

Oracle Fusion Cloud Applications

Groovy Scripting Reference

26D



Contents

Get Help

i

1 Introduction 1

Terminology	1
Where You'll Use Groovy in Your Application	2
Ensuring Your Scripts Are Easy to Maintain	2

2 Groovy Basics 5

Commenting Your Scripts	5
Defining Variables	6
Referencing the Value of a Field in the Current Object	7
Working with Numbers, Dates, and Strings	7
Using Substitution Expressions in Strings	8
Using Conditional Expressions	9
Using the Switch Statement	10
Returning a Boolean Result	11
Assigning a Value to a Field in the Current Object	11
Writing Null-Aware Expressions	12
Understanding the Difference Between Null and Empty String	12
Understanding Secondary Fields Related to a Lookup	13
Using Groovy's Safe Navigation Operator	14
Assigning a Value to a Field in a Related Object	14
Printing and Viewing Diagnostic Messages	15
Working with Lists	16
Working with Maps	18
Working with Ranges	19

3 Examples of Each Context Where You Can Use Groovy 21

Providing an Expression to Calculate a Custom Formula Field's Value	21
Providing an Expression to Calculate a Custom Field's Default Value	21
Providing an Expression to Make a Custom Field Conditionally Updateable	22

Providing an Expression to Make a Custom Field Conditionally Required	22
Defining a Field-Level Validation Rule	23
Defining an Object-Level Validation Rule	24
Defining Utility Code in a Global Function	24
Defining Reusable Behavior with an Object Function	25
Controlling the Visibility of an Object Function	26
Defining an Object-Level Trigger to Complement Default Processing	26
Defining a Field-Level Trigger to React to Value Changes	28

4 Groovy Tips and Techniques 29

Simplifying Code Authoring with the Expression Palette	29
Using the Related Object Accessor Field to Work with a Parent Object	34
Using the Related Object Accessor Field to Work with a Referenced Object	35
Using the Related Collection Accessor Field to Work with Child Rows	35
Accessing Current Date and Time from the Application Server	37
Accessing Current Date and Time from the Database	37
Understanding ADF's Additional Built-in Groovy Functions	38
Testing Whether a Field's Value Is Changed	43
Avoiding Validation Threshold Errors By Conditionally Assigning Values	43
Prefer "Before" Save-time Triggers to "After" Ones for Best Performance	44
Detecting Row State in After Changes Posted to Database Trigger	44
Avoiding Posting Threshold Errors By Conditionally Assigning Values	45
Functional Restrictions in Trigger Scripts	45
Passing the Current Object to a Global Function	46
Referencing Original Values of Changed Fields	46
Raising a Warning From a Validation Rule Instead of an Error	46
Throwing a Custom Validation Exception	47
Returning Locale-Sensitive Custom Strings	47
Raising a Trigger's Optional Declaratively-Configured Error Message	48
Accessing the View Object for Programmatic Access to Business Objects	48
Defining the Sort Order for Query Results	50
Finding an Object by Id	52
Finding Objects Using a View Criteria	53
Accomplishing More with Less Code	60
Creating a New Object	77
Updating an Existing Object	78
Permanently Removing an Existing Object	78

Reverting Changes in a Single Row	79
Understanding Why Using Commit or Rollback In Scripts Is Strongly Discouraged	79
Using the User Data Map	79
Referencing Information About the Current User	80
Using Aggregate Functions	80
Understanding When to Configure Field Dependencies	83
Enforcing Conditional Updateability of Custom Fields for Web Service Access	85
Implementing Non-Formula Field Changeable Only From Script	86
Understanding When Field Default Value Expressions Are Evaluated	86
Understanding the Difference Between Default Expression and Create Trigger	87
Deriving Values of a Field When Other Fields Change Value	87
Setting Invalid Fields for the UI in an Object-Level Validation Rule	88
Determining the State of a Row	89
Understanding How Local Variables Hide Object Fields	90
Invoking Web Services from Your Scripts	90
Accessing the Display Value of the Selected Item(s) in a List Field	100
Formatting Numbers and Dates Using a Formatter	101
Working with Field Values Using a Parameterized Name	102
Determining the Object Type of a Row	104
5 Best Practices for Groovy Performance	107
Search Using at Least One Indexed Field	107
Explicitly Select Only the Attributes You Need	107
Test for Existence by Retrieving a Single Row	108
Avoid Using <code>newView()</code> Inside a Loop	109
Set Field Values in Bulk	111
Avoid Revalidating Known Valid Data	112
Use Left Shift Operator To Append to Lists	113
6 Understanding Common JBO Exceptions in Groovy Scripts	115
JBO-25030: Detail entity X with row key Y cannot find or invalidate its owning entity	115
JBO-26020: Attempting to insert row with no matching EO base	116
7 Supported Classes and Methods for Use in Groovy	117
Supported Classes and Methods for Use in Groovy Scripts	117

Get Help

In addition to the resources available in Oracle Help Center, you can explore training, connect with the Oracle community, and share your feedback.

Get Training

Increase your knowledge of Oracle Cloud by taking courses at [Oracle University](#).

Join Our Community

Use [Cloud Customer Connect](#) to get information from industry experts at Oracle and in the partner community. You can join forums to connect with other customers, post questions, suggest [ideas](#) for product enhancements, and watch events.

Share Your Feedback

We welcome your feedback about Oracle Applications user assistance. If you need clarification, find an error, or just want to tell us what you found helpful, we'd like to hear from you.

You can email your feedback to oracle_fusion_applications_help_ww_grp@oracle.com.

Thanks for helping us improve our user assistance!

1 Introduction

Groovy is a standard, dynamic scripting language for the Java platform for which the 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: *Supported Classes and Methods for Use in Groovy Scripts* 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.

Terminology

Throughout the document the term script is used to describe one or more lines of Groovy code that your application using Oracle business objects 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 `return` keyword since the `return` keyword is always implied on the last line of a script:

```
CommissionPercentage < 0.40
```

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:

```
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 updatable, 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.

Ensuring Your Scripts Are Easy to Maintain

When writing a script, your first instinct is to get your business logic working correctly. Over time, as you iteratively add functionality, the script for a complex business process can grow very long.

However, Oracle recommends limiting each script to 400 lines. Since one script can invoke other functions, in practice this restriction does not hamper your ability to solve business problems. It is always possible to decompose a lengthy script into a shorter alternative that invokes other object functions or global functions as needed.

For example, instead of writing a 600-line trigger script, create a shorter trigger that invokes other object functions or global functions. In turn, if one of these functions starts getting long, reorganize its code into additional, smaller functions that the original function can invoke. For each function you create, choose a meaningful name that describes the task it performs.

Suppose you have written a “Before Insert” trigger that executes before each new `PurchaseOrder` object is created. Imagine it contains a large amount of code that conditionally creates a default set of `LineItem` child objects for new orders. To avoid exceeding the 400-line limit and improve readability of your code, reorganize it into two object functions named `requiresDefaultLineItems()` and `createDefaultLineItems()`. Then rewrite the original trigger to be:

```
// Before Insert Trigger on PurchaseOrder
if (requiresDefaultLineItems()) {
    createDefaultLineItems()
}
```

By following these recommendations, your code will avoid “*Code too large!*” errors and will become much easier for colleagues to understand and maintain as well.

2 Groovy Basics

This section highlights some important aspects of Groovy to allow you to better understand the examples in the sections that follow.

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:

```
// 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
   */
  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, your typical script would look like this:

```
/*
 * Object validation rule for BankAccount
 *
 * Ensures that account is not overdrawn
 */
def balance = CurrentBalance
// 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 your variable needs to hold a business object, you must to define the variable's type using the `def` keyword. See the tip in [Using Substitution Expressions in Strings](#) below for more information.

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 updatable 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:

```
// 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:

```
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 Substitution Expressions in Strings](#) for more information.

Working with Numbers, Dates, and Strings

Groovy makes it easy to work with numbers, dates and strings. The expression for a literal number is just the number itself:

```
// Default discount is 5%
```

```
def defaultDiscount = 0.05
// Assume 31 days in a month
def daysInMonth = 31
```

To create a literal date, use the `date()` or `dateTime()` function:

```
// Start by considering January 31st, 2019
def lastDayOfJan = date(2019,1,31)
// Tax forms are late after 15-APR-2019 23:59:59
def taxSubmissionDeadline = dateTime(2019,4,15,23,59,59)
```

Write a literal string using a matching pair of single quotes, as shown here.

```
// Direct users to the Acme support twitter account
def supportTwitterHandle = '@acmesupport'
```

It is fine if the string value contains double-quotes, as in:

```
// Default podcast signoff
def salutation = 'As we always say, "Animate from the heart."'
```

However, if your string value contains *single* quotes, then use a matching pair of *double*-quotes to surround the value like this:

```
// Find only gold customers with credit score over 750
customers.appendViewCriteria("status = 'Gold' and creditScore > 750")
```

You can use the normal `+` and `-` operators to do date, number, and string arithmetic like this:

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

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

// Increase an employee's Salary field value by 100 dollars
Salary = Salary + 100

// Decrement an salesman's commission field value by 100 dollars
Commission = Commission - 100

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

// 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:

```
// 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:

```
// 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` 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:

```
def textFragment = 'We will try to call you at ' + ContactPhoneNumber
if (ContactTwitterName != null) {
    textFragment += ', or send you a tweet at '+ContactTwitterName
}
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` 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:

```
// 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, '.gov')) {
    emailToUse = AlternateEmailAddress
}
else {
    emailToUse = EmailAddress
}
```

Using Groovy's handy inline `if / then / else` operator, you can write the same code in a lot fewer lines:

```
def emailToUse = endsWith(EmailAddress, '.gov') ? AlternateEmailAddress : EmailAddress
```

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

```
BooleanExpression ? If_True_Use_This_Expression : If_False_Use_This_Expression
```

Since you can use white space to format your code to make it more readable, consider wrapping inline conditional expressions like this:

```
def emailToUse = endsWith(EmailAddress, '.gov')
? AlternateEmailAddress
: EmailAddress
```

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.

```
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,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'
}
println(logMsg+' ['+maxDiscount+']')
// return expression that will be true when rule is valid
return Discount == null || Discount <= 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 updatable 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:

```
// 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:

```
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:

```
Commission_c > 1000
```

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

Assigning a Value to a Field in the Current Object

To assign the value of a field, use the Groovy assignment operator `=` and to compare expressions for equality, use the double-equals operator `==` as follows:

```
// Compare the ContactTwitterName field's value to the constant string 'steve'
if (ContactTwitterName == 'steve') {
    // Assign a new value to the ContactTwitterName field
    ContactTwitterName = 'stefano'
}
```

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 `nv1()` 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 = nv1(PostedDate,now()) + 3

// Increase an employee's custom Salary field value by 10 percent
// Use zero if current Salary is null
Salary = nv1(Salary,0) * 1.1
```

Tip: Both expressions you pass to the `nv1()` function must have the same datatype, or you will see type-checking warnings when saving your code. For example, if `salary` is a number field, then it is incorrect to use an expression like `nv1(Salary,'<No Salary>')` because the first expression is a number while the second expression is a string.

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 equal the value of `varB` (the empty string).

```
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:

```
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:

```
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:

```
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 dynamic choice list (DCL) 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 DCL field named `contact` that represents a foreign key reference to the specific `Contact` object that reported the trouble ticket.

When defining a DCL field, you specify a primary display field name from the reference object. For example, while defining the `contact` dynamic choice list field referencing the `Contact` object, you might specify the *Contact Name* field.

When you define a DCL field like `contact`, you get one primary field and two secondary fields:

- **The DCL Display 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 DCL 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 DCL 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 additional fields besides the display name from the related object, you can use the related object accessor field like this:

```
// Assume script runs in context of TroubleTicket object
def contactEmail = Contact_Obj_c?.EmailAddress_c
```

If you reference multiple fields from the related object, you can save the related object in a variable and then reference multiple fields using this object variable:

```
// Assume script runs in context of TroubleTicket object
def contact = Contact_Obj_c
def email = contact?.EmailAddress_c
def linkedIn = contact?.LinkedInUsername_c
```

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, then use this approach:

```
// Assume script runs in context of TroubleTicket object
def newId = /* ... Get the Id of the New Contact Here */
Contact_Id_c = newId
```

If you know the value of the contact's primary display field, then use this approach instead:

```
// Assume script runs in context of TroubleTicket object
Contact_c = 'James Smith'
```

Tip: If the value your script assigns to a dynamic choice list field (e.g. `Contact_c`) uniquely identifies a referenced object, then the corresponding value of the foreign keyfield (e.g. `Contact_Id_c`) will automatically update to reflect the primary key of the matching object.

Note: If the value your script assigns to a DCL field does not uniquely identify a referenced object, then the assignment is *ignored*.

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:

```
// 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 Dynamic Choice List Field](#)

Assigning a Value to a Field in a Related Object

To assign a value to a field in a related object, use the assignment operator like this:

```
// Assume script runs in context of an Activity_c object (child of TroubleTicket_c)
// and that TroubleTicket is the name of the parent accessor
TroubleTicket.Status_c = 'Open'
```

Since a child object must be owned by some parent row, you can assume that the `TroubleTicket` accessor will always return a valid parent row instead of ever returning null. This is a direct consequence of the fact that the parent foreign

key value in the `Activity` object's `TroubleTicket_Id` field is mandatory. In this situation, it is not *strictly* necessary to use the Groovy safe-navigation operator, but in practice is it always best to use it:

```
TroubleTicket?.Status_c = 'Open'
```

By following this advice, you can be certain your code will never fail with a `NullPointerException` error when you happen to work with an accessor to a related object that is optional. For example, suppose you added a custom dynamic choice field named `SecondaryAssignee_c` to the `Activity` object and that its value is optional. This means that referencing the related accessor field named `SecondaryAssignee_Obj_c` can return null if the related foreign key field `SecondaryAssignee_Id_c` is null. In this case, it is imperative that you use the safe-navigation operator so that the attempted assignment is ignored when there is no secondary assignee for the current activity.

```
SecondaryAssignee_Obj_c?.OpenCases_c = caseTotal
```

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 assist with debugging, use the diagnostic log to view the runtime diagnostic messages your scripts have written to the log, as well as any exception stack traces that coding errors have produced at runtime.

On entering the diagnostic log viewer, if the *Enable Application Script Logging* check box is checked, then all of the messages from your current session are shown. The diagnostic console toolbar reflects this by showing that you are viewing all messages *Since Session Started*. As shown in the figure below, the messages are shown by default in chronological order and the display is scrolled to view the most recent messages. When an exception has occurred, additional details on the call stack where the error occurred are visible in the *Message Details* panel at the right.

Enable Application Script Logging

View Format Since Session Started Refresh Clear Export All

Messages	Timestamp
Validating that P1/P2 ticket must be assigned	Sep 12, 2015 07:58:12 PM PDT
Ticket# TT-00001, priority = 2, status = Closed	Sep 12, 2015 07:58:13 PM PDT
Calling checkAssignmentForPriority()	Sep 12, 2015 07:58:14 PM PDT
Called checkAssignmentForPriority()	Sep 12, 2015 07:58:14 PM PDT
JBO-25184: Exception in expression "TroubleTicket_c" object function checkAssignmentForPriority(): java.lang.NullPointerException: Cannot invoke method toUpperCase() on null object	Sep 12, 2015 07:58:14 PM PDT

Message Details

Sep 12, 2015 07:58:14 PM PDT

JBO-25184: Exception in expression "TroubleTicket_c" object function checkAssignmentForPriority(): java.lang.NullPointerException: Cannot invoke method toUpperCase() on null object

at "TroubleTicket_c" object function checkAssignmentForPriority() line 6
at "TroubleTicket_c" object function checkPriority() line 6
at "TroubleTicket_c" validation rule "p2_or_p1_must_be_assigned" line 4

Writing Diagnostic Log Messages from Your Scripts

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 its value along with a newline. For example:

```
// Write a diagnostic message to the log. Notice how  
// convenient string substitution expressions are
```

```
println("Status = ${Status}")
```

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}")
```

Clearing the Diagnostic Log Message Viewer

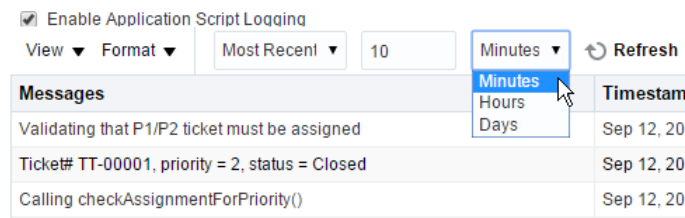
While you are debugging your code, it can be handy to clear the log message display before repeating a test so that only the most recent log messages appear in the console.

To do this, click the **Clear** button in the toolbar. The diagnostic console toolbar reflects your action by showing that you are viewing all messages **Since Log Cleared**.

Viewing the Most Recent Messages by Minute, Hour, or Day

To view the messages that have occurred in the last several minutes, hours, or days, change the diagnostic console toolbar selector from the value **Since** to the value **Most Recent** as shown in the figure below.

Choose a time unit from the list among **Minutes**, **Hours** or **Days**. Type in a number of time units into the field between the two selection lists. For example, if you want to see the messages for the last 10 minutes, type in the number **10** with the time unit selector set to **Minutes**. Finally, click the **Refresh** button to view the most recent messages in the new time span.



Working with Lists

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

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

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

```
// 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.

```
// 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 six element. Of course you can also pass a variable as the value of the operand like this:

```
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:

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

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

```
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:

```
// 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:

```
// 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.

```
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 an object function for each name processed
    if (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:

```
// 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:

```
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:

```
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:

```
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 map key value as if it were a field name on the map object:

```
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:

```
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:

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

```
// 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!

```
// 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 ${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.

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

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

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

Working with Ranges

Using ranges, you can conveniently create 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 `1..100`:

```
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:

```
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:

```
for (j in 1..100) {
    // do something with j here
}
```


3 Examples of Each Context Where You Can Use Groovy

This section provides a simple example of using Groovy in all of the different supported contexts in your application.

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

When you need a field whose value is calculated, 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:

```
(today() - CreationDate) as Integer /* truncate to whole number of days */
```

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:

```
CreationDate + 3
```

Providing an Expression to Make a Custom Field Conditionally Updatable

A custom field can be updatable or read-only. By default, any non-formula field is updatable. Or, you can configure a conditionally updatable expression.

If you do this, it's 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 updatable expression depends. By doing this, your conditionally updatable field will interactively enable or disable as appropriate when the user changes the values of fields on which the conditionally updatable 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 updatable expression for the `Justification` field as follows:

```
Status_c != 'Closed'
```

After configuring this expression, you must then indicate that the `Justification` field depends on the `Status` field as described in *Understanding Automatic Dependency Information for Cascading Fixed-Choice Lists*. 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.

Tip: A field configured with a conditionally updatable expression only enforces the conditional updateability through the web user interface in this release. See *Enforcing Conditional Updatability of Custom Fields for Web Service Access* for more information on how to ensure it gets enforced for web service access as well.

CAUTION: It's not recommended to assign a new value to any field as part of the code of a conditionally updatable expression. The script can consult the value of one or more fields, but should not assign a new value to any field in the row. Doing so will cause unexpected behavior in the user interface, may impact runtime performance, and can cause deadlocks.

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:

```
Priority_c <= 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.

CAUTION: It is not recommended to assign a new value to any field as part of the code of a conditionally required expression. The script can consult the value of one or more fields, but should not assign a new value to any field in the row. Doing so will cause unexpected behavior in the user interface, may impact runtime performance, and can cause deadlocks.

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.

Note: Dynamic choice lists don't support field-level validation rules.

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 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.

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

Rule Body

```
newValue == null || (1..5).contains(newValue as Integer)
```

Tip: If a validation rule for field `a` depends on the values of one or more other fields (e.g. `x` 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](#).

Defining an Object-Level Validation Rule

An object-level validation rule is a constraint you can define on any business object. It is evaluated whenever the framework attempts to validate the object.

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 `DueDate` fields. To validate that a trouble ticket of priority 1 or 2 cannot be saved without a due date, your object-level rule would look like this:

- **Rule Name:** `Validate_High_Priority_Ticket_Has_DueDate`
- **Error Message:** A trouble ticket of priority 1 or 2 must have a due date

Rule Body

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

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 $\pm 2^{63}-1$
- **BigInteger:** a integer of arbitrary precision
- **Double:** a floating-point decimal value in the range of $\pm 1.79769313486231570 \times 10^{308}$
- **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.

- **Function Name:** `logStart`
- **Return Type:** `void`
- **Parameters:** `scriptName String`

Function Definition

```
// Log the name of the script
println("[In: ${scriptName}]")
```

- **Function Name:** `log`
- **Return Type:** `void`
- **Parameters:** `message String`

Function Definition

```
// 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 appends a view criteria 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. Finally, if the new count is different from the existing value of the `OpenTroubleTickets_c` field, it updates this field's value to be the new count computed.

- **Function Name:** `updateOpenTroubleTicketCount`
- **Return Type:** `void`
- **Parameters:** *None*

Function Definition

```
adf.util.logStart('updateOpenTroubleTicketCount')
// Access the view object for TroubleTicket programmatic access
def tickets = newView('TroubleTicket_c')
tickets.appendViewCriteria("""
Contact_Id_c = ${Id} and Status_c in ('Working','Waiting')
""")
// Update OpenTroubleTickets field value
def newCount = tickets.getEstimatedRowCount()
if (OpenTroubleTickets_c != newCount) {
    OpenTroubleTickets_c = newCount
}
```

Controlling the Visibility of an Object Function

When you create an object function named `doSomething()` on an object named `Example`, the following is true by default:

- other scripts on the same object can call it,
- any script written on *another* object that obtains a row of type `Example` can call it
- external systems working with an `Example` object via Web Services, *cannot* call it
- it displays in the *Row* category of the *Functions* tab on the *Expression Palette*.

You can alter some of this default behavior by changing your `doSomething()` object function's *Visibility* setting. If you change its *Visibility* to the value *Callable by External Systems*, then an external system working with an `Example` object will be able to invoke your `doSomething()` via Web Services. Do this when the business logic it contains should be accessible to external systems.

If instead you change the *Visibility* to the value *Hidden in Expression Builder*, then `doSomething()` will not display in the *Row* category of the *Functions* tab of the *Expression Palette*, and it remains inaccessible to external systems. Do this when you want to discourage colleagues from inadvertently invoking the `doSomething()` function directly because you know that its correct use is limited to being called by some other script on the `Example` object.

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 Invalidate**
Fires on a valid parent object when a child row is created, removed, or modified, or also when the first persistent field is changed in an unmodified row.
- **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.
- **Before Update in Database**
Fires before an existing object is modified in the database
- **Before Delete in Database**
Fires before 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.
- **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 trigger on the `TroubleTicket` object that invokes the `updateOpenTroubleTicketCount()` object function described above.

- **Trigger Object:** `TroubleTicket`

- **Trigger:** *After Changes Posted to Database*
- **Trigger Name:** `After_Changes_Set_Open_Trouble_Tickets`

Trigger Definition

```
adf.util.logStart('After_Changes_Set_Open_Trouble_Tickets')
// 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 Values of a Field When Other Fields Change Value](#) for tips on using this trigger.

4 Groovy Tips and Techniques

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

Simplifying Code Authoring with the Expression Palette

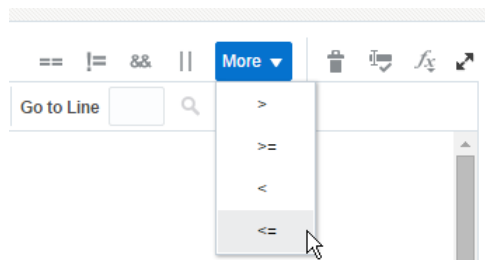
Before diving into the specifics of each example, keep in mind that the Application Composer always makes an Expression Palette available to assist you. It helps you insert the names of built-in functions, object fields, or function names.

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 the figure below, you have a number of useful toolbar buttons that can assist you to:

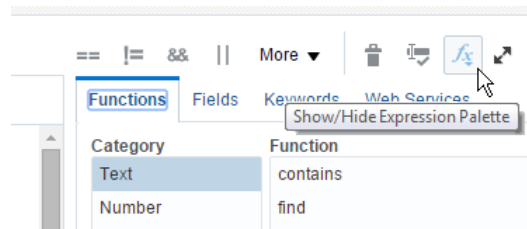
- insert one of the common comparison operators like equals (`==`), not equals (`!=`), less than (`<`), greater than (`>`), etc.
- erase the code editor to start over with a clean slate
- validate the correct syntax of your script, or
- show the expression palette for help with functions, field names, keywords, and web services, or
- maximize the code editor to give you the largest possible area in which to see and edit your script.



Showing and Hiding the Expression Palette

To expand the amount of assistance available to you, you can show the expression palette. As shown in the figure below, click the **Show/Hide Expression Palette** button to expand the palette, revealing its four tab display.

You can adjust how much horizontal space the palette occupies by using the slider between the code editor and the palette. To hide the palette again, just click again on the **Show/Hide Expression Palette** button.



Inserting an Expression for Fields of Related Objects

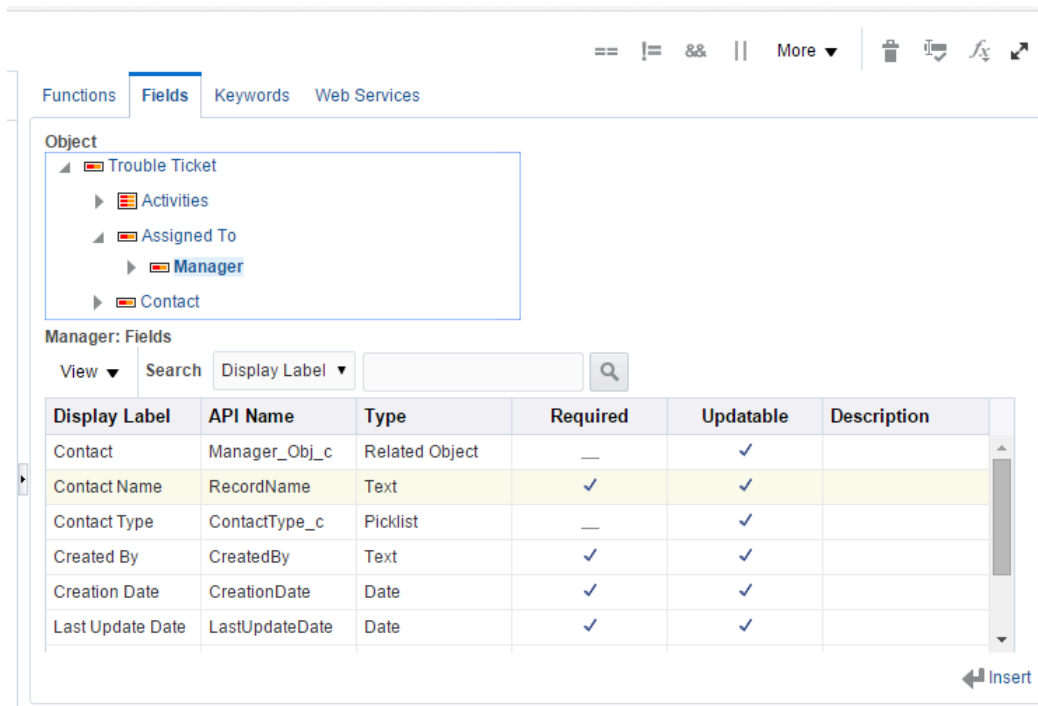
The Fields tab of the palette, as shown below, displays the current object at the root of the tree, and allows you to explore the related objects and collections of related objects by expanding the tree nodes.

If the selected node has the single-row icon , then it represents a single, related object. If the selected node has the multiple-row icon , 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.



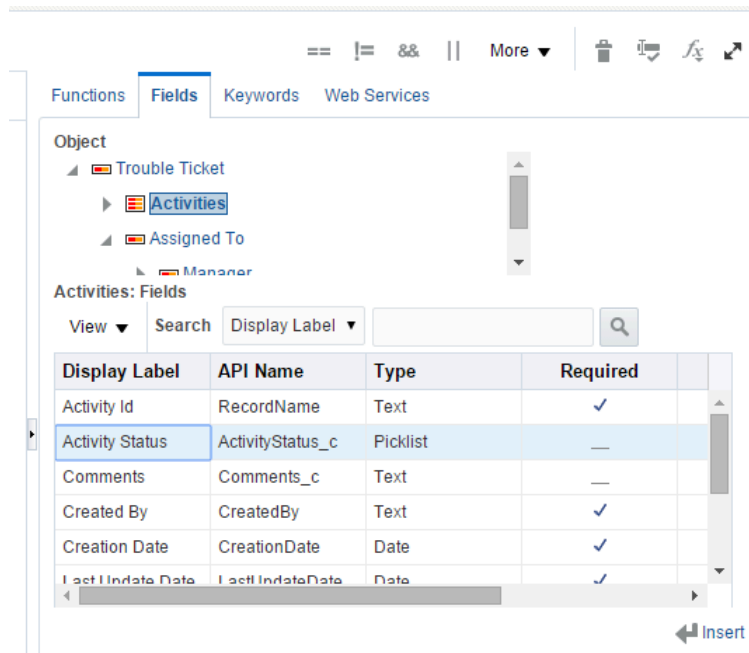
If the field on the selected object is directly accessible from the current object through "dot" navigation, then the expression palette will insert the expression for you. For example, if you click (*Insert*) after selecting the *Contact Name* field of the *Manager* object related to the *Assigned To* 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

As shown in the figure below, if you select the Activities collection and then select the **Activity Status** field from the object in that collection, only the API name `activityStatus_c` of the selected field is inserted.

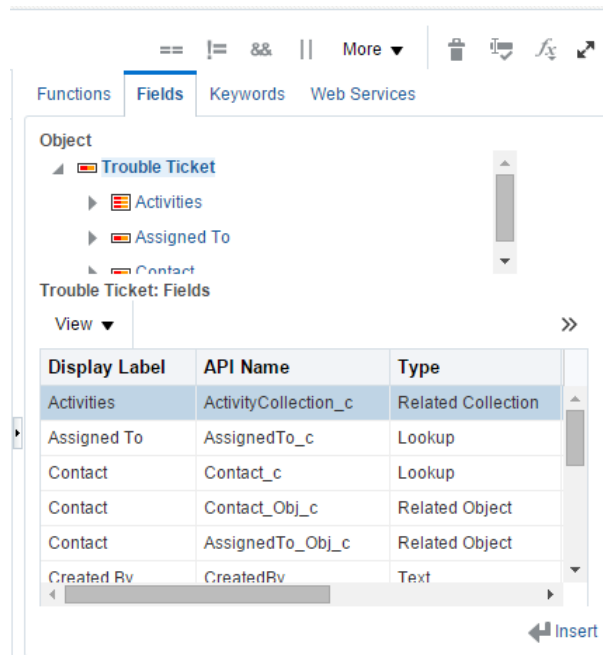
A complete expression can't be inserted in this case because you can't use "dot" notation to work with a specific row in a collection to access its field value. You need to use Groovy code to access the collection of rows and then work with one row at a time.



Inserting a Related Object Accessor or Related Collection Accessor Field Name

As shown in the figure below, to insert the name of a 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.

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.



Display Label	API Name	Type
Activities	ActivityCollection_c	Related Collection
Assigned To	AssignedTo_c	Lookup
Contact	Contact_c	Lookup
Contact	Contact_Obj_c	Related Object
Contact	AssignedTo_Obj_c	Related Object
Created By	CreatedBy	Text

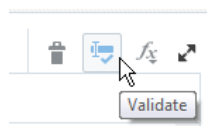
Once you have inserted the name of the related collection field, you can write a loop to process each row in the collection like this:

```
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*).

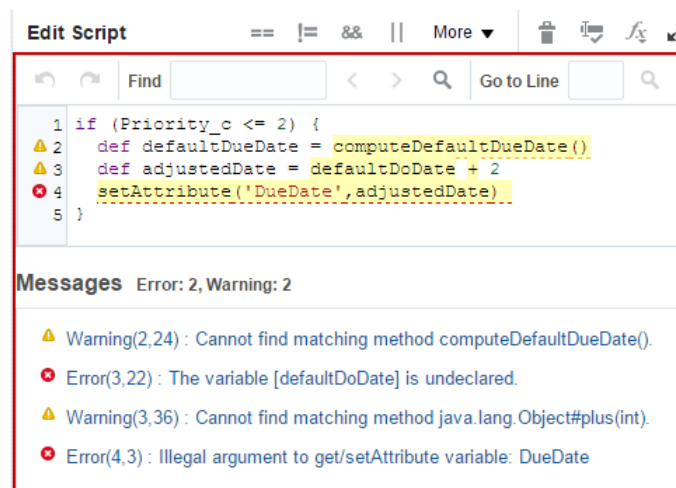
Validating the Syntax and Correctness of Your Script

To validate the syntax and correctness of any script you write, click on the Validate script button in the expression editor toolbar as shown in the figure below.



The validator highlights places in your script where you've violated the rules of Groovy scripting language syntax, making them easier to find and correct. For example, this helps you catch mismatched parenthesis or curly braces. The process also checks for the most common kinds of typographical errors like when you incorrectly spell the name of a built-in, global, or object function, or when you mistype the name of a field. Wherever your script calls a function, the system ensures that you have passed the correct number and types of arguments and that the security policy allows using that function.

The figure below shows the result of validating a script that contains three typographical errors. In line 2, the `computeDefaultDueDate()` function name is incorrect because the actual function name is `calculateDefaultDueDate()`. In line 3, the local variable `defaultDueDate` is misspelled as `defaultDoDate`. In line 4, the name of the field passed as the first argument to the `setAttribute()` function is incorrect, because the actual field name is `DueDate_c`. A fourth warning message appears related to the plus sign (+) operator. Since the `defaultDoDate` is not recognized, the validator assumes it must have the default type `object` which does not have a function named `plus()`. To quickly navigate to the line containing an error or warning, just click on the relevant line in the *Messages* panel. The cursor moves automatically to the line and column containing the problem in the expression editor.



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`. It is best practice to always store the parent object in a local variable as shown in the example below. This ensures that no matter how many fields you access from the parent object or how many times you reference it that you only *retrieve* it once.

```
// 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.
}
```

```
}
```

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 `ticket.Status_c`.

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 Dynamic Choice List Field](#) 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. The example below shows how to reference the two lookup objects from script written in the context of `TroubleTicket`, and access one of each's fields. It is best practice to always store the referenced lookup object in a local variable as shown in the example below. This ensures that no matter how many fields you access from the related object you only retrieve it once.

```
// 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'))
    &&
    (startsWith(supportRep?.PhoneNumber_c, '(202)') ||
    startsWith(supportRep?.PhoneNumber_c, '(206)'))
)
{
    // Do something here because contact's email address
    // is a government or business email and assigned-to
    // support rep is in 202 or 206 Washington DC area code
}
```

Notice that since the dynamic choice list fields `Contact_c` and `AssignedTo_c` might be optional, their value may be `null` and consequently the value of the related object accessor may be `null`, too. This is why the example is using the Groovy safe navigation operator (`?.`) to reference fields of the related object in case either related object might be `null`.

Using the Related Collection Accessor Field to Work with Child Rows

When a 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:

```
// 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 the table below to work with the rows. The row iterator tracks the current row in the collection that your code is working with.

Most Commonly Used RowIterator Methods

Method Name	Description
<code>hasNext()</code>	Returns: - <code>true</code> if the row iterator has more rows to iterate over, <code>false</code> if there are no rows in the iterator's row set or if the iterator is already on or beyond the last row.
<code>next()</code>	Returns: - the next row in the row iterator
<code>reset()</code>	Returns: - <code>void</code> . Resets the row iterator to the "slot" before the first row.
<code>first()</code>	Returns: - the first row in the row iterator, or <code>null</code> if the iterator's row set is empty

Putting the commonly used row iterator methods from this table into practice, the example below 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.

```
// 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 the example below, 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.

```
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

Oracle's application development framework exposes functionality to your business object scripts through the predefined adf variable. For example, to reference the application server's current date use the following expression:

```
adf.currentDate
```

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

```
adf.currentDateTime
```

Note: This function is valid in any Groovy script specific to a particular business object. If necessary to pass the information into other contexts, you can pass its value as a parameter to a function call.

Accessing Current Date and Time from the Database

Oracle's application development framework exposes functionality to your business object scripts through the predefined adf variable. For example, to reference the database's current date, use the following expression:

```
adf.currentDBDate
```

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

```
adf.currentDBDateTime
```

Note: This function is valid in any Groovy script specific to a particular business object. If necessary to pass the information into other contexts, you can pass its value as a parameter to a function call.

Understanding ADF's Additional Built-in Groovy Functions

This section explains a number of additional helper functions you can use in your scripts. Some provide a simple example as well.

Use the *Functions* tab of the code editor palette to insert any of the built-in functions into your script.

Built-in Date Functions

Function	Description
<code>today()</code>	Returns: the current date, with no time Return Type: <code>Date</code>
<code>now()</code>	The current date and time Return Type: <code>Timestamp</code>
<code>date(year , month , day)</code>	Returns: a date, given the year, month, and day Return Type: <code>Date</code> Parameters: • year - a positive integer • month - a positive integer between 1 and 12 • day - a positive integer between 1 and 31 Example: to return a date for February 8th, 1998, use <code>date(1998,2,8)</code>
<code>dateTime(y , m , d , hr , min , sec)</code>	Returns: a timestamp, given the year, month, day, hour, minute, and second Return Type: <code>Timestamp</code> Parameters: • 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 Example: to return a timestamp for February 8th, 1998, at 23:42:01, use <code>dateTime(1998,2,8,23,42,1)</code>
<code>year(date)</code>	Returns: the year of a given date

Function	Description
	Return Type: Integer Parameters: • <code>date</code> - date Example: if <code>curDate</code> represents April 19th, 1996, then <code>year (curDate)</code> returns 1996.
<code>month (date)</code>	Returns: the month of a given date Return Type: Integer Parameters: • <code>date</code> - a date Example: if <code>curDate</code> represents April 12th, 1962, then <code>month (curDate)</code> returns 4.
<code>day (date)</code>	Returns: the day for a given date Return Type: Integer Parameters: • <code>date</code> - a date Example: if <code>curDate</code> represents July 15th, 1968, then <code>day (curDate)</code> returns 15.

Built-in String Functions

Function	Description
<code>contains (s1 , s2)</code>	Returns: <code>true</code>, if string <code>s1</code> contains string <code>s2</code>, <code>false</code> otherwise Return Type: boolean Parameters: • <code>s1</code> - a string to search in • <code>s2</code> - a string to search for Example: if <code>twitterName</code> holds the value <code>@steve</code>, then <code>contains (twitterName, '@')</code> returns <code>true</code>.
<code>endsWith (s1 , s2)</code>	Returns: <code>true</code>, if string <code>s1</code> ends with string <code>s2</code>, <code>false</code> otherwise Return Type: boolean Parameters: • <code>s1</code> - a string to search in • <code>s2</code> - a string to search for For example, if <code>twitterName</code> holds the value <code>@steve</code>, then <code>endsWith (twitterName, '@')</code> returns <code>false</code>.

Function	Description
<code>find(s1 , s2)</code>	Returns: the integer position of the first character in string <code>s1</code> where string <code>s2</code> is found, or zero (0) if the string is not found Return Type: Integer Parameters: • <code>s1</code> - a string to search in • <code>s2</code> - a string to search for Example: if <code>twitterName</code> holds the value <code>@steve</code>, then <code>find(twitterName, '@')</code> returns 1 and <code>find(twitterName, 'ev')</code> returns 4.
<code>left(s , len)</code>	Returns: the first <code>len</code> characters of the string <code>s</code> Return Type: String Parameters: • <code>s</code> - a string • <code>len</code> - an integer number of characters to return Example: if <code>postcode</code> holds the value <code>94549-5114</code>, then <code>left(postcode, 5)</code> returns <code>94549</code>.
<code>length(s)</code>	Returns: the length of string <code>s</code> Return Type: Integer Parameters: • <code>s</code> - a string Example: if <code>name</code> holds the value <code>Julian Croissant</code>, then <code>len(name)</code> returns 16.
<code>lowerCase(s)</code>	Returns: the string <code>s</code> with any uppercase letters converted to lowercase Return Type: String Parameters: • <code>s</code> - a string Example: if <code>sku</code> holds the value <code>12345-10-WHT-XS</code>, then <code>lowerCase(sku)</code> returns <code>12345-10-wht-xs</code>.
<code>right(s , len)</code>	Returns: the last <code>len</code> characters of the string <code>s</code> Return Type: String Parameters: • <code>s</code> - a string • <code>len</code> - an integer number of characters to return Example: if <code>sku</code> holds the value <code>12345-10-WHT-XS</code>, then <code>right(sku, 2)</code> returns <code>XS</code>.
<code>startsWith(s1 , s2)</code>	Returns: <code>true</code>, if string <code>s1</code> starts with <code>s2</code>, <code>false</code> otherwise Return Type: boolean

Function	Description
	Parameters: <code>s1</code> - a string to search in <code>s2</code> - a string to search for Example: if <code>twitterName</code> holds the value <code>@steve</code>, then <code>startsWith(twitterName, '@')</code> returns <code>true</code>.
<code>substringBefore(s1 , s2)</code>	Returns: the substring of <code>s1</code> that precedes the <i>first</i> occurrence of <code>s2</code>, otherwise an empty string Return Type: <code>String</code> Parameters: <code>s1</code> - a string to search in <code>s2</code> - a string to search for Examples: if <code>sku</code> holds the value <code>12345-10-WHT-XS</code>, then <code>substringBefore(sku, '-')</code> returns the value <code>12345</code>, <code>substringBefore(sku, '12345')</code> returns an empty string, and <code>substringBefore(sku, '16-BLK')</code> also returns an empty string.
<code>substringAfter(s1 , s2)</code>	Returns: the substring of <code>s1</code> that follows the <i>first</i> occurrence of <code>s2</code>. otherwise an empty string Return Type: <code>String</code> Parameters: <code>s1</code> - a string to search in <code>s2</code> - a string to search for Example: if <code>sku</code> holds the value <code>12345-10-WHT-XS</code>, then <code>substringAfter(sku, '-')</code> returns the value <code>10-WHT-XS</code>, <code>substringAfter(sku, 'WHT-')</code> returns the value <code>XS</code>, <code>substringAfter(sku, 'XS')</code> returns an empty string, and <code>substringAfter(sku, 'BLK')</code> also returns an empty string.
<code>upperCase(s)</code>	Returns: the string <code>s</code> with any lowercase letters converted to uppercase Return Type: <code>String</code> Parameters: <code>s</code> - a string Example: if <code>sku</code> holds the value <code>12345-10-Wht-xs</code>, then <code>upperCase(sku)</code> returns <code>12345-10-WHT-XS</code>.

Other Built-in Functions

Function	Description
<code>newView(objectAPIName)</code>	Returns: a <code>ViewObject</code> reserved for programmatic use, or <code>null</code> if not available. Return Type: <code>ViewObject</code> Parameters:

Function	Description
	<code>objectAPIName</code> - the object API name whose rows you want to find, create, update, or remove Example: <code>newView('TroubleTicket')</code> returns a new view object instance you can use to find, create, update, or delete <code>TroubleTicket</code> rows.
<code>key(list)</code>	Returns: a multi-valued key object for use in the <code>ViewObject</code>'s <code>findByKey()</code> method. Return Type: <code>Key</code> Parameters: <code>list</code> - a list of values for a multi-field key Example: if a standard object has a two-field key, use <code>key([101, 'SAMBA'])</code>
<code>key(val)</code>	Returns: a key object for use in the <code>ViewObject</code>'s <code>findByKey()</code> method. Return Type: <code>Key</code> Parameters: <code>val</code> - a value to use as the key field Example: if a standard object has a single-field key, as all custom objects do, use <code>key(123456789)</code>
<code>nvl(o1 , o2)</code>	Returns: the object <code>o1</code> if it is not null, otherwise the object <code>o2</code>. Return Type: <code>Object</code> Parameters: <code>o1</code> - a value to use if not null <code>o2</code> - a value to use instead if <code>o1</code> is null Example: to calculate the sum of <code>Salary</code> and <code>Commission</code> fields that might be null, use <code>nvl(Salary,0) + nvl(Commission,0)</code>
<code>encodeToBase64(s)</code>	Returns: the base64 encoding of <code>s</code>. Return Type: <code>String</code> Parameters: <code>s</code> - string to encode
<code>decodeBase64(s)</code>	Returns: the base64 decoding of <code>s</code>. Return Type: <code>String</code> Parameters: <code>s</code> - string to decode
<code>decodeBase64ToByteArray(s)</code>	Returns: byte array decoding of <code>s</code>. Return Type: <code>byte[]</code> Parameters:

Function	Description
	• s - string to decode
<code>encodeByteArrayToBase64 (b)</code>	Returns: base64 encoding of b. Return Type: <code>String</code> Parameters: • b - <code>byte[]</code> to encode

Testing Whether a Field's Value Is Changed

You can test whether a 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:

```
if (isAttributeChanged('Status')) {  
    // 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 application development 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 field assignment (or `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:

```
// 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()
    // If new defaultRep is not the current rep, assign it
    if (PurchasingRep != defaultRep) {
        PurchasingRep = defaultRep
    }
}
return true
```

Note: This example illustrates how to avoid a typical problem that can occur when using a validation rule to perform field derivations. The recommended trigger to use for such purposes would be the field-level "After Value Changed" trigger, or alternatively the "Before Insert" and/or "Before Update" trigger. It is still a good practice to perform conditional field assignment in those cases, too. See *Deriving Values of a Field When Other Fields Change Value* for more information on deriving field values.

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 Changes Posted to Database.

When the After Changes Posted to Database trigger fires, the changes in the row have already been sent to the database, and performing further field assignments therein requires doing a second round trip to the database to permanently save your field updates to each row modified in this way. When possible, using the Before-save triggers sets the field values before the changes are sent the first time to the database, resulting in better performance.

Note: If your script utilizes the `getEstimatedRowCount()` function on view object query with a complex filter, then use the After Changes Posted to Database trigger for best results. The database `COUNT()` query the function performs to return the estimate is more accurate when performed over the already-posted data changes made during the current transaction.

Detecting Row State in After Changes Posted to Database Trigger

When writing an After Changes Posted to Database trigger, if your code needs to detect the effective row state of the current object, use the `getPrimaryRowState()` function.

For example, it can use `getPrimaryRowState().isNew()` to notice that the current object was created in the current transaction or `getPrimaryRowState().isModified()` to conclude instead that it was an existing row that was changed.

The `getPrimaryRowState()` function is covered in *Determining the State of a Row*.

Avoiding Posting Threshold Errors By Conditionally Assigning Values

Despite the recommendation in , if you still must use an After Changed Posted to Database trigger then you must be aware of how this affects the object's so-called "posting cycle".

For example, you might use it to perform field value assignments when your custom logic must perform a query that filters on the data being updated in the current transaction. 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 field assignment (or `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:

```
// After Changes Posted in Database Trigger
// If total score is 100 or more, set status to WON.
def totalScore = calculateTotalScoreUsingQuery()
if (totalScore >= 100) {
    // Only set the status to WON if it's not already that value
    if (Status != 'WON') {
        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.

- *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 drop-down 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`):

```
// 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:

```
// 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:

```
// 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 current locale (inferred from each end user's browser settings) as part of global function that encapsulates all of your error strings. Consider a `getMessage` 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:

```
// 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:** `stringKey String`

Function Definition

```
// 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, your trigger would look like this:

```
// 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
- Append a view criteria to restrict the view object to only return some desired *subset* of business objects rows that meet your needs, 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, the most common methods that you will use on the view object are shown in the following table.

Most Commonly Used View Object Methods

Method Name	Description
<code>findByKey()</code>	Allows you to find a row by unique id. Returns: an array of rows having the given key, typically containing either zero or one row. Parameters: • key - a key object representing the unique identifier for the desired row • maxRows - an integer representing the maximum number of rows to find (typically 1 is used) Example: See <i>Finding an Object by Id</i>
<code>findRowsMatchingCriteria()</code>	Allows you to find a set of matching rows based on a filter criteria. Returns: an iterator you can use to process the matching rows using methods <code>iter.hasNext()</code> and <code>iter.next()</code> for one row. Parameters: • viewCriteria - a view criteria representing the filter. The easiest way to create a new view criteria is to use the <code>newViewCriteria()</code> function. • maxRows - an integer representing the maximum number of rows to find (-1 means return all matching rows up to a limit of 500) Example: See <i>Finding Rows in a Child Rowset Using findRowsMatchingCriteria</i>

Method Name	Description
<code>appendViewCriteria()</code>	Appends an additional view criteria query filter. Parameters: • filterExpr - a String representing a filter expression. • ignoreNullBindVarValues - an optional boolean parameter indicating whether expression predicates containing null bind variable values should be ignored (defaults to false if not specified). <i>Returns:</i> - void. Alternatively, if you already have created a view criteria using <code>newViewCriteria()</code> you can pass that view criteria as the single argument to this function.
<code>executeQuery()</code>	Executes the view object's query with any currently appended view criteria filters. <i>Returns:</i> - void.
<code>hasNext()</code>	<i>Returns:</i> - true if the row iterator has more rows to iterate over, false if there are no further rows in the iterator or it is already on or beyond the last row.
<code>next()</code>	<i>Returns:</i> - the next row in the iterator
<code>reset()</code>	Resets the view object's iterator to the "slot" before the first row. <i>Returns:</i> - void.
<code>first()</code>	<i>Returns:</i> - the first row in the row iterator, or null if the iterator's row set is empty
<code>createRow()</code>	Creates a new row, automatically populating its system-generated Id primary key field. <i>Returns:</i> - the new row
<code>insertRow()</code>	Inserts a new row into the view object's set of rows. <i>Returns:</i> - void
<code>setSortBy()</code>	Set the sort order for query results. <i>Returns:</i> - void

Defining the Sort Order for Query 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.

To define the sort order for view object query results, call the `setSortBy()` method on the view object instance you're working with, before calling its `executeQuery()` method, to retrieve the results.

The following example shows how to use this method to sort by a single field:

```
def vo = newView('TroubleTicket')
// Use object function to simplify filtering by agent
applyViewCriteriaForSupportAnalyst(vo, analystId)
vo.setSortBy('Priority')
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.

The example below shows how to sort descending by the number of callbacks.

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

The `setSortBy()` method doesn't let you specify how null values are sorted. Instead, null values are sorted according to whether the sort order is ascending or descending:

- If the sort order is ascending, then null values display last
- If the sort order is descending, then null values display first

Note: If `setSortBy()` is called during the creation of a new row, then that new row isn't part of the sort and is always listed first.

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

```
def vo = newView('TroubleTicket')
// Use object function to simplify filtering by customer
applyViewCriteriaForCustomerCode(vo, custCode)
// Sort ascending by Priority, then descending by date created
vo.setSortBy('Priority 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 the following example.

```
def vo = newView('TroubleTicket')
// 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), Priority')
```

```
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. The following example shows how the steps fit together in practice.

```
// 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:** `oracle.jbo.Row`
- **Parameters:** `vo oracle.jbo.ViewObject, idValue Object`

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, the example below shows the simplified code for finding a row by key.

```
// 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 declarative data filter for the custom or standard objects you work with in your scripts. After creating a view object using the `newView()` function, but before calling `executeQuery()` on it, use the `appendCriteria()` method to add a filter so the query will return only the rows you want to work with.

This section explains the declarative syntax of view criteria filter expressions, and provides examples of how to use them. At runtime, the application development framework translates the view criteria into an appropriate SQL `WHERE` clause for efficient database execution.

At runtime, the application development framework translates the view criteria into an appropriate SQL `WHERE` clause for efficient database execution.

This section explains the declarative syntax of view criteria filter expressions, and provides examples of how to use them.

Using a Simple View Criteria

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

1. Create a view object with the `newView()` function
2. Append a view criteria with the `appendViewCriteria()` function, using an appropriate filter expression
3. Execute the query by calling `executeQuery()`
4. Process the results

The example below queries the `TroubleTicket` custom object to find the trouble tickets assigned to a particular staff member with id 100000000089003 and which have a status of `Working`.

```
/*
 * Query all 'Working'-status trouble tickets assigned to a staff member with id 100000000089003
 */
// 1. Use the newView() function to get a view object
def vo = newView('TroubleTicket_c')
// 2. Append a view criteria using a filter expression
vo.appendViewCriteria("AssignedTo_Id_c = 100000000089003 and Status_c = 'Working'")
// 3. Execute the query
vo.executeQuery()
// 4. Process the results
if (vo.hasNext()) {
    def row = vo.next()
    // Do something here with the current result row
}
```

Syntax of View Criteria Filter Expressions

You use a view criteria filter expression to identify the specific rows you want to retrieve from a view object.

Each expression includes the case-sensitive name of a field you can query, followed by an operator and one or more operand values (depending on the operator used). Each operand value can be either a literal value or a bind variable value. An attempt to filter on a field that you can't query, or a field name that doesn't exist in the current object will raise an error. The following are simple examples of filter expressions.

To test whether a value is `null` you must use the `is null` or the `is not null` keywords:

- `Comment is null`
- `Comment is not null`

For equality use the `=` sign, and for inequality use either the `!=` or the `<>` operators. Literal datetime values must adhere exclusively to the format shown here.

- `NextCallSchedule = '2015-07-15 16:26:30'`
- `Priority = 3`
- `Priority != 1`
- `Priority <> 1`
- `ActivityType != 'RS'`
- `ActivityType <> 'RS'`

For relational comparisons, use the familiar `<`, `<=`, `>`, or `>` operators, along with `between` or `not between`. Literal date values must adhere exclusively the format shown here.

- `CreationDate >= '2015-07-15'`
- `Priority <= 2`
- `Priority < 3`
- `Priority <> 1`
- `Priority > 1`
- `Priority >= 1`
- `TotalLoggedHours >= 12.75`
- `Priority between 2 and 4`
- `Priority not between 2 and 4`

For string matching, you can use the `like` operator, employing the percent sign `%` as the wildcard character to obtain "starts with", "contains", or "ends with" style filtering, depending on where you place your wildcard(s):

- `RecordName like 'TT-%'`
- `RecordName like '%-TT'`
- `RecordName like '%-TT-%'`

To test whether a field's value is in a list of possibilities, you can use the `in` operator:

- `ActivityType in ('OC','IC','RS')`

You can combine expressions using the conjunctions `and` and `or` along with matching sets of parentheses for grouping to create more complex filters like:

- `(Comment is null) or ( (Priority <= 2) and (RecordName like 'TT-99%'))`
- `(Comment is not null) and ( (Priority <= 2) or (RecordName like 'TT-99%'))`

When using the `between` or `in` clauses, you must surround them by parentheses when you join them with other clauses using `and` or `or` conjunctions.

You use a filter expression in one of two ways:

1. Append the view criteria filter expression using `appendViewCriteria()` to a view object created using `newView()`
2. Create the view criteria by passing a filter expression to `newViewCriteria()`, then filter a related collection with `findRowsMatchingCriteria()`

Filter expressions are not validated at design time, so if your expression contains typographical errors like misspelled field names, incorrect operators, mismatched parentheses, or other errors, you will learn of the problem at runtime when you test your business logic.

Tips for Formatting Longer Criteria Across Multiple Lines

Groovy does not allow carriage returns or newlines to appear inside of a quoted string, so for example, the following lines of script would raise an error:

```
def vo = newView('StaffMember')
// ERROR: Single-line quotes cannot contain carriage returns or new lines
vo.appendViewCriteria("
    (Salary between 10000 and 24000)
    and JobId <> 'AD_VP'
    and JobId <> 'PR_REP'
    and CommissionPct is null
    and Salary != 11000
    and Salary != 12000
    and (DepartmentId < 100
    or DepartmentId > 200)
")
vo.executeQuery()
```

Luckily, Groovy supports the triple-quote-delimited, multi-line string literal, so you can achieve a more readable long view criteria filter expression using this as shown:

```
def vo = newView('StaffMember')
vo.appendViewCriteria("""
    (Salary between 10000 and 24000)
    and JobId <> 'AD_VP'
    and JobId <> 'PR_REP'
    and CommissionPct is null
    and Salary != 11000
    and Salary != 12000
    and (DepartmentId < 100
    or DepartmentId > 200)
""")
vo.executeQuery()
```

Using String Substitution for Literal Values into a View Criteria Expression Used Only Once

If you're using a view object only a single time after calling `newView()`, use Groovy's built-in string substitution feature to replace variable or expression values directly into the view criteria expression text as shown below:

```
def vo = newView('StaffMember')
def loSal = 13500
def anon = 'Anonymous'
vo.appendViewCriteria("(Salary between ${loSal} and ${loSal + 1}) and LastName != '${anon}')"
vo.executeQuery()
```

Notice that you must still include single quotes around the literal string values. The string substitution occurs at the moment the string is passed to the `appendViewCriteria()` function, so if the values of the `loSal` or `anon` variables change, their new values are not reflected retroactively in the substituted string filter criteria expression. In this example below,

Groovy substitutes the values of the `loSal` and `anon` into the view criteria expression string before passing it to the `appendViewCriteria()` function. Even though their values have changed later in the script, when the `vo.executeQuery()` is performed a second time, the view object re-executes using the exact same filter expression as it did before, unaffected by the changed variable values.

```
def vo = newView('StaffMember')
def loSal = 13500
def anon = 'Anonymous'
vo.appendViewCriteria("(Salary between ${loSal} and ${loSal + 1}) and LastName != '${anon}'")
vo.executeQuery()
// ... etc ...
loSal = 24000
anon = 'Julian'
// The changed values of 'loSal' and 'anon' are not used by the
// view criteria expression because the one-time string substitutions
// were done as part of the call to appendViewCriteria() above.
vo.executeQuery()
```

If you need to use a view object with appended view criteria filter expression multiple times within the same script, use named bind variables as described in the following section instead of string substitution. Using named bind variables, the updated values of the variables are automatically used by the re-executed query.

Using Custom Bind Variables for View Criteria Used Multiple Times

Often you may need to execute the same view object multiple times within the same script.

If your operand values change from query execution to query execution, then named bind variables allow you to append a view criteria once, and use it many times with different values for the criteria expression operands. Just add one or more named bind variables to your view object, and then set the values of these bind variables as appropriate before each execution. The bind variables act as "live" placeholders in the appended filter expression, and their current values are used each time the view object's query is executed.

To add a named bind variable, use the `addBindVariable()` function. Pass a view object or rowset as the first argument and a string value to define the name of the bind variable as the second argument as shown in the example below. You can name your bind variable using any combination of letters, numbers, and underscores, as long as the name starts with a letter. When using a view criteria with a standard object like the `StaffMember` in this example, it is best practice to name your bind variables with the same suffix (`_c`) as custom objects and custom fields use.

```
def vo = newView("StaffMember")
addBindVariable(vo, "VarLastName_c")
setBindVariable(vo, "VarLastName_c", "King")
vo.appendViewCriteria("LastName = :VarLastName_c")
vo.executeQuery()
while (vo.hasNext()) {
    def r = vo.next();
    // Will return "Steven King" and "Janette King"
}
setBindVariable(vo, "VarLastName_c", "Higgins")
vo.executeQuery()
while (vo.hasNext()) {
    def r = vo.next();
    // Will return "Shelley Higgins"
}
```

You can reference a named bind variable in the view criteria expression anywhere a literal value can be used, prefacing its name by a colon (e.g. `:VarLastName_c`). After adding the bind variable, you use the `setBindVariable()` function one or

more times in your script to assign values to the variable. Until you explicitly set its value for the current view object or rowset, your bind variable defaults to having a value of `null`. Accidentally leaving the value `null` will result in retrieving no rows for most filter expressions involving a bind variable operand due to how the SQL language treats the `null` value in comparisons. The current value of the bind variable is used each time your script executes the view object. In the example below, this causes the rows for employees "Steven King" and "Janette King" to be returned during the first view object execution, and the row for "Shelly Higgins" to be returned on the second view object execution.

By default, the data type of the named bind variable is of type `Text`. If you need to use a bind variable in filter expressions involving number, date, or datetime fields, then you need to explicitly define a bind variable with the appropriate type for best performance. To add a bind variable of a specific datatype, pass one of the values `Text`, `Number`, `Date`, or `Datetime` as a string value to the optional third argument of the `addBindVariable()` function. For example, the following script uses two bind variables of type `Number` and another of type `Date`. Notice that the data type name is not case-sensitive (e.g. `Number`, `number`, or `NUMBER` are all allowed).

```
def vo = newView('TroubleTicket_c')
addBindVariable(vo, "VarLowPri_c", "number")
addBindVariable(vo, "VarHighPri_c", "Number")
addBindVariable(vo, "VarDueDate_c", "DATE")
setBindVariable(vo, "VarLowPri_c", 1)
setBindVariable(vo, "VarDueDate_c", 2)
setBindVariable(vo, "VarDueDate_c", today() + 3)
vo.appendViewCriteria("(Priority_c between :VarLowPri_c and :VarHighPri_c) and DueDate < :VarDueDate_c ")
vo.executeQuery()
while (vo.hasNext()) {
    def row = vo.next()
    // Returns trouble tickets with priorities 1 and 2 that are
    // due within three days from today
}
setBindVariable(vo, "VarLowPri_c", 3)
setBindVariable(vo, "VarDueDate_c", 4)
setBindVariable(vo, "VarDueDate_c", today() + 5)
vo.executeQuery()
while (vo.hasNext()) {
    def row = vo.next()
    // Returns trouble tickets with priorities 3 and 4 that are
    // due within five days from today
}
```

Using View Criteria to Query Case-Insensitively

If you want to filter in a case-insensitive way, you can use the `upper()` function around the field name in the filter.

If you're not sure whether the operand value is uppercase, you can also use the `upper()` function around the operand like this:

- `upper(JustificationCode) = 'BRK'`
- `upper(JustificationCode) = upper(:codeVar)`
- `upper(JustificationCode) like upper(:codeVar)||'%'`

Limitations of View Criteria Filter Expressions

While view criteria filter expressions are extremely convenient, they do not support every possible type of filtering that you might want to do.

This section describes several constructs that are not possible to express directly, and where possible, suggests an alternative way to achieve the filtering.

- *Only a case-sensitive field name is allowed before the operator*

On the left hand side of the operator, only a case-sensitive field name is allowed. So, for example, even a simple expression like `1 = 1` is considered illegal because the left-hand side is not a field name.

- *Cannot reference a calculated expression directly as an operand value*

You might be interested in querying all rows where one field is equal to a calculated quantity. For example, when querying trouble tickets you might want to find all open tickets whose Resolution Promised Date is less than three days away. Unfortunately, an expression like `ResolutionPromisedDate <= today() + 3` is not allowed because it uses a calculated expression on the right hand side of the operator. As an alternative, you can compute the value of the desired expression prior to appending the view criteria and use the already-computed value as a literal operand value string substitution variable in the string or as the value of a bind variable.

- *Cannot reference a field name as an operand value*

You might be interested in querying all rows where one field is equal to another field value. For example, when querying contacts you might want to find all contacts whose Home Phone Number is equal to their Work Phone Number. Unfortunately, an expression like `HomePhoneNumber = WorkPhoneNumber` is not allowed because it uses a field name on the right hand side of the operator. A clause such as this will be ignored at runtime, resulting in no effective filtering.

- *Cannot reference fields of related objects in the filter expression*

It is not possible to reference fields of related objects directly in the filter query expression. As an alternative, you can reference the value of a related expression prior to appending the view criteria and use the already-computed value as a literal operand value string substitution variable in the string or as the value of a bind variable.

- *Cannot use bind variable values of types other than Text, Number, Date, or Datetime*

It is not possible to use bind variable values of types other than the four supported types: Text, Number, Date, and Datetime. An attempt to use other data types as the value of a bind variable may result in errors or in the criteria's being ignored.

Finding Rows in a Child Rowset Using `findRowsMatchingCriteria`

In addition to using view criteria to filter a view object that you create using `newView()`, you can also use one to retrieve a subset of the rows in a related collection.

For example, if a `TroubleTicket` custom object contains a child object collection of related activities, you can process selected activities in the related collection using code as shown below:

```
def vo = newView('TroubleTicket_c')
vo.appendViewCriteria("Priority_c = 1 and Status_c = 'Open'")
vo.executeQuery()
def vc = null
// Process all open P1 trouble tickets
while (vo.hasNext()) {
    def curTicket = vo.next()
    def activities = curTicket.ActivityCollection_c
    if (vc == null) {
        addBindVariable(activities, 'TodaysDate', 'date')
        vc = newViewCriteria(activities, "ActivityType_c in ('OC','IC') and CreationDate > :TodaysDate")
    }
}
```

```
}
// Process the activities created today for inbound/outbound calls
setBindVariable('TodayDate', today())
def iter = activities.findRowsMatchingCriteria(vc, -1)
while (iter.hasNext()) {
    def activity = iter.next()
    // process the activity here
}
}
```

The `newViewCriteria()` function accepts an optional third parameter `ignoreNullBindVarValues` of boolean type that you can use to indicate whether filter expression predicates containing null bind variable values should be ignored. If omitted, the default value of this parameter is false.

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. The example below shows the code you need to work with a fictitious `ServiceTicket` standard object that predefines an `AllSeverityOneOpenTickets` named view criteria.

```
/*
 * 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 bind variables in their filter criteria. Your code *can* (or sometimes must!) assign a value to one or more these bind 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 bind variables must be set, then use the `setBindVariable()` function to assign a value to these in your script before executing the query. The example below shows the code you need to work with a fictitious `ServiceTicket` standard object that predefines an `AllOpenTicketsByAssigneeAndPriority` named view criteria. The `Bind_MaxPriority` bind variable might default to the value 4, so it may be optional to set it in your script. In contrast, the `Bind_Assignee` bind 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.

```
/*
 * 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 bind variable
setBindVariable(vo, 'Bind_MaxPriority', 2)
// 5. Set mandatory Bind_Assignee bind variable
setBindVariable(vo, 'Bind_Assignee', 'psmith')
```

```
// 6. Execute the query
vo.executeQuery()
```

Accomplishing More with Less Code

Your code will frequently work with collections and contain conditional logic and loops involving values that might be null. This section explains the simplest way of working with conditionals and loops when the value involved might be null.

It also covers how to define and pass functions around like objects using closures. Finally, it explains the most common collection methods and how to combine them with closures to gain maximum expressive power in minimum lines of code. Fewer lines of code makes your business logic easier to read and write.

Embracing Null-Handling in Conditions

You can avoid many extra lines of code by understanding how conditional statements behave with null values. If a variable `someFlag` is a Boolean variable that might be null, then the following conditional block executes only if `someFlag` is true.

A `String` variable can be null, an empty string (`""`), or can contain at least one character in it. If a variable `middleName` is a `String`, then the following conditional block executes only if `middleName` is not null and contains at least one character:

```
// If customer has a middle name...
if (middleName) {
    // Do something here if middleName has at least one character in it
}
```

If a variable `recentOrders` is a `List`, then the following conditional block executes only if `recentOrders` is not null and contains at least one element:

```
// If customer has any recent orders...
if (recentOrders) {
    // Do something here if recentOrders has at least one element
}
```

If a variable `recentTransactions` is a `Map`, then the following conditional block executes only if `recentTransactions` is not null and contains at least one map entry:

```
// If supplier has any recent transactions...
if (recentTransactions) {
    // Do something here if recentTransactions has at least one map entry
}
```

If a variable `customerId` can be null, and its data type is anything other than the ones described above then the following conditional block executes only if `customerId` has a non-null value:

```
// If non-boolean customerId has a value...
if (customerId) {
    // Do something here if customerId has a non-null value
}
```

If you need to test a `Map` entry in a conditional and there's a chance the `Map` might be null, then remember to use the safe-navigation operator (`?.`) when referencing the map key by name:

```
// Use the safe-navigation operator in case options Map is null
if (options?.orderBy) {
    // Do something here if the 'orderBy' key exists and has a non-null value
}
```

```
}
```

Embracing Null-Handling in Loops

You can avoid many extra lines of code by understanding how loops behave with null values.

If a variable `recentOrders` is a `List`, then the following loop processes each element in the list or gets skipped if the variable is `null` or the list is empty:

```
// Process recent customer orders (if any, otherwise skip)
for (order in recentOrders) {
    // Do something here with current order
}
```

If a variable `recentTransactions` is a `Map`, then the following conditional block executes only if `recentTransactions` is not `null` and contains at least one map entry:

```
// Process supplier's recent transaction (if any, otherwise skip)
for (transaction in recentTransactions) {
    // Do something here with each transaction referencing each map
    // entry's key & value using transaction.key & transaction.value
}
```

A `String` variable can be `null`, an empty string (`""`), or can contain at least one character in it. If a variable `middleName` is a `String`, then the following conditional block will execute only if `middleName` is not `null` and contains at least one character:

```
// Process the characters in the customer's middle name
for (c in middleName) {
    // Do something here with each character 'c'
}
```

If your `for` loop invokes a method directly on a variable that might be `null`, then use the safe-navigation operator (`?.`) to avoid an error if the variable is `null`:

```
// Split the recipientList string on commas, then trim
// each email to remove any possible whitespace
for (email in recipientList?.split(',')) {
    def trimmedEmail = email.trim()
    // Do something here with the trimmed email
}
```

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.

Examples of How null Is Less Than Everything

Left-Side Expression	Operator	Right-Side Expression	Comparison Result
'a'	>	null	true
'a'	<	null	false
100	>	null	true
100	<	null	false
-100	>	null	true
-100	<	null	false
now()	>	null	true
now()	<	null	false
now() - 7	>	null	true
now() - 7	<	null	false

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) like a spreadsheet user might expect — then use the `nvl()` function as part of your comparison logic as shown in the following example:

```
// 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,0) < -2000) {
    // do something for suspiciously large overdraft amount
}
```

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

Using Functions as Objects with Closures

While writing helper code for your application, you may find it handy to treat a function as an object called a closure.

It lets you to define a function you can store in a variable, accept as a function parameter, pass into another function as an argument, and later invoke on-demand, passing appropriate arguments as needed.

For example, consider an application that must support different strategies for calculating sales tax on an order's line items. The `order_c` object's `computeTaxForOrder()` function shown below declares a `taxStrategyFunction` parameter of type `closure` to accept a tax strategy function from the caller. At an appropriate place in the code, it invokes the function passed-in by applying parentheses to the parameter name, passing along any arguments.

```
// Object function on Order_c object
// Float computeTaxForOrder(Closure taxStrategyFunction)
Float totalTax = 0
// Iterate over order line items and return tax using
// taxStrategyFunction closure passed in
def orderLines = OrderLinesCollection_c
orderLines.reset()
while (orderLines.hasNext()) {
    // Invoke taxStrategyFunction() passing current line's LineTotal_c
    def currentLine = orderLines.next()
    totalTax += taxStrategyFunction(currentLine.LineTotal_c)
}
return totalTax
```

In one territory *ABC*, imagine that amounts under 25 euros pay 10% tax while items 25 euros or over pay 22%. In a second territory *DEF*, sales tax is a flat 20%. We could represent these two tax computation strategies as separate function variables as shown below. The closure is a function body enclosed by curly braces that has no explicit function name. By default the closure function body accepts a single parameter named `it` that will evaluate to `null` if no parameter is passed at all when invoked. Here we've saved one function body in the variable named `taxForTerritoryABC` and another in the variable `taxForTerritoryDEF`.

```
def taxForTerritoryABC = { return it * (it < 25 ? 0.10 : 0.22) }
def taxForTerritoryDEF = { return it * 0.20 }
```

When the function body is a one-line expression, you can omit the `return` keyword as shown below, since Groovy returns the last evaluated expression as the function return value if not explicitly returned using the `return` statement.

```
def taxForTerritoryABC = { it * (it < 25 ? 0.10 : 0.22) }
def taxForTerritoryDEF = { it * 0.20 }
```

The code inside each anonymous function body is not executed until later when it gets explicitly invoked. With the code in a variable, we can pass that variable as an argument to an object function like the `Order_c` object's `computeTaxForOrder()` as shown below. Here we're calling it from a *Before Insert* trigger on the `Order_c` object:

```
// Before Insert trigger on Order_c
def taxForTerritoryABC = { it * (it < 25 ? 0.10 : 0.22) }
// Assign the value of TotalTax_c field, using the taxForTerritoryABC
// function to compute the tax for each line item of the order.
TotalTax_c = computeTaxForOrder(taxForTerritoryABC)
```

If you don't like the default name `it` for the implicit parameter passed to the function, you can give the parameter an explicit name you prefer using the following "arrow" (`->`) syntax. The parameter name goes on the left, and the body of the function on the right of the arrow:

```
def taxForTerritoryABC = { amount -> amount * (amount < 25 ? 0.10 : 0.22) }
def taxForTerritoryDEF = { val -> val * 0.20 }
```

The closure is not limited to a single parameter. Consider the following slightly different tax computation function on the `Order_c` object named `computeTaxForOrderInCountry()`. It accepts a `taxStrategyFunction` that it invokes with two arguments: an amount to be taxed and a country code.

```
// Object function on Order_c object
// BigDecimal computeTaxForOrderInCountry(Closure taxStrategyFunction) {
BigDecimal totalTax = 0
// Iterate over order line items and return tax using
// taxStrategyFunction closure passed in
def orderLines = OrderLinesCollection_c
orderLines.reset()
while (orderLines.hasNext()) {
    // Invoke taxStrategyFunction() passing current line's LineTotal_c
    // and the CountryCode_c field value from the owning Order_c object
    def currentLine = orderLines.next()
```

```
totalTax += taxStrategyFunction(currentLine.LineTotal_c,  
currentLine.Order_c.CountryCode_c)  
}  
return totalTax
```

This means the closure you pass to `computeTaxForOrderInCountry` must declare *both* parameters and give each a name as shown in the example below. Notice that the function body can contain multiple lines if needed.

```
def taxForTerritoryABC = { amount, countryCode ->  
  if (countryCode == 'IT') {  
    return amount * (amount < 25 ? 0.10 : 0.22)  
  }  
  else {  
    return amount * (amount < 50 ? 0.12 : 0.25)  
  }  
}
```

There's no requirement that you store the closure function in a local variable before you pass it into a function. You can pass the closure directly inline like this:

```
// Before Insert trigger on Order_c: Assign TotalTax_c  
// using a flat 0.22 tax regardless of countryCode  
TotalTax_c = computeTaxForOrderInCountry( { amount, country -> return 0.22 } )
```

In this situation, to further simplify the syntax, Groovy allows omitting the extra set of surrounding parentheses like this:

```
TotalTax_c = computeTaxForOrderInCountry{ amount, country -> return 0.22 }
```

Many built-in collection functions — described in more details in the following sections — accept a closure to accomplish their job. For example, the `findAll()` function shown below finds all email addresses in the list that end with the `.edu` suffix.

```
def recipients = ['sjc@example.edu', 'dan@example.com',  
'spm@example.edu', 'jim@example.org']  
def eduAddresses = recipients.findAll{ it?.endsWith('.edu') }
```

Finally, in order to define a closure that accepts *no* parameters and should raise an error if any parameter is passed to it, you must use the arrow notation without mentioning any parameters on the left side of the arrow like this:

```
def logCurrentTime = { -> println("Current time is ${now()}") }
```

Some later code that invokes this closure by name by appending parentheses like this will succeed because it is passing no arguments:

```
// Invoke the closure's function body with no arguments  
logCurrentTime()
```

However, an attempt to pass it an argument will fail with an error:

```
// This will FAIL because the closure demands no arguments!  
logCurrentTime(123)
```

Working More Cleverly with Collections

Business logic frequently requires working with collections of values. This section explains the most useful functions you can use to work with your collections to keep code clean, readable, and easy to understand.

Finding Items in a Collection

To find all items matching a condition in a collection, use the `findAll()` function. It accepts a boolean closure identifying the items you're looking for.

The result is a `List` of all items in the collection for which the closure evaluates to `true`. If no item matches or the collection is empty, then an empty collection is returned.

As shown below, you can leave off the parentheses if passing the closure in-line. The result of this example is a list containing all recipient emails whose address ends with the `.edu` suffix:

```
def recipients = ['sjc@example.edu', 'dan@example.com',  
  'spm@example.edu', 'jim@example.org']  
// Pass boolean closure using implicit "it" parameter with find criteria  
// (using safe-navigation operator in case any element is null)  
def eduAddresses = recipients.findAll { it?.endsWith('.edu') }
```

When applied to a `List` of `Map` objects, your closure can reference the current map's keys by name as shown below. This example produces a list of phone book entries having a phone number that starts with the country code "+39-" for Italy.

```
def phonebook = [  
  [name: 'Steve', phone: '+39-123456789'],  
  [name: 'Joey', phone: '+1-234567890'],  
  [name: 'Sara', phone: '+39-345678901'],  
  [name: 'Zoe', phone: '+44-456789123']  
]  
def italianFriends = phonebook.findAll { it?.phone?.startsWith('+39-') }
```

If you call `findAll()` on a `Map`, then the parameter passed to the closure on each evaluation is the current `Map` entry. Each entry has a `key` and `value` property you can reference in the closure function body if necessary. The result is a `Map` containing only the entries for which the closure evaluates to `true`. In the example below, the result is a map containing the two users' map entries whose `name` is `Steve`.

```
def users = [  
  'smuench': [name: 'Steve', badge: 'A123'],  
  'jevans': [name: 'Joe', badge: 'B456'],  
  'sburns': [name: 'Steve', badge: 'C789']  
]  
def usersNamedSteve = users.findAll { it?.value.name == 'Steve' }
```

To find only the *first* matching item, use the `find()` function instead of `findAll()`. It accepts the same boolean closure but stops when the first match is identified. Note that in contrast to `findAll()`, when using `find()` if no item matches the predicate or the collection was empty to begin with then `null` is returned.

Companion functions exist to perform other searching operations like:

- `any { boolean_predicate }` — returns `true` if `boolean_predicate` returns `true` for any item
- `every { boolean_predicate }` — returns `true` if `boolean_predicate` returns `true` for every item

Generating One Collection from Another

You can use the `collect()` function to produce a new collection from an existing collection. The resulting one contains the results of evaluating a closure for each element in the original collection.

In the example below, the `uppercasedNames` collection is a list of the uppercase `name` property values of all the map entries in the phone book.

```
def phonebook = [  
  [name: 'Steve', phone: '+39-123456789'],  
  [name: 'Joey', phone: '+1-234567890'],  
  [name: 'Sara', phone: '+39-345678901'],  
]
```

```
[name: 'Zoe', phone: '+44-456789123']  
]  
def uppercasedNames = phonebook.collect { it?.name?.toUpperCase() }
```

You can combine collection functions in a chain to *first* filter then collect results of only the matching entries. For example, the code below produces a list of the values of the `name` property of `phonebook` entries with an Italian phone number.

```
// First filter phonebook collection, then collect the name values  
def italianNames = phonebook.findAll { it?.phone?.startsWith('+39-') }  
    .collect { it?.name }
```

Sorting Items in a Collections

To sort the items in a collection, use the `sort()` function. If the collection is a simple list then its items will be sorted ascending by their natural ordering.

For example, this line will sort the list of names in alphabetical order. The collection you invoke it on is updated to reflect the sorted ordering:

```
def names = ['Zane', 'Jasmine', 'Abigail', 'Adam']  
names.sort()
```

For a list of maps, if you want to sort on the value of a particular map property, pass a closure that returns the property to use for sorting. The following example shows how to sort a `users` collection based on the number of `accesses` a user has made.

```
def users = [  
    [userid: 'smuench', name: 'Steve', badge: 'A123', accesses: 135],  
    [userid: 'jevans', name: 'Joe', badge: 'B456', accesses: 1001],  
    [userid: 'sburns', name: 'Steve', badge: 'C789', accesses: 52]  
]  
// Sort the list of maps based on the accesses property of each map  
users.sort { it.accesses }
```

For a map of maps, the approach is similar but since the closure is passed a map entry key/value pair, this use case requires accessing the `value` property of the map entry before referencing its `accesses` property as shown here.

```
def users = [  
    'smuench': [name: 'Steve', badge: 'A123', accesses: 135],  
    'jevans': [name: 'Joe', badge: 'B456', accesses: 1001],  
    'sburns': [name: 'Steve', badge: 'C789', accesses: 52]  
]  
// Sort the map of maps based on the accesses property of map entry's value  
users.sort { it.value.accesses }
```

If you need more control over the sorting, you can pass a closure that accepts two parameters and returns:

- 0 — if they are equal
- -1 — if the first parameter is less than the second parameter
- 1 — if the first parameter is greater than the second parameter

The simplest way to implement a comparative closure is to use the Groovy "compare to" operator (`<=>`). In the example below, the two-parameter closure uses this operator to return the appropriate integer based on comparing the value of the `accesses` property of the first map entry's value with the corresponding value of the same property on the second map entry's value.

```
// Sort map of maps by comparing the accesses property of map entry's value  
users.sort { a, b -> a.value.accesses <=> b.value.accesses }
```

To reverse the sort order to be *descending* if needed, simply swap the roles of the two parameters passed to the closure. For example, to sort the user list descending by number of accesses, as shown below, swap the `a` and `b` parameters on the right side of the arrow:

```
// Sort map of maps DESCENDING by comparing the accesses property of map entry's value
users.sort { a, b -> b.value.accesses <=> a.value.accesses }
```

If your sorting needs are more complex, you can implement the comparative closure in any way you need to, so long as it returns one of the three expected integer values.

Grouping Items in a Collection

To group items in a collection, use the `groupBy()` function, providing a closure to evaluate as the grouping key.

For example, given a list of words you can group them based on the length of each word by doing the following:

```
def words = ['For', 'example', 'given', 'a', 'list', 'of', 'words', 'you', 'can',
'group', 'them', 'based', 'on', 'the', 'length', 'of', 'each', 'word']
def groupedByLength = words.groupBy{ it.length() }
```

This produces the following result of type `Map of List`:

```
[
  3:['For', 'you', 'can', 'the'],
  7:['example'],
  5:['given', 'words', 'group', 'based'],
  1:['a'],
  4:['list', 'them', 'each', 'word'],
  2:['of', 'on', 'of'],
  6:['length']
]
```

To produce a *count* of the number of items in each group, use the `countBy()` function, passing the same kind of closure to determine the grouping key:

```
def countsByLength = words.countBy{ it.length() }
```

This produces a map with the word lengths as the map key and the count as the value:

```
[3:4, 7:1, 5:4, 1:1, 4:4, 2:3, 6:1]
```

You can group and sort any collection as needed. For example, after grouping and counting the list of words above, you can group the resulting map into further groups based on whether the words have an even number of characters or an odd number of characters like this:

```
def evenOdd = countsByLength.groupBy{ it.key % 2 == 0 ? 'even' : 'odd' }
```

This produces a map of maps like this:

```
[odd:[3:4, 7:1, 5:4, 1:1],
 even:[4:4, 2:3, 6:1]]
```

These functions can be chained so you can produce a sorted list of words containing less than three letters and the count of their occurrences by doing:

```
def shortWordCounts = words.findAll{ it.length() < 3 }
    .countBy{ it }
    .sort{ it.key }
```

The code is compact and easy to understand, but if you want to rename the closure parameters to make them even more self-documenting:

```
def shortWordCounts =
```

```
words.findAll{ word -> word.length() < 3 }  
.countBy{ word -> word  
.sort{ wordCountMapEntry -> wordCountMapEntry.key }
```

For the final flourish, you could consider even adding additional comments like this:

```
def shortWordCounts =  
    // Find words less than 3 characters  
    words.findAll{ word -> word.length() < 3 }  
    // Then count how many times each resulting word occurs  
    .countBy{ word -> word }  
    // Then sort alphabetically by word  
    .sort{ wordCountMapEntry -> wordCountMapEntry.key }
```

This produces the desired result of:

```
[a:1, of:2, on:1]
```

Computing Aggregates Over a Collection

You can easily compute the count, sum, minimum, or maximum of items in a collection. This section describes how to use these four collection functions.

Computing the Count of Items in a Collection

To determine the number of items in a collection call its `size()` function. However, if you need to count a subset of items in a collection based on a particular condition, then use `count()`.

If you provide a single value, it returns a count of occurrences of that value in the collection. For example, the following use of `count('bbb')` returns the number 2.

```
def list = ['aa','bbb','cccc','defgh','bbb','aa','defgh','defgh']  
// If there are two or more 'bbb' then do something...  
if (list.count('bbb') >= 2){ /* etc. */ }
```

The `count()` function also accepts a boolean closure identifying which items to count. For example, to count the strings in a list whose lengths are an even number of characters, use code like the following. The count reflects the items for which the closure evaluates to `true`.

```
def list = ['aa','bbb','cccc','defgh','bbb','aa','defgh','defgh']  
def numEvenLengths = list.count{ it.length() % 2 == 0 }
```

To partition the collection into distinct groups by a grouping expression and then count the number of items in each group, use the `countBy()` function. It takes a closure that identifies the grouping key before computing the count of the items in each group. For example, to count the number of occurrences of items in the list above, use:

```
def entriesAndCounts = list.countBy{ it }
```

This will produce a resulting map like this:

```
[aa:2, bbb:2, cccc:1, defgh:3]
```

If you want to sort the result descending by the number of occurrences of the strings in the list, use:

```
def entriesAndCounts = list.countBy{ it }  
.sort{ a, b -> b.value <=> a.value }
```

Which produces the map:

```
[defgh:3, aa:2, bbb:2, cccc:1]
```

If you only care about the map entry containing the word that occurred the most frequently and its count of occurrences, then you can further chain the unqualified `find()` function that returns the first element.

```
def topWord = list.countBy{ it }
    .sort{ a, b -> b.value <=> a.value }
    .find()
println "Top word '${topWord.key}' appeared ${topWord.value} times"
```

Computing the Minimum of Items in a Collection

To determine the minimum item in a collection call its `min()` function with no arguments.

However, if you need to find the minimum from a subset of items in a collection based on a particular condition, then pass a closure to `min()` that identifies the expression for which to find the minimum value. For example, to find the minimum item in the following list of users based on the number of accesses they've made to a system, do the following:

```
def users = [
    'smuench': [name: 'Steve', badge: 'A123', accesses: 135],
    'sburns': [name: 'Steve', badge: 'C789', accesses: 52],
    'qbronson': [name: 'Quello', badge: 'Z231', accesses: 52],
    'jevans': [name: 'Joe', badge: 'B456', accesses: 1001]
]
// Return the map entry with the minimum value based on accesses
def minUser = users.min { it.value.accesses }
```

The `min()` function returns the *first* item having the minimum `accesses` value of 52, which is the map entry corresponding to `sburns`. However, to return all users having the minimum value requires first determining the minimum value of accesses and then finding all map entries having that value for their `accesses` property. This code looks like:

```
// Find the minimum value of the accesses property
def minAccesses = users.min { it.value.accesses }.value.accesses
// Return all map entries having that value for accesses
def usersWithMinAccesses = users.findAll { it.value.accesses == minAccesses }
```

There is often more than one way to solve a problem. Another way to compute the minimum number of accesses would be to first `collect()` all the `accesses` values, then call `min()` on that collection of numbers. That alternative approach looks like this:

```
// Find the minimum value of the accesses property
def minAccesses = users.collect { it.value.accesses }.min()
```

Using either approach to find the minimum accesses value, the resulting map produced is:

```
[
    sburns: [name: 'Steve', badge: 'C789', accesses: 52],
    qbronson: [name: 'Quello', badge: 'Z231', accesses: 52]
]
```

If the collection whose minimum item you seek requires a custom comparison to be done correctly, then you can pass the same kind of two-parameter comparative closure that the `sort()` function supports.

Computing the Maximum of Items in a Collection

To determine the maximum item in a collection call its `max()` function with no arguments.

However, if you need to find the maximum from a subset of items in a collection based on a particular condition, then pass a closure to `max()` that identifies the expression for which to find the maximum value. For example, to find the maximum item in the following list of users based on the number of accesses they've made to a system, do the following:

```
def users = [
  'smuench':[name:'Steve', badge:'A123', accesses: 1001],
  'sburns':[name:'Steve', badge:'C789', accesses: 52],
  'qbronson':[name:'Quello', badge:'Z231', accesses: 152],
  'jevans':[name:'Joe', badge:'B456', accesses: 1001]
]
// Return the map entry with the maximum value based on accesses
def maxUser = users.max { it.value.accesses }
```

The `max()` function returns the *first* item having the maximum `accesses` value of 1001, which is the map entry corresponding to `smuench`. However, to return all users having the maximum value requires first determining the maximum value of `accesses` and then finding all map entries having that value for their `accesses` property. This code looks like:

```
// Find the maximum value of the accesses property
def maxAccesses = users.max { it.value.accesses }.value.accesses
// Return all map entries having that value for accesses
def usersWithMaxAccesses = users.findAll{ it.value.accesses == maxAccesses }
```

There is often more than one way to solve a problem. Another way to compute the maximum number of accesses would be to first `collect()` all the `accesses` values, then call `max()` on that collection of numbers. That alternative approach looks like this:

```
// Find the maximum value of the accesses property
def maxAccesses = users.collect{ it.value.accesses }.max()
```

Using either approach to find the maximum accesses value, the resulting map produced is:

```
[
  smuench:[name:Steve, badge:A123, accesses:1001],
  jevans:[name:Joe, badge:B456, accesses:1001]
]
```

If the collection whose maximum element you seek requires a custom comparison to be done correctly, then you can pass the same kind of two-parameter comparative closure that the `sort()` function supports.

Computing the Sum of Items in a Collection

To determine the sum of items in a collection call its `sum()` function with no arguments. This works for any items that support a plus operator.

For example, you can sum a list of numbers like this to produce the result 1259.13:

```
def salaries = [123.45, 678.90, 456.78]
// Compute the sum of the list of salaries
def total = salaries.sum()
```

However, since strings also support a *plus* operator, it might surprise you that the following also works to produce the result `vincentvanGogh`:

```
def names = ['Vincent', 'van', 'Gogh']
def sumOfNames = names.sum()
```

If you need to find the sum of a subset of items in a collection based on a particular condition, then first call `findAll()` to identify the subset you want to consider, then `collect()` the value you want to sum, then finally call `sum()` on that collection. For example, to find the sum of all accesses for all users with over 100 accesses, do the following to compute the total of 2154:

```
def users = [
  'smuench':[name:'Steve', badge:'A123', accesses: 1001],
  'sburns':[name:'Steve', badge:'C789', accesses: 52],
  'qbronson':[name:'Quello', badge:'Z231', accesses: 152],
  'jevans':[name:'Joe', badge:'B456', accesses: 1001]
]
```

```
]
// Compute sum of all user accesses for users having more than 100 accesses
def total = users.findAll{ it.value.accesses > 100 }
    .collect{ it.value.accesses }
    .sum()
```

Joining Items in a Collection

To join the items in a collection into a single string, use its `join()` function as shown below, passing the string you want to be used as the separator between list items.

```
def paths = ['/bin', '/usr/bin', '/usr/local/bin']
// Join the paths in the list, separating by a colon
def pathString = recipients.join(':')
```

The result will be the string:

```
/bin:/usr/bin:/usr/local/bin
```

Using Optional Method Arguments

Using optional, named method arguments on your helper functions can make your code easier to read and more self-documenting. For example, consider a global helper function `queryRows()` that simplifies common querying use cases.

Sometimes your calling code only requires a `select` list and a `from` clause:

```
def rates = adf.util.queryRows(select: 'FromCurrency,ToCurrency,ExchangeRate',
    from: 'DailyRates_c')
```

On other occasions, you may need a `where` clause to filter the data and an `orderBy` parameter to sort it:

```
def euroRates = adf.util.queryRows(select: 'FromCurrency,ToCurrency,ExchangeRate',
    from: 'DailyRates_c',
    where: "FromCurrency = 'EUR'",
    orderBy: 'ExchangeRate desc')
```

By using optional, named arguments, your calling code specifies only the information required and clarifies the meaning of each argument. To adopt this approach, use a single parameter of type `Map` when defining your function:

```
// Global Function
List queryRows(Map options)
```

Of course, one way to call the `queryRows()` function is to explicitly pass a `Map` as its single argument like this:

```
// Passing a literal Map as the first argument of queryRows()
def args = [select: 'FromCurrency,ToCurrency,ExchangeRate',
    from: 'DailyRates_c']
def rates = adf.util.queryRows(args)
```

You can also pass a literal `Map` inline without assigning it to a local variable like this:

```
// Passing a literal Map inline as the first argument of queryRows()
def rates = adf.util.queryRows([select: 'FromCurrency,ToCurrency,ExchangeRate',
    from: 'DailyRates_c'])
```

However, when passing a literal `Map` directly inside the function call argument list you can omit the square brackets. This makes the code easier to read:

```
// Passing a literal Map inline as the first argument of queryRows()
// In this case, Groovy allows removing the square brackets
def rates = adf.util.queryRows(select: 'FromCurrency,ToCurrency,ExchangeRate',
```

```
from: 'DailyRates_c')
```

The `Map` argument representing your function's optional parameters must be first. If your function defines *additional* parameters, then when calling the function, pass the values of the other parameters first *followed by* any optional, named parameters you want to include. For example, consider the signature of following `findMatchingOccurrences()` object function that returns the number of strings in a list that match a search string. The function supports three optional `boolean` parameters `caseSensitive`, `expandTokens`, `useRegExp`.

```
Long findMatchingOccurrences(Map options, List stringsToSearch, String searchFor)
```

Calling code passes optional, named arguments *after* values for `stringsToSearch` and `searchFor` as shown below:

```
// Use an object function to count how many emails
// are from .org or .edu sites
def nonCommercial = findMatchingOccurrences(emails, '.*.org|.*.edu',
    caseSensitive: true,
    useRegExp: true)
```

Regardless of the approach the caller used to pass in the key/value pairs, your function body works with optional, named arguments as entries in the leading `Map` parameter. Be aware that if no optional argument is included, then the leading `Map` parameter evaluates to `null`. So assume the `options` parameter might be `null` and handle that case appropriately.

Your code should validate incoming optional arguments and, where appropriate, provide default values for options the caller did not explicitly pass in. The example below shows the opening lines of code for the `queryRows()` global function. Notice it uses the safe-navigation operator (`?.`) when referencing the `select` property of the `options` parameter just in case it might be `null` and signals an error using another global function named `error()`.

```
// Global Function: List queryRows( Map options )
// -----
// The options Map might be null if caller passes no named parameters
// so check uses the safe-navigation operator to gracefully handle the
// options == null case, too. We're assuming another global helper function
// named 'error()' exists to help throw exception messages.
if (!options?.select) {
    adf.util.error("Must specify list of field names in 'select' parameter")
}
if (!options?.from) {
    adf.util.error("Must specify object name in 'from' parameter")
}
// From here, we know that some options were supplied, so we do not
// need to continue using the "?.\" operator when using options.someName
def vo = newView(options.from)
// etc.
```

Simplifying Business Logic Queries

Querying data is a frequent task your business logic will perform. This section explains how to define a set of global helper functions to simplify the job of performing database queries.

It also illustrates examples of using the new set of functions.

Using Query Helper Functions

After defining the global functions in the following section, your business logic will be able to easily perform query tasks using a syntax that evokes the structure of a SQL select statement.

For example, to query the email, last name, and first name from the `StaffMember` business object's table where the job code is Sales Representative, you'll be able to write code like this:

```
def salesReps = adf.util.queryMaps(select: 'Email,LastName,FirstName',
    from: 'StaffMember',
    where: 'JobId = :JobCode',
    orderBy: 'LastName,FirstName',
    binds: [JobCode: 'SA_REP'])
```

Only the `select` and `from` named parameters are required. You specify the others only when you need to use them. The `queryMaps()` function returns a `List` of `Map` objects, each containing values for only the fields you've mentioned in the `select` parameter.

If you need to retrieve rows to update them as part of your business logic, then *instead* of `queryMaps()` use the `queryRows()` method as shown below. The result is a `List` of `Row` objects whose attributes are updatable and which will be validated and saved along with any other modified objects in the current transaction. Notice that since the result is a `List` we can use it directly in a `for` loop. Also note that we've included the `Salary` column in the `select` list that our code intends to update inside the loop.

```
// Update the salary of sales reps to increase it by 5 percent
for (curRep in adf.util.queryRows(select: 'Salary',
    from: 'StaffMember',
    where: 'JobId = :JobCode',
    binds: [JobCode: 'SA_REP'])) {
    // Round the salary to two digits after increasing by 5%
    curRep.Salary = (curRep.Salary * 1.05 as Double).round(2)
}
```

If your `where` clause will identify a single row, or if you only care to retrieve the first row of the result, then use the companion single-row-return functions:

- `Map queryMap(Map options)` — returns first query result row as a `Map`, or `null` if no row was returned
- `Object queryRow(Map options)` — returns first query result row, or `null` if no row was returned

If you only care to compute the total number of rows that would be returned by the query, use the `queryCount()` function. However, keep in mind that if your goal is to test for the existence of a single row it is more efficient to use `queryMap()` or `queryRow()` and test whether its return value is not `null`.

The complete list of named parameters the query methods support includes the following. All places that mention business object names and field names are *case-sensitive*.

- `select` — comma-separated list of business object field names (*string, required*)
- `from` — name of business object (*string, required*)
- `where` — view criteria filter predicate (*string*, optionally referencing bind variables)
- `orderBy` — comma-separated list of field names (*string*, optionally suffixed by " `desc`" for descending)
- `binds` — pairs of bind variable names/values referenced in the `where` predicate (*Map*)
- `ignoreNullBinds` — [*Boolean*] set to `true` to ignore `where` predicate elements involving null bind values

In addition to the user-defined by variables, your `where` clause can reference any of these built-in bind variable names:

- `SysUser` — name of the currently-logged-in user, or `anonymous` otherwise (*String*)
- `SysToday` — current date (*Date*)
- `SysNow` — current date and time (*Datetime*)

For example, to query the list of currency exchange rates for the current date, you could write a query like this. Since the use case expects to return a single row and is not planning to modify the data, it's using `queryMap()` to retrieve the single `Rate_c` attribute that the code requires to compute the converted currency value.

```
def rate = adf.util.queryMap(  
    select: 'Rate_c',  
    from: 'ExchangeRate_c',  
    where: 'From_c = :Base and To_c = :Other and Date_c = :SysToday',  
    binds: [Base: 'GBP', Other: 'EUR'])  
def convertedVal = sourceVal * rate.Rate_c
```

The helper methods `queryRows()`, `queryMaps()`, `queryRow()`, `queryMap()`, and `queryCount()` call the same core `query()` helper function in their implementation. This function centralizes:

- Validating that the `select` and `from` required options are present
- Creating a view object using `newView()` using the object name passed in the `from` parameter
- Tokenizing the `select` list field names and passing them to `selectAttributesBeforeQuery()`
- Defining any user-supplied bind variables if a `where` and `binds` parameter are supplied
- Applying the view criteria filter expression if a `where` parameter is supplied
- Setting the values of any system and/or user-supplied bind variables
- Returning the view object

If your code needs to re-execute the same query multiple times with different values for its bind variables, use the core `query()` function that returns a view object. You can iterate the results yourself, reassign appropriate bind variables, and then execute the view object again without creating multiple, distinct view objects. This technique is important to avoid a runtime exception for using too many view objects in a single trigger or object function.

For example, consider the following code that iterates over an unknown number of uncleared transaction records and for each one queries the exchange rate into the target currency. It uses the `queryMap()` function inside a loop. This approach creates one new view object for each loop iteration. If the number of rows being iterated is unpredictably large, this technique can produce a runtime resource exception when it hits the upper limit on number of view objects that can be created in a single trigger or function.

```
// Will be updating the queried rows, so use queryRows()  
for (txn in adf.util.queryRows(select: 'Id,Cleared_c,Currency_c,Amount_c,Date_c',  
    from: 'Transaction_c',  
    where: "Cleared_c = 'N'")) {  
    def rate = 1  
    // If transaction currency is different than GBP, lookup historical  
    // exchange rate for the date of the transaction to convert the  
    // transaction currency into GBP  
    if (txn.Currency_c != 'GBP') {  
        // NOT BEST PRACTICE: Using a query inside a loop!  
        rate = adf.util.queryMap(  
            select: 'Rate_c',  
            from: 'ExchangeRate_c',  
            where: "From_c = :Base and To_c = 'GBP' and Date_c = :ForDate",  
            binds: [Base: txn.Currency_c, ForDate: txn.Date_c ])?.Rate_c  
        )  
        if (rate) {  
            txn.Cleared_c = 'Y'  
            // Multiply original txn amount by rate and round to 2 decimal places  
            txn.AmountInGBP_c = (txn.Amount_c * rate as Double).round(2)  
        }  
    }  
}
```

Rather than using the query functions inside the body of a loop, the best practice is to call the core `query()` helper function. It returns a view object you can use over and over inside the loop. For each loop iteration, you set the bind variables to appropriate values and re-execute the query. This best-practice rewrite of the above routine is shown below.

```
// BEST PRACTICE: Create a reusable query for looking up the exchange rate using
// ~~~~~ bind variable values of the correct datatype. Inside the loop
// set the correct bind var values and execute the same view object over & over
def rateVO = adf.util.query(
    select: 'Rate_c',
    from: 'ExchangeRate_c',
    where: "From_c = :Base and To_c = 'GBP' and Date_c = :ForDate",
    binds: [Base: 'XXX', ForDate: today() ])
// Will be updating the queried rows, so use queryRows()
for (txn in adf.util.queryRows(select: 'Id,Cleared_c,Currency_c,Amount_c,Date_c',
    from: 'Transaction_c',
    where: "Cleared_c = 'N'")) {
    def rate = 1
    // If transaction currency is different than GBP, lookup historical
    // exchange rate for the date of the transaction to convert the
    // transaction currency into GBP
    if (txn.Currency_c != 'GBP') {
        // Set any bind variables to new values for current loop iteration
        setBindVariable(rateVO, 'Base', txn.Currency_c)
        setBindVariable(rateVO, 'ForDate', txn.Date_c)
        // Execute the same view object with the new bind variable values
        rateVO.executeQuery()
        rate = rateVO.first()?.Rate_c
    }
    if (rate) {
        txn.Cleared_c = 'Y'
        // Multiply original txn amount by rate and round to 2 decimal places
        txn.AmountInGBP_c = (txn.Amount_c * rate as Double).round(2)
    }
}
```

Defining Query Helper Functions

To use the query helper functions in your application, use Application Composer to define the following set of global functions in the order that they appear.

The order is important because later functions depend on calling the earlier ones in the list.

Global Function: `Map mapForRow(Object row, List selectList)`

```
def ret = null
if (row != null) {
    ret = [:]
    for (attrName in row.getAttributeNames()) {
        if (attrName in selectList) {
            ret[attrName] = row.getAttribute(attrName)
        }
    }
}
return ret as Map
```

Global Function: `void error(String message)`

```
throw new oracle.jbo.JboException(message)
```

Global Function: `Object query(Map options)`

```
// The options Map might be null if caller passes no named parameters
// so first two checks use the safe-navigation operator to gracefully
// handle the case when options == null
if (!options?.select) {
```

```
adf.util.error("Must specify list of field names in 'select' parameter")
}
if (!options?.from) {
    adf.util.error("Must specify object name in 'from' parameter")
}
// From here, we know that some options were supplied, so we do not
// need to continue using the "?." operator when using options.someName
def sysBindVars = [
    [name:"SysUser", type:"Text",
     defaultValue: {adf.context.getSecurityContext().getUserName()?:'anonymous'}],
    [name:"SysToday", type:"Date", defaultValue: {today()}],
    [name:"SysNow", type:"Datetime", defaultValue: {now()}]]
options.selectList = []
for (fieldNameWithWhitespace in options.select.split(',')) {
    def fieldName = fieldNameWithWhitespace.trim()
    options.selectList << fieldName
}
def vo = newView(options.from)
if (options.orderBy) {
    vo.setSortBy(options.orderBy)
}
if (options.where) {
    if (options.binds) {
        for (varName in options.binds.keySet()) {
            if (sysBindVars.find{ it.name == varName}) {
                adf.util.error("Cannot give a value for system bind variable ${varName}")
            }
            if (varName.startsWith('Sys')) {
                adf.util.error("${varName} is a bind variable reserved for future system use")
            }
            def value = options.binds[varName]
            def varType = 'Text'
            if (value) {
                if (value instanceof java.sql.Timestamp ||
                    value instanceof oracle.jbo.domain.Timestamp ) {
                    varType = 'Datetime'
                }
                else if (value instanceof java.util.Date ||
                    value instanceof oracle.jbo.domain.Date) {
                    varType = 'Date'
                }
                else if ( value instanceof java.lang.Number ||
                    value instanceof oracle.jbo.domain.Number) {
                    varType = 'Number'
                }
            }
            addBindVariable(vo,varName,varType)
        }
    }
    for (var in sysBindVars) {
        if (options.where.contains(":${var.name}") ) {
            addBindVariable(vo, var.name, var.type)
        }
    }
    if (options.ignoreNullBinds) {
        vo.appendViewCriteria(options.where, (Boolean)options.ignoreNullBinds)
    }
    else {
        vo.appendViewCriteria(options.where)
    }
}
selectAttributesBeforeQuery(vo,options.selectList)
if (options.limit) {
    vo.setMaxFetchSize(options.limit as short)
}
if (options.where) {
```

```
if (options.binds) {
  for (varName in options.binds.keySet()) {
    setBindVariable(vo, varName, options.binds[varName])
  }
}
for (var in sysBindVars) {
  if (options.where.contains(":${var.name}")) {
    def defaultValueClosure = var.defaultValue
    def value = defaultValueClosure()
    setBindVariable(vo, var.name, value)
  }
}
}
return vo
```

Global Function: List queryRows(Map options)

```
def ret = []
def vo = adf.util.query(options)
vo.executeQuery()
while (vo.hasNext()) {
  ret << vo.next()
}
return ret
```

Global Function: Object queryRow(Map options)

```
def ret = []
def vo = adf.util.query(options)
vo.executeQuery()
while (vo.hasNext()) {
  ret << vo.next()
}
return ret
```

Global Function: Map queryMap(Map options)

```
def vo = adf.util.query(options)
vo.executeQuery()
return adf.util.mapForRow(vo.first(), options.selectList)
```

Global Function: List queryMaps(Map options)

```
def ret = []
def vo = adf.util.query(options)
vo.executeQuery()
while (vo.hasNext()) {
  ret << adf.util.mapForRow(vo.next(), options.selectList)
}
return ret
```

Global Function: Long queryCount(Map options)

```
def vo = adf.util.query(options)
vo.executeQuery()
return vo.getEstimatedRowCount()
```

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. 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. The example below shows how the steps fit together in practice.

```
// Access the view object for the custom TroubleTicket object
def vo = newView('TroubleTicket')
// Create the new row
def newTicket = vo.createRow()
// Set the problem summary
newTicket.ProblemSummary = 'Cannot insert floppy disk'
// Assign the ticket a priority
newTicket.Priority = 2
// 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 just 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 requirements
3. Assign new values to fields on this row 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 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 `revertRowAndContaineers()` method as follows on the row

```
yourRow.revertRowAndContaineers()
```

Understanding Why Using Commit or Rollback In Scripts Is Strongly Discouraged

By design you cannot commit or rollback the transaction from within your scripts. Any changes made by your scripts get committed or rolled-back along with the rest of the current transaction.

If your script code were allowed to call `commit()` or `rollback()`, this would affect all changes pending in the current transaction, not only those performed by your script and could lead to data inconsistencies.

Using the User Data Map

The application development 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 reading a user data map key executes as part of different web request from the code that put the value in the map, then write the code in a resilient way to correctly handle the situation when the expected value is not present 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:

```
// 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 *Using Groovy Maps and Lists with Web Services* 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:

```
// 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

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

Five built-in aggregate functions allow summarizing 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.

The table below provides a description and example of the supported functions.

Supported Aggregate Functions

Aggregate Function	Description	Example (in Context of TroubleTicket Parent Object)
avg	Average value of an expression	<code>ActivityCollection.avg('Duration')</code>
min	Minimum value of an expression	<code>ActivityCollection.min('Duration')</code>
max	Maximum value of an expression	<code>ActivityCollection.max('Duration')</code>
sum	Sum of the value of an expression	<code>ActivityCollection.sum('Duration')</code>
count	Count of rows having a non-null expression value	<code>ActivityCollection.count('Duration')</code>

Aggregate Function	Description	Example (in Context of TroubleTicket Parent Object)

Understanding Why Aggregate Functions Are Appropriate Only to Small Numbers of Child Rows

The aggregate functions described in this section compute their result by retrieving the rows of a child collection from the database and iterating through all of these rows in memory.

This has two important consequences:

- These aggregate functions should only be used when you know the number of rows in the child collection will be reasonably small.
- Your calculation may encounter a runtime error related to exceeding a fetch limit if the child collection's query retrieves more than 500 rows.

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:

```
// Use nvl() Function in aggregate expression
def avgDuration = ActivityCollection.min('nvl(Duration,0)')
```

Performing Conditional Counting

In the case of the `count()` function, if the expression evaluates to null then the row is not counted.

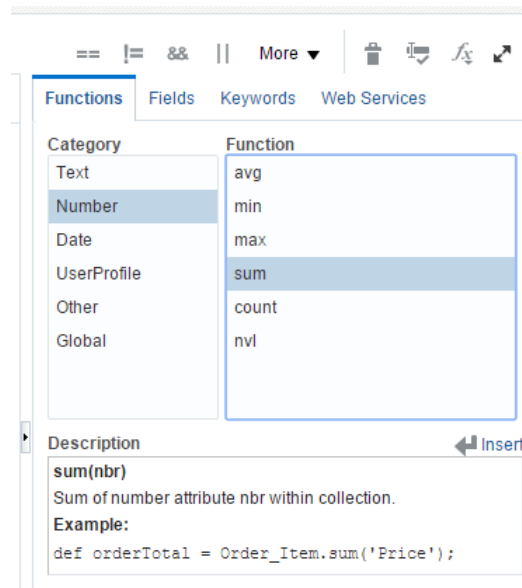
You can supply a conditional expression to the `count()` function which will count only the rows where the expression returns a non-null value.

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:

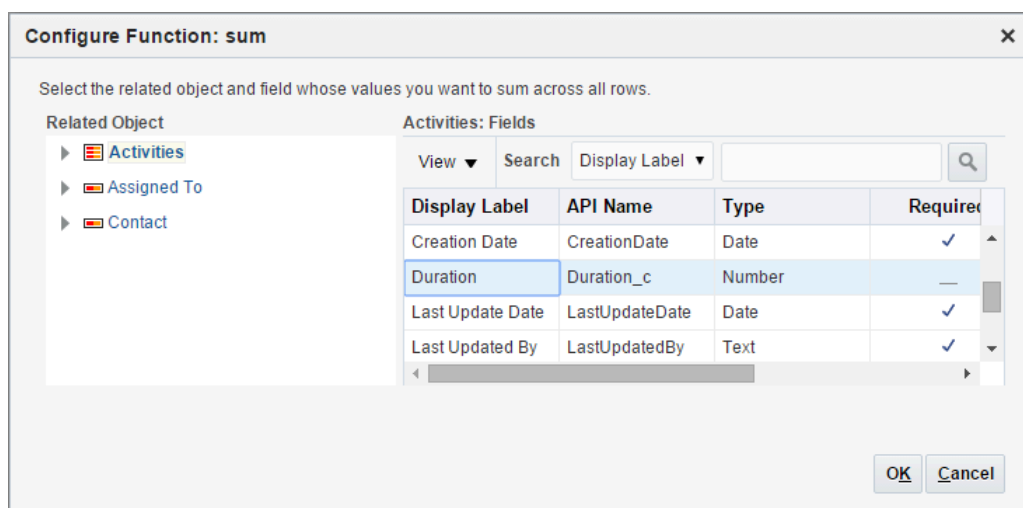
```
// Conditional expression returns non-null for rows to count
// Use the inline if/then/else operator to return 1 if the
// duration is over 0.5 hours, otherwise return null to avoid
// counting that the non-qualifying row.
def overHalfHourCount = ActivityCollection.count('nvl(Duration,0) > 0.5 ?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

Inserting an Aggregate Expression Using the Expression Palette

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



After selecting the desired function and clicking the (*Insert*) button, as shown in the following figure, 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 updatable and/or conditionally required expressions. If you fail to correctly configure this information, your application will not behave correctly at runtime.

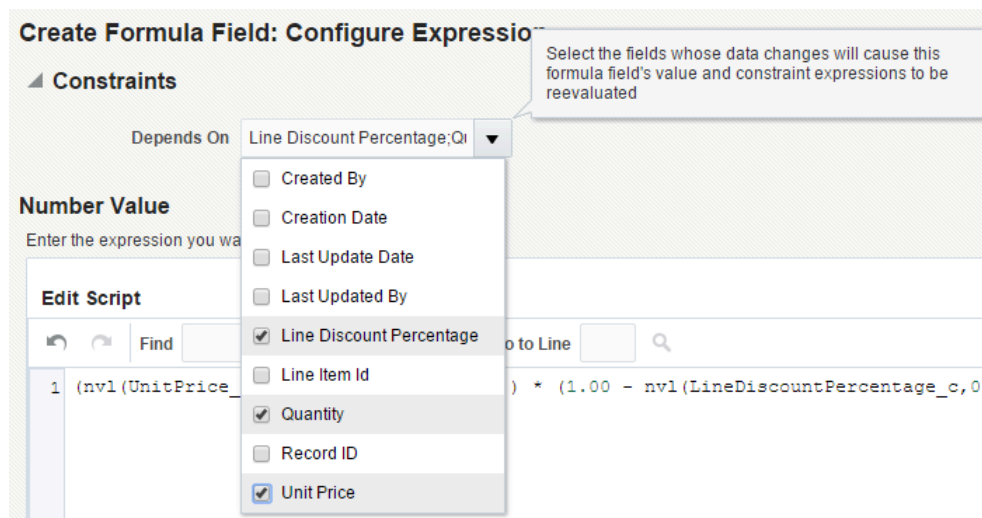
Configuring Depends On Fields for a Formula

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:

```
(nvl(UnitPrice_c,0) * nvl(Quantity_c,0)) * (1.00 - 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, the figure below 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 updatable expression or a conditionally required expression.

In the 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 updatable 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 updatable 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 updatable or not. The figure below 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.

Depends On: Fulfillment Center;Quantity

☐ Required

☒ Updatable

☒ Searchable

List of Values

Configure the list of values you want to display in the choice list. Click the search icon to search for values.

* **Lookup Type**: Warehouse Codes

☒ Constrain list by parent field value selection

Select the parent field whose value selection will drive the contents of this field, and the parent lookup type.

* **Parent Choice List**: Fulfillment Center

Parent Lookup Type:

* **Value Map**

Dropdown menu items:

- ☐ Created By
- ☐ Creation Date
- ☒ Fulfillment Center
- ☐ Last Update Date
- ☐ Last Updated By
- ☐ Line Discount Percentage
- ☐ Line Item Id
- ☐ Line Total
- ☒ Quantity
- ☐ Record ID
- ☐ Unit Price

Enforcing Conditional Updateability of Custom Fields for Web Service Access

Any conditional updatability 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 Updatability 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, 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 updatability 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`
- **Error Message:** `A warehouse can be specified only for quantities over five`

Rule Body

```
// 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
}
```

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 isn't 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 updatable expression for the new `LineEditedCount` with the expression:

```
false
```

This will cause the user interface to always see that the field isn't updatable, and the value will only be updatable from script. Note that configuring the conditionally updatable expression to always return `false` is semantically different from unchecking the *Updatable* checkbox. Doing the latter, your field would never be updatable (neither from the user interface nor from script). Using the conditionally updatable expression, the updatability enforcement is done only at the user interface level.

Finally, to derive the value of the `LineEditedCount` you would add the following trigger:

- **Trigger Object:** `OrderLineItem`
- **Trigger:** *Before Update In Database*
- **Trigger Name:** `Before_Update_Adjust_Edited_Count`

Trigger Definition

```
adf.util.logStart('Before_Update_Adjust_Edited_Count')
// Get the original value of the LineEditedCount field
def origCount = getOriginalAttributeValue('LineEditedCount_c')
def newCount = origCount + 1
// Only assign the value if it's not already what we want it to be
if (LineEditedCount_c != newCount) {
    LineEditedCount_c = newCount
}
```

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 field assignments or 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 field assignments or `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 assigns the field's value to your calculated value.

See *Testing Whether a Field's Value Is Changed* 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 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`
- **Error Message:** `A contact cannot be his/her 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
```

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 the following example.

```
// 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.

For example, assuming an object has a field named `status`, then consider the following object validation script:

```
// Assuming current object has a Status field, define local variable of the same name
def Status = '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 local variable takes precedence
// ----- so the Status field value is not used!
//
if (Status == 'Open') {
    Status = 'Pending'
}
```

At the top of the example, a variable named `status` is defined. After pages full of complex code, later in the script the author references the custom field named `status` without remembering that there is also a local variable named `status` defined above. Since the local variable named `status` will always take precedence, the script will never enter into the conditional block here, regardless of the current value of the `status` 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:

```
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 Lists](#) and [Working with Maps](#)

Registering a Web Service Connection

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:

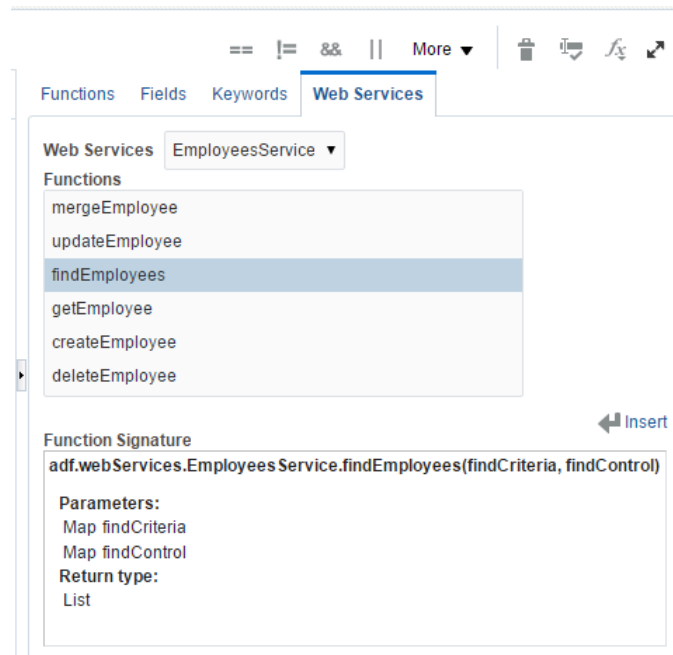
```
http://example.com:8099/Services/EmployeeService?WSDL
```

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 the figure below 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:

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

For instance, the example below 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.

```
// 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 the following example 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.

```
// 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 this example 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.

```
// 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)
```

If the web services you need to invoke expect or return data in base64 encoded format, you can use the base64-related helper functions explained in [Understanding Additional Built-in Groovy Functions](#).

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 start 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 the following example 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 white spaces used in these examples are purely to improve readability and assist with visually matching the square brackets.

```
// 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 the example below 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 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']]
                    ]
                ]
            ]
        ]
    ]
]
// 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 this example extends the one 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 criteria from the previous example, 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 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']]
          ]
        ]
      ],
      [
        item:
        [
          [
            upperCaseCompare : true,
            attribute : 'Ename',
            operator  : '=',
            value    : [[item:'allen']]
          ]
        ]
      ]
    ]
  ]
]
// 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 the example below enhances the original one above 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 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)
}
```

The script below extends the previous one 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 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 shown here 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 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)
}
```

Understanding View Criteria Item Operators

When building view criteria filters for use in a web service call, the following table provides the list of valid values that you can provide as an operator in a view criteria item.

Notice that some operators are valid only for fields of certain data types. For example, for a field {**Priority** of type number, you can use the **>=** operator, but in contrast, for a view criteria item related to a **ResolutionDate** field of type Date, you would use the **ONORAFTER** operator.

View Criteria Item Operators (Case-Sensitive!)

Operator	Description	Valid for Field Types
=	Equals	Number, Text, Date
<>	Not equals	Number, Text, Date
ISBLANK	Is null (no view criteria item value required)	Number, Text, Date
ISNOTBLANK	Is not null (no view criteria item value required)	Number, Text, Date
LIKE	Like	Text
STARTSWITH	Starts with	Text
>	Greater than	Number, Text
AFTER	Date comes after	Date
>=	Greater than or equal to	Number, Text
ONORAFTER	Date is on or comes after	Date
<	Less than	Number, Text
BEFORE	Date comes before	Date
<=	Less than or equal to	Number, Text
ONORBEFORE	Date is on or comes before	Date

Operator	Description	Valid for Field Types
BETWEEN	Is between (two view criteria item values required)	Number, Text, Date
NOTBETWEEN	Is not between (two view criteria item values required)	Number, Text, Date

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:

```
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:

```
// 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`.

Formatting Numbers and Dates Using a Formatter

Groovy provides the `Formatter` object that you can use in a text formula expression or anywhere in your scripts that you need for format numbers or dates.

The general pattern for using a `Formatter` is to first construct a new instance like this, passing the the expression for the current user's locale as an argument:

```
def fmt = new Formatter(adf.context.locale)
```

This `Formatter` object you've instantiated will generally be used to format a *single*, non-null value by calling its `format()` method like this:

```
def ret = fmt.format( formatString , arg1 [, arg2 , ..., argN ] )
```

Note that if you call the `format()` method of the same `Formatter` object multiple times, then the results are concatenated together. To format several distinct values without having their results be concatenated, instantiate a new `Formatter` for each call to a `format()` method.

The format string can include a set of special characters that indicate how to format each of the supplied arguments. Some simple examples are provided below, however the complete syntax is covered in the [documentation for the `Formatter` class](#).

Example of Formatting a Number Using a Formatter

To format a number `numberVal` as a floating point value with two (2) decimal places and thousands separator you can do:

```
Double dv = numberVal as Double
def fmt = new Formatter(adf.context.locale)
def ret = (dv != null) ? fmt.format('%,.2f', dv) : null
```

If the value of `numberVal` were 12345.6789, and the current user's locale is US English, then this would produce a formatted string like:

12,345.68

If instead the current user's locale is Italian, it would produce a formatted string like:

12.345,68

To format a number `numberVal` as a floating point value with three (3) decimal places and no thousands separator you can do:

```
Double dv = numberVal as Double
def fmt = new Formatter(adf.context.locale)
def ret = (dv != null) ? fmt.format('%f', dv) : null
```

If the value of `numberVal` were 12345.6789, and the current user's locale is US English, then this would produce a formatted string like:

12345.679

To format a number value with no decimal places to have a zero-padded width of 8, you can do:

```
Long lv = numberVal as Long
def fmt = new Formatter(adf.context.locale)
def ret = (lv != null) ? fmt.format('%08d', lv) : null
```

If the value of `numberVal` were 5543, then this would produce a formatted string like:

00005543

Formatting a Date Using a Formatter

To format a datetime value `datetimeVal` to display only the hours and minutes in 24-hour format, you can do:

```
Date dv = datetimeVal as Date
def fmt = new Formatter(adf.context.locale)
def ret = (dv != null) ? fmt.format('%tH:%tM', dv, dv) : null
```

If the value of `datetimeVal` were 2014-03-19 17:07:45, then this would produce a formatted string like:

17:07

To format a date value `dateVal` to display the day of the week, month name, the day, and the year, you can do:

```
Date dv = dateVal as Date
def fmt = new Formatter(adf.context.locale)
def ret = (dv != null) ? fmt.format('%tA, %tB %te, %tY', dv, dv, dv, dv) : null
```

If the value of `dateVal` were 2014-03-19, and the current user's locale is US English, then this would produce a formatted string like:

Wednesday, March 19, 2014

Working with Field Values Using a Parameterized Name

When writing reusable code, if your object function needs to perform the same operations on different fields, you can parameterize the field name.

Start by defining a function parameter of type `string` whose value at runtime will be the name of a field in the current object. Then, when your code needs to access the value of the parameterized field, just call `getAttribute(fieldNameParam)`. To assign a new value to that field, call `setAttribute(fieldNameParam, newValue)`. In either case, if the value of the field name parameter passed in does not match the name of some field in the current object, a `NoDefException` will be thrown to signal an error.

Consider the following example of an object function named `conditionalIncrement()` that increments the value of the number field whose name is passed in only if the field's value is less than a maximum value also passed in:

```
// Object function: void conditionalIncrement(fieldName String, maxValue Long)
// -----
def fieldValue = getAttribute(fieldName)
if (fieldValue < maxValue) {
    setAttribute(fieldName, fieldValue + 1)
}
```

The first line defines a `fieldValue` variable to store the value of the field whose name is passed in. If its value is less than `maxValue`, then line three assigns the field a new value that is one greater than its current value. Once you define

an object function like `conditionalIncrement()`, then any Groovy scripts on the same object can invoke it, passing in appropriate argument values. For example, in one script suppose you need to increment the value of a field named `UsageCount` if its value is less than 500:

```
// Increment the usage count if it is less than 500
conditionalIncrement('UsageCount', 500)
```

In another script, imagine you need to increment the value of a `DocumentVersionNumber` field if its value is less than 1000. You can use the same object function: just pass in different values for the field name and maximum value parameters:

```
// Increment the document version number if it is less than 1000
conditionalIncrement('DocumentVersionNumber', 1000)
```

Of course the `getAttribute()` and `setAttribute()` functions can also accept a literal `string` value as their first argument, so you could theoretically write conditional logic like:

```
// Ensure document is not locked before updating request-for-approval date
// NOTE: more verbose get/setAttribute() approach
if (getAttribute('DocumentStatus') != 'LOCKED') {
    setAttribute('RequestForApprovalDate', today())
}
```

However, in the example above, when the name of the field being evaluated and assigned is not coming from a parameter or local variable, then it is simpler and more readable to write this equivalent code instead:

```
// Ensure document is not locked before updating request-for-approval date
// NOTE: More terse, elegant direct field name access
if (DocumentStatus != 'LOCKED') {
    RequestForApprovalDate = today()
}
```

When invoked on their own, the `getAttribute()` and `setAttribute()` functions operate on the current object. However, anywhere in your script code where you are working with a business object `Row`, you can also call these functions on that particular row as shown in the following example of an object function. Notice that it also parameterizes the name of the object passed to the `newView()` function:

```
// Object function: String getRowDescription(objectName String, displayFieldName String, id Long)
// -----
// Create a new view object to work with the business object whose name is
// passed in the objectName parameter
def view = newView(objectName)
// Find the row in that view whose key is given by the value of the id parameter
def rows = view.findByKey(key(id), 1)
// If we found exactly one row, return the value of the display field name on
// that row, whose field name is given by the value in the displayFieldName parameter
return rows.size() == 1 ? return rows[0].getAttribute(displayFieldName) : null
```

With such a function defined, we can invoke it from any script in the object to access the display field value of different objects we might need to work with:

```
// Get RecordName of the Task object with key 123456
def taskName = getRowDescription('Task', 'RecordName', 123456)
// Get the Name of the Territory object with key 987654
def optyName = getRowDescription('Territory', 'Name', 987654)
```

If you use the `getAttribute()` or `setAttribute()` to access field values on a related object, remember that the first argument must represent the name of a single field on the 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?.Status` is not the name of a single field on the current `Activity` object:

```
// Assume script runs in context of an Activity object (child of TroubleTicket)
// INCORRECT way to set a parent field's value using setAttribute()
setAttribute('TroubleTicket?.Status', 'Open')
```

Instead, first access the related object and store it in a local variable. Then you can assign a field on the related object as follows:

```
// Assume script runs in context of an Activity object (child object TroubleTicket)
// First access the parent object
def parentTicket = TroubleTicket
// Then call the setAttribute on that parent object
parentTicket?.setAttribute('Status', 'Open')
```

Determining the Object Type of a Row

When writing a global helper function, you may find it useful to accept a business object data row as an argument.

Inside your function, if you need to determine the object type of the row passed in, use the `objectName()` function.

For example, this might be useful if you need to create a new row of the same type as the row passed in, copying selected data from the existing row.

```
// Global Function:
// -----
// Object duplicateRow(existingRow Object,
// fieldNamesToCopy List)

// Find the name of the object for the existingRow
def rowObjectName = objectName(existingRow)

// In order to create a new row of the same type, we
// need a new view object for the right business object
// whose name we just determined above.
def vo = newView(rowObjectName)

// Setup a map to hold any attribute names whose values
// we want to copy from the existing row
def attrs = [:]

// If any fieldNamesToCopy were specified, prepare a
// map containing the field names and values from the
// existing row to use while creating the new row.
// Groovy elegantly handles the case where the
// fieldNamesToCopy might be null, keep code simple.
for (fieldName in fieldNamesToCopy)
{
    attrs[fieldName] = existingRow[fieldName]
}

// Now create and return a new row using that view object
// passing in the map of fields whose values we've been
// asked to copy from the existing row.
return vo.createAndInitRowFromMap(attrs)
```

Once you define the `duplicateRow()` global function, you can use it anywhere in your application where you need to duplicate an existing row, optionally copying along one or more attributes from an existing row. For example, as shown below, you can write the code in a *Before Update* trigger on an `Order_c` custom object that duplicates the existing `Order_c` object, copying just the fields `CustomerId_c` and `PrimaryShippingAddress_c` into the newly-created order:

```
// Before Update trigger on Order_c
// Call an object function to determine any backordered items
def backorderedItems = determineBackorderedItems()
if (backorderedItems)
{
    // ...
}
```

```
// Use adf.source to pass this current Order_c object
def newOrder = adf.util.duplicateRow(adf.source,
['CustomerId_c',
'PrimaryShippingAddress_c'])
// Call an object function to add backordered items
// to the newly-created order.
newOrder.addItem(backorderedItems)
// Call an object function to remove backordered items
// from the current order
removeItems(backorderedItems)
}
```

Of course, you can also pass any other `order_c` row as the first argument. In the example below, we've used a `queryRow()` global helper function to find the current customer's most recent shipped order that contains a specific product id, and pass that order if found to the `duplicateRow()` function.

```
// Assume custId variable holds current customer id
// and prodId variable holds current product id
def recentOrder = adf.util.queryRow(
  select: 'CustomerId_c,PrimaryShippingAddress_c',
  from: 'Order_c',
  where: ""
  CustomerId_c = :CurCust
  and OrderStatus_c = 'SHIPPED'
  and OrderLinesCollection_c.ProductId_c = :CurProd
  "",
  orderBy: 'OrderDate_c desc',
  binds: [CurCust: custId, CurProd: prodId])
if (recentOrder) {
  // Pass the recent order we found above into duplicateRow()
  def newOrder = adf.util.duplicateRow(recentOrder,
['CustomerId_c',
'PrimaryShippingAddress_c'])
  // etc.
}
```


5 Best Practices for Groovy Performance

Following the advice in this section will ensure your application has the best performance possible.

Search Using at Least One Indexed Field

Whenever you perform a query, make sure that your view object's view criteria filter includes at least one indexed field in the predicate.

Especially when the amount of data is large, using at least one index to filter the data makes a meaningful difference in application query performance.

Note: Failure to use an index of any kind for your application business logic query implies the database will perform a full table scan that can be a recipe for slow response times and unhappy end users.

Explicitly Select Only the Attributes You Need

When performing a business object query, it's important to indicate which fields your code will access from the results. This includes fields your logic plans to update as well. By doing this proactively, your application gains two advantages:

1. You retrieve only the data you need from the database, and
2. You avoid an *additional* system-initiated query to "fault-in" missing data on first reference

Call the `selectAttributesBeforeQuery()` function to select the attributes your code will access before it performs a view object's query. As shown in the example below, the first parameter is a view object you have created with `newView()` and the second argument is a case-sensitive list of field names. If you are including a sort in your query by calling `setSortBy()`, make sure to include the sort field name(s) in the selected attributes list as well.

```
def employees = newView('StaffMember')
addBindVariable(employees, 'Job', 'Text')
addBindVariable(employees, 'Dept', 'Number')
// Make sure that JobId or DepartmentId is indexed!
employees.appendViewCriteria('JobId = :Job and DepartmentId = :Dept')
employees.setSortBy('Salary desc')
selectAttributesBeforeQuery(employees, ['Email', 'LastName', 'FirstName', 'Salary'])
setBindVariable(employees, 'Job', 'SH_CLERK')
setBindVariable(employees, 'Dept', 50)
employees.executeQuery()
while (employees.hasNext()) {
    def employee = employees.next()
    // Work with employee.Email, employee.LastName, employee.FirstName, employee.Salary
}
```

Note: If you fail to call the `selectAttributesBeforeQuery()` function before executing a view object for a custom object you've created with `newView()`, then by default the query will retrieve only the primary key field from the database when initially performing the query, and then *for each row* of the `while` loop as soon as your code references one of the other attributes like `Email`, `LastName`, `FirstName`, or `Salary`, the system is forced to perform an additional query to retrieve *all* of the current row's fields from the database using the primary key. If your object has 200 fields, this means retrieving 200 fields of data even though your code may actually reference only four of them. This can quickly lead to your application's performing many, many avoidable extra queries and fetching much unnecessary data. Neither of these situations is good for performance.

Using the `queryMaps()` or `queryRows()` helper functions described in *Simplifying Business Logic Queries* you can perform the same optimized query in the example above with fewer lines of code like this:

```
// Make sure that JobId or DepartmentId is indexed!
for (employee in adf.util.queryRows(select: 'Email,LastName,FirstName,Salary',
    from: 'StaffMember',
    where: 'JobId = :Job and DepartmentId = :Dept',
    orderBy: 'Salary desc',
    binds:[Job:'SH_CLERK',Dept:50])) {
    // Work with employee.Email, employee.LastName, employee.FirstName, employee.Salary
}
```

Under the covers, the query helper function calls all of the other functions above, including `selectAttributesBeforeQuery()`, to produce the data your application needs to process.

Related Topics

- [Simplifying Business Logic Queries](#)

Test for Existence by Retrieving a Single Row

When you need to check if at least one row matches a particular criteria, for best performance select only the primary key field and just the first row of the result. If it's not null, then the existence test succeeds.

Object Function: `Boolean employeeExistsInDepartmentWithJob( Long department, String jobCode)`

```
def employees = newView('StaffMember')
addBindVariable(employees, 'Job', 'Text')
addBindVariable(employees, 'Dept', 'Number')
// Make sure that either JobId or DepartmentId is indexed!
employees.appendViewCriteria('JobId = :Job and DepartmentId = :Dept')
// Retrieve only the primary key field
selectAttributesBeforeQuery(employees, ['EmployeeId'])
setBindVariable(employees, 'Job', jobCode)
setBindVariable(employees, 'Dept', department)
employees.executeQuery()
// Retrieve just the first row!
return employees.first() != null
```

Using the `queryMap()` or `queryRow()` helper functions described in *Simplifying Business Logic Queries* you can perform the same optimized query in the example above with fewer lines of code like this:

Object Function: `Boolean employeeExistsInDepartmentWithJob( Long department, String jobCode)`
`// Retrieve only the primary key field and just the first row`

```
return adf.util.queryRow(select:'EmployeeId',
    from: 'StaffMember',
    where: 'JobId = :Job and DepartmentId = :Dept',
    binds:[Job:jobCode,Dept:department]) != null
```

With the `employeeExistsInDepartmentWithJob()` helper function in place, our business logic in the `StaffMember` object can use it like this:

```
if (employeeExistsInDepartmentWithJob(50,'SH_CLERK')) { /* etc. */ }
```

Related Topics

- [Simplifying Business Logic Queries](#)

Avoid Using `newView()` Inside a Loop

Using `newView()` inside a loop can lead to unpredictable `ExprResourceException` errors when you inadvertently create more view objects than the system allows in a single trigger or object function.

For example, consider the following code that iterates over an unknown number of uncleared transaction records. For each uncleared transaction processed, if the transaction currency is not `GBP` then it queries the historical exchange rate based on the transaction date to convert the non-`GBP` currency amount in question into `GBP`. It uses the `newView()` function inside the loop to query the `ExchangeRate_c` business object and does so without using bind variables. This approach creates one new view object for each loop iteration. If the number of transaction rows being iterated over is unpredictably large, this technique can produce an `ExprResourceException` error when it hits the upper limit on number of view objects that can be created in a single trigger or function.

```
// NON-BEST-PRACTICE EXAMPLE: USES newView() INSIDE A LOOP !!
// ~~~~~ May lead to unpredictable ExprResourceException error
// Create view object for processing uncleared transactions
def txns = newView('Transaction_c')
txns.appendViewCriteria("Cleared_c = 'N'")
selectAttributesBeforeQuery(txns,['Id','Cleared_c','Currency_c','Amount_c','Date_c'])
txns.executeQuery()
// Process each uncleared transaction
while (txns.hasNext()) {
    def rate = 1
    def txn = txns.next()
    def curr = txn.Currency_c
    if (curr != 'GBP') {
        def date = txn.Date_c
        // NON-BEST PRACTICE: USE OF newView() INSIDE A LOOP !!
        def rates = newView('ExchangeRate_c')
        rates.appendViewCriteria("From_c = '${curr}' and To_c = 'GBP' and Date_c = '${date}'")
        rates.executeQuery()
        rate = rates.first()?.Rate_c
    }
    if (rate) {
        txn.Cleared_c = 'Y'
        // Multiply original txn amount by rate and round to 2 decimal places
        txn.AmountInGBP_c = (txn.Amount_c * rate as Double).round(2)
    }
}
```

In these situations, use this approach instead:

- Create a single view object outside the loop that references bind variables in its filter criteria

- Inside the loop, set the values of the bind variables for the current loop iteration
- Execute the query on the single view object once per loop iteration

By adopting this technique, you use a single view object instead of an unpredictably large number of view objects and you avoid encountering the `ExprResourceException` when iterating over a larger number of rows. The code below implements the same functionality as above, but follows these best practice guidelines.

```
// BEST PRACTICE: Single VO with bind variables outside the loop
// ~~~~~
// Create view object to be reused inside the loop for exchange rates
def rates = newView('ExchangeRate_c')
addBindVariable(rates, 'Base', 'Text')
addBindVariable(rates, 'FromDate', 'Date')
rates.appendViewCriteria("From_c = :Base and To_c = 'GBP' and Date_c = :FromDate")
// Create view object for processing uncleared transactions
def txns = newView('Transaction_c')
txns.appendViewCriteria("Cleared_c = 'N'")
selectAttributesBeforeQuery(txns, ['Id', 'Cleared_c', 'Currency_c', 'Amount_c', 'Date_c'])
txns.executeQuery()
// Process each uncleared transaction
while (txns.hasNext()) {
    def rate = 1
    def txn = txns.next()
    def curr = txn.Currency_c
    if (curr != 'GBP') {
        def date = txn.Date_c
        // BEST PRACTICE: Set bind variables & execute view object created outside loop
        setBindVariable(rates, 'Base', curr)
        setBindVariable(rates, 'FromDate', date)
        rates.executeQuery()
        rate = rates.first()?.Rate_c
    }
    if (rate) {
        txn.Cleared_c = 'Y'
        // Multiply original txn amount by rate and round to 2 decimal places
        txn.AmountInGBP_c = (txn.Amount_c * rate as Double).round(2)
    }
}
```

If the functionality inside the loop becomes more involved, you may benefit by refactoring it into an object function. The object function below shows an `exchangeRateForCurrencyOnDate()` helper function that accepts the single view object created outside the loop as a parameter of type `Object`. Inside the function, it sets the bind variables, executes the view object's query, and returns the resulting exchange rate.

Object Function: `Float exchangeRateForCurrencyOnDate( Object rates, String curr, Date date)`

```
// Set bind variables and execute view object passed in
setBindVariable(rates, 'Base', curr)
setBindVariable(rates, 'FromDate', date)
rates.executeQuery()
return rates.first()?.Rate_c
```

After refactoring the code into this object function, the if block in the original best-practice code above can be changed to:

```
// etc.
if (curr != 'GBP') {
    def date = txn.Date_c
    // Pass single 'rates' view object into the helper function
    rate = exchangeRateForCurrencyOnDate(rates, curr, date)
}
```

```
// etc.
```

Using the `queryMaps()` Or `queryRows()` helper functions described in *Simplifying Business Logic Queries* you can perform the same optimized best-practice technique described in this section with fewer lines of code like this:

```
// BEST PRACTICE: Create a reusable query for looking up the exchange rate using
// ~~~~~ bind variable values of the correct datatype. Inside the loop
// set the correct bind var values and execute the same view object over & over
def rateVO = adf.util.query(
    select: 'Rate_c',
    from: 'ExchangeRate_c',
    where: "From_c = :Base and To_c = 'GBP' and Date_c = :ForDate",
    binds: [Base: 'XXX', ForDate: today() ])
// Will be updating the queried rows, so use queryRows()
for (txn in adf.util.queryRows(select: 'Id,Cleared_c,Currency_c,Amount_c,Date_c',
    from: 'Transaction_c',
    where: "Cleared_c = 'N'")) {
    def rate = 1
    // If transaction currency is different than GBP, lookup historical
    // exchange rate for the date of the transaction to convert the
    // transaction currency into GBP
    if (txn.Currency_c != 'GBP') {
        // Set any bind variables to new values for current loop iteration
        setBindVariable(rateVO, 'Base', txn.Currency_c)
        setBindVariable(rateVO, 'ForDate', txn.Date_c)
        // Execute the same view object with the new bind variable values
        rateVO.executeQuery()
        rate = rateVO.first()?.Rate_c
    }
    if (rate) {
        txn.Cleared_c = 'Y'
        // Multiply original txn amount by rate and round to 2 decimal places
        txn.AmountInGBP_c = (txn.Amount_c * rate as Double).round(2)
    }
}
```

Related Topics

- [Simplifying Business Logic Queries](#)

Set Field Values in Bulk

Wherever possible in your code, for best performance set the values of all fields in a row in a single call to the `setAttributeValuesFromMap()` function.

Using these bulk-assignment functions saves processing time and can eliminate avoidable queries related to your Dynamic Choice List and Fixed Choice List attribute validation when compared to the equivalent job performed one field at a time.

For example, the following code examples sets the values of five fields of an existing staff member row. The code is using the `queryRow()` helper function described in *Simplifying Business Logic Queries* to find the row by employee id

```
// Find an existing staff member by the indexed primary key field EmployeeId
// Then bulk-assign 5 field values whose names are also included in the
// view object's select list to avoid unnecessary "fault-in" queries.
def emp = adf.util.queryRow(
    select: 'EmployeeId,Email,CarMake,CarModel,Vacation,AccrualDate',
    from: 'StaffMember',
    where: 'EmployeeId = :id',
```

```
    binds: [id: 123456789])
if (emp) {
    emp.setAttributeValuesFromMap(
        Email: emp.Email.replace('old.org', 'new.org'),
        CarMake: 'VW',
        CarModel: 'GLF',
        Vacation: 160,
        AccrualDate: today())
}
```

When creating a new row, you can accomplish the same bulk assignment task using the `createAndInitRowFromMap()` function. The following example creates a new staff member assigning all fields in bulk:

```
def emps = newView('StaffMember')
// Insert a new staff member, setting all necessary fields in bulk
emps.insertRow(emps.createAndInitRowFromMap(
    Email: 'jane.barnes@example.org',
    CarMake: 'AUD',
    CarModel: 'A8',
    Vacation: 200,
    AccrualDate: today()))
```

Both examples in this section illustrate Groovy's support for removing the square brackets around a literal `map` passed inline to a function with a leading `map` argument. To learn more about how your own functions can leverage this feature, see [Using Optional, Named Method Arguments](#).

When writing generic helper code, if you find it more convenient to process the field *names* to assign and corresponding *values* to assign in separate lists, then consider using the `setAttributeValues()` function. The example below shows how it may fit your situation better than `setAttributeValuesFromMap()`. This alternative function accomplishes the same performance improvement.

```
// void doBulkAssignment(Object row, List fieldNames, List fieldValues)
// accepting row to assign, field names and field values as separate Lists
if (fieldNames.size() == fieldValues.size()) {
    row.setAttributeValues(fieldNames, fieldValues)
}
else {
    adf.util.error("Must supply same number of fields and values to assign!")
}
```

Related Topics

- [Using Optional, Named Method Arguments](#)
- [Simplifying Business Logic Queries](#)

Avoid Revalidating Known Valid Data

Normally your business logic will use the equals sign assignment operator to set a single field's value, or use the `setAttributeValuesFromMap()` or `createRowAndInitFromMap()` functions to set two or more field values in bulk.

Any fields assigned through these methods will be validated by any field and object-level validation rules that are defined to ensure that the business object data saved to the database is always 100% valid.

On special occasions, you may know that the value your code assigns to a field is already valid. In cases where you are 100% certain the value being assigned to a field is valid, you can consider using the `populateValidAttribute()` function to knowingly assign a valid value to a field without causing additional validation to occur.

For example, your code can change the value of an order's `OrderStatus` field to one of the values that you know is valid like `CLOSED` by using the following code:

```
// NOTE: Consciously assigning a known-valid value
// ~~~~ without further validation!
order.populateValidAttribute('OrderStatus','CLOSED')
```

Use Left Shift Operator To Append to Lists

To append elements to an existing list, use the left shift operator (`<<`) or call the list's `add()` function for best performance.

For example, the following code processes a collection of products and adds the value of the `id` field from a subset of the products encountered to a new list:

```
list productIdsToProcess = []
for (prod in products) {
    if (prod.Status == 'RETURNED') {
        // Append the current product id to the list
        // Same as calling productIdsToProcess.add(prod.Id)
        productIdsToProcess << prod.Id
    }
}
```

This technique is better than using the *plus* or *plus-equals* operator to do the same job because both of those create a new list each time.

6 Understanding Common JBO Exceptions in Groovy Scripts

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

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 `x row` 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 `TroubleTicket` that has a child object named `Activity`. The following script that tries to create a new activity would generate this error:

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

This generates a *JBO-25030: Detail entity Activity 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 `createRow()` and `insertRow()` combination. For example, to create a new `Activity` row in the context of `TroubleTicket` with an `id` of 100000000272002 you can do the following, using the helper function mentioned in *Finding an Object by Id*. When you use this approach, the parent object context is implicit since you're performing the action on the parent row's child collection.

```
def idParent = 100000000272002
def ttVO = newView('TroubleTicket')
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
    // 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.

```
def idParent = 100000000272002
// Create an name/value pairs object to pass the parent id
def parentAttrs = new oracle.jbo.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')
def newActivity = activityVO.createAndInitRow(parentAttrs)
activityVO.insertRow(newActivity);
// Set other field values of the new activity here...
```

JBO-26020: Attempting to insert row with no matching EO base

- **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:

```
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"
deptVO.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.

7 Supported Classes and Methods for Use in Groovy

Supported Classes and Methods for Use in Groovy Scripts

When writing Groovy scripts, only use the classes and methods that are documented in this topic.

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()`

Groovy Allowed Classes, Methods, and Packages

Class Name	Allowed Methods	Package
<code>ADFContext</code>	• <code>getLocale()</code> • <code>getSecurityContext()</code>	<code>oracle.adf.share</code>
<code>Array</code>	• <i>Any constructor</i> • <i>Any method</i>	<code>java.sql</code>
<code>Array</code>	• <code>getArray()</code> • <code>getElemType()</code> • <code>getList()</code>	<code>oracle.jbo.domain</code>
<code>ArrayList</code>	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>

Class Name	Allowed Methods	Package
Arrays	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
AttributeDef	• <code>getAttributeKind()</code> • <code>getIndex()</code> • <code>getName()</code> • <code>getPrecision()</code> • <code>getProperty()</code> • <code>getScale()</code> • <code>getUIHelper()</code> • <code>getUpdateableFlag()</code> • <code>isMandatory()</code> • <code>isQueryable()</code>	<code>oracle.jbo</code>
AttributeHints	• <code>getControlType()</code> • <code>getDisplayHeight()</code> • <code>getDisplayHint()</code> • <code>getDisplayWidth()</code> • <code>getFormat()</code> • <code>getFormattedAttribute()</code> • <code>getFormatter()</code> • <code>getFormatterClassName()</code> • <code>getHint()</code> • <code>getLocaleName()</code> • <code>parseFormattedAttribute()</code>	<code>oracle.jbo</code>
AttributeList	• <code>getAttribute()</code> • <code>getAttributeIndexOf()</code> • <code>getAttributeNames()</code> • <code>setAttribute()</code>	<code>oracle.jbo</code>
BaseLobDomain	• <code>closeCharacterStream()</code> • <code>closeInputStream()</code> • <code>closeOutputStream()</code> • <code>getInputStream()</code> • <code>getLength()</code> • <code>getOutputStream()</code> • <code>getcharacterStream()</code>	<code>oracle.jbo.domain</code>
BigDecimal	• <i>Any constructor</i> • <i>Any method</i>	<code>java.math</code>

Class Name	Allowed Methods	Package
BigInteger	- Any constructor - Any method	<code>java.math</code>
BitSet	- Any constructor - Any method	<code>java.util</code>
Blob	- Any constructor - Any method	<code>java.sql</code>
BlobDomain	- Any constructor <code>getBinaryOutputStream()</code> <code>getBinaryStream()</code> <code>getBufferSize()</code>	<code>oracle.jbo.domain</code>
Boolean	- Any constructor - Any method	<code>java.lang</code>
Byte	- Any constructor - Any method	<code>java.lang</code>
Calendar	- Any constructor - Any method	<code>java.util</code>
Char	- Any constructor <code>bigDecimalValue()</code> <code>bigIntegerValue()</code> <code>booleanValue()</code> <code>doubleValue()</code> <code>floatValue()</code> <code>getValue()</code> <code>intValue()</code> <code>longValue()</code>	<code>oracle.jbo.domain</code>
Clob	- Any constructor - Any method	<code>java.sql</code>
ClobDomain	- Any constructor <code>toCharArray()</code>	<code>oracle.jbo.domain</code>
Collection	- Any constructor - Any method	<code>java.util</code>
Collections	- Any constructor - Any method	<code>java.util</code>

Class Name	Allowed Methods	Package
Comparator	- Any constructor - Any method	java.util
Currency	- Any constructor - Any method	java.util
DBSequence	- Any constructor - getValue()	oracle.jbo.domain
Date	- Any constructor - Any method	java.util
Date	- Any constructor - Any method	java.sql
Date	- Any constructor - compareTo() - dateValue() - getValue() - stringValue() - timeValue() - timestampValue()	oracle.jbo.domain
Dictionary	- Any constructor - Any method	java.util
Double	- Any constructor - Any method	java.lang
Enum	- Any constructor - Any method	java.lang
EnumMap	- Any constructor - Any method	java.util
EnumSet	- Any constructor - Any method	java.util
Enumeration	- Any constructor - Any method	java.util
EventListener	- Any constructor - Any method	java.util
EventListenerProxy	Any constructor	java.util

Class Name	Allowed Methods	Package
	Any method	
EventObject	- Any constructor - Any method	java.util
Exception	Any method	java.lang
ExprValueErrorHandler	- addAttribute() - clearAttributes() - raise() - raiseLater() - warn()	oracle.jbo
Float	- Any constructor - Any method	java.lang
Formattable	- Any constructor - Any method	java.util
FormattableFlags	- Any constructor - Any method	java.util
Formatter	- Any constructor - Any method	java.util
GregorianCalendar	- Any constructor - Any method	java.util
HashMap	- Any constructor - Any method	java.util
HashSet	- Any constructor - Any method	java.util
Hashtable	- Any constructor - Any method	java.util
IdentityHashMap	- Any constructor - Any method	java.util
Integer	- Any constructor - Any method	java.lang
Iterator	- Any constructor - Any method	java.util

Class Name	Allowed Methods	Package
JboException	• <code>getDetails()</code> • <code>getErrorCode()</code> • <code>getErrorParameters()</code> • <code>getLocalizedMessage()</code> • <code>getMessage()</code> • <code>getProductCode()</code> • <code>getProperty()</code>	<code>oracle.jbo</code>
JboWarning	• <i>Any constructor</i> • <code>getDetails()</code> • <code>getErrorCode()</code> • <code>getErrorParameters()</code> • <code>getLocalizedMessage()</code> • <code>getMessage()</code> • <code>getProductCode()</code> • <code>getProperty()</code>	<code>oracle.jbo</code>
Key	• <code>toStringFormat()</code>	<code>oracle.jbo</code>
LinkedHashMap	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
LinkedHashSet	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
LinkedList	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
List	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
ListIterator	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
ListResourceBundle	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
Locale	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
Long	• <i>Any constructor</i> • <i>Any method</i>	<code>java.lang</code>

Class Name	Allowed Methods	Package
Map	- Any constructor - Any method	<code>java.util</code>
Math	- Any constructor - Any method	<code>java.lang</code>
MathContext	- Any constructor - Any method	<code>java.math</code>
NClob	- Any constructor - Any method	<code>java.sql</code>
NameValuePairs	- Any constructor <code>getAttribute()</code> <code>getAttributeIndexOf()</code> <code>getAttributeNames()</code> <code>setAttribute()</code>	<code>oracle.jbo</code>
NativeTypeDomainInterface	<code>getNativeObject()</code>	<code>oracle.jbo.domain</code>
Number	- Any constructor <code>bigDecimalValue()</code> <code>bigIntegerValue()</code> <code>booleanValue()</code> <code>byteValue()</code> <code>doubleValue()</code> <code>floatValue()</code> <code>getValue()</code> <code>intValue()</code> <code>longValue()</code> <code>shortValue()</code>	<code>oracle.jbo.domain</code>
Number	<code>abs()</code> <code>and()</code> <code>compareTo()</code> <code>div()</code> <code>downto()</code> <code>intdiv()</code> <code>leftShift()</code> <code>minus()</code> <code>mod()</code>	<code>java.lang</code>

Class Name	Allowed Methods	Package
	• <code>multiply()</code> • <code>next()</code> • <code>or()</code> • <code>plus()</code> • <code>power()</code> • <code>previous()</code> • <code>rightShift()</code> • <code>rightShiftUnsigned()</code> • <code>step()</code> • <code>times()</code> • <code>toBigDecimal()</code> • <code>toBigInteger()</code> • <code>toDouble()</code> • <code>toInteger()</code> • <code>toLong()</code> • <code>unaryMinus()</code> • <code>upto()</code> • <code>xor()</code>	
Object	• <code>any()</code> • <code>asBoolean()</code> • <code>asType()</code> • <code>collect()</code> • <code>each()</code> • <code>eachWithIndex()</code> • <code>every()</code> • <code>find()</code> • <code>findAll()</code> • <code>findIndexOf()</code> • <code>findIndexValues()</code> • <code>findLastIndexOf()</code> • <code>findResult()</code> • <code>getAt()</code> • <code>grep()</code> • <code>identity()</code> • <code>inject()</code> • <code>inspect()</code> • <code>is()</code> • <code>isCase()</code>	java.lang

Class Name	Allowed Methods	Package
	• <code>iterator()</code> • <code>print()</code> • <code>printf()</code> • <code>println()</code> • <code>putAt()</code> • <code>split()</code> • <code>sprintf()</code> • <code>toString()</code> • <code>with()</code>	
Observable	• Any constructor • Any method	<code>java.util</code>
Observer	• Any constructor • Any method	<code>java.util</code>
PriorityQueue	• Any constructor • Any method	<code>java.util</code>
Properties	• Any constructor • Any method	<code>java.util</code>
PropertyPermission	• Any constructor • Any method	<code>java.util</code>
PropertyResourceBundle	• Any constructor • Any method	<code>java.util</code>
Queue	• Any constructor • Any method	<code>java.util</code>
Random	• Any constructor • Any method	<code>java.util</code>
RandomAccess	• Any constructor • Any method	<code>java.util</code>
Ref	• Any constructor • Any method	<code>java.sql</code>
ResourceBundle	• Any constructor • Any method	<code>java.util</code>
Row	• <code>getAttribute()</code> • <code>getAttributeHints()</code>	<code>oracle.jbo</code>

Class Name	Allowed Methods	Package
	• <code>getKey()</code> • <code>getLookupDescription()</code> • <code>getOriginalAttributeValue()</code> • <code>getPrimaryRowState()</code> • <code>getSelectedListDisplayValue</code> • <code>getSelectedListDisplayValue</code> • <code>getStructureDef()</code> • <code>isAttributeChanged()</code> • <code>isAttributeUpdateable()</code> • <code>remove()</code> • <code>revertRow()</code> • <code>revertRowAndContainees()</code> • <code>setAttribute()</code> • <code>setAttributeValues()</code> • <code>setAttributeValuesFromMap()</code> • <code>validate()</code>	
<code>RowId</code>	• <i>Any constructor</i> • <i>Any method</i>	<code>java.sql</code>
<code>RowIterator</code>	• <code>createAndInitRow()</code> • <code>createAndInitRowFromMap()</code> • <code>createRow()</code> • <code>findByKey()</code> • <code>findRowsMatchingCriteria()</code> • <code>first()</code> • <code>getAllRowsInRange()</code> • <code>getCurrentRow()</code> • <code>getEstimatedRowCount()</code> • <code>hasNext()</code> • <code>hasPrevious()</code> • <code>insertRow()</code> • <code>last()</code> • <code>next()</code> • <code>previous()</code> • <code>reset()</code>	<code>oracle.jbo</code>
<code>RowSet</code>	• <code>avg()</code> • <code>count()</code> • <code>createAndInitRow()</code>	<code>oracle.jbo</code>

Class Name	Allowed Methods	Package
	• <code>createRow()</code> • <code>executeQuery()</code> • <code>findByKey()</code> • <code>findRowsMatchingCriteria()</code> • <code>first()</code> • <code>getAllRowsInRange()</code> • <code>getCurrentRow()</code> • <code>getEstimatedRowCount()</code> • <code>hasNext()</code> • <code>hasPrevious()</code> • <code>insertRow()</code> • <code>last()</code> • <code>max()</code> • <code>min()</code> • <code>next()</code> • <code>previous()</code> • <code>reset()</code> • <code>sum()</code>	
Scanner	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
SecurityContext	• <code>getUserName()</code> • <code>getUserProfile()</code> • <code>isUserInRole()</code>	<code>oracle.adf.share.security</code>
Session	• <code>getLocale()</code> • <code>getLocaleContext()</code> • <code>getUserData()</code>	<code>oracle.jbo</code>
Set	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
Short	• <i>Any constructor</i> • <i>Any method</i>	<code>java.lang</code>
Short	• <i>Any constructor</i> • <i>Any method</i>	<code>java.lang</code>
SimpleTimeZone	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
SortedMap	• <i>Any constructor</i>	<code>java.util</code>

Class Name	Allowed Methods	Package
	Any method	
SortedSet	- Any constructor - Any method	java.util
Stack	- Any constructor - Any method	java.util
StackTraceElement	- Any constructor - Any method	java.lang
StrictMath	- Any constructor - Any method	java.lang
String	- Any constructor - Any method	java.lang
StringBuffer	- Any constructor - Any method	java.lang
StringBuilder	- Any constructor - Any method	java.lang
StringTokenizer	- Any constructor - Any method	java.util
Struct	- Any constructor - Any method	java.sql
Struct	- getAttribute() - setAttribute()	oracle.jbo.domain
StructureDef	- findAttributeDef() - getAttributeIndexof()	oracle.jbo
Time	- Any constructor - Any method	java.sql
TimeZone	- Any constructor - Any method	java.util
Timer	- Any constructor - Any method	java.util
TimerTask	- Any constructor - Any method	java.util

Class Name	Allowed Methods	Package
Timestamp	- Any constructor - Any method	java.sql
Timestamp	- Any constructor - compareTo() - dateValue() - getValue() - stringValue() - timeValue() - timestampValue()	oracle.jbo.domain
TreeMap	- Any constructor - Any method	java.util
TreeSet	- Any constructor - Any method	java.util
UUID	- Any constructor - Any method	java.util
UserProfile	- getBusinessCity() - getBusinessCountry() - getBusinessEmail() - getBusinessFax() - getBusinessMobile() - getBusinessPOBox() - getBusinessPager() - getBusinessPhone() - getBusinessPostalAddr() - getBusinessPostalCode() - getBusinessState() - getBusinessStreet() - getDateofBirth() - getDateofHire() - getDefaultGroup() - getDepartment() - getDepartmentNumber() - getDescription() - getDisplayName() - getEmployeeNumber() - getEmployeeType()	oracle.adf.share.security.identitymanagment

Class Name	Allowed Methods	Package
	• <code>getFirstName()</code> • <code>getGUID()</code> • <code>getGivenName()</code> • <code>getHomeAddress()</code> • <code>getHomePhone()</code> • <code>getInitials()</code> • <code>getJpegPhoto()</code> • <code>getLastName()</code> • <code>getMaidenName()</code> • <code>getManager()</code> • <code>getMiddleName()</code> • <code>getName()</code> • <code>getNameSuffix()</code> • <code>getOrganization()</code> • <code>getOrganizationalUnit()</code> • <code>getPreferredLanguage()</code> • <code>getPrincipal()</code> • <code>getProperties()</code> • <code>getProperty()</code> • <code>getTimeZone()</code> • <code>getTitle()</code> • <code>getUIAccessMode()</code> • <code>getUniqueName()</code> • <code>getUserID()</code> • <code>getUserName()</code> • <code>getWirelessAccountNumber()</code>	
ValidationException	• <code>getDetails()</code> • <code>getErrorCode()</code> • <code>getErrorParameters()</code> • <code>getLocalizedMessage()</code> • <code>getMessage()</code> • <code>getProductCode()</code> • <code>getProperty()</code>	<code>oracle.jbo</code>
Vector	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>
ViewCriteria	• <code>createAndInitRow()</code> • <code>createRow()</code>	<code>oracle.jbo</code>

Class Name	Allowed Methods	Package
	• <code>createViewCriteriaRow()</code> • <code>findByKey()</code> • <code>findRowsMatchingCriteria()</code> • <code>first()</code> • <code>getAllRowsInRange()</code> • <code>getCurrentRow()</code> • <code>getEstimatedRowCount()</code> • <code>hasNext()</code> • <code>hasPrevious()</code> • <code>insertRow()</code> • <code>last()</code> • <code>next()</code> • <code>previous()</code> • <code>reset()</code>	
<code>ViewCriteriaItem</code>	• <code>getValue()</code> • <code>makeCompound()</code> • <code>setOperator()</code> • <code>setUpperColumns()</code> • <code>setValue()</code>	<code>oracle.jbo</code>
<code>ViewCriteriaItemCompound</code>	• <code>ensureItem()</code> • <code>getValue()</code> • <code>makeCompound()</code> • <code>setOperator()</code> • <code>setUpperColumns()</code> • <code>setValue()</code>	<code>oracle.jbo</code>
<code>ViewCriteriaRow</code>	• <code>ensureCriteriaItem()</code> • <code>getConjunction()</code> • <code>isUpperColumns()</code> • <code>setConjunction()</code> • <code>setUpperColumns()</code>	<code>oracle.jbo</code>
<code>ViewObject</code>	• <code>appendViewCriteria()</code> • <code>avg()</code> • <code>count()</code> • <code>createAndInitRow()</code> • <code>createRow()</code> • <code>createViewCriteria()</code> • <code>executeQuery()</code>	<code>oracle.jbo</code>

Class Name	Allowed Methods	Package
	• <code>findByKey()</code> • <code>findRowsMatchingCriteria()</code> • <code>first()</code> • <code>getAllRowsInRange()</code> • <code>getCurrentRow()</code> • <code>getEstimatedRowCount()</code> • <code>getMaxFetchSize()</code> • <code>hasNext()</code> • <code>hasPrevious()</code> • <code>insertRow()</code> • <code>last()</code> • <code>max()</code> • <code>min()</code> • <code>next()</code> • <code>previous()</code> • <code>reset()</code> • <code>setMaxFetchSize()</code> • <code>setSortBy()</code> • <code>sum()</code>	
WeakHashMap	• <i>Any constructor</i> • <i>Any method</i>	<code>java.util</code>