

# Oracle9i Forms in a Java World

An Oracle White Paper

*July 2002*

## INTRODUCTION

Oracle9i Forms is a product with a long pedigree that has constantly kept up with the latest trends in application deployment from green screen “Block Mode” 3270 and “Character Mode” VT terminals, through Client-Server and most recently to deployment on the Web. Throughout this long evolution, applications written in Forms have been able to adapt and move forward as well, with customer investment in code being preserved throughout the whole process. No other declarative tool has been able to carry its code base forwards in such a consistent and painless manner.

However, the world is changing, all focus now is on the two rival architectures: Java 2 Enterprise Edition (J2EE) and Microsoft’s proprietary .NET platform. Standards rule the day and buzzwords like XML and Web Services are on everyone’s lips. Where does Forms go now?

Oracle’s strategy has changed somewhat as well. For a long, long time, Forms was sold as a standalone product, loosely coupled with its stable mates, Oracle Reports and Graphics, but now Forms is just one small part of a much larger integrated product set, the Oracle9i Application Server (Oracle9iAS).

In response to the changing market and the bundling of Forms into the greater whole, the next stage in the evolution of Forms is clear. The focus has to be in providing integration points between Forms applications and its new deployment platforms, specifically Oracle9iAS and J2EE.

Within Forms itself, this has already taken place. Forms consumes many of the services provided by the Oracle9iAS platform, such as single sign-on, enterprise management, and source control. But what about applications written in Forms and the Forms development environment? That is the area that this paper aims to cover. How you can integrate your forms Applications into a wider J2EE environments, and how the Java skills that you will leverage to do that can also help in the day to day development process.

This paper is split into three topic areas:

- Java Integration on the Client
- Java Integration on the Application Server

Which deal with the problems of writing applications that can integrate into a J2EE world, and;

- Java and XML at Design Time

Which shows how skills need never be wasted and how Java and XML can make a programmer's life simpler at design time.

## **JAVA INTEGRATION ON THE CLIENT**

Oracle9i Forms Services enables applications to be run as Internet applications through the use of a unique Java based client and application server architecture. This architecture employs the use of a powerful application server to execute the application logic and a generic Java client that is able to render the user interface for any Forms application. The Oracle9i Forms Java client facilitates the interaction between the end user and the application running on the application server using an intelligent message passing mechanism which utilizes caching and compression to reduce network traffic.

Because this Java client is generic, there is no opportunity, by default, to code for certain activities to happen on the client tier rather than on the application server tier where the Form is actually executed. However, there is a need for Forms programmers to be able to interact with the client browser machine, and in this section of the paper we'll discuss how that can be done.

### **Why Execute Code on the Client?**

One of the great benefits of the Forms architecture is the fact that the Java client is generic and highly optimized. You don't have to generate separate applets (each of which would need to be downloaded and cached separately) for each Form or application that you wish to deploy. However, realistically you may find that the supplied Forms applet is too restrictive for your particular needs. With that in mind, Forms has been designed with extensibility built-in.

The common reasons that developers want to extend the capabilities of the client are:

- Customizing the User Interface in some way. It may be that there is a requirement to enhance an existing Forms Widget, for instance, adding spell checking capability to a text area. Or a totally new type of widget that Forms does not supply is required, such as a spin-box control.
- Interacting with the client-browser operating system. When a Form is deployed on the Web, there is often the requirement to be able to interact with the machine that is running the browser that hosts the Forms applet, for instance to allow file upload, or to integrate with some device attached to the client machine such as a barcode scanner.
- Handling asynchronous events. The architecture of the Forms Java client provides opportunities for the interaction between external

programs or queues and Forms. An example of this is the ability to write code that will listen for events coming from the Oracle advanced queuing (AQ) mechanism, which can then be passed through to the Forms runtime process. This is a new area of connectivity that was impossible with conventional character mode or client server deployed Forms.

## **How to Do Client Side Integration**

Oracle9i Forms provides two mechanisms for extending the client with Java. The Pluggable Java Component (PJC) mechanism, which was introduced with Forms 6i, and the newer Oracle9i Forms specific, enhanced JavaBean support mechanism.

### **Pluggable Java Components**

Pluggable Java Components provide a low level API, which allows you to extend the Forms Java client. Each Oracle Forms native UI component, as defined in the builder – a field, a button and so on, has an equivalent representative in Java. The Java representations of the Oracle Forms native UI components are created using the Java lightweight component model. Using the lightweight component model means that the Oracle Forms Java UI components are rendered completely, rather than relying on the peer UI objects provided by the windowing system of the client operating system. This means that the Oracle Forms Java UI components appear visually the same and have similar behavior across different client operating systems.

Each Oracle Forms Java UI component is implemented using two different classes; a Handler class and a View class. This two class representation is a variation of the standard Model-View-Controller design (MVC) pattern but still adheres to the general goal of separating the data storage aspect from the visual display aspect. In the Oracle9i Forms services architecture the Handler class acts both as the Model and Controller whilst the View class acts as the View.

The Handler class is responsible for both maintaining the current value of any data and controlling the visual representation of the data. All server based interaction with the View class is conducted through the Handler class. The Handler class may register itself as an event listener for events that are generated by the View class. The Handler class itself interacts directly with the message dispatcher to send and receive messages to and from the Oracle9i Forms Services.

The View class is singularly responsible for presenting the data to the user in some manner and handling user input. The View class may allow the data to be changed by the user. This is dependent on the type of UI component the View class is representing and the properties it has set. The View class propagates any data changes made back to the Handler class using the Java event model. It is this view class that the Pluggable Java Component mechanism allows you to replace.

### ***The IView Interface***

To enable the Handler class to interact with and control the View class, the View class implements a public interface; `oracle.forms.ui.IView`. This interface describes all of the methods the Handler class uses to manage and interact with a View class including lifecycle, property manipulation, event handling and component display methods.

Any Java class that is to be used within the Oracle Forms Java Client must provide an implementation of this interface. All Oracle Forms Java UI components implement this interface. Because a Pluggable Java Component is just a different View class, it must also provide an implementation of this interface.

Figure 1 contains the definition of the IView interface.

```
public void init(IHandler handler);
```

This method is called immediately after the object is constructed. This method passes the object a reference to its Handler and gives it a chance to perform any initialization that it requires.

```
public void destroy();
```

This method is called when the object is no longer required. This method gives the object a chance to free up any system resources, that it holds.

```
public Object getProperty(PropertyID id);
```

This method returns the value of the requested property. Each View class must support the properties listed in the following sections. If the requested property is not supported by this Object, this method must return null.

```
public boolean setProperty(PropertyID id, Object value);
```

This method sets the value of the specified property. Each View must support the properties listed in the following sections. If the requested property is not supported by this Object, this method must return false, otherwise it must return true.

```
public void addListener(Class type, EventListener listener);
```

This method adds a listener of the specified type.

```
public void removeListener(Class type, EventListener listener);
```

This method removes a listener of the specified type.

```
public void paint(Graphics g);
```

In this method, the View must paint itself, using the AWT Graphics object provided. For subclasses of Component, this method is called by the Component's Container. For other Objects this method will be called by the Object's Handler

```
public void repaint(Rectangle r);
```

In this method, the View must invalidate the rectangle provided. If the rectangle is null, the entire object should be invalidated.

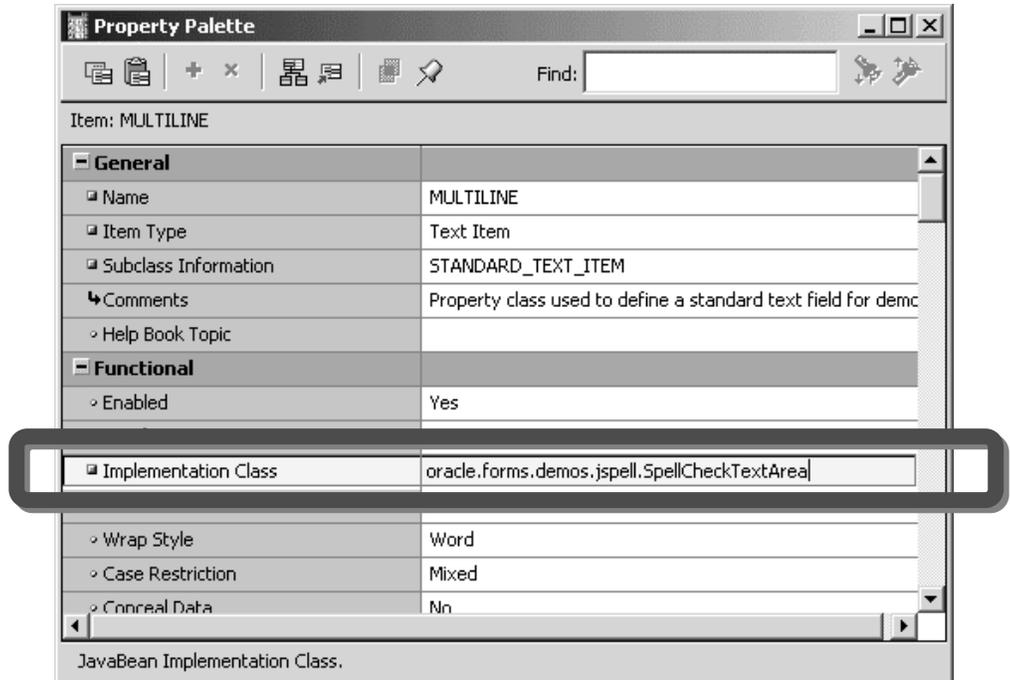
### Figure 1: the IView interface definition

Although this all seems a little complex, it is generally a lot simpler than writing code which implements this IView interface from scratch. You can simply sub-class one of the “standard” item types such as a button and just make the changes you require, whilst letting the “VButton” super-class handle all of the complex issues such as events and all the standard button properties.

To make the process even simpler, Oracle9i JDeveloper has a PJC wizard which will take you through the process of sub-classing a existing item type to create a specialized widget, or will even help you write a JavaBean from scratch that correctly implements the IView interface.

### Using the PJC

Once you have written a PJC to extend the Forms client, you actually have to tell Forms to use it. To do this you need to set the Implementation Class property of the item that you wish to have replaced with your custom code at runtime.



### Figure 2 - Setting the Implementation Class

In this case a multi-line text item is being implemented with a custom PJC class `oracle.forms.demos.jspell.SpellCheckTextArea`. Note how the class name is fully qualified with the package that contains the class (if it has one).

The final piece of the puzzle is to include the Java class somewhere where Forms can find it to download at runtime. This will usually involve placing the code into a JAR file and adding it to the Archive tag in the Formsweb.cfg file.

### ***Interacting with the PJC at Runtime***

The API for interacting with a PJC from PL/SQL is very simple: there are two built-ins and a trigger, which comprise the interface.

The built-ins are `Get_Custom_Property()` and `Set_Custom_Property()` which call directly through to the `getProperty()` and `setProperty()` methods defined in the `IView` interface. You can react to calls to these built-ins by setting or getting properties or executing Java methods inside the PJC code.

The Trigger is the `When-Custom-Item-Event` trigger, which is executed on the PJC item in the application server tier when the PJC Java code in the client makes a call to the `dispatchEvent()` method provided by the PJC mechanism.

For more information about writing PJCs, and many samples of code, see the Forms pages on [otn.oracle.com](http://otn.oracle.com).

### **Enhanced JavaBean Support in Oracle9i Forms**

Although the PJC mechanism provides an extremely useful API it requires that you write bespoke Java code to extend the client. So in Oracle9i Forms a new simpler mechanism was introduced to allow you to integrate JavaBeans and Java Applets as custom components without writing any Java Code at all.

This is all handled using a PL/SQL package in Forms called `FBean`.

`FBean` provides a full PL/SQL API to allow interaction with any public methods, properties and events that a `JavaBean` or `Applet` exposes.

### ***Enabling a Bean***

The Enhanced `JavaBean` support feature works entirely at runtime from PL/SQL code. In order to embed a `JavaBean` into your Form, all you will need to do is create a `Bean Area` of the size and in the position required for the bean you want to use. There is no need to set the implementation class property of the `Bean Area`, the assignment of the bean happens in code.

In order for a `JavaBean` to be displayed in the `Bean Area`, you have to register it with the Form. This uses a function in the `FBean` package called `FBean.Register_Bean()`. This takes the name and instance number of the `Bean Area`, and the fully qualified class name of the `JavaBean`.

For example to register an `FTP JavaBean` in a `BeanArea` called `FTPArea` in the block `B1` you would use a command like:

```
FBean.register_bean( 'B1.FTPArea' , 1 , 'ftp.FtpBean' );
```

The second argument (1 in this case) refers to the instance number of the field. In a single row block this will always have the value of 1, but in a multi-row block

there will be 'n' instances of the field where n is the number of physical rows displayed in the block and you can register different beans for each instance.

The JavaBean being used in this case is called FtpBean and it's in the Java package ftp.

When Register\_Bean() is called, Forms will instantiate the JavaBean into the specified Bean Area in the Form (note the JavaBean may not have any actual User Interface, that does not matter) and Forms will introspect the JavaBean to find out what methods, properties and events it exposes.

### **Setting and Getting Properties**

Once the JavaBean is instantiated using Register\_Bean() then code can be used to manipulate it. The FBean package has overloaded functions and procedures for getting and setting properties on the JavaBean:

- FBean.Get\_Char\_Property()
- FBean.Get\_Bool\_Property()
- FBean.Get\_Num\_Property()
- FBean.Set\_Property() (Overloaded to take Chars, Numbers or Boolean).

As well as these simple gets and sets, the FBean package also has methods for getting and setting *indexed* properties, where an indexed property is actually an array rather than a simple scalar value.

Back to the FTP example, here is a line of code used to set the target directory on the remote FTP server:

```
FBean.Set_Property( 'B1.FTPArea', 1, 'directory',  
                  '/downloads/patch1' );
```

The property being set here is called "directory" (note that the method names and property names used by the JavaBean are case sensitive), the code has set the destination directory to be the downloads/patch1 directory

### **Executing Methods**

Methods on the JavaBean are executed using one of the variety of FBean.Invoke methods. Like the property getters and setters there are various versions of this method which handle calling methods on the JavaBean that return different datatypes (or return void):

- FBean.Invoke\_Char()
- FBean.Invoke\_Bool()
- FBean.Invoke\_Num()
- FBean.Invoke()

Arguments can be passed to methods in one of two ways; either a special argument list can be built, to which each argument is passed in turn, and the completed list given to the FBean.Invoke command, or the arguments can be concatenated together into a delimited string, which the FBean package will parse into the individual arguments. To illustrate this, the following code fragments do exactly the same thing, which is connect to a remote FTP server with a username and password.

Using an arglist:

```
Declare
  argList FBean.ArgList;
Begin
  argList := FBean.Create_ArgList;
  FBean.Add_Arg(argList, 'downloads.oracle.com');
  FBean.Add_Arg(argList, 'anonymous');
  FBean.Add_Arg(argList, 'user@anyco');
  FBean.Invoke('B1.FTPArea',1, 'ftpConnect', argList);
End;
```

**Figure 3 – Calling a method on a JavaBean using an argument list**

Using a delimited argument string:

```
Begin
  FBean.Invoke('B1.FTPArea',1, 'ftpConnect',
    "downloads.oracle.com", "anonymous", "user@anyco");
End;
```

**Figure 4 – Calling a method on a JavaBean using a delimited argument string**

In most cases the delimited argument string is the simplest to use!

#### ***Enabling and Reacting to Events***

JavaBeans can raise events to be handled by their container. The enhanced JavaBean support allows you to choose which events you want Forms to be interested in. By default, all events are ignored, this is to reduce the amount of network traffic that is sent between the Application Server and the Forms Java Client. To register an interest in the event, FBean has an Enable\_Event() procedure which can be used to register (or un-register) to receive events. When an interesting event is raised by the JavaBean, a When-Custom-Item-Event trigger will fire in the Form, and information about the event and the associated event object from the JavaBean are made available. The programmer can then write code in the When-Custom-Item-Event trigger to examine this information and react accordingly.

## **JAVA INTEGRATION ON THE APPLICATION SERVER**

In this section of the paper we'll look at Java Integration from the perspective of coding on the Application Server, or "Middle Tier" as it's sometimes known. To facilitate the integration of Java Code into the PL/SQL coding used by Forms Programmers, Forms provides an interface package called Ora\_Java. This package acts as a wrapper around the functionality provided by the Java Native Interface (JNI) which allows other languages such as 'c' to call into Java.

Raw Ora\_Java or JNI coding is not much fun, however, so the Form builder helps things along with a facility called the Java Importer.

### **The Java Importer**

The Java Importer is the name given to a set of components that has been added to the Oracle9i Forms Developer and Oracle9i Forms Server. 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:

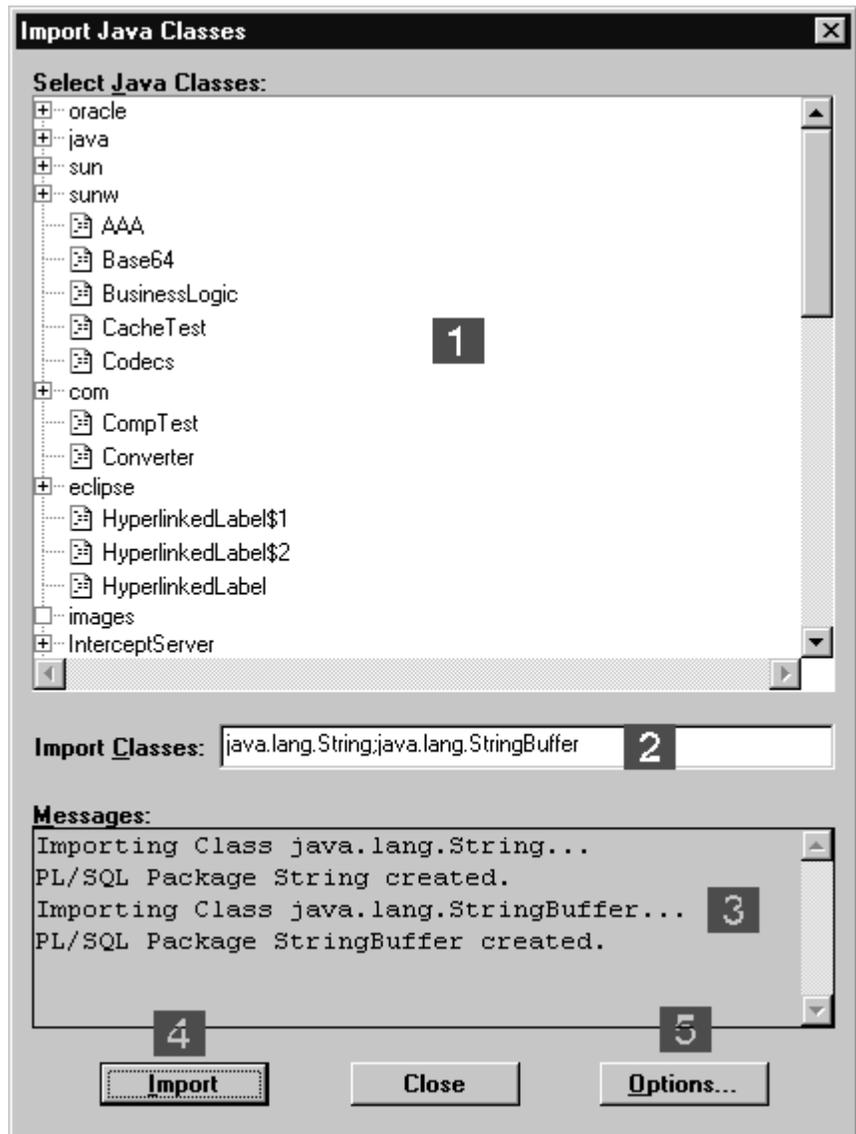
- The Java Importer Tool, which allows a developer to select and specify which Java classes they wish to access in their application.
- The Java Importer Generator, which creates PL/SQL packages that provide access to the specified Java classes.
- The ORA\_JAVA package, which provides a set of convenience functions that assist a developer in working with the selected Java packages.
- The Oracle Forms 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 Forms builder and Forms Services.

### **The Java Importer Tool**

The Java Importer Tool is a dialog in the Forms builder 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 5 - 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

#### ***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_OBJECT` 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 6 - 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 7 - Snippet of PL/SQL package spec for String**

The `valueOf()` PL/SQL function represents a Java class (Static) 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 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 8 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 8 - 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.

### **When to use Java Code From Forms on the Application Server**

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.

- **Memory:** Each connected processes that calls imported Java code will have to instantiate 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.
- **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 its recommend that you don't try and attempt to replace large portions of default Forms functionality with imported code.

- Maintenance: It's a good idea to only use Java code where you are attempting to carry out an 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.

## **JAVA AND XML AT DESIGN TIME**

In this final section of the paper I'll deviate somewhat from the theme so far which has been how to integrate Forms with Java Code in various ways. Here, instead we'll have a quick look at some of the changes to the design time environment, which touch onto the Java and XML worlds, showing that the skills you learn to help with solving runtime coding problems can also benefit by making the build-time more productive.

### **The Forms Design-Time Java API**

The Forms Java Design-Time API, known as the JDAPI, is a new feature introduced in Oracle9i Forms. It is a Java API for programmatically loading, creating, manipulating, saving and compiling Oracle Forms applications.

The JDAPI is built on top of the existing Forms C API and gives Java programmers a way to work with Forms without needing to learn C. Likewise, JDAPI is a complementary tool to the Form Builder and can be used to programmatically accomplish anything which can be achieved visually using the Builder.

It is important to note that the JDAPI is not a run-time API and cannot be used to program, interact with or control running Forms applications. It is a design time only tool

Oracle9i Forms is an object-based 4GL (declarative) development tool for building GUI database applications. All the components of the environment are represented in the development environment as individual, self-contained objects, that have properties which define their appearance and behavior. These objects are normally created and manipulated visually using the Form Builder; they are also represented as classes in JDAPI in order to achieve the same ends programmatically.

### **Advantages of Using JDAPI**

JDAPI offers all the advantages of the Java 2 platform and APIs for writing high-level, object-oriented programs which process Forms modules (Form Modules, Menu Modules, PL/SQL Libraries and Object Libraries). Java is increasing in popularity and JDAPI allows Java developers to apply their knowledge and the standard Java APIs to Forms programming.

In addition there are some utility features and constructs not available elsewhere. For example, it is possible to construct Forms objects directly on Object Library Tabs, and thus build Object Libraries without using intermediate Forms Modules, a feature not available either in the C API or the Form Builder.

Using JDAPI it is possible to create utility programs to perform actions on a large number of forms automatically. For example, you could write a Servlet, which

takes a named Forms application and publishes information (for instance its external dependencies) about it to the web, or renders a layout preview bitmap.

#### Example of JDAPI code

Although there is not room in this paper to give anything other than a short overview of the JDAPI I've included a short piece of example code to illustrate how simple and understandable JDAPI code is even for programmers with minimal Java experience. A large number of samples of more complex JDAPI coding are shipped with the Oracle9i Forms Demos.

```
try
{
    // Open the module - this implicitly starts JDAPI
    FormModule form = FormModule.open("C:\APPS\EMPF\M\FMB");
    // Call the getName method on the new Form object
    System.out.println("Form Name is " + form.getName());
    // Clean Up
    Jdapi.shutdown();
}
catch (JdapiException jdex)
{
    System.out.println(jdex.toString());
}
```

**Figure 9 – Using the JDAPI to extract the name of a Form**

#### Forms as XML

As a final foray into the world of Java and XML we'll look at the rendering of Forms source code as XML files.

Oracle9i Forms offers a collection of tools that allow you to:

- Convert a Forms Module into a specified XML format
- Convert an XML file into a Forms Module
- Generate a file containing the Forms XML Schema
- Validate a specified XML file against the Forms XML Schema

The tools are written in Java and use the Forms JDAPI as the interface to Forms. When you convert a Forms Module, the Forms to XML conversion tool uses the Oracle XDK to create and parse the XML. You can then use any standard tool to parse and manipulate the resulting XML (Even a text editor!). The tool also lets you generate a Forms XML Schema document, which you can use to validate any XML against.

These conversion tools make it possible to perform operations on Forms files that were impossible before. For example, you cannot diff and merge Forms binary source files themselves. However, you can use the tools to convert two FormsModule files to XML, diff and merge them, validate the result, and convert it back to a Forms .FMB.

The conversion tools processes modules generically. This means that the Forms to XML conversion tool can convert a Forms module, Object Library, or Menu module to its XML representation. Similarly, the XML-to-Forms tool can convert an XML file to a Forms module, Object Library or Menu module. PL/SQL Libraries are not handled by this tool as they have no structure that can be usefully represented in an XML format. The existing text based format (.pld) of a PL/SQL library is sufficient for browsing and editing.

As the action of Converting modules to XML, editing the XML Definition, and converting back to binary format is fully supported (providing the XML is validated), the use of these XML utilities provides a real alternative to both the C and JDAPI APIs as a way of carrying out bulk edits to module as well as being a documentation tool.

## **CONCLUSION**

This paper has only been able to scratch the surface of the opportunities to use Java both within the Oracle9i Forms runtime, and the development environment as well. Access to Java from within the product opens up a vast range of possibilities for application integration and functionality that have never been available until now. Thus Forms, as a product, is able to coexist and even thrive even in this Java-centric world.

Much, much, more information and detail on PJC's, Enhanced JavaBean support, the Java Importer, JDAPI and the Forms to XML converter are available on the Oracle Technology Network – <http://otn.oracle.com> and in the online help for Forms. If your interest has been piqued go and have a look!



Oracle9i Forms in a Java World

05/JUL/2002

Duncan Mills

Contributing Authors:

Steve Button

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

[www.oracle.com](http://www.oracle.com)

Oracle is a registered trademark of Oracle Corporation. Various product and service names referenced herein may be trademarks of Oracle Corporation. All other product and service names mentioned may be trademarks of their respective owners.

Oracle is the world's largest enterprise software company. For more information about Oracle, visit our Web site at <http://www.oracle.com>.

Copyright © 2001 Oracle Corporation

All rights reserved.