An Oracle White Paper
March 2009

Introduction to Groovy Support in JDeveloper and Oracle ADF 11*g*

Introduction

Oracle JDeveloper and Oracle ADF 11*g* introduce support for the Groovy scripting language, allowing Groovy expressions to be used in attribute validation and as a source for attribute values. This paper gives and overview and examples of the support.

Introduction to Groovy

Groovy is as an agile dynamic language for the Java platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax. Being a dynamic language, many features of the language are executed at runtime as opposed to the compile-time, strongly typed language of Java.

The main points to take from this are that: where Java can be very "exacting" in how you develop, Groovy is a little more forgiving and hence "agile". Groovy runs inside the JVM, and so can make use of the Java environment libraries; and give a Java-like language (strictly speaking, it is a superset of Java) that provides a convenient and agile development experience.

For a starter's guide on Groovy you can visit
http://groovy.codehaus.org/Getting+Started+Guide

Groovy in JDeveloper and Oracle ADF.

Where Groovy comes into its own is within Oracle ADF Business Components is helping to bridge the gap between Oracle ADF's declarative business logic, and where you have to get into full blown Java coding. Furthermore, because Groovy is interpreted at runtime, it can be treated as metadata and customized. So, you could have an application that has customized business rules in Groovy and those customizations stored in Oracle ADF's meta data services (MDS). This means, for example, if you built your application for reselling then your application could be using different business rules depending on the customer instance of the application.

Oracle ADF provides a number of different declarative points into which you can drop Groovy code:

- Values for view and entity objects attributes
- Validation rules on entity object attributes
- Expressions for error messages

- Values for view object bind variables
- View object transient attribute validation

Groovy expressions in ADF Business Components Entity Objects

Groovy expressions can be placed in a number of different places in ADF Business Components. For the main part, you will probably be applying them in entity and view object attribute values and for entity object validations.

Referencing attributes in the same entity

The simplest example is to reference attributes in the same entity. So, for the transient attribute *AnnualSalary* you could define a Groovy expression for the default to be:

```
Sal * 12
```

You can also use a Groovy expression for a validator. So, on the *Sal* attribute you might want to define that a salesman must have a salary of less than 1000:

```
if (Job == "SALESMAN")

{return newValue < 1000}

else

return true
```

When you reference an attribute in a validator, the attribute-level validator fires before the new value being validated is actually assigned to the attribute (which makes sense, since you want to validate it before the attribute is set). So, this means if you want to refer to the new value you use the expression *newValue*. In the Groovy support in ADF Business Components, *newValue* returns the value the user has just input. You can also use *oldValue* to return the value before it was changed.

Referencing custom methods in the EntityImpl class

If you have defined a method in your EntityImpl class, then this can be called as part of your expression for an attribute default:

```
adf.object.getDefaultSalaryForGrade()
```

Unlike when referencing an attribute of the entity object, which can be done by simply referencing the attribute name, when referencing a method, you are required to prefix the method name with `adf.object`.

If you want to reference the same method from a validator, you have to use the `source` prefix.

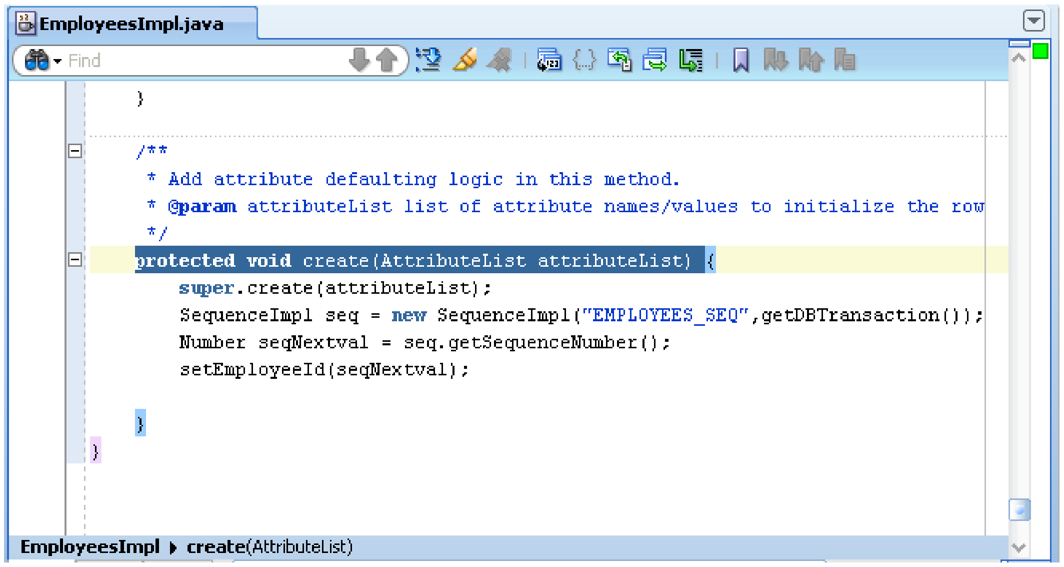
```
source.getDefaultSalaryForGrade()
```

For example, you might use the following expression for a more dynamic check of the salary for a salesman.

```
if (Job == "SALESMAN")
{
return newValue < source.getMaxSalaryForGrade(Job)
}
else
return true
```

Referencing base class methods in the EntityImpl class

In the previous example you were referencing your own custom methods in the EntityImpl class. You can also reference base class methods as you would if you were coding directly in the EntityImpl. For example, in the *create* method you may want to set the *employeeId* to a sequence number retrieved from the database.



```
    }

    /**
     * Add attribute defaulting logic in this method.
     * @param attributeList list of attribute names/values to initialize the row
     */
    protected void create(AttributeList attributeList) {
        super.create(attributeList);
        SequenceImpl seq = new SequenceImpl("EMPLOYEES_SEQ",getDBTransaction());
        Number seqNextval = seq.getSequenceNumber();
        setEmployeeId(seqNextval);

    }
}
```

Figure 1. Setting employeeId from a database sequence

This is done in the EntityImpl by creating a sequence object and then calling *getDBTransaction*. – as shown in figure 1

The same calls can be made using a Groovy expression as the value for the *EmployeeId* attribute:

```
(new oracle.jbo.server.SequenceImpl("EMPLOYEES_SEQ",
adf.object.getDBTransaction())).
getSequenceNumber()
```

Note that you are required to use the full name for the class (*object.jbo.server.SequenceImpl*) and *object* is used as a reference to the EntityImpl.

Of course, in practice you might create an EntityImpl helper method:

```
protected oracle.jbo.domain.Number nextVal(String
sequenceName) {
SequenceImpl s = new
SequenceImpl(sequenceName,getDBTransaction());
       return s.getSequenceNumber();
    }
```

Then, the groovy expression can just be:

```
adf.object.nextVal("YOUR_SEQUENCE_NAME")
```

So, any methods that you call in the EntityImpl can also be called using Groovy. So, consider the (slightly contrived) example of wanting to set the value of an attribute to the label of another attribute. Using the following code you are calling the EntityImpl method and getting the label, using the correct locale so you can pick up translated labels, and assigning to the *lastName* field.

```
adf.object.setEmail(adf.object.getAttributeHints("LastName").g
etLabel(adf.object.getDBTransaction().getSession().getLocaleCo
ntext()))
```

However, there is also another way of accessing the same information:

```
adf.object.hints.LastName.label
```

The keyword *adf.object* evaluates to the EntityImpl instance. The keyword *hints* refers to a Java bean property getter for a property named *hints*. So, *adf.object.hints* accesses the hints bean property and returns a *java.util.Map*. Like Groovy does with any Map object, it uses the next "element" in the dotted syntax to lookup the map key of that name.

So *adf.object.hints.LastName* looks up the entry in the hints Map named "LastName", which is another Map of all the UI hints available for this particular attribute. So, again as Groovy does with any Map, it uses the next element in the dotted syntax to lookup the map key of that name.

So *adf.object.hints.LastName.label* looks up the entry in the *adf.object.hints.LastName* Map with the key of "label".

You can compare how much simpler:

```
adf.object.hints.LastName.label
```

is to its Java equivalent:

```
getAttributeHints("LastName").getLabel(getDBTransaction().getS
ession().getLocaleContext());
```

Using Groovy expressions in error messages

As part of the validation for an entity object, you can define placeholders in the error messages that are evaluated through Groovy expressions. So, for example, if you want to reference the label and tooltip of a field as part of a validation error message you could create an error message with placeholders as: "This is not a valid value for {x}. For this field you need to {y}." The values of x and y could come from Groovy expression (as noted before) such as:

```
source.getAttributeHints("HireDate").getLabel(source.getDBTran
saction().getSession().getLocaleContext())
```

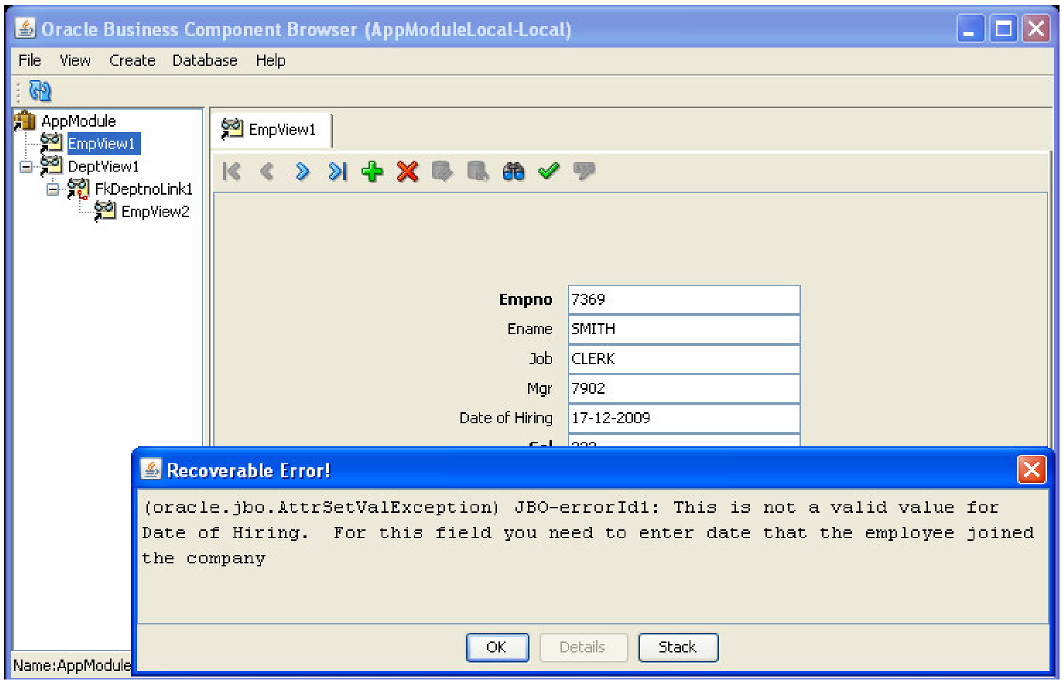
```
source.getAttributeHints("HireDate").getTooltip(source.getDBTr
ansaction().getSession().getLocaleContext())
```

or using the briefer notation of:

```
source.hints.HireDate.label
```

```
source.hints.HireDate.tooltip
```

Thus, for an error on *Hiredate* you can refer to the label for the field and the tooltip text as part of the error message.



```
(oracle.jbo.AttrSetValException) JBO-errorId1: This is not a valid value for
Date of Hiring.  For this field you need to enter date that the employee joined
the company
```

Figure 2. Customizing an error message by calling an entity attribute control hint

Referencing attribtues in other entities

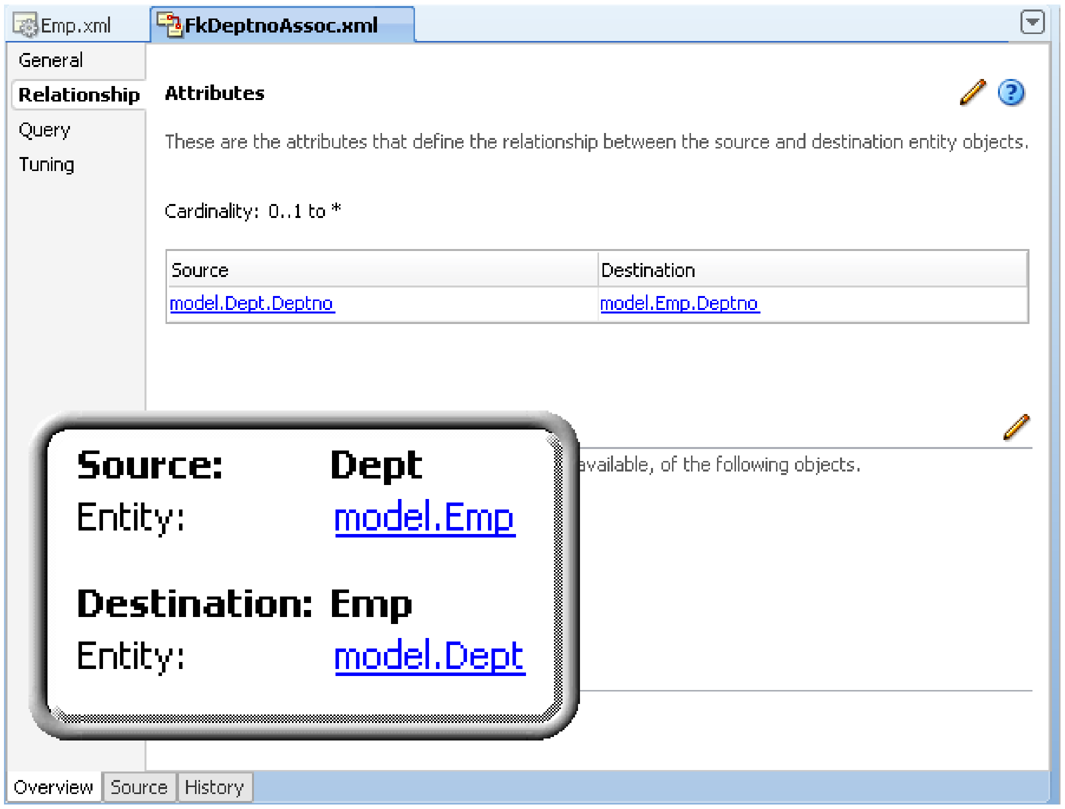
You can also reference attributes in other entities. For example, a new employee's default salary has a weighting depending on their location. So, the call to `getDefaultSalaryForGrade` will pass the department location as a string.

This is achieved by referencing the accessor in the entity association. This allows the Groovy expression for the employee entity object to "walk back" to the master department entity. For example:

```
adf.object.getDefaultSalaryForGrade(Dept.Loc)
```

So, assuming a master/detail relationship between departments and employees with this expression as the default for the *Employees.Sal*, it will call the Employees EntityImpl method and pass in the value of *Loc* from the master.

Note, you are not referencing the name of the entity (Dept), it just happens to be that, by default, when reverse engineering from the database, the name of the association that links Dept, and Emp is called *FkDeptnoAssoc* and the accessors are called *Dept* and *Emp*. This is shown in figure 3.



Attributes

These are the attributes that define the relationship between the source and destination entity objects.

Cardinality: 0..1 to *

Source	Destination
model.Dept.Deptno	model.Emp.Deptno

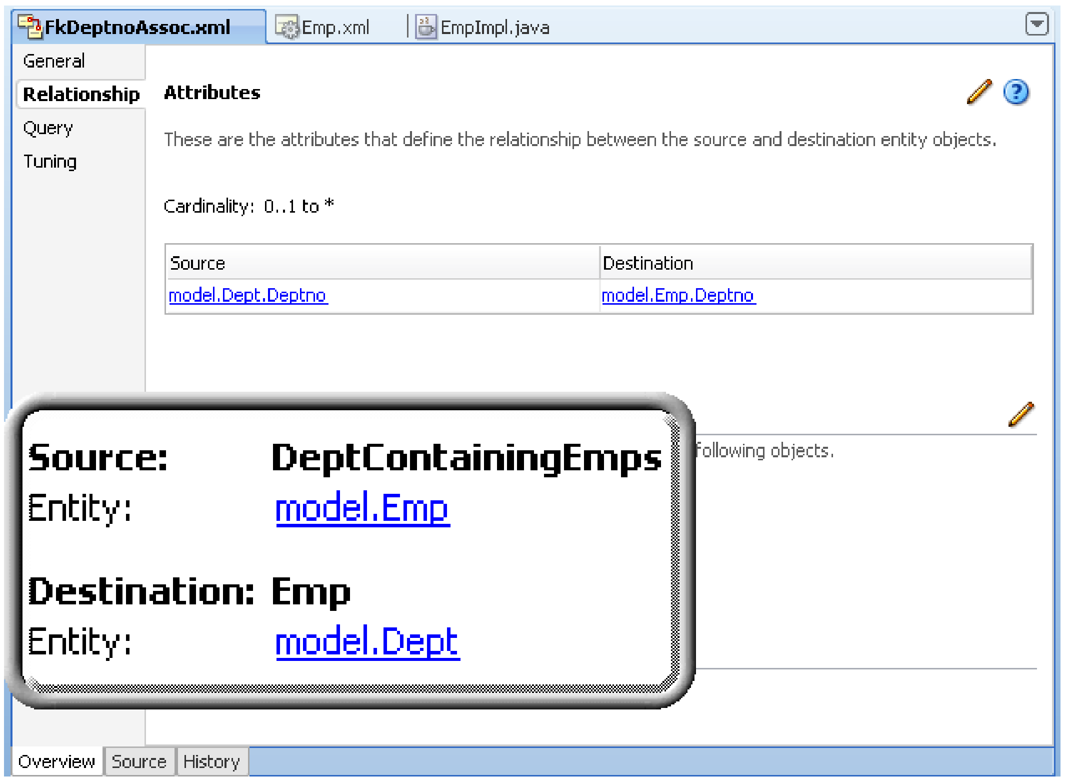
available, of the following objects.

Source:	**Dept**
Entity:	model.Emp
Destination:	**Emp**
Entity:	model.Dept

Figure 3. Entity association

However, if you were to change the name of the source accessor to *DeptContainingEmps* as shown in figure 4, then the Groovy call would be:

```
adf.object.getDefaultSalaryForGrade(DeptContainingEmps.Loc)
```



Attributes

These are the attributes that define the relationship between the source and destination entity objects.

Cardinality: 0..1 to *

Source	Destination
model.Dept.Deptno	model.Emp.Deptno

following objects.

Source:	**DeptContainingEmps**
Entity:	model.Emp
Destination:	**Emp**
Entity:	model.Dept

Figure 4. Entity association using a different name for the accessor

So, it is the name of the accessor that is allowing the Groovy expression to "walk" the relationship between objects.

Note that there currently does not exist a mechanism to refactor changes in the accessor name and to carry those through to the Groovy expressions. So, if after writing a Groovy expression you change an accessor name (or infact any other artifacts such as attribute name) then you will have to manually change the Groovy expression as well.

Referencing built-in calls

There are also a number of "built-in" features that you can reference.

Date and time

```
adf.currentDate
```

```
adf.currentDateTime
```

The return the current data and time from the middle tier. So, if you wish the default hire date for a new employee to be defined as today's date you would simply define the expression as:

```
adf.currentDate
```

If you want to check that the hire date for a new employee is before today's date you would set a validator on *Hiredate* as:

```
return (newValue < adf.currentDate)
```

Aggregate functions

ADF Business Components provides aggregate functions for rows of data.

```
<Accessor>.sum(Groovyexpression)

<Accessor>.count(Groovyexpression)

<Accessor>.avg(Groovyexpression)

<Accessor>.min(Groovyexpression)

<Accessor>.max(Groovyexpression)
```

So, for the Dept entity object you could add a new transient attribute that displays the sum of all employees' salary in that department.

```
Emp.sum("Sal")
```

As with accessing attributes in other entities, you are using the name of the accessor (Emp) to reference the employees for a specific department. For an accessor called EmpAccessor you would call the following and could also pass in an expression:

```
EmpAccessor.sum("Sal + 20")
```

In the example the string provided as the argument to the sum() function is interpreted in the context of each detail entity in the collection over which the sum is occurring. So in this case, if this expression is on the Dept entity object, the parameter is being interpreted for each record of the Emp entity. This means that any expression you put here, and any attributes or methods you reference are on the Emp entity.

Imagine the use case where you want to add a new attribute to your Dept entity that calculates the salary for all of the employees in that department. The calculation of the salary is based on each employee's salary plus their benefits package, which is different for each job role.

Therefore if the following expression is put as the value of an attribute on the Dept entity object:

```
EmpAccessor.sum("Sal + adf.object.getBenefitsValue(Job)")
```

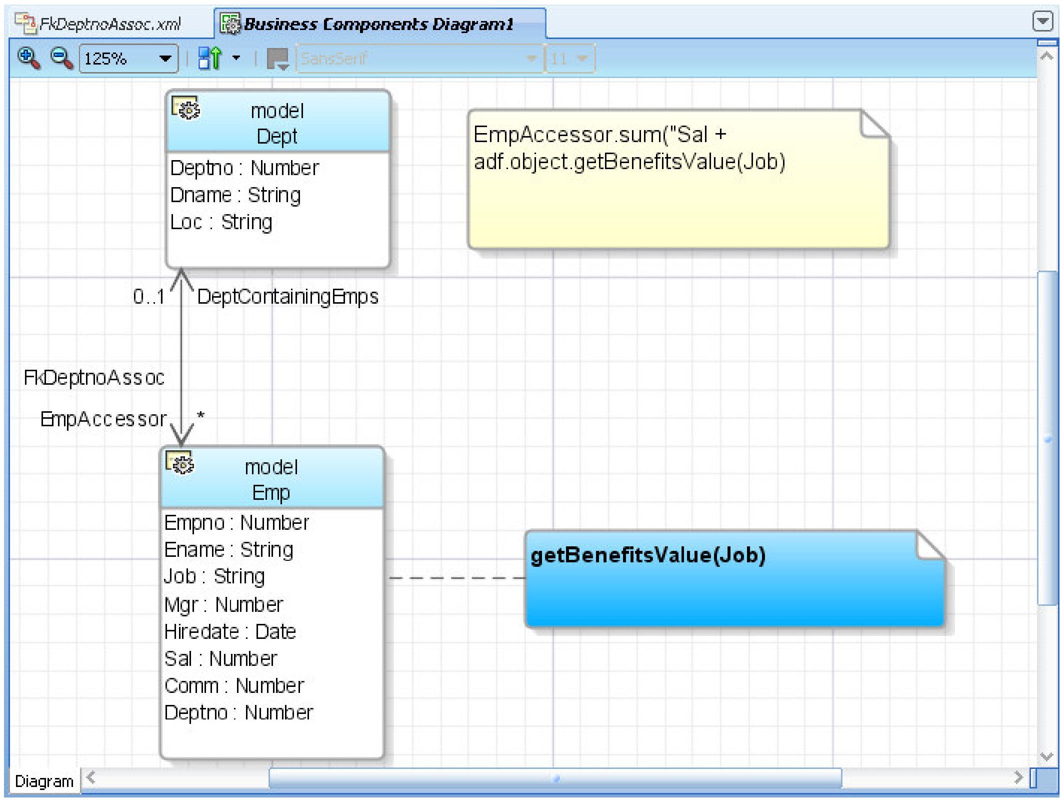


Figure 5. Business components diagram showing the object structure

Figure 5 shows that for this Groovy expression on the Dept entity (yellow box top right), the accessor EmpAccessor "walks" from the Dept entity to the Emp entity. From this point on the context is now within the Emp entity and so *getBenefitsValue* is referenced in the Emp EntityImp and the attribute *Job* is reference from the Emp entity object.

Walking objects using the dot notation

Taking these concepts one-step further, you can use accessors to "walk" up and down the business components. Consider a master detail Dept and Emp where you want to ensure that that the salary for an employee must be greater than the department minimum. You could have an Emp validation on *Sal* that is:

```
return newValue > DeptContainingEmps.Emp.min("Sal")
```

Thus, from the detail attribute, *Sal*, you are walking up the object tree to the master Dept and performing an aggregate function on the details of that master, which in this case in Emp.

Of course it is worth noting that if there were ten thousand employees you would be iterating over all those rows and so you may consider a more efficient means.

Ternary Operators

At this point it might be useful to introduce Groovy's ternary operators. This provides a shorthand way of implementer an if-then-else construct.

```
(<Conditional expression> ? <action> : <else-action>)
```

For example:

```
 (Job != "SALESMAN" ? newValue > 100 : newValue > 0)
```

This can be read as "if job is not a SALESMAN then the new value of the salary must be greater than 100 else, greater than 0.

This can be used to good effect in aggregate functions to ensure you don't calculate nulls:

```
EmpAccessor.sum("Comm != null ? Comm : 0")
```

Or if you wanted to count only the employees with a non-null commission over 300 you could do:

```
EmpAccessor.count("Comm != null && Comm > 300 ? Comm : null")
```

Raising exceptions and warnings

For a validation rule on an entity attribute, you can use a Groovy expression to decide whether an error or a warning message should be displayed using the following expression:

```
adf.error.raise("THIS_IS_AN_ERROR_MESSAGE")

adf.error.warn("THIS_IS_A_WARNING_MESSAGE")
```

So, for employee salary, you can define a rule that will allow the input of salary between 1000 and 5000 but should display a warning. Any salary over that value should display an error and require the input to be corrected.

```
if (newValue > 1000)

{

  if (newValue > 5000)

  {adf.error.raise("SALARY_TOO_HIGH_ERROR")

   return false}

 adf.error.warn("SALARY_LIMIT_WARNING")

 return true

}
```

```
else
{
return true
}
```

Entity object Groovy diagram

Figure 6 summarizes the object structure for using Groovy expressions in entity objects.

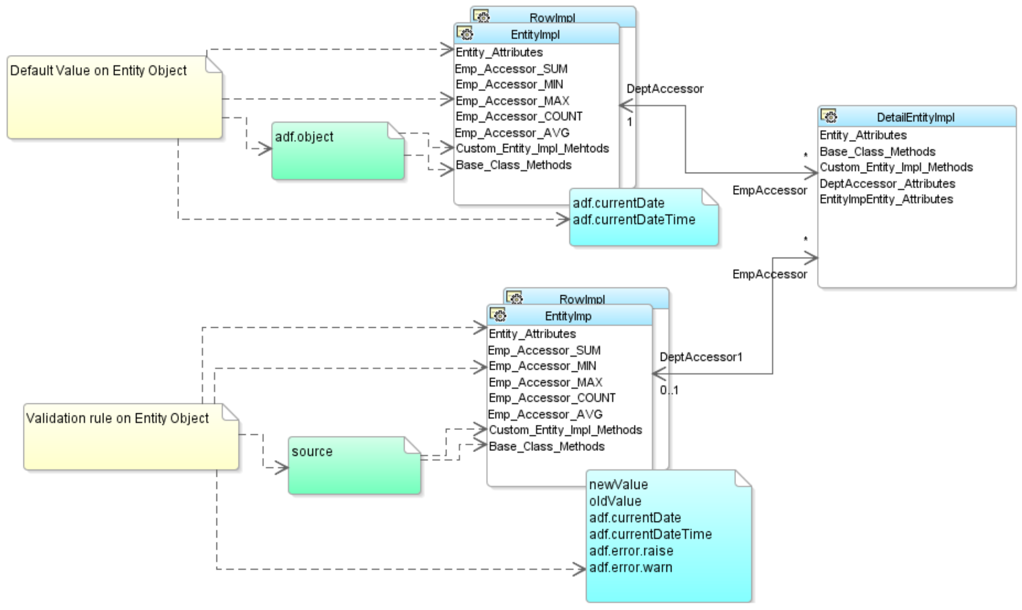


Figure 6. Object structure for Groovy expressions in entity objects

Groovy Expressions in ADF Business Components View Objects

ADF Business Components view objects also provide the ability to use Groovy expressions. As with entity objects, view objects can support:

- Referencing attributes in the same view object
- Referencing attributes in other view objects
- Transient attribute validation

- Referencing methods in the Java class that backs the view object
- Referencing built in calls such as *sum* and *min*

View objects also allow Groovy expressions to be used in bind variables. However, when a bind variable is being evaluated it depends on its context. So, if the default value of a view object's bind variable is being evaluated, then the context is the ViewObjectImpl class, so values of any particular view row (e.g. attributes of the view objects) are not available.

As noted, you can call methods in the Java class that backs the view object. The thing to note is that for the entity object it is the EntityImpl class. In a view object it is the ViewRowImpl not the ViewImpl that can be referenced. The EntityImpl and the ViewRowImpl both extend the RowImpl class. They both represents rows, whereas the ViewObjectImpl is an object that represents the query and has a default rowset embedded in it to manage the resulting ViewRowImpl instances returned from the query.

Conclusion

The Groovy scripting language gives a powerful, yet agile way of harnessing the power of the Java language in a way that it allows it to be easily incorporated into the productive framework provided by Oracle ADF. While Oracle ADF provides a framework that is helping abstract the complexities of the platform, Groovy expressions give the developer the ability to exploit that power in an agile development manner.

Introduction to Groovy Support in JDeveloper
and Oracle ADF 11g
March] 2009
Author: Grant Ronald
Contributing Authors:

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
oracle.com

Oracle is committed to developing practices and products that help protect the environment