Oracle® Cloud
Groovy Scripting Reference for Oracle Sales Cloud
Release 13.2
E51622-03

July 2014
This document explains the basics of using the Groovy scripting language to enhance your Oracle Sales Cloud offerings.
Contents

Preface ................................................................................................................................................................ v
  Audience ............................................................................................................................................................ vii
  Documentation Accessibility ............................................................................................................................... vii
  Conventions ........................................................................................................................................................ vii

1 Introduction
  Overview .......................................................................................................................................................... 1-1
  Terminology .................................................................................................................................................... 1-1
  Where You'll Use Groovy in Your Application .............................................................................................. 1-2

2 Groovy Basics
  Commenting Your Scripts ................................................................................................................................. 2-1
  Defining Variables ......................................................................................................................................... 2-3
  Referencing the Value of a Field in the Current Object ............................................................................... 2-3
  Working with Numbers, Dates, and Strings ................................................................................................. 2-4
  Using Substitution Expressions in Strings ................................................................................................... 2-5
  Using Conditional Expressions ................................................................................................................... 2-5
  Using the Switch Statement ......................................................................................................................... 2-6
  Returning a Boolean Result .......................................................................................................................... 2-7
  Assigning a Value to a Field in the Current Object ...................................................................................... 2-7
  Writing Null-Aware Expressions .................................................................................................................. 2-8
  Understanding the Difference Between Null and Empty String .................................................................. 2-8
  Understanding Secondary Fields Related to a Lookup .................................................................................. 2-9
  Using Groovy's Safe Navigation Operator .................................................................................................. 2-10
  Assigning a Value to a Field in a Related Object ........................................................................................ 2-10
  Printing and Viewing Diagnostic Messages ............................................................................................... 2-11
  Working with Lists ...................................................................................................................................... 2-12
  Working with Maps ..................................................................................................................................... 2-13
  Working with Ranges .................................................................................................................................. 2-14

3 Examples of Each Context Where You Can Use Groovy
  Providing an Expression to Calculate a Custom Formula Field's Value .................................................. 3-1
    Read-Only Calculated Fields ......................................................................................................................... 3-1
    Transient Value Holder Fields with Optional Calculated Initial Value ......................................................... 3-2
  Providing an Expression to Calculate a Custom Field's Default Value .................................................... 3-2
4 Groovy Tips and Techniques

Simplifying Code Authoring with the Expression Palette .............................................. 4-2
  The Expression Editor Toolbar .................................................................................. 4-2
  Showing the Expression Palette ............................................................................... 4-3
  Inserting an Expression for Fields of Related Objects ............................................. 4-3
  Inserting Field Name of Object in Related Collection ............................................ 4-4
  Inserting a Related Object Accessor or Related Collection Accessor Field Name ...... 4-4
  Hiding the Expression Palette .................................................................................. 4-5
Using the Related Object Accessor Field to Work with a Parent Object ......................... 4-5
Using the Related Object Accessor Field to Work with a Referenced Object ................. 4-6
Using the Related Collection Accessor Field to Work with Child Rows ..................... 4-7
Accessing Current Date and Time from the Application Server .................................. 4-8
Accessing Current Date and Time from the Database ................................................. 4-9
Understanding ADF's Additional Built-in Groovy Functions ....................................... 4-9
Understanding Groovy's Null-Safe Comparison Operators ....................................... 4-14
Testing Whether an Field's Value Is Changed ............................................................ 4-15
Avoiding Validation Threshold Errors By Conditionally Assigning Values .................... 4-15
Prefer "Before" Save-time Triggers to "After" Ones for Best Performance ....................... 4-16
Avoiding Posting Threshold Errors By Conditionally Assigning Values ....................... 4-16
Functional Restrictions in Trigger Scripts ................................................................. 4-17
Passing the Current Object to a Global Function ....................................................... 4-17
Referencing Original Values of Changed Fields ......................................................... 4-17
Raising a Warning From a Validation Rule Instead of an Error .................................... 4-18
Throwing a Custom Validation Exception .................................................................... 4-18
Returning Locale-Sensitive Custom Strings .................................................................. 4-18
Raising a Trigger's Optional Declaratively-Configured Error Message ......................... 4-19
Accessing the View Object for Programmatic Access to Business Objects .................. 4-19
Defining the Sort Order for Query Results ................................................................... 4-21
Finding an Object by Id ................................................................................................. 4-23
Finding Objects Using a View Criteria ........................................................................ 4-24
  Overview of Basic Steps to Use a View Criteria ....................................................... 4-24
  Simplifying Most Common View Criteria Creation with a Global Function ............... 4-25
  Understanding View Criteria Item Operators .......................................................... 4-26
  Using View Criteria Items with the ISBLANK Operator ........................................... 4-27
  Using View Criteria Items with the BETWEEN Operator ......................................... 4-27
  Understanding View Criteria Row and Item Conjunctions ...................................... 4-28
  Using Compound Criteria Item to Search for Multiple Alternatives ....................... 4-28
Preface

This document explains how to use the Groovy scripting language to enhance your Application Composer applications.

Audience

This document is intended for application developers that are new to Groovy scripting.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Conventions

The following text conventions are used in this document:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>boldface</strong></td>
<td>Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.</td>
</tr>
<tr>
<td><em>italic</em></td>
<td>Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.</td>
</tr>
<tr>
<td><code>monospace</code></td>
<td>Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.</td>
</tr>
</tbody>
</table>
This chapter provides a brief overview of the different contexts in which you can use Groovy scripts.

This chapter contains the following:

- "Overview"
- "Terminology"
- "Where You’ll Use Groovy in Your Application"

**Overview**

Groovy is a standard, dynamic scripting language for the Java platform for which the CRM Application Composer provides deep support. This document explains the basics of how you will use the Groovy scripting language to enhance your applications. This section provides a brief overview of the different contexts in which you can use Groovy scripts. The second section covers the basics of Groovy. The third section gives a concrete example of each type of Groovy script you can write. The fourth section offers a compendium of tips and techniques for getting the most out of Groovy in your applications, and the final section documents the supported classes and methods you are allowed to use in your Groovy code.

**Note:** Please read Supported Groovy Classes and Methods which documents the only classes and methods you may use in your Groovy scripts. Using any other class or method will raise a security violation error when you migrate your code to later Fusion CRM maintenance releases. Therefore, we strongly suggest that the Groovy code you write uses only the classes and methods shown there to avoid the unpleasant and possibly time-consuming task of having to rewrite your code in the future.

**Terminology**

Throughout the document the term *script* is used to describe one or more lines of Groovy code that the Oracle ADF framework executes at runtime. Often a very-short script is all that is required. For example, to validate that a CommissionPercentage field’s value does not exceed 40%, you might use a one-line script like:

```
return CommissionPercentage < 0.40
```

In fact, this one-liner can be conveniently shortened by dropping the keyword since the *return* keyword is always implied on the last line of a script:
For slightly more complicated logic, your script might require some conditional handling. For example, suppose the maximum commission percentage is 40% if the salesperson’s job grade is less than or equal to 3, but 60% if the job grade is higher. Your script would grow a little to look like this:

```groovy
if (JobGrade <= 3) {
    return CommissionPercentage < 0.40
} else {
    return CommissionPercentage < 0.60
}
```

Scripts that you’ll write for other purposes like complex validation rules or reusable functions may span multiple pages, depending on your needs.

When a context requiring a Groovy script will typically use a short (often, one-line) script, we emphasize that fact by calling it an *expression*, however technically the terms *script* and *expression* are interchangeable. Anywhere you can provide a one-line expression is also a valid context for providing a multi-line script if the need arises. Whether you provide a short expression or a multi-line script, the syntax and features at your disposal are the same. You need only pay attention that your code returns a value of the appropriate type for the context in which you use it. Each section below highlights the expected return type for the script in question.

## Where You’ll Use Groovy in Your Application

There are a number of different contexts where you will use Groovy scripts as you customize existing objects or create new custom ones. You will write shorter scripts to provide an expression to:

- calculate a custom formula field’s value
- calculate a custom field’s default value
- make a custom field conditionally updateable, or
- make a custom field conditionally required
- define the condition for executing an object workflow

You will generally write somewhat longer scripts to define:

- a field-level validation rule
- an object-level validation rule
- a trigger to complement default processing
- utility code in a global function, or
- reusable behavior in an object function

If you anticipate calling the same code from multiple different contexts, any of your scripts can call the reusable code you write in either global functions or object functions. As their name implies, global functions can be called from scripts in any object or from other global functions. Object functions can be called by any scripts in the same object, or even triggered by a button in the user interface.

After exploring the Groovy basic techniques needed to understand the examples, see "Examples of Each Context Where You Can Use Groovy" for a concrete example of each of these usages, and “Groovy Tips and Techniques” for additional tips and
techniques on getting the most out of Groovy in your application.
This chapter highlights some important aspects of Groovy to allow you to better understand the examples in the sections that follow.

This chapter contains the following:

- "Commenting Your Scripts"
- "Defining Variables"
- "Referencing the Value of a Field in the Current Object"
- "Working with Numbers, Dates, and Strings"
- "Using Substitution Expressions in Strings"
- "Using Conditional Expressions"
- "Using the Switch Statement"
- "Returning a Boolean Result"
- "Assigning a Value to a Field in the Current Object"
- "Writing Null-Aware Expressions"
- "Understanding the Difference Between Null and Empty String"
- "Understanding Secondary Fields Related to a Lookup"
- "Using Groovy's Safe Navigation Operator"
- "Assigning a Value to a Field in a Related Object"
- "Printing and Viewing Diagnostic Messages"
- "Working with Lists"
- "Working with Maps"
- "Working with Ranges"

**Commenting Your Scripts**

It is important that you document your scripts so that you and your colleagues who might view the code months from now will remember what the logic is doing. You can use either a double-slash combination // which makes the rest of the current line a comment, or you can use the open-comment and close-comment combination of /* followed later by */. The latter style can span multiple lines.

Here is an example of both styles in action:

```groovy
// Loop over the names in the list
```
for (name in listOfNames) {
    /*
    * Update the location for the current name.
    * If the name passed in does not exist, will result in a no-op
    */
    adf.util.updateLocationFor(name, // name of contact
        'Default', /* location style */
    )
}

When using multi-line comments, it is illegal for a nested /* ... */ comment to appear inside of another one. So, for example, the following is not allowed:

// Nested, multi-line comment below is not legal
def interest = 0
/*
18-MAY-2001 (smuench) Temporarily commented out calculation!
/

    * Interest Accrual Calculation Here
    */
    interest = complexInterestCalculation()
*/

Instead, you can comment out an existing multi-line block like this:

// Nested, multi-line comment below is legal
def interest = 0
//
// 18-MAY-2001 (smuench) Temporarily commented out calculation!
//
// /*
// * Interest Accrual Calculation Here
// */
// interest = complexInterestCalculation()
//

Or, alternatively had your initial code used the // style of comments, the following is also legal:

// Nested, multi-line comment below is not legal
def interest = 0
/*
18-MAY-2001 (smuench) Temporarily commented out calculation!

//
// Interest Accrual Calculation Here
//
// interest = complexInterestCalculation()
*/

The most common style-guide for comments would suggest to use multi-line comments at the beginning of the script, and single-line comments on subsequent lines. This allows you to most easily comment out code for debugging purposes. Thus, you typical script would look like this:

/*
* Object validation rule for BankAccount
*
* Ensures that account is not overdrawn
*/
def balance = CurrentBalance_c
// Use an object function to calculate uncleared charges
def unclearedCharges = unclearedChargesAmountForAccount()
// Perform some other complicated processing
performComplicatedProcessing()
// return true if the account is not overdrawn
return balance > unclearedCharges

**Defining Variables**

Groovy is a dynamic language, so variables in your scripts can be typed dynamically using the `def` keyword as follows:

// Assign the number 10 to a variable named "counter"
def counter = 10

// Assign the string "Hello" to a variable named "salutation"
def salutation = 'Hello'

// Assign the current date and time to a variable named "currentTime"
def currentTime = now()

Using the `def` keyword you can define a local variable of the right type to store *any* kind of value, not only the three examples above. Alternatively you can declare a specific type for variables to make your intention more explicit in the code. For example, the above could be written like this instead:

// Assign the number 10 to a variable of type Integer named "counter"
Integer counter = 10

// Assign the string "Hello" to a variable named "salutation"
String salutation = 'Hello'

// Assign the current date and time to a variable named "currentTime"
Date currentTime = now()

---

**Note:** You can generally choose to use the `def` keyword or to use a specific type for your variables according to your own preference, however when working with ADF objects you must to use the `def` keyword to define a variable to hold them. See the tip in the "Using the Related Collection Accessor Field to Work with Child Rows" section below for more details.

---

**Referencing the Value of a Field in the Current Object**

When writing scripts that execute in the context of the current business object, you can reference the value of any field in the current object by simply using its API name. This includes all of the following contexts:

- object validation rules
- field-level validation rules
- formula field expressions
- custom field conditionally updateable expressions
- custom field conditionally required expressions
- object triggers
- field triggers
object functions, and

- conditions for executing an object workflow

The API name of custom fields that you have added to a standard object will be suffixed with _c to distinguish them from standard field names. So, for example to write a script that references the value of a standard field named ContactPhoneNumber and a custom field named ContactTwitterName, you would use the following code:

```groovy
// Assign value of standard field "ContactPhoneNumber" to "phone" var def phone = ContactPhoneNumber

// Assign value of custom field "ContactTwitterName" to "twitterName" var def twitterName = ContactTwitterName_c

// Assemble text fragment by concatenating static text and variables def textFragment = 'We will try to call you at ' + phone +
' or send you a tweet at ' + twitterName
```

Defining a local variable to hold the value of a field is a good practice if you will be referencing its value more than once in your code. If you only need to use it once, you can directly reference a field’s name without defining a local variable for it, like this:

```groovy
def textFragment = 'We will try to call you at ' + ContactPhoneNumber +
' or send you a tweet at ' + ContactTwitterName_c
```

**Note:** When referencing a field value multiple times, you can generally choose to use or not to use a local variable according to your own preference, however when working with an ADF RowIterator object, you must to use the def keyword to define a variable to hold it. See the tip in the “Using the Related Collection Accessor Field to Work with Child Rows” section below for more details.

---

**Working with Numbers, Dates, and Strings**

Groovy makes it easy to work with numbers, dates and strings. You can use the normal + and - operators to do date, number, and string arithmetic like this:

```groovy
// Assign a date three days after the CreatedDate def targetDate = CreatedDate + 3

// Assign a date one week (seven days) before the value of the custom SubmittedDate field def earliestAcceptedDate = SubmittedDate_c - 7

// Increase an employee’s custom Salary field value by 100 dollars setAttribute('Salary_c', Salary_c + 100)

// Decrement a salesman’s commission field value by 100 dollars setAttribute('Commission_c', Commission_c - 100)

// Subtract (i.e. remove) any '@'-sign that might be present // in the contact’s twitter name def twitNameWithoutAtSign = ContactTwitterName_c - '@'

// Add the value of the twitter name to the message def message = 'Follow this user on Twitter at @' + twitNameWithoutAtSign
```
Using Substitution Expressions in Strings

Groovy supports using two kinds of string literals, normal strings and strings with substitution expressions. To define a normal string literal, use single quotes to surround the contents like this:

```groovy
// These are normal strings
def name = 'Steve'
def confirmation = '2 message(s) sent to ' + name
```

To define a string with substitution expressions, use double-quotes to surround the contents. The string value can contain any number of embedded expressions using the `$(expression)` syntax. For example, you could write:

```groovy
// The confirmation variable is a string with substitution expressions
def name = 'Steve'
def numMessages = 2
def confirmation = "${numMessages} message(s) sent to ${name}"
```

Executing the code above will end up assigning the value `2 messages(s) sent to Steve` to the variable named confirmation. It does no harm to use double-quotes all the time, however if your string literal contains no substitution expressions it is slightly more efficient to use the normal string with single-quotes.

**Tip:** As a rule of thumb, use normal (single-quoted) strings as your default kind of string, unless you require the substitution expressions in the string.

Using Conditional Expressions

When you need to perform the conditional logic, you use the familiar `if/else` construct. For example, in the text fragment example in the previous section, if the current object's `ContactTwitterName_c` returns null, then you won’t want to include the static text related to a twitter name. You can accomplish this conditional text inclusion using `if/else` like this:

```groovy
def textFragment = 'We will try to call you at ' + ContactPhoneNumber
if (ContactTwitterName_c != null) {
    textFragment += ', or send you a tweet at ' + ContactTwitterName_c
} else {
    textFragment += '. Give us your twitter name to get a tweet'
}
textFragment += '.
```

While sometimes the traditional `if/else` block is more easy to read, in other cases it can be quite verbose. Consider an example where you want to define an `emailToUse` variable whose value depends on whether the `EmailAddress` custom field ends with a `.gov` suffix. If the primary email ends with `.gov`, then you want to use the `AlternateEmailAddress` instead. Using the traditional `if/else` block your script would look like this:

```groovy
// Define emailToUse variable whose value is conditionally assigned. If the primary email address contains a '.gov' domain, then use the alternate email, otherwise use the primary email.
def emailToUse
if (endsWith(EmailAddress_c,'.gov')) {
    emailToUse = AlternateEmailAddress_c
}
```

Tip: As a rule of thumb, use normal (single-quoted) strings as your default kind of string, unless you require the substitution expressions in the string.
Using Groovy’s handy inline if/then/else operator, you can write the same code in a lot fewer lines:

```groovy
def emailToUse = endsWith(EmailAddress_c,'.gov') ? AlternateEmailAddress_c : EmailAddress_c
```

The inline if/then/else operator has the following general syntax:

```
BooleanExpression ? If_True_Use_This_Expression : If_False_Use_This_Expression
```

### Using the Switch Statement

If the expression on which your conditional logic depends may take on many different values, and for each different value you’d like a different block of code to execute, use the `switch` statement to simplify the task. As shown in the example below, the expression passed as the single argument to the `switch` statement is compared with the value in each `case` block. The code inside the first matching `case` block will execute. Notice the use of the `break` statement inside of each `case` block. Failure to include this `break` statement results in the execution of code from subsequent `case` blocks, which will typically lead to bugs in your application.

Notice, further, that in addition to using a specific value like 'A' or 'B' you can also use a range of values like 'C'..'P' or a list of values like ['Q','X','Z']. The `switch` expression is not restricted to being a string as is used in this example; it can be any object type.

```groovy
def logMsg
def maxDiscount = 0
// warehouse code is first letter of product SKU
// uppercase the letter before using it in switch
def warehouseCode = upperCase(left(SKU_c,1))
// Switch on warehouseCode to invoke appropriate
// object function to calculate max discount
switch (warehouseCode) {
  case 'A':
    maxDiscount = Warehouse_A_Discount()
    logMsg = "Used warehouse A calculation"
    break
  case 'B':
    maxDiscount = Warehouse_B_Discount()
    logMsg = "Used warehouse B calculation"
  case 'C'..'P':
    maxDiscount = Warehouse_C_through__P_Discount()
    logMsg = "Used warehouse C-through-P calculation"
    break
  case ['Q','X','Z']:
    maxDiscount = Warehouse_Q_X_Z_Discount()
    logMsg = "Used warehouse Q-X-Z calculation"
    break
  default:
    maxDiscount = Default_Discount()
    logMsg = "Used default max discount"
}
```

```groovy
adf.util.log(logMsg+' ['+maxDiscount+'])
```

// return expression that will be true when rule is valid
return Discount_c == null || Discount_c <= maxDiscount
Returning a Boolean Result

Several different contexts in ADF runtime expect your groovy script to return a boolean true or false result. These include:

- custom field's conditionally updateable expressions
- custom field's conditionally required expressions
- object-level validation rules
- field-level validation rules
- conditions for executing an object workflow

Groovy makes this easy. One approach is to use the groovy `true` and `false` keywords to indicate your return as in the following example:

```groovy
// Return true if value of the Commission custom field is greater than 1000
if (Commission_c > 1000) {
    return true
} else {
    return false
}
```

However, since the expression `Commission_c > 1000` being tested above in the `if` statement is itself a boolean-valued expression, you can write the above logic in a more concise way by simply returning the expression itself like this:

```groovy
return Commission_c > 1000
```

Furthermore, since Groovy will implicitly change the last statement in your code to be a return, you could even remove the `return` keyword and just say:

```groovy
Commission_c > 1000
```

This is especially convenient for simple comparisons that are the only statement in a validation rule, conditionally updateable expression, conditionally required expression, formula expression, or the condition to execute an object workflow.

Assigning a Value to a Field in the Current Object

The most logical way to assign the value of a field would be to using the groovy assignment operator like this:

```groovy
// Logical, but incorrect, way to assign the value "steve"
// to a custom field named ContactTwitterName in the current object
ContactTwitterName_c = 'steve'
```

However, since it is easy for you to inadvertently substitute the assignment operator `=` with the equals operator `==` ADF requires that your Groovy scripts explicitly use the `setAttribute()` function to avoid any accidental assignments from being silently performed without your knowledge. It has the following syntax:

```groovy
// Correct way to assign the value "steve" to a custom field
// named ContactTwitterName in the current object
setAttribute('ContactTwitterName_c', 'steve')
```

Notice above that the second argument `'steve'` passed to `setAttribute()` is a literal string, so it requires the single quotes around it. However, if the value to be assigned to the field is a literal number, or a groovy expression, or a value that is already stored in a variable `myNewVal` then the second argument to `setAttribute()` does not require any quotes like this:

```groovy
// Correct way to assign the value 3 to a custom field
// named ContactNumberOfChildren in the current object
setAttribute('ContactNumberOfChildren_c', 3)
```
setAttribute('ContactNumberOfChildren_c', 3)

// Correct way to assign the value 'one year from today's date'
// to a custom field ContactCallbackDate in the current object
setAttribute('ContactCallbackDate_c', today() + 365)

// Correct way to assign the value in the variable myNewVal
// to a custom field ContactNumberOfChildren in the current object
def myNewVal = /* compute new number of children value here */
setAttribute('ContactNumberOfChildren_c', myNewVal)

Notice that the name of the field whose value you want to assign is passed as a literal string (enclosed in quotes). Failure to enclose the value in quotes will result in either an error or unexpected functionality. For example, if you did the following:

// Incorrect way to assign the value 'steve' to a custom field
// named ContactTwitterName in the current object.
// The name of the field is not being passed as a literal string
setAttribute( ContactTwitterName_c, 'steve')

Using this incorrect syntax, at runtime, ADF will evaluate the value of the ContactTwitterName custom field and use its value as the name of the field to assign. So, as a silly example, if the ContactTwitterName field currently had the value "NoComment", then ADF would attempt to assign the value "steve" to the field in the current object named NoComment. If there were such a field, its value will be assigned without error. If no such field exists, then a runtime exception will be thrown.

---

**Tip:** See "Avoiding Validation Threshold Errors By Conditionally Assigning Values" for a tip about how to avoid your field assignments from causing an object to hit its validation threshold.

---

### Writing Null-Aware Expressions

When writing your scripts, be aware that field values can be null. You can use the nvl() null value function to easily define a value to use instead of null as part of any script expressions you use. Consider the following examples:

// Assign a date three days after the PostedDate
// Use the current date instead of the PostedDate if the
// PostedDate is null
def targetDate = nvl(PostedDate_c, now()) + 3

// Increase an employee's custom Salary field value by 10 percent
// Use zero if current Salary is null
setAttribute('Salary_c', nvl(Salary_c, 0) * 1.1)

---

### Understanding the Difference Between Null and Empty String

In Groovy, there is a subtle difference between a variable whose value is null and a variable whose value is the empty string. The value null represents the absence of any object, while the empty string is an object of type String with zero characters. If you try to compare the two, they are not the same. For example, any code inside the following conditional block will not execute because the value of varA(null) does not equals the value of varB (the empty string).

```groovy
def varA = null
def varB = "" /* The empty string */
if (varA == varB) {
    // Do something here when varA equals varB
```
Another common gotcha related to this subtle difference is that trying to compare a variable to the empty string does not test whether it is null. For example, the code inside the following conditional block will execute (and cause a NullPointerException at runtime) because the null value of varA is not equal to the empty string:

```groovy
def varA = null
if (varA != '') {
    // set varB to the first character in varA
    def varB = varA.charAt(0)
}
```

To test whether a string variable is neither null nor empty, you could explicitly write out both conditions like this:

```groovy
if (varA != null && varA != '') {
    // Do something when varA is neither null nor empty
}
```

However, Groovy provides an even simpler way. Since both null and the empty string evaluate to false when interpreted as a boolean, you can use the following instead:

```groovy
if (varA) {
    // Do something when varA has a non-null and non-empty value
}
```

If varA is null, the condition block is skipped. The same will occur if varA is equal to the empty string because either condition will evaluate to boolean false. This more compact syntax is the recommended approach.

---

### Understanding Secondary Fields Related to a Lookup

A lookup field represents a many-to-1 foreign key reference between one object and another object of the same or different type. For example, a TroubleTicket object might have a lookup field named Contact that represents a foreign key reference to the specific Contact object that reported the trouble ticket. When defining a lookup field, you specify a primary display field name from the reference object. For example, while defining the Contact lookup field referencing the Contact object, you might specify the Contact Name field.

When you define a lookup field like Contact, you get one primary field and two secondary fields:

**The Lookup Field**
This primary field is named Contact_c and it holds the value of the primary display field related to the referenced object, for example the name of the related contact.

**The Foreign Key Field**
This secondary field is named Contact_Id_c and it holds the value of the primary key of the reference contact.

**The Related Object Accessor Field**
This secondary field is named Contact_Obj_c and it allows you to programmatically access the related contact object in script code.

To access other fields on the related object, you can use the related object accessor field like this:

```groovy
// Assume script runs in context of TroubleTicket object
```

---
def contactEmail = Contact_Obj_c?.EmailAddress_c

If you know the primary key of the new contact:

To change which contact the TroubleTicket is related to, you can set a new contact by using one of the following techniques:

**If you know the primary key value of the new contact...**

```groovy
// Assume script runs in context of TroubleTicket object
def newId = /* ... Get the Id of the New Contact Here */
setAttribute('Contact_Id_c', newId)
```

**If you know the value of the contact's primary display field...**

```groovy
// Assume script runs in context of TroubleTicket object
setAttribute('Contact_c', 'James Smith')
```

### Using Groovy's Safe Navigation Operator

If you are using "dot" notation to navigate to reference the value of a related object, you should use Groovy's safe-navigation operator `?` instead of just using the `.` operator. This will avoid a `NullPointerException` at runtime if the left-hand-side of the operator happens to evaluate to null. For example, consider a TroubleTicket object with a lookup field named `AssignedTo` representing the staff member assigned to work on the trouble ticket. Since the `AssignedTo` field may be `null` before the ticket gets assigned, any code referencing fields from the related object should use the safe-navigation operator as shown here:

```groovy
// access related lookup object and access its record name
// Using the ?. operator, if related object is null,
// the expression evaluates to null instead of throwing
// NullPointerException
def assignedToName = AssignedTo_Obj_c?.RecordName
```

#### Tip:
For more information on why the code here accesses `AssignedTo_Obj_c` instead of a field named `AssignedTo_c`, see "Understanding Secondary Fields Related to a Lookup"

### Assigning a Value to a Field in a Related Object

When using the `setAttribute()` method — see "Assigning a Value to a Field in the Current Object" — the first argument must be a string that represents the name of a single field in object on which you invoke it. For example, the following is not a correct way to use the `setAttribute()` function to set the `Status` field of the parent TroubleTicket object for an activity because `TroubleTicket_c?.Status_c` is not the name of a field on the current activity object:

```groovy
// Assume script runs in context of Activity object
// which is a child object of parent TroubleTicket
setAttribute('TroubleTicket_c?.Status_c', 'Open')
```

Instead, you must first assign the parent object to a variable, then invoke the `setAttribute()` method on this parent object as shown here:

```groovy
// Assume script runs in context of Activity object
// which is a child object of parent TroubleTicket
```
// First access the parent object
def parentTicket = TroubleTicket_c
// Then call the setAttribute on that parent object
parentTicket?.setAttribute('Status_c', 'Open')

Tip: For more information on accessing related objects see "Using the Related Object Accessor Field to Work with a Parent Object" and "Using the Related Object Accessor Field to Work with a Referenced Object".

Printing and Viewing Diagnostic Messages

To write messages to the diagnostic log, use the `print` or `println` function. The former writes its value without any newline character, while the latter writes it value along with a newline. For example:

```
// Write a diagnostic message to the log. Notice how
// convenient string substitution expressions are
println("Status = \{Status_c\}")
```

Use the Diagnostic Dashboard shown in to view the diagnostic messages your scripts have written to the log to assist with debugging. As shown in Figure 2–1, after accessing the diagnostic dashboard, click the (Get Latest Log Messages) button to retrieve the latest Groovy script messages from the log file. To sort in reverse chronological order — to see the most recent messages first — click the down-pointing arrow in the Timestamp column header.

Figure 2–1 Viewing Messages Using the Diagnostics Dashboard

In this release, the diagnostic messages in the log are not identified by context, so it can be helpful to include information in the printed diagnostic messages to identify what code was executing when the diagnostic message was written. For example:

```
// Write a diagnostic message to the log, including info about the context
println("[In: BeforeInsert] Status = \{Status_c\}")
```
Working with Lists

A list is an ordered collection of objects. You can create a list of objects using Groovy’s square-bracket notation and a comma separating each list element like this:

```groovy
// Define a list of numbers
def list = [101, 334, 1208, 20]
```

Of course, the list can be of strings as well:

```groovy
// Define a list of strings
def names = ['Steve','Paul','Jane','Josie']
```

If needed, the list can contain objects of any type, including a heterogeneous set of object types, for example a mix of strings and numbers.

To refer to a specific element in the list, use the square brackets with an integer argument like this.

```groovy
// Store the third name in the list in a variable
def thirdName = names[2] // zero based index!
```

Remember that the list is zero-based so `list[0]` is the first element of the list and `list[5]` is the sixth element. Of course you can also pass a variable as the value of the operand like this:

```groovy
for (j in 2..3) {
    def curName = names[j]
    // do something with curName value here
}
```

To update a specific list item’s value, you can use the combination of the subscript and the assignment operator:

```groovy
names[2] = 'John'
```

To add an entry to the end of the list, use the `add()` method:

```groovy
names.add('Ringo')
```

A list can contain duplicates, so if you write code like the following, then the string Ringo will be added twice to the list:

```groovy
// This will add 'Ringo' twice to the list!
names.add('Ringo')
names.add('Ringo')
```

To test if an entry already exists in the list, use the `contains()` function. This way, you can ensure that you don’t add the same item twice if duplicates are not desirable for your purposes:

```groovy
// The exclamation point is the 'not' operator, so this
// first checks if the 'names' list does NOT contain 'Ringo' before
// adding it to the list
if (!names.contains('Ringo')) {
    names.add('Ringo')
}
```

To remove an entry from the list, use the `remove()` method.

```groovy
names.remove('Ringo')
```

Note that this only removes the first occurrence of the item in the list, returning a boolean result indicating true if the desired item was found and removed. Therefore, if your list allows duplicates and you need to remove them all, you’ll need to write a loop to call `remove()` until it returns false.

You can iterate over the entries in a list using the `for...in` loop like this:
// Process each name in the list, returning
// false if any restricted name is encountered
for (name in names) {
    // call a custom global function for each name processed
    if (adf.util.isNameRestricted(name)) {
        return false
    }
}
return true

You can define an empty list using the square-bracket notation with nothing inside like this:
def foundElements = [] // empty list!

### Working with Maps

A map is an unordered collection of name/value pairs. The name in each name/value pair is called the map's **key** for that entry since it is the key to looking up the value in the map later. You can create a map using Groovy's square-bracket notation, using a colon to separate each key and value, and a comma between each key/value pair like this:

```groovy
// Define a map of name/value pairs that associate
// a status value (e.g. "Open", "Closed", "Pending") with a
// maximum number of days
def maxDaysByStatus = [Open:30, Closed:90, Pending:45]
```

Notice that by default, the map key is assumed to be a string so you don't need to include the key values in quotes. However, if any key value contains spaces you will need to use quotes around it like this:

```groovy
def maxDaysByStatus = [Open:30, Closed:90, Pending:45, 'On Backorder':10]
```

If you want to use another type as the map key, you need to surround the key with parentheses. Consider the following example without the parentheses:

```groovy
def x = 1
def y = 2
def xvalue = 'One'
def yvalue = 'Two'
// this creates a map with entries ('x'->'One') and ('y'->'Two')
def m = [x:xvalue,y:yvalue]
```

The above example creates a map with key values of the strings `x` and `y`, rather than using the value of the variable `x` and the value of the variable `y` as map keys. To obtain this effect, surround the key expressions with parentheses like this:

```groovy
def x = 1
def y = 2
def xvalue = 'One'
def yvalue = 'Two'
// this creates a map with entries (1->'One') and (2->'Two')
def m = [(x):xvalue,(y):yvalue]
```

This creates a map with key values of the numbers 1 and 2.

To reference the value of a map entry, use dot notation like this, using the may key value as if it were a field name on the map object:

```groovy
def closedDayLimit = maxDaysByStatus.Closed
```
If the key value contains a literal dot character or contains spaces or special characters, you can also use the square-bracket notation, passing the key value as the operand:

```groovy
def onBackorderDayLimit = maxDaysByStatus['On Backorder']
```

This square bracket notation is also handy if the key value is coming from the value of a variable instead of a literal string, for example:

```groovy
// Loop over a list of statuses to process
for (curStatus in ['Open','On Backorder']) {
    def limitForCurStatus = maxDaysByStatus[curStatus]
    // do something here with the current status' limit
}
```

To add a new key/value pair to the map, use the `put()` method:

```groovy
// Add an additional status to the map
maxDaysByStatus.put('Ringo')
```

A map cannot contain duplicate key entries, so if you use `put()` to put the value of an existing element, the existing value for that key is overwritten. You can use the `containsKey()` function to test whether or not a particular map entry already exists with a given key value, or you can use the `containsValue()` function to test if any map entry exists that has a given value — there might be zero, one, or multiple entries!

```groovy
// Test whether a map key matching the value of the
// curKey variable exists or not
if (maxDaysByStatus.containsKey(curKey)) {
    def dayLimit = maxDaysByStatus[curKey]
    // do something with dayLimit here
} else {
    println("Unexpected error: key \\
    \\
    Unexpected error: key \${curKey} not found in maxDaysByStatusMap!")
}
```

To remove an entry from the map, use the `remove()` method. It returns the value that was previously associated with the key passed in, otherwise it returns null if it did not find the given key value in the map to remove.

```groovy
maxDaysByStatus.remove('On Backorder')
```

You can define an empty map using the square-bracket notation with only a colon inside like this:

```groovy
def foundItemCounts = [:] // empty map!
```

---

**Working with Ranges**

Using ranges, you can conveniently creates lists of sequential values. If you need to work with a list of integers from 1 to 100, rather than creating a list with 100 literal numbers in it, you can use the `..` operator to create a range like this:

```groovy
def indexes = 1..100
```

The range is particularly useful in performing iterations over a list of items in combination with a `for` loop like this:

```groovy
def indexes = 1..100
for (j in indexes) {
    // do something with j here
}
```
Of course, you need not assign the range to a variable to use it in a loop, you can use it inline like this:

```groovy
for (j in 1..100) {
    // do something with j here
}
```
This chapter provides a simple example of using Groovy in all of the different supported contexts in your application.

This chapter contains the following:

- "Providing an Expression to Calculate a Custom Formula Field's Value"
- "Providing an Expression to Calculate a Custom Field's Default Value"
- "Providing an Expression to Make a Custom Field Conditionally Updateable"
- "Providing an Expression to Make a Custom Field Conditionally Required"
- "Defining a Field-Level Validation Rule"
- "Defining an Object-Level Validation Rule"
- "Defining Utility Code in a Global Function"
- "Context Information Available in a Global Function"
- "Defining Reusable Behavior with an Object Function"
- "Defining an Object-Level Trigger to Complement Default Processing"
- "Defining a Field-Level Trigger to React to Value Changes"

Providing an Expression to Calculate a Custom Formula Field's Value

When you need a calculated field or a transient value-holder field with an optional initial, calculated value, use a formula field.

Read-Only Calculated Fields

A formula field defaults to being a read-only, calculated value. It displays the value resulting from the runtime evaluation of the calculation expression you supply. By using the Depends On multi-select list in the field create or edit page, you can configure the names of fields on which your expression depends. By doing this, its calculated value will update dynamically when any of those Depends On fields' value changes. The expected return type of the formula field's expression must be compatible with the formula field type you specified (Number, Date, or Text).

For example, consider a custom TroubleTicket object. If you add a formula field named DaysOpen, you can provide its calculated value with the expression:

\[(\text{today}()) - \text{CreationDate}) \text{ as Integer} /* \text{truncate to whole number of days} */\]
Transient Value Holder Fields with Optional Calculated Initial Value

If you want to allow the end user to override the calculated value, then mark your formula to be updateable. An updateable formula field is a "transient value holder" whose expression provides the value of the field until the user overrides it. If the user overrides the value, the object remembers this user-entered value for the duration of the current transaction so that your validation rules and triggers can reference it. If you have configured one or more Depends On fields for your updateable formula field, then note that the value of the formula will revert back to the calculated value should any of the dependent fields' value change. If you want a transient field whose initial value is null until the user fills it in, simply provide no formula expression for your updateable formula field to achieve this.

Providing an Expression to Calculate a Custom Field's Default Value

When a new row is created for an object, the value of a custom field defaults to null unless you configure a default value for it. You can supply a literal default value of appropriate type or supply an expression to calculate the default value for new rows. The default value expression is evaluated at the time the new row is created. The expected return type of your field's default value expression must be compatible with the field's type (Number, Date, Text, etc.)

For example, consider a custom CallbackDate field in a TroubleTicket object. If you want the callback back for a new trouble ticket to default to 3 days after it was created, then you can provide a default expression of:

\[ \text{CreationDate} + 3 \]

Providing an Expression to Make a Custom Field Conditionally Updateable

A custom field can be updateable or read-only. By default, any non-formula field is updateable. Alternatively, you can configure a conditionally updateable expression. If you do this, it is evaluated each time a page displaying the field is rendered or refreshed. The expected return type the expression is boolean. If you define one for a field, you must also configure the Depends On list to indicate the names of any fields on which your conditionally updateable expression depends. By doing this, your conditionally updateable field will interactively enable or disable as appropriate when the user changes the values of fields on which the conditionally updateable expression depends.

For example, consider a custom TroubleTicket object with Status and Justification fields. Assume you want to prevent a user from editing the justification of a closed trouble ticket. To achieve this, configure the conditionally updateable expression for the Justification field as follows:

\[ \text{Status}_c \neq \text{Closed} \]

After configuring this expression, you must then indicate that the Justification field depends on the Status field as described in "Understanding When to Configure Field Dependencies". This ensures that if a trouble ticket is closed during the current transaction, or if a closed trouble ticket is reopened, that the Justification field becomes enable or disabled as appropriate.
Providing an Expression to Make a Custom Field Conditionally Required

A custom field can be optional or required. By default it is optional. Alternatively, you can configure a conditionally required expression. If you do this, it is evaluated each time a page displaying the field is rendered or refreshed, as well as when the object is validated. The expected return type of the expression boolean. If you define one for a field, you must also configure the Depends On list to indicate the names of any fields on which your conditionally required expression depends. By doing this, your conditionally required field will interactively show or hide the visual indicator of the field’s being required as appropriate when the user changes the values of fields on which the conditionally required expression depends.

For example, consider a custom TroubleTicket object with Priority and Justification fields. Assume that priority is an integer from 1 to 5 with priority 1 being the most critical kind of problem to resolve. To enforce that a justification is required for trouble tickets whose priority is 1 or 2, configure the conditionally required expression for the Justification field as follows:

\[ \text{Priority} \leq 2 \]

After configuring this expression, you must then indicate that the Justification field depends on the Priority field as described in "Understanding When to Configure Field Dependencies". This ensures that if a trouble ticket is created with priority 2, or an existing trouble ticket is updated to increase the priority from 3 to 2, that the Justification field becomes mandatory.

Defining a Field-Level Validation Rule

A field-level validation rule is a constraint you can define on any standard or custom field. It is evaluated whenever the corresponding field’s value is set. When the rule executes, the field’s value has not been assigned yet and your rule acts as a gatekeeper to its successful assignment. The expression (or longer script) you write must return a boolean value that indicates whether the value is valid. If the rule returns true, then the field assignment will succeed so long as all other field-level rules on the same field also return true. If the rule returns false, then this prevents the field assignment from occurring, the invalid field is visually highlighted in the UI, and the configured error message is displayed to the end user. Since the assignment fails in this situation, the invalid field retains its current value (possibly null, if the value was null before), however the UI component in the web page allows the user to see and correct their invalid entry to try again. Your script can use the newValue keyword to reference the new value that will be assigned if validation passes. To reference the existing field value, use the oldValue keyword. A field-level rule is appropriate when the rule to enforce only depends on the new value being set. As shown in Figure 3–1, you can use the Keywords tab of the Expression Palette to insert the newValue and oldValue keywords.
Defining an Object-Level Validation Rule

An object-level validation rule is a constraint you can define on any standard or custom object. It is evaluated whenever the framework attempts to validate the object. This can occur upon submitting changes in a web form, when navigating from one row to another, as well as when changes to an object are saved. Use object-level rules to enforce conditions that depend on two or more fields in the object. This ensures that regardless of the order in which the user assigns the values, the rule will be consistently enforced. The expression (or longer script) you write must return a boolean value that indicates whether the object is valid. If the rule returns true, then the object validation will succeed so long as all other object-level rules on the same object return true. If the rule returns false, then this prevents the object from being saved, and the configured error message is displayed to the end user.

For example, consider a TroubleTicket object with a Priority field. To validate that the number entered is between 1 and 5, your field-level validation rule would look like this:

**Field Name**
*Priority*

**Rule Name**
Validate_Priority_Range

**Rule Body**
newValue == null || (1..5).contains(newValue as Integer)

**Error Message**
The priority must be in the range from 1 to 5

---

**Tip:** If a validation rule for field A depends on the values of one or more other fields (e.g. Y and Z), then create an object-level rule and programmatically signal which field or fields should be highlighted as invalid to the user as explained in "Setting Invalid Fields for the UI in an Object-Level Validation Rule".

---

For example, consider a custom TroubleTicket object with a Priority field. To validate that a trouble ticket of priority 1 or 2 cannot be saved without being assigned to a staff member, your object-level rule would look like this:

Defining an Object-Level Validation Rule

An object-level validation rule is a constraint you can define on any standard or custom object. It is evaluated whenever the framework attempts to validate the object. This can occur upon submitting changes in a web form, when navigating from one row to another, as well as when changes to an object are saved. Use object-level rules to enforce conditions that depend on two or more fields in the object. This ensures that regardless of the order in which the user assigns the values, the rule will be consistently enforced. The expression (or longer script) you write must return a boolean value that indicates whether the object is valid. If the rule returns true, then the object validation will succeed so long as all other object-level rules on the same object return true. If the rule returns false, then this prevents the object from being saved, and the configured error message is displayed to the end user.

For example, consider a TroubleTicket object with Priority and AssignedTo fields, where the latter is a dynamic choice list field referencing Contact objects whose Type field is a 'Staff Member'. To validate that a trouble ticket of priority 1 or 2 cannot be saved without being assigned to a staff member, your object-level rule would look like this:
Rule Name
Validate_High_Priority_Ticket_Has_Owner

Rule Body
// Rule depends on two fields, so must be written as object-level rule
if (Priority_c <= 2 && AssignedTo_Id_c == null) {
    // Signal to highlight the AssignedTo field on the UI as being in error
    adf.error.addAttribute('AssignedTo_c')
    return false
} else {
    return true
}

Error Message
A trouble ticket of priority 1 or 2 must have a staff member assigned to it

Defining Utility Code in a Global Function
Global functions are useful for code that multiple objects want to share. To call a global function, preface the function name with the adf.util. prefix. When defining a function, you specify a return value and can optionally specify one or more typed parameters that the caller will be required to pass in when invoked. The most common types for function return values and parameters are the following:

String
a text value

Boolean
a logical true or false value

Long
an integer value in the range of ±2
63
-1

BigInteger
an integer of arbitrary precision

Double
a floating-point decimal value in the range of ±1.79769313486231570 x 10308

BigDecimal
a decimal number of arbitrary precision

Date
a date value with optional time component

List
an ordered collection of objects

Map
an unordered collection of name/value pairs

Object
any object
In addition, a function can define a `void` return type which indicates that it returns no value.

**Tip:** A global function has no current object context. To write global functions that work on a particular object, refer to "Passing the Current Object to a Global Function".

For example, you could create the following two global functions to define standard helper routines to log the start of a block of groovy script and to log a diagnostic message. Examples later in this document will make use of them.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Return Type</th>
<th>Parameters</th>
<th>Function Definition</th>
</tr>
</thead>
</table>
| logStart      | void        | None       | // Log the name of the script
println("[In: ${scriptName}]") |
| log           | void        | None       | // Log the message, could add other info println(message) |

### Context Information Available in a Global Function

In global functions, your Groovy script can reference only the following elements in the `adf` namespace:

- `adf.context`, as described in "Referencing Information About the Current User".
- `adf.util`, to invoke other global functions as described in the section above.
- `adf.webServices`, to invoke registered web services as described in "Calling Web Service Methods in Groovy".

In particular, other `adf`-prefixed elements like `adf.source` to reference the current object cannot be directly referenced in a global function’s script. You need to reference them in the calling script and pass them in as a parameter to the global function. See "Passing the Current Object to a Global Function", for an example.

### Defining Reusable Behavior with an Object Function

Object functions are useful for code that encapsulates business logic specific to a given object. You can call object functions by name from any other script code related to the same object. In addition, you can invoke them using a button or link in the user interface. The supported return types and optional parameter types are the same as for global functions (described above).

For example, you might define the following `updateOpenTroubleTicketCount()` object function on a `Contact` custom object. It begins by calling the `logStart()` global function above to log a diagnostic message in a standard format to signal the beginning of a block of custom Groovy script. It calls the `newView()` built-in function (described in "Accessing the View Object for Programmatic Access to Business Objects") to access the view object for programmatic access of trouble tickets, then calls another global function `applyFilter()` (described in "Simplifying Most Common View Criteria Creation with a Global Function") to apply a filter to find trouble tickets related to the current contact’s id and having either 'Working' or 'Waiting' as their current status. Finally, it calls `getEstimatedRowCount()` to retrieve the count of trouble tickets that qualify for the filter criteria.
Defining an Object-Level Trigger to Complement Default Processing

Triggers are scripts that you can write to complement the default processing logic for a standard or custom object. You can define triggers both at the object-level and the field-level. The object-level triggers that are available are described below. See "Defining a Field-Level Trigger to React to Value Changes" for the available field-level triggers.

### After Create
Fires when a new instance of an object is created. Use to assign programmatic default values to one or more fields in the object.

### Before Modify
Fires when the end-user first modifies a persistent field in an existing, queried row

### Before Invalidate
Fires on an existing object when its first persistent field is modified. Fires on a valid parent object when a child row is created, removed, or modified.

### Before Remove
Fires when an attempt is made to delete an object. Returning false stops the row from being deleted and displays the optional trigger error message.

### Before Insert in Database
Fires before a new object is inserted into the database.

---

**Function Name** | **Return Type** | **Parameters** | **Function Definition**
---|---|---|---
updateOpenTroubleTicketCount | void | None | afd.util.logStart('updateOpenTroubleTicketCount')

```java
// Access the view object for TroubleTicket programmatic access
def tickets = newView('TroubleTicket_c')
// Use a global function to apply a query filter to TroubleTicket
// view object where Contact_Id = Id of current Contact and
// Status is either 'Working' or 'Waiting'
adf.util.applyFilter(tickets,[Status__c:['Working','Waiting'],
Contact_Id__:Id])
// Update OpenTroubleTickets field value
setAttribute('OpenTroubleTickets_c',tickets.getEstimatedRowCount())
```
**After Insert in Database**
Fires after a new object is inserted into the database. Best to use **Before Insert** for field derivations.

**Before Update in Database**
Fires before an existing object is modified in the database

**After Update in Database**
Fires after an existing object is modified in the database. Best to use **Before Update** for field derivations.

**Before Delete in Database**
Fires before an existing object is deleted from the database

**After Delete in Database**
Fires after an existing object is deleted from the database

**After Changes Posted to Database**
Fires after all changes have been posted to the database, but before they are permanently committed. Can be used to make additional changes that will be saved as part of the current transaction.

**Before Commit in Database**
Fires before the change pending for the current object (insert, update, delete) is made permanent in the current transaction. Any changes made in this trigger will **not** be part of the current transaction. Use "After Changed Posted to Database" trigger if your trigger needs to make changes.

**After Commit in Database**
Fires after the change pending for the current object (insert, update, delete) is made permanent in the current transaction.

**Before Rollback in Database**
Fires before the change pending for the current object (insert, update, delete) is rolled back

**After Rollback in Database**
Fires after the change pending for the current object (insert, update, delete) is rolled back

For example, consider a **Contact** object with a **OpenTroubleTickets** field that needs to be updated any time a trouble ticket is created or modified. You can create the following two triggers on the **TroubleTicket** object which both invoke the **updateOpenTroubleTicketCount()** object function described above.

**Trigger Object**
TroubleTicket

**Trigger**
*After Insert In Database*

**Trigger Name**
After_Insert_Set_Open_Trouble_Tickets

**Trigger Definition**
```groovy
adf.util.logStart('After_Insert_Set_Open_Trouble_Tickets')
// Get the related contact for this trouble ticket
def relatedContact = Contact_Obj_c
```
// Update its OpenTroubleTickets field value
relatedContact?.updateOpenTroubleTicketCount()

**Trigger Object**
TroubleTicket

**Trigger**
_After Update In Database_

**Trigger Name**
_After_Update_Set_Open_Trouble_Tickets_

**Trigger Definition**
// Get the related contact for this trouble ticket
def relatedContact = Contact_Obj_c
// Update its OpenTroubleTickets field value
relatedContact?.updateOpenTroubleTicketCount()

---

**Defining a Field-Level Trigger to React to Value Changes**

Field-level triggers are scripts that you can write to complement the default processing logic for a standard or custom field. The following field-level trigger is available:

**After Field Changed**
Fires when the value of the related field has changed (implying that it has passed any field-level validation rules that might be present).

Use the _After Field Changed_ trigger to calculate other derived field values when another field changes value. Do not use a field-level validation rule to achieve this purpose because while your field-level validation rule may succeed, _other_ field-level validation rules may fail and stop the field's value from actually being changed. Since generally you only want your field-change derivation logic to run when the field's value actually changes, the _After Field Changed_ trigger guarantees that you get this desired behavior.

See "Deriving a Non-Formula Field Using an After Field Changed Trigger" for tips on using this trigger.
This chapter provides a compendium of tips and techniques for getting the most out of Groovy in your application.

This chapter contains the following:

- "Simplifying Code Authoring with the Expression Palette"
- "Using the Related Object Accessor Field to Work with a Parent Object"
- "Using the Related Object Accessor Field to Work with a Referenced Object"
- "Using the Related Collection Accessor Field to Work with Child Rows"
- "Accessing Current Date and Time from the Application Server"
- "Accessing Current Date and Time from the Database"
- "Understanding ADF’s Additional Built-in Groovy Functions"
- "Understanding Groovy’s Null-Safe Comparison Operators"
- "Testing Whether an Field’s Value Is Changed"
- "Avoiding Validation Threshold Errors By Conditionally Assigning Values"
- "Prefer "Before" Save-time Triggers to "After" Ones for Best Performance"
- "Avoiding Posting Threshold Errors By Conditionally Assigning Values"
- "Functional Restrictions in Trigger Scripts"
- "Passing the Current Object to a Global Function"
- "Referencing Original Values of Changed Fields"
- "Raising a Warning From a Validation Rule Instead of an Error"
- "Throwing a Custom Validation Exception"
- "Returning Locale-Sensitive Custom Strings"
- "Raising a Trigger’s Optional Declaratively-Configured Error Message"
- "Accessing the View Object for Programmatic Access to Business Objects"
- "Defining the Sort Order for Query Results"
- "Finding an Object by Id"
- "Finding Objects Using a View Criteria"
- "Creating a New Object"
- "Updating an Existing Object"
Simplifying Code Authoring with the Expression Palette

Before diving into the specifics of each example, keep in mind that the CRM Application Composer always makes an Expression Palette available to assist you. It helps you insert the names of built-in functions, object fields, or function names.

The Expression Editor Toolbar

Typically the expression editor opens with the expression palette collapsed to give you more space to your script. In this minimized, hidden state, as shown in Figure 4–1, you have a number of useful toolbar buttons that can assist you to:

- insert one of the common comparison operators like equals (==), not equals (!=), less than (<), greater than (>), etc.
- erase the code editor to start over with a clean slate
- validate the correct syntax of your script, or
- detach the code editor into a floating window to give you even more space to see your script.

This section provides a compendium of tips and techniques for getting the most out of Groovy in your application.
Showing the Expression Palette

To expand the amount of assistance available to you, you can show the expression palette. As shown in Figure 4–2, click the right-pointing arrow to the left of the Show Palette label to expand the palette, revealing its three tab display.

Inserting an Expression for Fields of Related Objects

As shown in Figure 4–3, the Fields tab of the palette shows you the current object at the root of the tree, and allows you to explore the related objects and collections of related objects by expanding the tree nodes. If the selected node has the single-row icon [inlinegraphic]graphics/singlerow_icon.png[/inlinegraphic] then it represents a single, related object. If the selected node has the multiple-row icon [inlinegraphic]graphics/multirow_icon.png[/inlinegraphic] then it represents a collection of related objects. For example, in the figure the tree shows Activities with the multiple-row icon, so this represents the collection of Activity objects related to the current trouble ticket. The Contact node has a single-row icon so it represents a single Contact object that is the customer who reported the trouble ticket. The Assigned To node represents the staff member to whom the trouble ticket has been assigned, and lastly, the Manager node represents the manager of that staff member.

As shown in the numbered steps in the figure, to insert the name of a field, do the following:

1. Select the object you want to work with, for example the Manager of the staff member Assigned To this Trouble Ticket

2. In the fields table, select the field you want to insert, for example Contact Name

3. Click the (Insert) button to insert the field’s name.
If the field on the selected object is directly accessible from the current object through "dot" navigation, then the expression palette will insert the expression for you. For example, if you click (Insert) after selecting the Contact Name field of the Manager object related to the Assigned To staff member assigned to the current Trouble Ticket, then the expression inserted will be:

```
AssignedTo_c?.Manager_c?.RecordName
```

### Inserting Field Name of Object in Related Collection

In contrast, as shown in Figure 4–4, if you were to select the Activities collection and then select the Activity Status field from the object in that collection, when you click (Insert) only the API name ActivityStatus_c of the selected field will be inserted. The reason a complete expression cannot be inserted in this case is due to the fact that you cannot use "dot" notation to work with a specific row in a collection in order to access its field value. You need to use Groovy code to access the collection of rows and then work with one row at a time.

### Inserting a Related Object Accessor or Related Collection Accessor Field Name

As shown in Figure 4–5, to insert the name of an Related Object or Related Collection field, select the object and field in the expression palette, and click (Insert) as you do with any other type of field. For example, to insert the API name of the collection of Activity objects related to a TroubleTicket, do the following:

1. Select the Trouble Ticket node in the tree (representing the current object in this example)
2. Select the desired Related Collection or Related Object field in the table at the right, for example, the ActivityCollection_c field.
3. Click the *Insert* button.

Figure 4–5  *Selecting a Field from an Object in a Related Collection*

Once you have inserted the name of the related collection field, you can write a loop to process each row in the collection like this:

```groovy
def activities = ActivityCollection_c
while (activities.hasNext()) {
    def activity = activities.next()
    // if the activity status is 'Open'...
    if (activity.Status_c == 'Open') {
        // do something here to the current child activity
    }
}
```

When writing the code in the loop that processes *Activity* rows, you can use the Expression Palette again to insert field names of the object in that collection by selecting the *Activities* collection in the tree, choosing the desired field, then clicking *Insert*.

### Hiding the Expression Palette

When you’re done using the Expression Palette, as shown in Figure 4–6, click the down-pointing arrow to the left of the *Hide Palette* label to collapse the palette.

Figure 4–6  *Collapsing the Expression Palette to Hide It*

### Using the Related Object Accessor Field to Work with a Parent Object

When writing business logic in a child object like *Activity*, you can access its owning parent *TroubleTicket* object using the related object accessor field. If the parent object is named *TroubleTicket*, the related object accessor field in *Activity* will be named *TroubleTicket_c*. Example 4–1 shows how to reference the parent *TroubleTicket* object and access one of its fields.

Example 4–1  *Accessing a Field from Parent Object*

```groovy
// Assume code in context of Activity
if (TroubleTicket_c.Status_c == 'Open') {
    // Do something here because the owning parent trouble ticket's status is open
```
Notice that since the child object cannot exist without an owning parent object, the reference to the parent object will never be null, so here instead of the Groovy safe navigation operator (?.) we can just use the normal dot operator in the expression TroubleTicket_c.Status_c.

If you will be referencing multiple fields from the parent, or calling any of its object functions, then it is best practice to store the parent object in a local variable as shown in Example 4-2.

**Example 4-2  Storing Parent Object in Local Variable Before Multiple References**

```groovy
// Store the parent object in a local variable
def ticket = TroubleTicket_c
// Now reference one or more fields from the parent
if (ticket.Status_c == 'Working' && ticket.Priority_c >= 2) {
    // Do something here because the owning parent
    // trouble ticket is high priority and being worked on.
}
```

**Using the Related Object Accessor Field to Work with a Referenced Object**

When writing business logic for an object like TroubleTicket that has a lookup field like Contact or AssignedTo, you can access the object referenced by the lookup field using the respective lookup field's secondary related object accessor field. See "Understanding Secondary Fields Related to a Lookup" for more information on this. For the Contact and AssignedTo lookup fields, the secondary related object accessor fields are named Contact_Obj_c and AssignedTo_Obj_c, respectively. Example 4-3 shows how to reference the two lookup objects from script written in the context of TroubleTicket, and access one of each's fields.

**Example 4-3  Accessing a Field from Two Lookup Objects**

```groovy
// Assume code in context of TroubleTicket
if (endsWith(Contact_Obj_c?.EmailAddress_c,'.gov') &&
    startsWith(AssignedTo_Obj_c?.PhoneNumber_c,'202')) {
    // Do something here because contact's email address
    // is a government mail address and assigned-to staff member's
    // phone number is in the 202 area code for Washington DC.
}
```

If you will be referencing multiple fields from the referenced lookup object, or calling any of its object functions, then it is best practice to store the referenced lookup object in a local variable as shown in Example 4-4.

**Example 4-4  Storing Related Lookup Objects in Local Variable Before Multiple References**

```groovy
// Store the contact object and assignedTo object in a local variable
def customer = Contact_Obj_c
def supportRep = AssignedTo_Obj_c
// Now reference one or more fields from the parent
if (endsWith(customer?.EmailAddress_c,'.gov') ||
    endsWith(customer?.EmailAddress_c,'.com'))
    #
```
Using the Related Collection Accessor Field to Work with Child Rows

When an parent object like TroubleTicket has a child object Activity, the parent object will have a related collection accessor field whose name you decided when you created the child object. For example, if when creating the child Activity object for the TroubleTicket parent, you decided to name the related collection accessor field ActivityCollection, then you can write business logic in the context of the parent TroubleTicket object that works with the one or more Activity child rows. To do this, your code accesses the related collection accessor field by name like this:

```groovy
// Assume code in context of TroubleTicket
// define a variable to hold activities collection
def activities = ActivityCollection_c
// work with activities here...
```

**Tip:** Always store a child collection you want to work with in a local variable. Failure to do this will result in your code that does not behave as you expect.

The related collection accessor field returns a row iterator object, so you can use methods like those listed in Table 4–1 to work with the rows. The row iterator tracks the current row in the collection that your code is working with.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext()</td>
<td>Returns: true if the row iterator has more rows to iterate over, false if there are no rows in the iterator’s row set or if the iterator is already on or beyond the last row.</td>
</tr>
<tr>
<td>next()</td>
<td>Returns: the next row in the row iterator</td>
</tr>
<tr>
<td>reset()</td>
<td>Returns: void. Resets the row iterator to the &quot;slot&quot; before the first row.</td>
</tr>
<tr>
<td>first()</td>
<td>Returns: the first row in the row iterator, or null if the iterator's row set is empty</td>
</tr>
</tbody>
</table>

Putting the commonly used row iterator methods from Table 4–1 into practice, Example 4–5 shows the typical code you will use to work with the child row iterator. This example accesses the child row iterator using the related collection field's API name, and saves it in a local variable. Then, it resets the iterator so that it sits on the "slot" before the first row in the row iterator. Next, it uses a while loop in combination with the hasNext() method to iterate over each row in the row iterator.
Example 4–5  Processing Each Row in a Related Child Collection

// store the child row iterator in a local variable
def activities = ActivityCollection_c
// ensure iterator is on slot before first row
activities.reset()
// loop while there are more rows to process
while (activities.hasNext()) {
  // access the next row in the row iterator
  def curActivity = activities.next()
  // reference fields or object functions from the current row
  if (curActivity.Status_c == 'Open') {
    // do something here to the current child activity
  }
}
// to process the same row iterator again in this block of code,
// call activities.reset() method again to reset the
// iterator to the slot before the first row

To detect whether the child row iterator is empty or not, you can use the first() method. If it returns null then the row iterator's row set is empty. As shown in Example 4–6, if you call the first() method and there are rows in the row iterator's row set, this method sets the iterator to point at the first row. So, if your script uses the first() method, then plans to iterate over all the rows in the iterator again using the typical while (rowiterator.hasNext()) idiom, you need to call the reset() method on the row iterator to move the current row pointer back to the slot before the first row. Failure to do this could result in inadvertently not processing the first row in the row set.

Example 4–6  Testing Whether a Related Child Row Iterator is Empty

def activities = ActivityCollection_c
// If there are no child activities...
if (activities.first() == null) {
  // Do something here because there are no child activities
}
else {
  // There are some child activities, call reset() to set
  // iterator back to slot before first row
  activities.reset()
  while (activities.hasNext()) {
    def curActivity = activities.next();
    // Do something here with the current activity
  }
}

Accessing Current Date and Time from the Application Server

To reference the application server's current date in any groovy expression, use:

adf.currentDate

To reference the application server's current date including the current time, use the expression:

adf.currentDateTime
Accessing Current Date and Time from the Database

To reference the database's current date in any groovy expression, use:

```groovy
adf.currentDBDate
```

To reference the application server's current date including the current time, use the expression:

```groovy
adf.currentDBDateTime
```

**Note:** This function cannot currently be referenced inside code for a Global Function. An object function can reference it and pass the value, if necessary, into the global function.

Understanding ADF's Additional Built-in Groovy Functions

Oracle ADF adds a number of additional helper functions that you can use in your Groovy scripts. This section defines each one and provides a simple example of its use. As shown in Figure 4–7, you can use the Functions tab of the Expression Palette to insert any of the built-in functions and get a quick example of their use.

**Table 4–2 Built-in Date Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>today()</code></td>
<td>Returns: the current date, with no time</td>
</tr>
<tr>
<td><code>now()</code></td>
<td>The current date and time</td>
</tr>
</tbody>
</table>

`Return Type: Date`  
`Return Type: Timestamp`
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Return Type</th>
<th>Parameters</th>
<th>Example</th>
</tr>
</thead>
</table>
| date(year, month, day)     | Returns: a date, given the year, month, and day | Date        | year - a positive integer  
month - a positive integer between 1 and 12  
day - a positive integer between 1 and 31 | to return a date for February 8th, 1998, use date(1998, 2, 8)                                   |
| dateTime(year, month, day, hour, minute, second) | Returns: a timestamp, given the year, month, day, hour, minute, and second | Date        | year - a positive integer  
month - a positive integer between 1 and 12  
day - a positive integer between 1 and 31  
hour - a positive integer between 0 and 23  
minute - a positive integer between 0 and 59  
second - a positive integer between 0 and 59 | to return a timestamp for February 8th, 1998, at 23:42:01, use dateTime(1998, 2, 8, 23, 42, 1) |
<p>| year(date)                 | Returns: the year of a given date              | Integer     | date - date                                                                | if curDate represents April 19th, 1996, then year(curDate) returns 1996.                     |
| month(date)                | Returns: the month of a given date             | Integer     | date - a date                                                              | if curDate represents April 12th, 1962, then month(curDate) returns 4.                       |
| day(date)                  | Returns: the day for a given date              | Integer     | date - a date                                                              | if curDate represents July 15th, 1968, then day(curDate) returns 15.                         |</p>
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains(s1, s2)</td>
<td>Returns: true, if string s1 contains string s2, false otherwise</td>
</tr>
<tr>
<td></td>
<td>Return Type: boolean</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>s1 - a string to search in</td>
</tr>
<tr>
<td></td>
<td>s2 - a string to search for</td>
</tr>
<tr>
<td></td>
<td>Example: if twitterName holds the value @slow, then contains(twitterName, '@') returns true.</td>
</tr>
<tr>
<td>endsWith(s1, s2)</td>
<td>Returns: true, if string s1 ends with string s2, false otherwise</td>
</tr>
<tr>
<td></td>
<td>Return Type: boolean</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>s1 - a string to search in</td>
</tr>
<tr>
<td></td>
<td>s2 - a string to search for</td>
</tr>
<tr>
<td></td>
<td>For example, if twitterName holds the value @steve, then endsWith(twitterName, '@') returns false.</td>
</tr>
<tr>
<td>find(s1, s2)</td>
<td>Returns: the integer position of the first character in string s1 where string s2 is found, or zero (0) if the string is not found</td>
</tr>
<tr>
<td></td>
<td>Return Type: Integer</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>s1 - a string to search in</td>
</tr>
<tr>
<td></td>
<td>s2 - a string to search for</td>
</tr>
<tr>
<td></td>
<td>Example: if twitterName holds the value @steve, then find(twitterName, '@') returns 1 and find(twitterName, 'ev') returns 4.</td>
</tr>
<tr>
<td>left(s, len)</td>
<td>Returns: the first len characters of the string s</td>
</tr>
<tr>
<td></td>
<td>Return Type: String</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>s - a string</td>
</tr>
<tr>
<td></td>
<td>len - an integer number of characters to return</td>
</tr>
<tr>
<td></td>
<td>Example: if postcode holds the value 94549-5114, then left(postcode, 5) returns 94549.</td>
</tr>
<tr>
<td>length(s)</td>
<td>Returns: the length of string s</td>
</tr>
<tr>
<td></td>
<td>Return Type: Integer</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>s - a string</td>
</tr>
<tr>
<td></td>
<td>Example: if name holds the value Julian Croissant, then len(name) returns 16.</td>
</tr>
</tbody>
</table>
### Table 4–3  (Cont.) Built-in String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lowerCase(s)</code></td>
<td>Returns: the string <code>s</code> with any uppercase letters converted to lowercase</td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> String</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>■ <code>s</code> - a string</td>
</tr>
<tr>
<td></td>
<td><strong>Example:</strong> if <code>sku</code> holds the value 12345-10-WHT-XS, then <code>lowerCase(sku)</code> returns 12345-10-wht-xs.</td>
</tr>
<tr>
<td><code>right(s, len)</code></td>
<td>Returns: the last <code>len</code> characters of the string <code>s</code></td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> String</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>■ <code>s</code> - a string</td>
</tr>
<tr>
<td></td>
<td>■ <code>len</code> - an integer number of characters to return</td>
</tr>
<tr>
<td></td>
<td><strong>Example:</strong> if <code>sku</code> holds the value 12345-10-WHT-XS, then <code>right(sku, 2)</code> returns XS.</td>
</tr>
<tr>
<td><code>startsWith(s1, s2)</code></td>
<td>Returns: true, if string <code>s1</code> starts with <code>s2</code>, false otherwise</td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> boolean</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>■ <code>s1</code> - a string to search in</td>
</tr>
<tr>
<td></td>
<td>■ <code>s2</code> - a string to search for</td>
</tr>
<tr>
<td></td>
<td><strong>Example:</strong> if <code>twitterName</code> holds the value <code>@steve</code>, then <code>startsWith(twitterName, '@')</code> returns true.</td>
</tr>
<tr>
<td><code>substringBefore(s1, s2)</code></td>
<td>Returns: the substring of <code>s1</code> that precedes the first occurrence of <code>s2</code>, otherwise an empty string</td>
</tr>
<tr>
<td></td>
<td><strong>Return Type:</strong> String</td>
</tr>
<tr>
<td></td>
<td><strong>Parameters:</strong></td>
</tr>
<tr>
<td></td>
<td>■ <code>s1</code> - a string to search in</td>
</tr>
<tr>
<td></td>
<td>■ <code>s2</code> - a string to search for</td>
</tr>
<tr>
<td></td>
<td><strong>Examples:</strong> if <code>sku</code> holds the value 12345-10-WHT-XS, then <code>substringBefore(sku, '-')</code> returns the value 12345, <code>substringBefore(sku, '12345')</code> returns an empty string, and <code>substringBefore(sku, '16-BLK')</code> also returns an empty string.</td>
</tr>
</tbody>
</table>
Understanding ADF’s Additional Built-in Groovy Functions

Table 4–3  (Cont.) Built-in String Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>substringAfter(s1,s2)</td>
<td>Returns: the substring of s1 that follows the first occurrence of s2. Otherwise an empty string</td>
</tr>
<tr>
<td></td>
<td>Return Type: String</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>■ s1 - a string to search in</td>
</tr>
<tr>
<td></td>
<td>■ s2 - a string to search for</td>
</tr>
<tr>
<td></td>
<td>Example: if sku holds the value 12345-10-WHT-XS, then substringAfter(sku,'-') returns the value 10-WHT-XS, substringAfter(sku,'WHT-') returns the value XS, substringAfter(sku,'XS') returns an empty string, and substringAfter(sku,'BLK') also returns an empty string.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>upperCase(s)</td>
<td>Returns: the string s with any lowercase letters converted to uppercase</td>
</tr>
<tr>
<td></td>
<td>Return Type: String</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>■ s - a string</td>
</tr>
<tr>
<td></td>
<td>Example: if sku holds the value 12345-10-Wht-xs, then upperCase(sku) returns 12345-10-WHT-XS.</td>
</tr>
</tbody>
</table>

Table 4–4  Other Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>newView(objectAPIName)</td>
<td>Returns: a ViewObject reserved for programmatic use, or null if not available.</td>
</tr>
<tr>
<td></td>
<td>Return Type: ViewObject</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>■ objectAPIName - the object API name whose rows you want to find, create, update, or remove</td>
</tr>
<tr>
<td></td>
<td>Example: if TroubleTicket is a custom object, then newView('TroubleTicket_c') returns a new view object instance you can use to find, create, update, or delete TroubleTicket rows.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>key(list)</td>
<td>Returns: a multi-valued key object for use in the ViewObject’s findByKey() method.</td>
</tr>
<tr>
<td></td>
<td>Return Type: Key</td>
</tr>
<tr>
<td></td>
<td>Parameters:</td>
</tr>
<tr>
<td></td>
<td>■ list - a list of values for a multi-field key</td>
</tr>
<tr>
<td></td>
<td>Example: if a standard object has a two-field key, use key([101, 'SAMBA'])</td>
</tr>
</tbody>
</table>
Understanding Groovy's Null-Safe Comparison Operators

It's important to know that Groovy's comparison operators == and != handle nulls gracefully so you don't have to worry about protecting null values in equality or inequality comparisons. Furthermore, the >, >=, <, and <= operators are also designed to avoid null-related exceptions, however you need to be conscious of how Groovy treats null in these order-dependent comparisons. Effectively, a null value is "less than" any other non-null value in the natural ordering, so for example observe the following comparison results.

Table 4–5 Examples of How null Is Less Than Everything

<table>
<thead>
<tr>
<th>Left-Side Expression</th>
<th>Operator</th>
<th>Right-Side Expression</th>
<th>Comparison Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a'</td>
<td>&gt;</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>'a'</td>
<td>&lt;</td>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>100</td>
<td>&gt;</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>100</td>
<td>&lt;</td>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>-100</td>
<td>&gt;</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>-100</td>
<td>&lt;</td>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>now()</td>
<td>&gt;</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>now()</td>
<td>&lt;</td>
<td>null</td>
<td>false</td>
</tr>
<tr>
<td>now() - 7</td>
<td>&gt;</td>
<td>null</td>
<td>true</td>
</tr>
<tr>
<td>now() - 7</td>
<td>&lt;</td>
<td>null</td>
<td>false</td>
</tr>
</tbody>
</table>

If you want a comparison to treat a null-valued field with different semantics — for example, treating a null MaximumOverdraftAmount field as if it were zero (0), you can use the nvl function.

```
nvl(o1, o2)
```

Returns: the object o1 if it is not null, otherwise the object o2.

Return Type: Object

Parameters:
- o1 - a value to use if not null
- o2 - a value to use instead if o1 is null

Example: to calculate the sum of Salary and Commission custom fields that might be null, use `nvl(Salary_c,0) + nvl(Commission_c,0)`.
spreadsheet user might expect — then use the nvl() function as part of your comparison logic as shown in the following example:

```groovy
// Change default comparison semantics for the MaximumOverdraftAmount custom field in
// case its value is null by using nvl() to treat null like zero (0)
if (nvl(MaximumOverdraftAmount_c, 0) < -2000) {
    // do something for suspiciously large overdraft amount
}
```

As illustrated by the table above, without the nvl() function in the comparison any MaximumOverdraftAmount_c value of null would always be less than -2000 — since by default null is less than everything.

### Testing Whether an Field's Value Is Changed

You can test whether an field’s value has changed in the current transaction by using the built-in isAttributeChanged() function. As shown in this example, it takes a single string argument that provides the name of the field whose changed status you want to evaluate:

```groovy
if (isAttributeChanged('Status_c')) {
    // perform some logic here in light of the fact
    // that status has changed in this transaction
}
```

### Avoiding Validation Threshold Errors By Conditionally Assigning Values

When you write scripts for validation rules that modify the values of fields in the current object, you must be aware of how this affects the object’s so-called "validation cycle". Before allowing an object to be saved to the database, the Oracle ADF framework ensures that its data passes all validation rules. The act of successfully running all defined validation rules results in the object’s being marked as valid and allows the object to be saved along with all other valid objects that have been modified in the current transaction. If as part of executing a validation rule your script modifies the value of a field, this marks the object “dirty” again. This results in ADF’s subjecting the object again to all of the defined validation rules to ensure that your new changes do not result in an invalid object. If the act of re-validating the object runs your scripts that modify the field values again, this process could result in a cycle that would appear to be an infinite loop. ADF avoids this possibility by imposing a limit of 10 validation cycles on any given object. If after 10 attempts at running all the rules the object still has not been able to be successfully validated due to the object’s being continually modified by its validation logic, ADF will throw an exception complaining that you have exceeded the validation threshold:

*Validation threshold limit reached. Invalid Entities still in cache*

A simple way to avoid this from happening is to test the value of the field your script is about to assign and ensure that you perform the setAttribute() call to modify its value only if the value you intend to assign is different from its current value. An example script employing this approach would look like this:

```groovy
// Object-level validation rule on a PurchaseOrder object
// to derive the default purchasing rep based on a custom
// algorithm defined in an object function named
determinePurchasingRep() if both the Discount and NetDaysToPay
// fields have changed in the current transaction.
if (isAttributeChanged("Discount") &&
    isAttributeChanged("NetDaysToPay")) {
    def defaultRep = determinePurchasingRep()
```
Prefer "Before" Save-time Triggers to "After" Ones for Best Performance

When you write a trigger to derive field values programmatically, wherever possible use the Before Insert or Before Update triggers instead of After Insert, After Update, or After Changed Posted to Database. When the After-save triggers fire, the changes in the row have already been sent to the database, and performing further field assignments therein requires doing a second database DML operation per row modified to communicate your field updates to the database for permanent storage. Using the Before-save triggers sets the field values before the changes are sent the first time to the database, resulting in better performance.

Avoiding Posting Threshold Errors By Conditionally Assigning Values

Despite the recommendation in "Prefer "Before" Save-time Triggers to "After" Ones for Best Performance", if for some reason you still must use an After Insert, After Update, or After Changed Posted to Database trigger to perform field value assignments — for example, your custom logic must perform a query that filters on the data being updated in the current transaction — then you must be aware of how this affects the object's so-called "posting cycle". If your trigger modifies the value of a field, this marks the object "dirty" again. This results in ADF's subjecting the object again to all of the defined validation rules to ensure that your new changes do not result in an invalid object. If the object passes validation, then your trigger's most recent field value changes must be posted again to the database. In the act of re-posting the object's changes, your trigger may fire again. If your trigger again unconditionally modifies one or more field values again, this process could result in a cycle that would appear to be an infinite loop. ADF avoids this possibility by imposing a limit of 10 posting cycles on any given object. If after 10 attempts to post the (re)validated object to the database it remains "dirty," due to the object's being continually modified by your trigger logic, then ADF will throw an exception complaining that you have exceeded the posting threshold:

Post threshold limit reached. Some entities yet to be posted

A simple way to avoid this from happening is to test the value of the field your script is about to assign and ensure that you perform the setAttribute() call to modify its value only if the value you intend to assign is different from its current value. An example script employing this approach would look like this:

```groovy
// After Changes Posted in Database Trigger
// If total score is 100 or more, set status to WON.
```
Referencing Original Values of Changed Fields

```groovy
def totalScore = calculateTotalScoreUsingQuery()
if (totalScore >= 100) {
    // Only set the status to WON if it's not already that value
    if (Status != 'WON') {
        setAttribute('Status', 'WON')
    }
}
```

### Functional Restrictions in Trigger Scripts

This section documents functional restrictions of which you should be aware when writing custom Groovy script in triggers.

- **After Delete in Database** Trigger
  
  Your trigger cannot set the value of fields in the deleted object. An attempt to do so will raise the `InvalidOperException` exception.

- **Before Commit in Database** Trigger
  
  Your trigger should not set the value of any fields in this trigger. The changes are too late to be included in the current transaction.

- **After Commit in Database** Trigger
  
  Your trigger should not set the value of any fields in this trigger. The changes are too late to be included in the current transaction.

- **Before Rollback in Database** Trigger
  
  Your trigger should not set the value of any fields in this trigger. The changes are too late to be included in the current transaction.

- **After Rollback in Database** Trigger
  
  Your trigger should not set the value of any fields in this trigger. The changes are too late to be included in the current transaction.

### Passing the Current Object to a Global Function

As you begin to recognize that you are writing repetitive code in more than one object's functions, triggers, or validation rules, you can refactor that common code into a global function that accepts a parameter of type `Object` and then have your various usages call the global function. Since a global function does not have a "current object" context, you need to pass the current object as an argument to your global function to provide it the context you want it to work with. When you define the global function, the `Object` type is not shown in the dropdown list of most commonly used argument types, but you can type it in yourself. Note that the value is case-sensitive, so `Object` must have an initial capital letter "O".

When writing code in an object trigger, object function, or other object script, you can use the expression `adf.source` to pass the current object to a global function that you invoke. "Referencing Original Values of Changed Fields" shows an example of putting these two techniques into practice.

### Referencing Original Values of Changed Fields

When the value of a field gets changed during the current transaction, the ADF framework remembers the so-called "original value" of the field. This is the value it had when the existing object was retrieved from the database. Sometimes it can be
useful to reference this original value as part of your business logic. To do so, use
the getOriginalAttributeValue() function as shown below (substituting your field's
name for the example's Priority_c):

```groovy
// Assume we're in context of a TroubleTicket
if (isAttributeChanged('Priority_c')) {
    def curPri = Priority_c
    def origPri = getOriginalAttributeValue("Priority_c")
    adf.util.log("Priority changed: ${origPri} -> ${curPri}")
    // do something with the curPri and origPri values here
}
```

### Raising a Warning From a Validation Rule Instead of an Error

When your validation rule returns `false`, it causes a validation error that stops normal
processing. If instead you want to show the user a warning that does *not* prevent the
data from being saved successfully, then your rule can signal a warning and then
return `true`. For example, your validation rule would look like this:

```groovy
// if the discount is over 50%, give a warning
if (Discount > 0.50) {
    // raise a warning using the default declarative error message
    adf.error.warn(null)
}
return true
```

### Throwing a Custom Validation Exception

When defining object level validation rules or triggers, normally the
declaratively-configured error message will be sufficient for your needs. When your
validation rule returns `false` to signal that the validation has failed, the error message
you've configured is automatically shown to the user. The same occurs for a trigger
when it calls the `adf.error.raise(null)` function. If you have a number of different
conditions you want to enforce, rather than writing one big, long block of code that
enforces several distinct conditions, instead define a separate validation rule or trigger
(as appropriate) for each one so that each separate check can have its own appropriate
error message.

That said, on occasion you may require writing business logic that does not make
sense to separate into individual rules, and which needs to conditionally determine
*which* among several possible error messages to show to the user. In this case, you can
throw a custom validation exception with an error string that you compose on the fly
using the following technique:

```groovy
// Throw a custom object-level validation rule exception
// The message can be any string value
throw new oracle.jbo.ValidationException('Your custom message goes here')
```

Note that choose this approach, your error message is not translatable in the standard
way, so it becomes your responsibility to provide translated versions of the
custom-thrown error messages. You could use a solution like the one presented in
"Returning Locale-Sensitive Custom Strings" for accomplishing the job.

### Returning Locale-Sensitive Custom Strings

When you throw custom validation error messages, if your end users are
multi-lingual, you may need to worry about providing a locale-specific error message
string. To accomplish this, you can reference the end user's current locale as part of
global function that encapsulates all of your error strings. Consider a `getMessages`
function like the one below. Once it is defined, your validation rule or trigger can throw a locale-sensitive error message by passing in the appropriate message key:

```java
// context is trigger or object-level validation rule
throw new oracle.jbo.ValidationException(adf.util.getMessage('BIG_ERROR'))
```

The global function is defined as follows.

### Function Name
getMessage

### Return Type
String

### Parameters

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>stringKey</td>
<td>String</td>
</tr>
</tbody>
</table>

### Function Definition

```java
// Let 'en' be the default lang
// Get the language part of the locale
// e.g. for locale 'en_US' lang part is 'en'
def defaultLang = 'en';
def userLocale = adf.context.getLocale() as String
def userLang = left(userLocale,2)
def supportedLangs=['en','it']
def lookupLang = supportedLangs.contains(userLang)
    ? userLang : defaultLang
def messages =
    [BIG_ERROR: [en:'A big error occurred',
    it:'È successo un grande errore'],
    SMALL_ERROR:[en:'A small error occurred',
    it:'È successo un piccolo errore']
]
return messages[stringKey][lookupLang]
```

### Raising a Trigger's Optional Declaratively-Configured Error Message

In contrast with a validation rule where the declarative error message is mandatory, when you write a trigger it is optional. Since any return value from a trigger's script is ignored, the way to cause the optional error message to be shown to the user is by calling the `adf.error.raise()` method, passing `null` as the single argument to the function. This causes the default declarative error message to be shown to the user and stops the current transaction from being saved successfully. For example, you trigger would look like this:

```java
// Assume this is in a Before Insert trigger
if (someComplexCalculation() == -1) {
    // raise an exception using the default declarative error message
    adf.error.raise(null)
}
```

### Accessing the View Object for Programmatic Access to Business Objects

A "view object" is an Oracle ADF component that simplifies querying and working with business object rows. The `newView()` function allows you to access a view object dedicated to programmatic access for a given business object. By default, any custom
object you create is enabled to have such a view object, and selected standard objects will be so-enabled by the developers of the original application you are customizing. Each time the `newView(objectAPIName)` function is invoked for a given value of object API name, a new view object instance is created for its programmatic access. This new view object instance is in a predictable initial state. Typically, the first thing you will then do with this new view object instance is:

- Call the `findByKey()` function on the view object to find a row by key, or
- Use a view criteria to restrict the view object to only return some desired *subset of* business objects rows that meet your needs. Filtering the rows returned by the view object involves creating and configuring a new view criteria, appending this view criteria to the view object, and then executing the view object as described in "Finding Objects Using a View Criteria".

A view object will typically be configured to return its results in sorted order. If the default sort order does not meet your needs, you can use the `setSortBy()` method on the view object to provide a comma-separated list of field names on which to sort the results. The new sort order will take effect the next time you call the `executeQuery()` method on the view object. See "Defining the Sort Order for Query Results" for further details on sorting options available.

A view object instance for programmatic access to a business object is guaranteed not to be used by any application user interface pages. This means that any iteration you perform on the view object in your script will not inadvertently affect the current row seen in the user interface. That said, the end user will see the results of any field values that you change, any new rows that you add, and any existing rows that you modify, presuming that they are presently on a page where said objects and fields are visible.

For example, suppose the user interface is displaying an employee along with the number and associated name of the department in which she works. If a script that you write...

- uses `newView()` to obtain the view object for programmatic access for the `Department` object, then
- uses `findByKey()` to find the department whose id matches the current employee’s department, and finally
- changes the name of the current employee’s department

then this change should be reflected in the screen the next time it is updated. Once you’ve accessed the view object, in addition to your use of the built-in `newViewCriteria()` function, the most common methods that you will use on the view object are shown in Table 4–6.
Defining the Sort Order for Query Results

To define the sort order for view object query results, call the `setSortBy()` method on the view object instance you are working with before calling its `executeQuery()` method to retrieve the results. The `setSortBy()` function takes a single string argument whose value can be a comma-separated list of one or more field names in the object. Example 4–7 shows how to use this method to sort by a single field.
**Example 4–7 Setting a Single Sort Field**

```groovy
def vo = newView('TroubleTicket_c')
// Use object function to simplify filtering by agent
applyViewCriteriaForSupportAnalyst(vo, analystId)
vo.setSortBy('Priority_c')
vo.executeQuery()
while (vo.hasNext()) {
    def curRow = vo.next()
    // Work with current row curRow here
}
```

By default the sort order will be *ascending*, but you can make your intention explicit by using the `asc` or `desc` keyword after the field’s name in the list, separated by a space. Example 4–8 shows how to sort descending by the number of callbacks.

**Example 4–8 Controlling the Sort Order**

```groovy
def vo = newView('TroubleTicket_c')
// Use object function to simplify filtering by customer
applyViewCriteriaForCustomerCode(vo, custCode)
vo.setSortBy('NumberOfCallbacks_c desc')
vo.executeQuery()
while (vo.hasNext()) {
    def curRow = vo.next()
    // Work with current row curRow here
}
```

As mentioned before, the string can be a comma-separated list of two or more fields as well. Example 4–9 shows how to sort by multiple fields, including explicitly specifying the sort order.

**Example 4–9 Sorting on Multiple Fields**

```groovy
def vo = newView('TroubleTicket_c')
// Use object function to simplify filtering by customer
applyViewCriteriaForCustomerCode(vo, custCode)
// Sort ascending by Priority, then descending by date created
vo.setSortBy('Priority_c asc, CreationDate desc')
vo.executeQuery()
while (vo.hasNext()) {
    def curRow = vo.next()
    // Work with current row curRow here
}
```

By default, when sorting on a text field, its value is sorted case-sensitively. A value like 'Blackberry' that starts with a capital 'B' would sort before a value like 'apple' with a lower-case 'a'. To indicate that you’d like a field’s value to be sorted case-insensitively, surround the field name in the list by the `UPPER()` function as shown in Example 4–10.
Example 4–10  Sorting Case-Insensitively

def vo = newView('TroubleTicket_c')
// Use object function to simplify filtering by customer
applyViewCriteriaForCustomerCode(vo, custCode)
// Sort case-insensitively by contact last name, then by priority
vo.setSortBy('UPPER(ContactLastName_c),Priority_c')
vo.executeQuery()
while (vo.hasNext()) {
    def curRow = vo.next()
    // Work with current row curRow here
}

**Tip:** While it is possible to sort on a formula field or dynamic choice field by specifying its name, don’t do so unless you can guarantee that only a small handful of rows will be returned by the query. Sorting on a formula field or dynamic choice list must be done in memory, therefore doing so on a large set of rows will be inefficient.

Finding an Object by Id

To find an object by id, follow these steps:

1. Use the `newView()` function to obtain the view object for programmatic access for the business object in question

2. Call `findByKey()`, passing in a key object that you construct using the `key()` function

The new object will be saved the next time you save your work as part of the current transaction. Example 4–11 shows how the steps fit together in practice.

Example 4–11  Finding an Object by Id

```
// Access the view object for the custom TroubleTicket object
def vo = newView('TroubleTicket_c')
def foundRows = vo.findByKey(key(100000000272002),1)
def found = foundRows.size() == 1 ? foundRows[0] : null;
if (found != null) {
    // Do something here with the found row
}
```

To simplify the code involved in this common operation, you could consider defining the following `findRowByKey()` global helper function:

**Function Name**
findRowByKey

**Return Type**
Object

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vo</td>
<td>Object</td>
</tr>
<tr>
<td>idValue</td>
<td>Object</td>
</tr>
</tbody>
</table>
Find objects using a View Criteria

Function Definition

```
adf.util.logStart('findRowByKey')
def found = vo.findByKey(key(idValue),1)
return found.size() == 1 ? found[0] : null;
```

After defining this helper function, Example 4–12 shows the simplified code for finding a row by key.

**Example 4–12  Finding an Object by Id Using Global Function**

```
// Access the view object for the custom TroubleTicket object
def vo = newView('TroubleTicket_c')
def found = adf.util.findRowByKey(vo,100000000272002)
if (found != null) {
  // Do something here with the found row
}
```

Finding Objects Using a View Criteria

A *view criteria* is a collection of one or more Query-By-Example (QBE) rows that define a data filter for a view object. Each QBE row in a view criteria is called a *view criteria row*. Each view criteria row has the same fields as the object you are filtering, and you assign values to the desired fields in the view criteria row to provide an example of what you want to find. At runtime, the Oracle ADF framework turns the view criteria into an appropriate SQL *WHERE* clause to pass your filter criteria to the database for execution.

Overview of Basic Steps to Use a View Criteria

To find objects using a view criteria, perform the following steps:

1. Use the `newView()` function to obtain the view object for programmatic access for the business object in question
2. Use the `newViewCriteria()` function to create a new view criteria on the view object
3. Create a new view criteria row by calling `createRow()` on the view criteria
4. Define one or more view criteria items by doing the following for each one:
   a. Create a new view criteria item by calling `ensureCriteriaItem()` on the view criteria row
   b. Define the criteria operator on the view criteria item by calling `setOperator()` on the view criteria item
   c. Set the value of the view criteria item operand by calling `setValue()` on the view criteria item
   d. For text field criteria items, if the search should be case-insensitive, then call `setUpUpperColumns(true)` on the view criteria item
5. Add the new view criteria row by calling `insertRow()` on the view criteria
6. Append the view criteria by calling `appendViewCriteria()` on the view object

Tip: See Table 4–7 in "Understanding View Criteria Item Operators" for a list of valid operators.
7. Execute the query by calling `executeQuery()`

Example 4–13 demonstrates how all the pieces fit together in an example for querying the `TroubleTicket` custom object to find the trouble tickets assigned to a particular staff member and which have a status of `Working`.

Example 4–13  Basic Example of Using View Criteria

```groovy
/*
 * Query all 'Working'-status trouble tickets assigned to a given staff member
 */
def staffMemberId = 100000000089003
// 1. Use the newView() function to get a view object
def vo = newView('TroubleTicket_c')
// 2. Create a new view criteria
def vc = newViewCriteria(vo)
// 3. Create a new view criteria row
def vcr = vc.createRow()
// 4. Define two view criteria items
// 4a. (#1) Create a view criteria item for AssignedTo_Id
def vci1 = vcr.ensureCriteriaItem('AssignedTo_Id_c')
// 4b. (#1) Set the operator of the first view criteria item
vci1.setOperator('=')
// 4c. (#1) Set the value of the first view criteria item
vci1.setValue(staffMemberId)
// 4a. (#2) Create a view criteria item for Status
def vci2 = vcr.ensureCriteriaItem('Status_c')
// 4b. (#2) Set the operator of the first view criteria item
vci2.setOperator('=')
// 4c. (#2) Set the value of the first view criteria item
vci2.setValue('working')
// 4d. (#2) Set the text field criteria to be case-insensitive
vci2.setUpperColumns(true)
// 5. Add the new view criteria row
vc.insertRow(vcr)
// 6. Append the view criteria to the view object
vo.appendViewCriteria(vc)
// 7. Execute the query
vo.executeQuery()
```

Simplifying Most Common View Criteria Creation with a Global Function

Very often, the kinds of a view criteria that you will use will be simple ones with equals operators, so you can create a global function to simplify creating common data filters like this. The `applyFilter()` function shown below would allow you to simplify the code in Example 4–13 to look like this:

```groovy
/*
 * Query all 'Working'-status trouble tickets assigned to a given staff member
 */
def staffMemberId = 100000000089003
// 1. Use the newView() function to get a view object
def vo = newView('TroubleTicket_c')
// 2. Filter on AssignedTo_Id = Create a new view criteria
adf.util.applyFilter(vo,[Status_c:'Working',AssignedTo_Id_c:staffMemberId])
// 3. Execute the query
vo.executeQuery()
```
The global function looks like this:

**Function Name**
applyFilter

**Return Type**
void

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>vo</td>
<td>Object</td>
</tr>
<tr>
<td>fieldsToFilter</td>
<td>Map</td>
</tr>
</tbody>
</table>

**Function Definition**

```groovy
/*
 * Defines a new view criteria with one view criteria row
 * having a view criteria item for each map entry in
 * the 'fieldsToFilter' parameter
 */
adf.util.logStart('applyFilter')
def criteria = newViewCriteria(vo)
def criteriaRow = criteria.createRow()
adf.util.log("fieldsToFilter = ${fieldsToFilter}")
criteria.insertRow(criteriaRow)
for (curField in fieldsToFilter?.keySet()) {
    def filterValue = fieldsToFilter[curField]
    adf.util.log("Filter on ${curField} = ${filterValue}")
    // if the value to filter on is a List then
    if (filterValue instanceof List) {
        adf.util.addMultiValueCriteriaItem(criteriaRow,
                                            curField,
                                            filterValue,
                                            true)
    }
    else {
        def criteriaItem = criteriaRow.ensureCriteriaItem(curField)
        criteriaItem.setValue(filterValue)
    }
}
vo.appendViewCriteria(criteria)
```

### Understanding View Criteria Item Operators

When you create a view criteria item, Table 4–7 provides the list of valid values that you can provide as the single argument to the `setOperator()` method on the view criteria item object. Notice that some operators are valid only for fields of certain data types. For example, for a field `Priority` of type number, to set the "Greater than or equals" operator on the related view criteria item, you would write a line of code like this:

```groovy
// Use the 'greater than or equals' operator
priorityVCItem.setOperator('>=')
```

In contrast, for a view criteria item related to a `ResolutionDate` field of type Date, you would use the `ONORAFTER` operator like this:

```groovy
// Use the 'on or after' operator
resolutionDateVCItem.setOperator('ONORAFTER')
```
Using View Criteria Items with the ISBLANK Operator

The `ISBLANK` and `ISNOTBLANK` view criteria item operators do not require any operand value to be set. For example, assuming that `vcr` represents a view criteria row in which you want to create a criteria item to filter where `Justification` is null, you would write script like this:

```groovy
// Assume vcr is a view criteria row
def vci = vcr.ensureCriteriaItem('Justification_c')
vci.setOperator('ISBLANK')
// no need to call vci.setValue() with the ISBLANK operators
```

The same consideration holds for the `ISNOTBLANK` operator.

Using View Criteria Items with the BETWEEN Operator

The `BETWEEN` and `NOTBETWEEN` view criteria item operators require two arguments. For example, assuming that `vcr` represents a view criteria row in which you want to create a criteria item to filter where `Priority` is between 2 and 4, you would write script like this:

```groovy
// Assume vcr is a view criteria row
def vci = vcr.ensureCriteriaItem('Priority_c')
vci.setOperator('BETWEEN')
// no need to call vci.setValue() with the BETWEEN operators
```

The table below summarizes the view criteria item operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Valid for Field Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>=</code></td>
<td>Equals</td>
<td>Number, Text, Date</td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td>Not equals</td>
<td>Number, Text, Date</td>
</tr>
<tr>
<td><code>ISBLANK</code></td>
<td>Is null (no view criteria item value required)</td>
<td>Number, Text, Date</td>
</tr>
<tr>
<td><code>ISNOTBLANK</code></td>
<td>Is not null (no view criteria item value required)</td>
<td>Number, Text, Date</td>
</tr>
<tr>
<td><code>LIKE</code></td>
<td>Like</td>
<td>Text</td>
</tr>
<tr>
<td><code>STARTSWITH</code></td>
<td>Starts with</td>
<td>Text</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
<td>Number, Text</td>
</tr>
<tr>
<td><code>AFTER</code></td>
<td>Date comes after</td>
<td>Date</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater than or equal to</td>
<td>Number, Text</td>
</tr>
<tr>
<td><code>ONORAFTER</code></td>
<td>Date is on or comes after</td>
<td>Date</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
<td>Number, Text</td>
</tr>
<tr>
<td><code>BEFORE</code></td>
<td>Date comes before</td>
<td>Date</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Less than or equal to</td>
<td>Number, Text</td>
</tr>
<tr>
<td><code>ONORBEFORE</code></td>
<td>Date is on or comes before</td>
<td>Date</td>
</tr>
<tr>
<td><code>BETWEEN</code></td>
<td>Is between (two view criteria item values required)</td>
<td>Number, Text, Date</td>
</tr>
<tr>
<td><code>NOTBETWEEN</code></td>
<td>Is not between (two view criteria item values required)</td>
<td>Number, Text, Date</td>
</tr>
</tbody>
</table>
// Assume vcr is a view criteria row
def vci = vcr.ensureCriteriaItem('Priority_c')
vci.setOperator('BETWEEN')
// The BETWEEN operator requires two values...
// Set first value to 2 (zero-based index!)
vci.setValue(0 /* index */, 2 /* value */)  
// Set second value to 4 (zero-based index!)
vci.setValue(1, 4)

Understanding View Criteria Row and Item Conjunctions

For many purposes, a view criteria with a single view criteria row will be ample. By default, when the view criteria is converted into a SQL WHERE clause, all of the view criteria items in the same view criteria row are joined by an AND conjunction to create a SQL fragment that looks like this (assuming the simplest equals operator on each view criteria item):

(FieldName1 = Value1 AND FieldName2 = Value2) /* from View Criteria Row#1 */

If you need to define multiple sets of criteria, you can add additional view criteria rows to the view criteria. By default, when the view criteria is converted into a SQL WHERE clause, each of the SQL fragments created for the individual view criteria rows is joined by an OR conjunction like this:

(FieldName1 = Value1 AND FieldName2 = Value2) /* from View Criteria Row#1 */ OR (FieldName3 = Value3 AND FieldName4 = Value4) /* from View Criteria Row#2 */

If you need to change the default AND conjunction used to join a view criteria item with the one processed just before it in the same view criteria row, then you can call the setConjunction() method on it to change the conjunction to an OR. The method accepts a numerical argument with number 1 corresponding to the AND conjunction and the number 0 (zero) corresponding to the OR conjunction. Therefore, to set a particular view criteria item to use the OR conjunction instead, use the code:

vci.setConjunction(0 /* OR */)  

If you need to change the default OR conjunction used to join the SQL fragment produced by a given view criteria row with the view criteria row processed just before it in the same view criteria, then you can call the setConjunction() method on it to change the conjunction to an AND. Therefore, to set a particular view criteria row to use the AND conjunction instead, use the code:

vcr.setConjunction(1 /* AND */)  

Using Compound Criteria Item to Search for Multiple Alternatives

To create a view criteria item that includes multiple values for the same field — for example, to query where the value of the Status field is either Working or Waiting, then you can use a compound criteria item value. You make a criteria item into a compound item by calling the makeCompound() method on it. Once you’ve done this, you use the ensureItem() method on the compound criteria item to create nested criteria items.

For example, the addMultiValueCriteriaItem function below simplifies creating multi-valued criteria item. You may have noticed that it is used by the applyFilter() global function in "Simplifying Most Common View Criteria Creation with a Global Function". Once you put the two global functions to work together, you can create a
criteria like "find the trouble tickets assigned to Jane Jedeye with a status Working or Waiting" by using a script like:

```groovy
// Query all 'Working'- or 'Waiting'-status trouble tickets assigned to a staff member
def staffMemberId = 100000000089003
// 1. Use the newView() function to get a view object
def vo = newView('TroubleTicket_c')
// 2. Filter on AssignedTo_Id = Create a new view criteria
adf.util.applyFilter(vo, [Status_c: ['Working', 'Waiting'], AssignedTo_Id_c: staffMemberId])
// 3. Execute the query
vo.executeQuery()
```

The code in the `applyFilter()` global function detects that the map entry for `Status_c` has a list as its value. When it detects this it uses `addMultiValueCriteriaItem()` to add the compound view criteria item to query for where `Status = 'Working' OR Status = 'Waiting'`. The relevant SQL `WHERE` clause criteria will look something like:

```sql
(STAFF_MEMBER_ID = 100000000089003
AND
(UPPER(STATUS) = UPPER('Working')
OR
  UPPER(STATUS) = UPPER('Waiting'))
)
```

The function definition looks like this:

**Function Name**

`addMultiValueCriteriaItem`

**Return Type**

`void`

**Parameters**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>criteriaRow</td>
<td>Object</td>
</tr>
<tr>
<td>fieldName</td>
<td>String</td>
</tr>
<tr>
<td>listOfValues</td>
<td>List</td>
</tr>
<tr>
<td>caseInsensitive</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
**Function Definition**

```groovy
daflow.util.logStart('addMultiValueCriteriaItem')
def compoundVCI =
criteriaRow.ensureCriteriaItem(fieldName).makeCompound();
compoundVCI.setOperator('=');
if (listOfValues != null) {
    for (int z=0; z < listOfValues.size(); z++) {
        def nestedVCI = compoundVCI.ensureItem(z);
        nestedVCI.setOperator('=');
        nestedVCI.setUpperColumns(caseInsensitive);
        // Nested items beyond first use 'OR' instead of 'AND'
        if (z > 0) {
            nestedVCI.setConjunction(0 /* OR */);
        }
        nestedVCI.setValue(listOfValues.get(z));
    }
}
```

**Using a Predefined View Criteria on a Standard Object**

Standard objects may predefine named view criteria you can use in your scripts to simplify common searches. See a particular standard object's documentation to learn whether it defines any named view criteria. After learning the name of the view criteria you want to employ, use the `copyNamedViewCriteria()` function to use it in your script. Example 4-14 shows the code you need to work with a fictitious `ServiceTicket` standard object that predefines an `AllSeverityOneOpenTickets` named view criteria.

**Example 4-14  Using Predefined Named View Criteria**

```groovy
/*
 * Query all open severity 1 service tickets
 * using a predefined view criteria
 */
// 1. Use newView() to get a view object
def vo = newView('ServiceTicket')
// 2. Copy predefined named view criteria
def vc = copyNamedViewCriteria(vo,'AllSeverityOneOpenTickets')
// 3. Append view criteria to the view object
vo.appendViewCriteria(vc)
// 4. Execute the query
vo.executeQuery()
```

Named view criteria may reference named view variables in their filter criteria. Your code can (or sometimes must!) assign a value to one or more these view variables for the view criteria to work correctly. After consulting the documentation for the standard object you are working with, if it mentions that named view variables must be set, then use the `setViewVariable()` function to assign a value to these in your script before executing the query. Example 4-15 shows the code you need to work with a fictitious `ServiceTicket` standard object that predefines an `AllOpenTicketsByAssigneeAndPriority` named view criteria. The `Bind_MaxPriority` view variable might default to the value 4, so it may be optional to set it in your script. In contrast, the `Bind_Assignee` view variable would likely not have a default value and your script must provide a value. Failure to do this would result in the query’s returning no rows.
Example 4–15 Using Named View Criteria with View Variables

`/*
 * Query all open tickets less than a
 * given priority assigned to a
 * given support engineer.
 */
// 1. Use newView() to get a view object
def vo = newView('ServiceTicket')
// 2. Copy predefined, named view criteria
def vc = copyNamedViewCriteria(vo,'AllOpenTicketsByAssigneeAndPriority')
// 3. Append view criteria to the view object
vo.appendViewCriteria(vc)
// 4. Set optional Bind_Priority view variable
setViewVariable(vo,'Bind_MaxPriority',2)
// 5. Set mandatory Bind_Assignee view variable
setViewVariable(vo,'Bind_Assignee','psmith')
// 6. Execute the query
vo.executeQuery();`

Creating a New Object

To create a new object, follow these steps:

1. Use the `newView()` function to obtain the view object for programmatic access for the business object in question
2. Call the `createRow()` function on the view object to create a new row
3. Call `setAttribute()` as needed to set the desired field values in the new row
4. Call `insertRow()` on the view object to insert the row.

The new object will be saved the next time you save your work as part of the current transaction. Example 4–16 shows how the steps fit together in practice.

Example 4–16 Creating a New Object

```
// Access the view object for the custom TroubleTicket object
def vo = newView('TroubleTicket_c')
// Create the new row
def newTicket = vo.createRow()
// Set the problem summary
newTicket.setAttribute('ProblemSummary_c','Cannot insert floppy disk')
// Assign the ticket a priority
newTicket.setAttribute('Priority_c',2)
// Relate the ticket to a customer by name
newTicket.setAttribute('Contact_c','Tim Tubbington')
// Relate the ticket to a staff member by name
newTicket.setAttribute('AssignedTo_c','Jane Jedeye')
// Insert the new row into the view object
vo.insertRow(newTicket)
// The new data will be saved to the database as part of the current transaction when it is committed.
```

Updating an Existing Object

If the object you want to update is the current row in which your script is executing, then you need only use the `setAttribute()` function to assign new values to the fields as needed.
However, if you need to update an object that is different from the current row, perform these steps:

1. Use `newView()` to access the appropriate view object for programmatic access
2. Find the object by id or find one or more objects using a view criteria, depending on your use requirements
3. Call the `setAttribute()` method on the row or rows to assign new values as needed

The changes will be saved as part of the current transaction when the user commits it.

---

**Tip:** See "Avoiding Validation Threshold Errors By Conditionally Assigning Values" for a tip about how to avoid your field assignments from causing an object to hit its validation threshold.

---

**Permanently Removing an Existing Object**

To permanently remove an existing object, perform these steps:

1. Use `newView()` to access the appropriate view object for programmatic access
2. Find the object by id or find one or more objects using a view criteria, depending on your use requirements
3. Call the `remove()` method on the row or rows as needed

The changes will be saved as part of the current transaction when the user commits it.

**Reverting Changes in a Single Row**

To revert pending changes to an existing object, perform these steps:

1. Use `newView()` to access the appropriate view object for programmatic access
2. Find the object by id
3. Call the `revertRowAndContainees()` method as follows on the row

   ```java
   yourRow.revertRowAndContainees()
   ```

**Understanding Why Using Commit or Rollback In Scripts Is Strongly Discouraged**

It is strongly advised to not commit or rollback the transaction from within your scripts, allowing instead that any changes made by your script to be committed or rolled-back along with the rest of the current transaction. If your script code were to call `commit()` or `rollback()`, this would affect all changes pending in the current transaction, not only those performed by your script. Forewarned is forearmed.

**Using the User Data Map**

The Oracle ADF framework provides a map of name/value pairs that is associated with the current user's session. You can use this map to temporarily save name/value pairs for use by your business logic. Be aware that the information that you put in the user data map is never written out to a permanent store, and is not replicated under failover conditions. If the code that reads the value in the user data map executes as part of different web request from the code that put the value in the map, then that
code reading the map should be written to expect that the value that has been put their may not be there as expected (due to server failover).

To access the server map from a validation rule or trigger, use the expression `adf.userSession.userData` as shown in the following example:

```groovy
// Put a name/value pair in the user data map
adf.userSession.userData.put('SomeKey', someValue)

// Get a value by key from the user data map
def val = adf.userSession.userData.SomeKey
```

**Tip:** See “Working with Maps” for more information on using maps in your scripts.

---

**Referencing Information About the Current User**

The `adf.context.getSecurityContext()` expression provides access to the security context, from which you can access information about the current user like her user name or whether she belongs to a particular role. The following code illustrates how to reference these two pieces of information:

```groovy
// Get the security context
def secCtx = adf.context.getSecurityContext()
// Check if user has a given role
if (secCtx.isUserInRole('MyAppRole')) {
    // get the current user's name
    def user = secCtx.getUserName()
    // Do something if user belongs to MyAppRole
}
```

---

**Using Aggregate Functions**

Oracle ADF’s built-in support for row iterator aggregate functions can simplify a number of common calculations you will perform in your scripts, especially in the context of scripts written in a parent object which has one or more collections of child objects.

---

**Understanding the Supported Aggregate Functions**

Oracle ADF supports five built-in aggregate functions over all the rows in a row set. The most common use case is to calculate an aggregate value of a child collection in the context of a parent object. Table 4-8 provides a description and example of the supported functions.

**Table 4-8 Supported Aggregate Functions**

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Description</th>
<th>Example (in Context of TroubleTicket Parent Object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg</td>
<td>Average value of an expression</td>
<td>ActivityCollection_c.avg('Duration_c')</td>
</tr>
<tr>
<td>min</td>
<td>Minimum value of an expression</td>
<td>ActivityCollection_c.min('Duration_c')</td>
</tr>
<tr>
<td>max</td>
<td>Maximum value of an expression</td>
<td>ActivityCollection_c.max('Duration_c')</td>
</tr>
</tbody>
</table>
Understanding How Null Values Behave in Aggregate Calculation

When an ADF aggregate function executes, it iterates over each row in the row set. For each row, it evaluates the Groovy expression provided as an argument to the function in the context of the current row. If you want a null value to be considered as zero for the purposes of the aggregate calculation, then use the `nvl()` function like this:

```groovy
// Use nvl() Function in aggregate expression
def avgDuration = ActivityCollection_c.min('nvl(Duration_c,0)')
```

Performing Conditional Counting

In the case of the `count()` function, if the expression evaluates to null then the row is not counted. You can supply a conditional expression to the `count()` function which will count only the rows where the expression returns a non-null value. For example, to count the number of child activities for the current trouble-ticket where the Duration was over half an hour, you can use the following expression:

```groovy
// Conditional expression returns non-null for rows to count
// Use the inline if/then/else operator to return 1 if the duration is over 0.5 hours, otherwise return null to avoid counting that the non-qualifying row.
def overHalfHourCount = ActivityCollection_c.count('nvl(Duration_c,0) > 0.5 ? 1 : null')
```

Inserting an Aggregate Expression Using the Expression Palette

As shown in Figure 4–8, the Expression Palette's Functions tab allows you to insert any of the aggregate functions. They appear in the Number category.

**Figure 4–8 Selecting an Aggregate Function to Insert**

After selecting the desired function and clicking the (Insert) button, as shown in Figure 4–9, and additional Configure Function dialog appears to allow you to select the field name over which the aggregate function will be performed. Note that in this dialog, the Fields table only shows field names when you select an object representing a multi-row collection in the tree at the left.
Understanding When to Configure Field Dependencies

You need to correctly configure dependent field information when you define a formula field or when you configure conditionally updateable and/or conditionally required expressions. If you fail to correctly configure this information, your application will not behave correctly at runtime.

Configuring Depends On Fields for a Formula

The ADF framework — unlike, say, a spreadsheet program — does not automatically infer what field(s) your Groovy scripts depend on. For example, suppose that in a OrderLineItem object, you added a formula field named LineTotal object having the following Groovy formula:

\[(\text{nvl(UnitPrice}_c,0) \times \text{nvl(Quantity}_c,0)) \times (1.00 - \text{nvl(LineDiscountPercentage}_c,0))\]

On its own, the ADF framework would not be able to automatically recalculate the line total when the user changed the UnitPrice, Quantity, or LineDiscountPercentage. Instead, you manually configure the Depends On information for any field formulas need it. For example, Figure 4–10 shows the Depends On multi-select list configured to reflect the fields on which the LineTotal formula depends. With this information correctly configured, the LineTotal will automatically update to reflect the new value whenever any of the values of the field on which it depends is changed.
Configuring Depends On Fields for Conditional Updateability or Conditional Mandatory

When you define a formula field, the Depends On multi-select list is always visible. However, for other field types it is only visible when you have configured either a conditionally updateable expression or a conditionally required expression. In this latter case, it appears to remind you that you must configure the information that tells the Oracle ADF framework on which other fields your conditionally updateable and/or conditionally required expression depends. If you fail to configure this information, your application will not behave correctly.

Understanding Automatic Dependency Information for Cascading Fixed-Choice Lists

Assume that you have defined two fixed-choice fields on the OrderLineItem objects named FulfillmentCenter and FulfillmentWarehouse. When defining the FulfillmentWarehouse field, further assume that you configured it to be constrained based on the value of the FulfillmentCenter field. This combination allows the end-user to first pick a fulfillment center, then to pick an appropriate fulfillment warehouse related to that choice of fulfillment center. If the user changes the value of the fulfillment center, then the current value of the fulfillment warehouse is set to null to force the end-user to select an appropriate fulfillment warehouse based on the new value of the constraining FulfillmentCenter field. This type of cascading fixed-choice list dependency is automatically configured when you setup the “constrained by” field as part of defining the second fixed-choice field. In this example, the FulfillmentWarehouse field depends on the FulfillmentCenter field.

Next, assume that your application has the requirement to allow configuring the fulfillment warehouse only for line items with a quantity greater than five. To support this functionality, you would configure the following conditionally updateable expression on the FulfillmentWarehouse field:

```
Quantity_c > 5
```

After doing this, the Depends On multi-select list will appear in the Constraints section so you can configure the fact that your FulfillmentWarehouse now also depends on the Quantity field. After doing this, when the user changes the value of the quantity at runtime, this will dynamically affect whether the fulfillment warehouse field is updateable or not. Figure 4–11 shows the process of configuring the Quantity field as one on which the FulfillmentWarehouse field now depends. Notice that the Fulfillment Center field appears selected in the Depends On list, yet is disabled. This is because that dependency is required for the correct runtime functionality of the constrained fixed-choice fields and so you are not allowed to remove it.

![Figure 4–11 Cascading Fixed-Choice Dependencies Cannot Be Removed](image)
Enforcing Conditional Updateability of Custom Fields for Web Service Access

In this release any conditional updateability expression you provide for a field is enforced only within the web application’s user interface. Depending on the condition you’ve provided — assuming you have correctly configured the Depends On information as described in “Configuring Depends On Fields for Conditional Updateability or Conditional Mandatory” — the field in the user interface will enable or disable as appropriate. However, the same cannot be said for updates to that field performed through the web service interface. Using the example from “Understanding Automatic Dependency Information for Cascading Fixed-Choice Lists”, if the line item’s quantity field is less than or equal to five (5), then the fulfillment warehouse field will be disabled in the user interface. In contrast, if a system attempts to update the same line item using the web service interface, an attempt to update FulfillmentWarehouse will succeed regardless of the value of the Quantity field!

To enforce this conditional updateability through the web service as well, you need to add an object-level validation rule that enforces the same condition. For example, you would configure the following rule to prevent web service updates to the fulfillment warehouse value if the quantity is not greater than five:

**Rule Name**

Warehouse_Updateable_Only_With_Quantity_Over_Five

**Rule Body**

```groovy
// If the fulfillment warehouse is changed, ensure the quantity is greater than five (5)
if (isAttributeChanged('FulfillmentWarehouse_c')) {
  if (Quantity_c <= 5) {
    return false
  }
}
return true
```

**Error Message**

A warehouse can be specified only for quantities over five

Implementing Non-Formula Field Changeable Only From Script

Assume you want to add a LineEditedCount field to the OrderLineItem object to track how many times it has been edited. The field value needs to be stored in the database, so a formula field is not appropriate. Start by adding a custom field of type Number to the object, configuring its default value to be the literal value 0 (zero). Next, configure a conditionally updateable expression for the new LineEditedCount with the expression:

false

This will cause the user interface to always see that the field is not updateable, and the value will only be updateable from script. Note that configuring the conditionally updateable expression to always return false is semantically different from unchecking the Updateable checkbox. Doing the latter, your field would never be updateable (neither from the user interface nor from script). Using the conditionally updateable expression, the updateability enforcement is done only at the user interface level.

Finally, to derive the value of the LineEditedCount you would add the following trigger:
Understanding When Field Default Value Expressions Are Evaluated

A default value expression provides you the ability to dynamically calculate the initial value of a field in a newly-created row. If you configure a default value expression for a field, it is evaluated only when a new row is created. The expression is not evaluated at any other time. If your use case requires the value of a field to change automatically when the value of one or more other fields is changed, see "Deriving Values of a Field When Other Fields Change Value".

Understanding the Difference Between Default Expression and Create Trigger

There are two ways you can assign default values to fields in a newly-created row and it is important to understand the difference between them.

The first way is to provide a default value expression for one or more fields in your object. Your default value expression should not depend on other fields in the same object since you cannot be certain of the order in which the fields are assigned their default values. The default value expression should evaluate to a legal value for the field in question and it should not contain any setAttribute() calls as part of the expression. The framework evaluates your default expression and assigns it to the field to which it is associated automatically at row creation time.

On the other hand, if you need to assign default values to one or more fields after first allowing the framework to assign each field’s literal default values or default value expression, then the second way is more appropriate. Define a Create trigger on the object and inside that trigger you can reference any field in the object as well as perform any setAttribute() calls to assign default values to one or more fields.

Deriving Values of a Field When Other Fields Change Value

There are three different use cases where you might want to derive the value of a field. This section assists you in determining which one is appropriate for your needs.
Deriving the Value of a Formula Field When Other Fields Change Value

If the value of your derived field is calculated based on other fields and its calculated value does not need to be permanently stored, then use a formula field. To derive the value of the formula field, perform these two steps:

1. Configure the formula expression
2. Configure the `Depends On` information to indicate the fields on which your formula expressions depends

Deriving the Value of Non-Formula Field When Other Fields Change Value

If the value of your derived field must be stored, then use one of strategies in this section to derive its value.

Deriving a Non-Formula Field Using a Before Trigger

The simplest way to derive a stored field’s value is to create an appropriate “before” trigger (Before Insert in Database and/or Before Update in Database) which sets the field’s value to your calculated value by calling the `setAttribute()`. See "Assigning a Value to a Field in the Current Object” for more information on this function and “Avoiding Validation Threshold Errors By Conditionally Assigning Values” for a tip about how to avoid your field assignments from causing an object to hit its validation threshold.

Deriving a Non-Formula Field Using an After Field Changed Trigger

If you want the end-user to see the derived field’s value update on the user interface immediately, then you need to perform the assignment of the derived field’s value inside an After Field Changed trigger. When this trigger fires, the value of the field in question has already changed. Therefore, you can simply reference the new value of the field by name instead of using the special `newValue` expression (as would be required in a field-level validation rule to reference the field’s candidate new value that is attempting to be set).

Setting Invalid Fields for the UI in an Object-Level Validation Rule

When a field-level validation rule that you’ve written returns `false`, ADF signals the failed validation with an error and the field is highlighted in red in the user interface to call the problem to the user’s attention. However, since object-level validation rules involve multiple fields, the framework does not know which field to highlight in the user interface as having the problematic value. If you want your object-level validation rule to highlight one or more fields as being in need of user review to resolve the validation error, you need to assist the framework in this process. You do this by adding a call to the `adf.error.addAttribute()` function in your validation rule script before returning `false` to signal the failure. For example, consider the following rule to enforce: A contact cannot be his/her own manager. Since the `Id` field of the `Contact` object cannot be changed, it will make sense to flag the `Manager_Id` field — a secondary field related to the `Manager` lookup field — as the field in error to highlight in the user interface. Here is the example validation rule.

Rule Name

`Contact_Cannot_Be_Own_Manager`
Rule Body
// Rule depends on two fields, so must be
// written as object-level rule
if (Manager_Id_c == Id) {
    // Signal to highlight the Manager field on the UI
    // as being in error. Note that Manager_Id field
    // is not shown in the user interface!
    adf.error.addAttribute('Manager_c')
    return false
}
return true

Error Message
A contact cannot be his/her own manager

Determining the State of a Row

A row of data can be in any one of the following states:

New
A new row that will be inserted into the database during the next save operation.

Unmodified
An existing row that has not been modified

Modified
An existing row where one or more values has been changed and will be updated in
the database during the next save operation

Deleted
An existing row that will be deleted from the database during the next save operation

Dead
A row that was new and got removed before being saved, or a deleted row after it has
been saved

To determine the state of a row in your Groovy scripts, use the function
getPrimaryRowState() and its related helper methods as shown in Example 4–17.

Example 4–17 Making Logic Conditional Based on the Row State

// Only perform this business logic if the row is new
if (getPrimaryRowState().isNew())
{
    // conditional logic here
}

The complete list of helper methods that you can use on the return value of
getPrimaryRowState() is shown below:

isNew()
Returns boolean true if the row state is new, false otherwise.

isUnmodified()
Returns boolean true if the row state is unmodified, false otherwise.

isModified()
Returns boolean true if the row state is modified, false otherwise.
isDeleted()
Returns boolean true if the row state is deleted, false otherwise.

isDead()
Returns boolean true if the row state is dead, false otherwise.

Understanding How Local Variables Hide Object Fields
If you define a local variable whose name is the same as the name of a field in your object, then be aware that this local variable will take precedence over the current object’s field name when evaluated in your script. For example, assuming your object has a custom field named Status_c, then consider the following object validation script:

```groovy
// Assuming current object has a Status_c field
def Status_c = 'Closed'
/*
 * imagine pages full of complex code here
 * /

// If the object's current status is Open, then change it to 'Pending'
// ------------------
// POTENTIAL BUG HERE: The Status_c local variable takes precedence
// ------------------ so the Status_c field value is not used!
//
if (Status_c == 'Open') {
    setAttribute('Status_c','Pending')
}
```

At the top of the example, a variable named Status_c is defined. After pages full of complex code, later in the script the author references the custom field named Status_c without remembering that there is also a local variable named Status_c defined above. Since the local variable named Status_c will always take precedence, the script will never enter into the conditional block here, regardless of the current value of the Status_c field in the current object. As a rule of thumb, use a naming scheme for your local variables to ensure their names never clash with object field names.

Invoking Web Services from Your Scripts
Calling web service methods from your scripts involves two high-level steps:

1. Registering a variable name for the web service you want to invoke
2. Writing Groovy code that prepares the inbound arguments, calls the web service function, and then processes the return value

Using Groovy Maps and Lists with Web Services
When passing and receiving structured data from a web service, a Groovy Map represents an object and its properties. For example, an Employee object with properties named Empno, Ename, Sal, and Hiredate would be represented by a Map object having four key/value pairs, where the names of the properties are the keys.

You can create an empty Map using the syntax:

```groovy
def newEmp = [:]
```

Then, you can add properties to the map using the explicit put() method like this:
newEmp.put("Empno",1234)
newEmp.put("Ename","Sean")
newEmp.put("Sal",9876)
newEmp.put("Hiredate",date(2013,8,11))

Alternatively, and more conveniently, you can assign and/or update map key/value pairs using a simpler direct assignment notation like this:

newEmp.Empno = 1234
newEmp.Ename = "Sean"
newEmp.Sal = 9876
newEmp.Hiredate = date(2013,8,11)

Finally, you can also create a new map and assign some or all of its properties at once using the constructor syntax:

def newEmp = [Empno : 1234,
              Ename : "Sean",
              Sal : 9876,
              Hiredate : date(2013,8,11)]

To create a collection of objects you use the Groovy `List` object. You can create one object at a time and then create an empty list, and call the list's add() method to add both objects to the list:

def dependent1 = [Name : "Dave",
                   BirthYear : 1996]
def dependent2 = [Name : "Jenna",
                   BirthYear : 1999]
def listOfDependents = []
listOfDependents.add(dependent1)
listOfDependents.add(dependent2)

To save a few steps, the last three lines above can be done in a single line by constructing a new list with the two desired elements in one line like this:

def listOfDependents = [dependent1, dependent2]

You can also create the list of maps in a single go using a combination of list constructor syntax and map constructor syntax:

def listOfDependents = [[Name : "Dave",
                         BirthYear : 1996],
                      [Name : "Jenna",
                         BirthYear : 1999]]

If the employee object above had a property named Dependents that was a list of objects representing dependent children, you can assign the property using the same syntax as shown above (using a list of maps as the value assigned):

newEmp.Dependents = [[Name : "Dave",
                     BirthYear : 1996],
                      [Name : "Jenna",
                     BirthYear : 1999]]

Lastly, note that you can also construct a new employee with nested dependents all in one statement by further nesting the constructor syntax:
def newEmp = [Empno : 1234,
    Ename : "Sean",
    Sal : 9876,
    Hiredate : date(2013,8,11),
    Dependents : [
        [Name : "Dave",
            BirthYear : 1996],
        [Name : "Jenna",
            BirthYear : 1999]]
]

For more information on Maps and Lists, see “Working with Maps”.

**Registering a Web Service Variable**

The composer allows you to register a web service for use in your scripts. This capability lets you associate a web service *variable name* with a URL that provides the location of the Web Service Description Language (WSDL) resource that represents the service you want to invoke. For example, you might register a web service variable name of EmployeeService for a web service that your application needs to invoke for working with employee data from another system. This service’s WSDL resource URL might look something like:

http://example.com:8099/Services/EmployeeService?WSDL

Of course, the server name, the port number, and path name for your actual service will be different. If the port number is omitted, then it will assume the service is listening on the default HTTP port number 80.

**Browsing Available Web Service Methods**

When writing your scripts, the Web Services tab in the expression builder shown in Figure 4–12 displays the available web service methods for a given web service. The Web Services drop down list displays the set of registered web service variable names. After you select a particular web service, the Functions list displays the function names that are available to invoke for that service. As shown in the figure, after selecting a particular method, the Function Signature panel lists name of the expected arguments as well as their data types and that of the function's return value. In addition, a code example appears for each parameter that requires a Map value. It illustrates the structure the argument must have, include the map key names and expected data type of the map values for each key. You can copy and paste the variable definition statements into your code to replace the tokens like integerValue, booleanValue, and stringValue with appropriate literal values or expressions as necessary. Finally, click on the (Insert) button to insert the syntax to invoke the web service method.
Calling Web Service Methods in Groovy

To invoke a web service method named `someMethodName()` on a web service that you registered with the variable name `YourServiceVariableName`, use the syntax:

```groovy
adf.webServices.YourServiceVariableName.someMethodName(args)
```

For instance, Example 4–18 shows how to invoke a `getEmployee()` method on a web service registered with the web service variable name `EmployeeService`, passing the integer 7839 as the single argument to the function.

**Example 4–18  Retrieving an Employee by Id**

```groovy
// retrieve Employee object by id from remote system
def emp = adf.webServices.EmployeeService.getEmployee(7839)
// log a message, referencing employee fields with "dot" notation
println('Got employee '+emp.Ename+' with id '+emp.Empno)
// access the nested list of Dependent objects for this employee
def deps = emp.Dependents
if (deps != null) {
    println("Found "+deps.size()+" dependents")
    for (dep in deps) {
        println("Dependent:"+dep.Name)
    }
}
```

The code in Example 4–19 illustrates how to use Groovy's convenient Map and List construction notation to create a new employee with two nested dependents. The `newEmp` object is then passed as the argument to the `createEmployee()` method on the service.
Example 4–19  Creating a new Employee, Including new Dependents

// Create a new employee object using a Groovy map. The
// nested collection of dependents is a Groovy list of maps
def newEmp = [ Ename:"Steve",
              Deptno:10,
              Job:"CLERK",
              Sal:1234,
              Dependents: [[Name:"Timmy",BirthYear:1996],
                            [Name:"Sally",BirthYear:1998]]

// Create the new employee by passing this object to a web service
newEmp = adf.webServices.EmployeeService.createEmployee(newEmp)
// The service returns a new employee object which may have
// other attributes defaulted/assigned by the service, like the Empno
println('New employee created was assigned Empno = '+ newEmp.Empno)

The script in Example 4–20 shows how to use the mergeEmployee() method to update fields in an employee object that is retrieved at the outset via another call to the getEmployee() method. The script updates the Ename field on the emp object retrieved, updates the names of the existing dependents, and then adds a new dependent before calling the mergeEmployee() method on the same service to save the changes.

Example 4–20  Merging Updates to an Employee, Adding a New Child Dependent Object

// Merge updates and inserts on Employee and nested Dependents
def emp = adf.webServices.EmployeeService.getEmployee(7839)
// update employee's name to add an exclamation point!
emp.Ename = emp.Ename + '!

def deps = emp.Dependents
// Update dependent names to add an exclamation point!
for (dep in deps) {
    dep.Name = dep.Name + '!
}
// Add a new dependent
def newChild = [Name:"Jane", BirthYear:1997]
deps.add(newChild)
emp = adf.webServices.EmployeeService.mergeEmployee(emp)

Calling the Find Method on an Oracle Service Interface

When your Groovy script needs to invoke a find method on the service interface for an Oracle Fusion Applications object, you need to pass a structured Map value for its findCriteria parameter. This section provides some simple examples you can use to jumpstart your usage of the find method. Assume that you've already registered a web service variable named EmployeesService for an Oracle Applications web service that provides information about staff members. Furthermore, assume that its standard find method is named findEmployees.

The script in Example 4–21 shows the simplest possible example of creating a correctly-structured findCriteria parameter to pass as the first argument to this findEmployees() method. This example retrieves all employees whose Deptno field is equal to the value 30. The results are returned using the default sorting order. All fields of the employee object are returned to your calling script by default. Note that the carriage returns and other whitespace used in these examples are purely to improve readability and assist with visually matching the square brackets.
Example 4–21 Web Service FindCriteria with a Single View Criteria Item

// Example findCriteria parameter definition
def findCriteria =
[
  filter:
  |
  group:
  |
  |
  item:
  |
  |
  attribute :'Deptno',
  operator :'=',
  value :[[item:30]]
],
|
|
// findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
  // Do something with each 'emp' row
  println(emp)
}

The script in Example 4–22 expands on the previous one by adding two additional view criteria items to further constrain the search. To accomplish this, we added two additional maps to the comma-separated list-of-maps value provided for the item map entry in the group list-of-maps entry of the filter map entry. The default conjunction between multiple view criteria items in the same view criteria row is AND, so this example finds all employees whose Deptno field is equal to 30, and whose commission field named Comm is greater than 300, and whose Job field starts with the value sales (using a case-insensitive comparison).

Example 4–22 Using Web Service FindCriteria with Multiple View Criteria Items

// Example findCriteria parameter definition
def findCriteria =
[
  filter:
  |
  group:
  |
  |
  item:
  |
  |
  attribute :'Deptno',
  operator :'=',
  value :[[item:30]]
],
|
|

value :[[item:300]]
],
[
  upperCaseCompare :true,
  attribute :'Job',
  operator :'STARTSWITH',
  value :[[item:'sales']]
]}
]
// findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
  println(emp)
}

The script in Example 4–23 extends the example above to add a second view criteria row to the filter. To accomplish this, we added an additional map to the comma-separated list-of-maps value provided for the group list-of-maps entry of the filter map entry. The default conjunction between separate view criteria rows in a view criteria filter is OR, so the filter in this example finds all employees matching the previous criteria from Example 4–22, or any employee whose Ename field equals allen. The upperCaseCompare : true entry ensures that a case-insensitive comparison is performed. For more information on the valid values you can pass for the operator entry for the view criteria item, see "Understanding View Criteria Item Operators".

Example 4–23 Using Web Service FindCriteria with Multiple View Criteria Rows

// Example findCriteria parameter definition
def findCriteria =
[
  filter:
  [  
    group:
    [ 
      item:
      [  
        attribute :'Deptno',
        operator :'=',
        value :[[item:30]]
      ],
      [ 
        attribute :'Comm',
        operator :'>',
        value :[[item:300]]
      ],
      [ 
        upperCaseCompare :true,
        attribute :'Job',
        operator :'STARTSWITH',
        value :[[item:'sales']]
      ]
    ]
  ]
]
The script in Example 4–24 enhances the original Example 4–21 to explicitly specify a single field sort order. The results will be sorted ascending by the value of their Ename field. Since the value of the sortAttribute entry is a list of maps, you could add additional maps separated a commas to perform a sort on multiple fields.

**Example 4–24 Using Web Service FindCriteria to Control Sort Order**

```groovy
// Example findCriteria parameter definition
def findCriteria =
[
  filter:
  [
    group:
    [
      item:
      [
        [attribute :'Deptno',
          operator :'=',
          value :[[item:30]]
        ]
      ]
    ]
  ],
  sortOrder:
  [sortAttribute:
   [
     name :'Ename',
     descending : false
   ]
  ]
]
// findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
  println(emp)
}
```

// Example 4–24 Using Web Service FindCriteria to Control Sort Order
// findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
    println(emp)
}

The script in Example 4–25 extends Example 4–24 to add a specific find attribute criteria in order to request that only a subset of employee object fields should be returned in the result. In this example, for each employee object in department 30, only its Empno and Ename field values will be returned to the calling groovy script.

**Example 4–25 Using Web Service FindCriteria to Return Only a Subset of Object Field Values**

// Example findCriteria parameter definition
def findCriteria =
[
    filter:
    [
        group:
        [
            item:
            [
                [attribute : 'Deptno',
                 operator : '=',
                 value : [[item : 30]]
            ]
        ]
    ],
    sortOrder:
    [
        sortAttribute:
        [
            [name : 'Ename',
             descending : false]
        ]
    ],
    findAttribute:
    [
        [item : 'Empno'],
        [item : 'Ename']
    ]
]
// findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
    println(emp)
}
The script in Example 4–26 shows how to use the `fetchSize` map entry to limit the number of rows returned to only the first 3 rows that match the supplied criteria in the requested sort order, including only the requested field values in the result. This example returns the `EmpNo`, `Ename`, and `Sal` fields of the top 3 employees whose `Job` fields equals `CLERK` (performed case-sensitively this time), ordered descending by `Sal`.

**Example 4–26 Using Web Service FindCriteria to Limit Number of Rows Returned**

```groovy
// Example findCriteria parameter definition
def findCriteria = [
    fetchSize : 3,
    filter: [
        group: [
            item: [
                attribute : 'Job',
                operator : '=',
                value : [[item: 'CLERK']]
            ]
        ],
    ],
    sortOrder: [
        sortAttribute: [
            name : 'Sal',
            descending : true
        ],
    ],
    findAttribute: [
        [item : 'Empno'],
        [item : 'Ename'],
        [item : 'Sal']
    ]
] // findControl needs to be an empty list rather than null
def findControl = []
def emps = adf.webServices.EmployeesService.findEmployees(findCriteria, findControl)
for (emp in emps) {
    println(emp)
}
```

**Accessing the Display Value of the Selected Item(s) in a List Field**

When your object contains a single-selection fixed-choice list field named `SomeListField`, you can use the `getSelectedListDisplayValue()` function to access the description that corresponds to the current field's code value. The function takes a
single argument that is the name of the list field for whose selected value you want the
description. For example, consider a single-selection fixed-choice field called
RequestStatus. You could write conditional logic based on the selected status'
description string using code like this:

```groovy
def meaning = getSelectedListDisplayValue('RequestStatus')
if (meaning.contains("Emergency")) {
    // do something here when description contains the string "Emergency"
}
```

If your object contains a multiple-selection fixed-choice field named
MultiChoiceField, you can use the getSelectedListDisplayValues() function to
access a list of the descriptions that correspond to the current field’s one or more
selected code values. As above, the function takes a single argument that is the name
of the multiple-selection fixed-choice list for whose selected values you want the
descriptions. For example, consider a custom multiple-selection fixed-choice list
named Category_c. You could write conditional logic based on the selected status'
description string using code like this:

```groovy
// return true if any of the selected meanings contains the
// string 'wood' case-insensitively (using Groovy regular expressions)
def meaningsList = getSelectedListDisplayValues('Category_c')
for (meaning in meaningsList) {
    // if current meaning contains *wood* case-insensitively
    if (meaning =~ /(?i)wood/) {
        return true
    }
    return false
}
```

If you call the multi-selection list function getSelectedListDisplayValues() on a
single-selection list field, it returns a list containing a single value. If you call the
single-selection list function getSelectedListDisplayValue() on a multi-selection list
field, it returns the String value of the first of the multiple choices selected. If you call
either of these functions on a list field whose value is null or whose value is not a
valid choice for the list, then the function returns null.
Supported Groovy Classes and Methods

This chapter contains information about the Groovy supported classes and methods. This chapter contains the following:

- "Overview"

Overview

When writing Groovy scripts, you may only use the classes and methods that are documented in Table A–1. Using any other class or method may work initially, but will throw a runtime exception when you migrate your code to later versions. Therefore, we strongly suggest that you ensure the Groovy code you write adheres to the classes and methods shown here. For each class, in addition to the method names listed in the table, the following method names are also allowed:

- equals()
- hashCode()
- toString()

In contrast, the following methods are never allowed on any object:

- finalize()
- getClass()
- getMetaClass()
- notify()
- notifyAll()
- wait()

Table A–1  Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADFContext</td>
<td></td>
<td>oracle.adf.share</td>
</tr>
<tr>
<td></td>
<td>getLocale()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSecurityContext()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getUserRoles()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isInRole()</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td></td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>Any constructor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>■ getArray()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>■ getElemType()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getList()</td>
<td></td>
</tr>
<tr>
<td>ArrayList</td>
<td>■ <em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ <em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>Arrays</td>
<td>■ <em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ <em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>AttributeDef</td>
<td>■ getAttributeKind()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getIndex()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getJavaType()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getPrecision()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getProperty()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getScale()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getUIHelper()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getUpdateableFlag()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ isMandatory()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ isQueriable()</td>
<td></td>
</tr>
<tr>
<td>AttributeHints</td>
<td>■ getControlType()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getDisplayHeight()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getDisplayHint()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getDisplayWidth()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getFormat()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getFormattedAttribute()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getFormatter()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getFormatterClassName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ get Hint()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getLocaleName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ parseFormattedAttribute()</td>
<td></td>
</tr>
<tr>
<td>AttributeList</td>
<td>■ getAttribute()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getAttributeIndexOf()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getAttributeNames()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setAttribute()</td>
<td></td>
</tr>
<tr>
<td>BaseLobDomain</td>
<td>■ closeCharacterStream()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>■ closeInputStream()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ closeOutputStream()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getInputStream()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getLength()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getOutputStream()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getcharacterStream()</td>
<td></td>
</tr>
<tr>
<td>Class Name</td>
<td>Allowed Methods</td>
<td>Package</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>BigDecimal</td>
<td>■ Any constructor ■ Any method</td>
<td>java.math</td>
</tr>
<tr>
<td>BigInteger</td>
<td>■ Any constructor ■ Any method</td>
<td>java.math</td>
</tr>
<tr>
<td>BitSet</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Blob</td>
<td>■ Any constructor ■ Any method</td>
<td>java.sql</td>
</tr>
<tr>
<td>BlobDomain</td>
<td>■ Any constructor ■ getBinaryOutputStream() ■ getBinaryStream() ■ getBufferSize()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td>Boolean</td>
<td>■ Any constructor ■ Any method</td>
<td>java.lang</td>
</tr>
<tr>
<td>Byte</td>
<td>■ Any constructor ■ Any method</td>
<td>java.lang</td>
</tr>
<tr>
<td>Calendar</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Char</td>
<td>■ Any constructor ■ bigDecimalValue() ■ bigIntegerValue() ■ booleanValue() ■ doubleValue() ■ floatValue() ■ getValue() ■ intValue() ■ longValue()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td>Clob</td>
<td>■ Any constructor ■ Any method</td>
<td>java.sql</td>
</tr>
<tr>
<td>BlobDomain</td>
<td>■ Any constructor</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td>Collection</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Collections</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Comparator</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Currency</td>
<td>■ Any constructor ■ Any method</td>
<td>java.util</td>
</tr>
<tr>
<td>Class Name</td>
<td>Allowed Methods</td>
<td>Package</td>
</tr>
<tr>
<td>-------------</td>
<td>------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>DBSequence</td>
<td>■ Any constructor</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>■ getValue()</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>■ Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Date</td>
<td>■ Any constructor</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>■ compareTo()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ dateValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ stringValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ timeval()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ timestampValue()</td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Double</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Enum</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>EnumMap</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>EnumSet</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Enumeration</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>EventListener</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>EventListenerProxy</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>EventObject</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Exception</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>ExprValueErrorHandler</td>
<td>■ addAttribute()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ clearAttributes()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ raise()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ raiseLater()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ warn()</td>
<td></td>
</tr>
<tr>
<td>Float</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1  (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formattable</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>FormattableFlags</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Formatter</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>GregorianCalendar</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>HashMap</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>HashSet</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Hashtable</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>IdentityHashMap</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td>Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Iterator</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>JboException</td>
<td>getDetails()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>getErrorCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getErrorParameters()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLocalizedMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProductCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProperty()</td>
<td></td>
</tr>
<tr>
<td>JboWarning</td>
<td>Any constructor</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>getDetails()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getErrorCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getErrorParameters()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLocalizedMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProductCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProperty()</td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>toStringFormat()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td>LinkedHashMap</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>LinkedHashSet</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>ListIterator</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>ListResourceBundle</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>Locale</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>Long</td>
<td>• Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>Map</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>Math</td>
<td>• Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>MathContext</td>
<td>• Any constructor</td>
<td>java.math</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>NClob</td>
<td>• Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>NameValuePairs</td>
<td>• Any constructor</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>• getAttribute()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• getAttributeIndexOf()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• getAttributeNames()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• setAttribute()</td>
<td></td>
</tr>
<tr>
<td>NativeTypeDomainInterface</td>
<td>• getNativeObject()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td>Number</td>
<td>• Any constructor</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>• bigDecimalValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• bigIntegerValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• booleanValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• byteValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• doubleValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• floatValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• getValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• intValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• longValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• shortValue()</td>
<td></td>
</tr>
<tr>
<td>Observable</td>
<td>• Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>• Any method</td>
<td></td>
</tr>
<tr>
<td>Class Name</td>
<td>Allowed Methods</td>
<td>Package</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------</td>
<td>---------------</td>
</tr>
<tr>
<td>Observer</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>PriorityQueue</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Properties</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>PropertyPermission</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>PropertyResourceBundle</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Queue</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Random</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>RandomAccess</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Ref</td>
<td>Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>ResourceBundle</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Row</td>
<td>getAttribute</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>getAttributeHints</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getKey</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLookupDescription</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getOriginalAttributeValue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getPrimaryRowState</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSelectedListDisplayValue</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getSelectedListDisplayValues</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isAttributeChanged</td>
<td></td>
</tr>
<tr>
<td></td>
<td>isAttributeUpdateable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>remove</td>
<td></td>
</tr>
<tr>
<td></td>
<td>revertRow</td>
<td></td>
</tr>
<tr>
<td></td>
<td>revertRowAndContainees</td>
<td></td>
</tr>
<tr>
<td></td>
<td>setAttribute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>validate</td>
<td></td>
</tr>
<tr>
<td>RowId</td>
<td>Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>Class Name</td>
<td>Allowed Methods</td>
<td>Package</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------------------------------</td>
<td>------------------------------</td>
</tr>
<tr>
<td>RowIterator</td>
<td>createAndInitRow()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>createRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findByKey()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findRowsMatchingCriteria()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>first()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAllRowsInRange()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCurrentRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getEstimatedRowCount()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hasNext()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hasPrevious()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insertRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>last()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>next()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>previous()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset()</td>
<td></td>
</tr>
<tr>
<td>RowSet</td>
<td>avg()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>count()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>createAndInitRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>createRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>executeQuery()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findByKey()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>findRowsMatchingCriteria()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>first()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getAllRowsInRange()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getCurrentRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getEstimatedRowCount()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hasNext()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>hasPrevious()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>insertRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>last()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>min()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>next()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>previous()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>reset()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sum()</td>
<td></td>
</tr>
<tr>
<td>Scanner</td>
<td>Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>Any method</td>
<td></td>
</tr>
<tr>
<td>SecurityContext</td>
<td>getUserName()</td>
<td>oracle.adf.share.security</td>
</tr>
<tr>
<td></td>
<td>getUserProfile()</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session</td>
<td>■ getLocale()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getLocaleContext()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getUserData()</td>
<td></td>
</tr>
<tr>
<td>Set</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Short</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>SimpleTimeZone</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>SortedMap</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>SortedSet</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Stack</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>StackTraceElement</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>StrictMath</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>String</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>StringBuffer</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>StringBuilder</td>
<td>■ Any constructor</td>
<td>java.lang</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>StringTokenizer</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Struct</td>
<td>■ Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Struct</td>
<td>■ getAttribute()</td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>■ setAttribute()</td>
<td></td>
</tr>
<tr>
<td>StructureDef</td>
<td>■ findAttributeDef()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getAttributeIndexOf()</td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td>■ Any constructor</td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>TimeZone</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>Class Name</td>
<td>Allowed Methods</td>
<td>Package</td>
</tr>
<tr>
<td>------------</td>
<td>-----------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Timer</td>
<td><em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>TimerTask</td>
<td><em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td><em>Any constructor</em></td>
<td>java.sql</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>Timestamp</td>
<td><em>Any constructor</em></td>
<td>oracle.jbo.domain</td>
</tr>
<tr>
<td></td>
<td>compareTo()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dateValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stringValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timeValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>timestampValue()</td>
<td></td>
</tr>
<tr>
<td>TreeMap</td>
<td><em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>TreeSet</td>
<td><em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
<tr>
<td>UUID</td>
<td><em>Any constructor</em></td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td><em>Any method</em></td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserProfile</td>
<td>getBusinessCity()</td>
<td>oracle.adf.share.security.identitymanagement</td>
</tr>
<tr>
<td></td>
<td>getBusinessCountry()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessEmail()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessFax()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessMobile()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessPOBox()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessPager()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessPhone()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessPostalAddr()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessPostalCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessState()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getBusinessStreet()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDateofBirth()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDateofHire()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDefaultGroup()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDepartment()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDepartmentNumber()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDescription()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getDisplayName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getEmployeeNumber()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getEmployeeType()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getFirstName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getGUID()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getGivenName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getHomeAddress()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getHomePhone()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getInitials()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getJpegPhoto()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getLastName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getMaidenName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getManager()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getMiddleName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getNameSuffix()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getOrganization()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getOrganizationalUnit()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getPreferredLanguage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getPrinciple()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProperties()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getProperty()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTimeZone()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getTitle()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getUIAccessMode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getUniqueName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getUserID()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getName()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>getUserName()</td>
<td></td>
</tr>
</tbody>
</table>
### Table A–1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>ValidationException</td>
<td>■ getDetails()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getErrorCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getErrorParameters()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getLocalizedMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getMessage()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getProductCode()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getPropertyName()</td>
<td></td>
</tr>
<tr>
<td>Vector</td>
<td>■ Any constructor</td>
<td>java.util</td>
</tr>
<tr>
<td></td>
<td>■ Any method</td>
<td></td>
</tr>
<tr>
<td>ViewCriteria</td>
<td>■ createAndInitRow()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ createRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ createViewCriteriaRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ findByKey()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ findRowsMatchingCriteria()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ first()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getAllRowsInRange()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getCurrentRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ getEstimatedRowCount()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ hasNext()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ hasPrevious()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ insertRow()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ last()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ next()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ previous()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ reset()</td>
<td></td>
</tr>
<tr>
<td>ViewCriteriaItem</td>
<td>■ getValue()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ makeCompound()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setOperator()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setUpperColumns()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setValue()</td>
<td></td>
</tr>
<tr>
<td>ViewCriteriaItemCompound</td>
<td>■ ensureItem()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getValue()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ makeCompound()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setOperator()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setUpperColumns()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setValue()</td>
<td></td>
</tr>
<tr>
<td>ViewCriteriaRow</td>
<td>■ ensureCriteriaItem()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td></td>
<td>■ getConjunction()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ isUpperColumns()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setConjunction()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>■ setUpperColumns()</td>
<td></td>
</tr>
</tbody>
</table>
### Table A-1 (Cont.) Classes/Methods Allowed for Groovy Scripts

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Allowed Methods</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>ViewObject</td>
<td>appendViewCriteria() avg() count() createAndInitRow() createRow() createViewCriteria() executeQuery() findFirst() findRowsMatchingCriteria() first() getAllRowsInRange() getCurrentRow() getMaxFetchSize() hasNext() hasPrevious() insertRow() last() max() min() next() previous() reset() setMaxFetchSize() setSortBy() sum()</td>
<td>oracle.jbo</td>
</tr>
<tr>
<td>WeakHashMap</td>
<td>Any constructor Any method</td>
<td>java.util</td>
</tr>
</tbody>
</table>
Understanding Common JBO Exceptions in Groovy Scripts

This chapter provides background information on ADF exceptions that might occur while your Groovy scripts are executing and attempts to explain the most common causes.

This chapter contains the following:

■ "JBO-25030: Detail entity X with row key Y cannot find or invalidate its owning entity"
■ "JBO-26020: Attempting to insert row with no matching EO base"

JBO-25030: Detail entity X with row key Y cannot find or invalidate its owning entity

Problem Description
You tried to create a new child object row of type \texttt{X} without providing the necessary context information to identify its owning parent object. At the moment of child row creation the correct owning parent context must be provided, otherwise the new child row created would be an "orphan".

For example, consider a custom object named \texttt{TroubleTicket} that has a child object named \texttt{Activity}. The following script that tries to create a new activity would generate this error:

```groovy
def activityVO = newView('Activity_c')
// PROBLEM: Attempting to create a new child activity row
// ------- without providing context about which owning
// TroublemTicket row this activity belongs to.
def newActivity = activityVO.createRow()
```

This generates a \texttt{JBO-25030: Detail entity Activity\_c with row key null cannot find or invalidate its owning entity} exception because the script is trying to create a new Activity row without providing the context that allows that new row to know which TroubleTicket it should belong to.

Resolution
There are two ways to provide the appropriate parent object context when creating a new child object row. The first approach is to get the owning parent row to which you want to add a new child row and use the parent row’s child collection attribute to perform the \texttt{createRow()} and \texttt{insertRow()} combination. For example, to create a new Activity row in the context of \texttt{TroubleTicket} with an \texttt{Id} of 10000000272002 you can do the following, using the helper function mentioned in Example 4-11. When you use...
this approach, the parent object context is implicit since you’re performing the action on the parent row’s child collection.

```groovy
def idParent = 100000000272002
def ttVO = newView('TroubleTicket_c')
def parent = adf.util.findRowByKey(ttVO,idParent)
if (parent != null) {
    // Access the collection of Activity child rows for
    // this TroubleTicket parent object
    def activities = parent.ActivityCollection_c
    // Use this child collection to create/insert the new row
    def newActivity = activities.createRow()
    activities.insertRow(newActivity);
    // Set other field values of the new activity here...
}
```

The second approach you can use is to pass the context that identifies the id of the parent TroubleTicket row when you create the child row. You do that using an alternative function named `createAndInitRow()` as shown below. In this case, you don’t need to have the parent row in hand or even use the parent TroubleTicket view object. Providing the id to the owning parent row at the moment of child activity row creation is good enough.

```groovy
def idParent = 100000000272002
// Create an name/value pairs object to pass the parent id
def parentAttrs = new NameValuePairs()
parentAttrs.setAttribute("Id",idParent)
// Use this name/value pairs object to pass the parent
// context information while creating the new child row
def activityVO = newView('Activity_c')
def newActivity = activityVO.createAndInitRow(parentAttrs)
activityVO.insertRow(newActivity);
// Set other field values of the new activity here...
```

---

**Problem Description**

You inadvertently added a row that was created or queried from one view object to another view object of a different type. For example, the following script would generate this error:

```groovy
def empVO = newView('Employees')
def newEmp = empVO.createRow()
def deptVO = newView('Department')
// PROBLEM: Incorrectly adding a row of type "Employees"
// ------- to the view of type "Department"
depthVO.insertRow(newEmp)
```

This generates a JBO-26020: Attempting to insert row with no matching EO base exception because the script is trying to insert a row from "Employees" view into a view that is expecting rows of type "Department". This leads to a type mismatch that is not supported.

**Resolution**

Ensure that when you call `insertRow()` on a view object that the row you are trying to insert into the collection is of the correct view type.