Netcool/Impact
Version 6.1.0.2

Policy Reference Guide

SC14-7553-00

Netcool/Impact
Version 6.1.0.2

Policy Reference Guide

SC14-7553-00

Note
Before using this information and the product it supports, read the information in "Notices".

Edition notice

This edition applies to version 6.1.0.2 of IBM Tivoli Netcool/Impact and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Policy Reference Guide

The Netcool/Impact *Policy Reference Guide* contains descriptions and complete syntax references for the Impact Policy Language (IPL) and JavaScript.

Intended audience

This publication is for users who are responsible for writing Netcool/Impact policies.

Publications

This section lists publications in the Netcool/Impact library and related documents. The section also describes how to access Tivoli® publications online and how to order Tivoli publications.

Netcool/Impact library

- *Quick Start Guide*, CF39PML

  Provides concise information about installing and running Netcool/Impact for the first time.
- *Administration Guide*, SC14755100

  Provides information about installing, running and monitoring the product.
- *User Interface Guide*, SC14755400

  Provides instructions for using the Graphical User Interface (GUI).
- *Policy Reference Guide*, SC14755300

  Contains complete description and reference information for the Impact Policy Language (IPL).
- *DSA Reference Guide*, SC14755500

  Provides information about data source adaptors (DSAs).
- *Operator View Guide*, SC14755600

  Provides information about creating operator views.
- *Solutions Guide*, SC14755200

  Provides end-to-end information about using features of Netcool/Impact.
- *Integrations Guide*, SC14755700

  Contains instructions for integrating Netcool/Impact with other IBM® software and other vendor software.
- *Troubleshooting Guide*, GC14755800

  Provides information about troubleshooting the installation, customization, starting, and maintaining Netcool/Impact.

Accessing terminology online

The IBM Terminology Web site consolidates the terminology from IBM product libraries in one convenient location. You can access the Terminology Web site at the following Web address:

http://www.ibm.com/software/globalization/terminology

Accessing publications online

Publications are available from the following locations:

- The *Quick Start* DVD contains the publications that are in the product library. The format of the publications is PDF, HTML, or both. Refer to the readme file on the DVD for instructions on how to access the documentation.
- Tivoli Information Center web site at http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/topic/com.ibm.netcoolimpact.doc6.1/welcome.html. IBM posts publications for all Tivoli products, as they become available and whenever they are updated to the Tivoli Information Center Web site.

  Note: If you print PDF documents on paper other than letter-sized paper, set the option in the **File → Print** window that allows Adobe Reader to print letter-sized pages on your local paper.
- Tivoli Documentation Central at http://www.ibm.com/developerworks/wikis/display/tivolidoccentral/Impact. You can also access publications of the previous and current versions of Netcool/Impact from Tivoli Documentation Central.
- The Netcool/Impact wiki contains additional short documents and additional information and is available at https://www.ibm.com/developerworks/mydeveloperworks/wikis/home?lang=en#/wiki/Tivoli%20Netcool%20Impact.

Ordering publications

You can order many Tivoli publications online at http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss.

You can also order by telephone by calling one of these numbers:

- In the United States: 800-879-2755
- In Canada: 800-426-4968

In other countries, contact your software account representative to order Tivoli publications. To locate the telephone number of your local representative, perform the following steps:

1. Go to http://www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss.
2. Select your country from the list and click **Go**.
3. Click **About this site** in the main panel to see an information page that includes the telephone number of your local representative.

Accessibility

Accessibility features help users with a physical disability, such as restricted mobility or limited vision, to use software products successfully. With this product, you can use assistive technologies to hear and navigate the interface. You can also use the keyboard instead of the mouse to operate all features of the graphical user interface.

For additional information, see Appendix A, “Accessibility,” on page 179.

Tivoli technical training

For Tivoli technical training information, refer to the following IBM Tivoli Education Web site at http://www.ibm.com/software/tivoli/education.

Support for problem solving

If you have a problem with your IBM software, you want to resolve it quickly. This section describes the following options for obtaining support for IBM software products:

- "Obtaining fixes"
- "Receiving weekly support updates"
- "Contacting IBM Software Support" on page x

Obtaining fixes

A product fix might be available to resolve your problem. To determine which fixes are available for your Tivoli software product, follow these steps:

1. Go to the IBM Software Support Web site at http://www.ibm.com/software/support.
2. Navigate to the **Downloads** page.
3. Follow the instructions to locate the fix you want to download.
4. If there is no **Download** heading for your product, supply a search term, error code, or APAR number in the search field.

For more information about the types of fixes that are available, see the *IBM Software Support Handbook* at http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html.

Receiving weekly support updates

To receive weekly e-mail notifications about fixes and other software support news, follow these steps:

1. Go to the IBM Software Support Web site at http://www.ibm.com/software/support.
2. Click the **My IBM** in the toobar. Click **My technical support**.
3. If you have already registered for **My technical support**, sign in and skip to the next step. If you have not registered, click **register now**. Complete the registration form using your e-mail address as your IBM ID and click **Submit**.
4. The **Edit profile** tab is displayed.
5. In the first list under **Products**, select **Software**. In the second list, select a product category (for example, **Systems and Asset Management**). In the third list, select a product sub-category (for example, **Application Performance & Availability** or **Systems Performance**). A list of applicable products is displayed.
6. Select the products for which you want to receive updates.
7. Click **Add products**.
8. After selecting all products that are of interest to you, click **Subscribe to email** on the **Edit profile** tab.
9. In the **Documents** list, select **Software**.
10. Select **Please send these documents by weekly email**.
11. Update your e-mail address as needed.
12. Select the types of documents you want to receive.
13. Click **Update**.

If you experience problems with the **My technical support** feature, you can obtain help in one of the following ways:

Online

Send an e-mail message to erchelp@u.ibm.com, describing your problem.

By phone

Call 1-800-IBM-4You (1-800-426-4409).

World Wide Registration Help desk

For word wide support information check the details in the following link: https://www.ibm.com/account/profile/us?page=reghelpdesk

Contacting IBM Software Support

Before contacting IBM Software Support, your company must have an active IBM software maintenance contract, and you must be authorized to submit problems to IBM. The type of software maintenance contract that you need depends on the type of product you have:

- For IBM distributed software products (including, but not limited to, Tivoli, Lotus®, and Rational® products, and DB2® and WebSphere® products that run on Windows or UNIX operating systems), enroll in Passport Advantage® in one of the following ways:

  Online

  Go to the Passport Advantage Web site at http://www-306.ibm.com/software/howtobuy/passportadvantage/pao_customers.htm .

  By phone

  For the phone number to call in your country, go to the IBM Worldwide IBM Registration Helpdesk Web site at https://www.ibm.com/account/profile/us?page=reghelpdesk.

- For customers with Subscription and Support (S & S) contracts, go to the Software Service Request Web site at https://techsupport.services.ibm.com/ssr/login.
- For customers with IBMLink, CATIA, Linux, OS/390®, iSeries®, pSeries, zSeries, and other support agreements, go to the IBM Support Line Web site at http://www.ibm.com/services/us/index.wss/so/its/a1000030/dt006.
- For IBM eServer™ software products (including, but not limited to, DB2 and WebSphere products that run in zSeries, pSeries, and iSeries environments), you can purchase a software maintenance agreement by working directly with an IBM sales representative or an IBM Business Partner. For more information about support for eServer software products, go to the IBM Technical Support Advantage Web site at http://www.ibm.com/servers/eserver/techsupport.html.

If you are not sure what type of software maintenance contract you need, call 1-800-IBMSERV (1-800-426-7378) in the United States. From other countries, go to the contacts page of the *IBM Software Support Handbook* on the Web at http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html and click the name of your geographic region for phone numbers of people who provide support for your location.

To contact IBM Software support, follow these steps:

1. "Determining the business impact" on page xi
2. "Describing problems and gathering information" on page xi
3. "Submitting problems" on page xi

Determining the business impact

When you report a problem to IBM, you are asked to supply a severity level. Use the following criteria to understand and assess the business impact of the problem that you are reporting:

Severity 1
: The problem has a *critical* business impact. You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.

Severity 2
: The problem has a *significant* business impact. The program is usable, but it is severely limited.

Severity 3
: The problem has *some* business impact. The program is usable, but less significant features (not critical to operations) are unavailable.

Severity 4
: The problem has *minimal* business impact. The problem causes little impact on operations, or a reasonable circumvention to the problem was implemented.

Describing problems and gathering information

When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- Which software versions were you running when the problem occurred?
- Do you have logs, traces, and messages that are related to the problem symptoms? IBM Software Support is likely to ask for this information.
- Can you re-create the problem? If so, what steps were performed to re-create the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, and so on.
- Are you currently using a workaround for the problem? If so, be prepared to explain the workaround when you report the problem.

Submitting problems

You can submit your problem to IBM Software Support in one of two ways:

Online
: Click **Submit and track problems** on the IBM Software Support site at http://www.ibm.com/software/support/probsub.html. Type your information into the appropriate problem submission form.

By phone
: For the phone number to call in your country, go to the contacts page of the *IBM Software Support Handbook* at http://www14.software.ibm.com/webapp/set2/sas/f/handbook/home.html and click the name of your geographic region.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Conventions used in this publication

This publication uses several conventions for special terms and actions, operating system-dependent commands and paths, and margin graphics.

Typeface conventions

This publication uses the following typeface conventions:

Bold

- Lowercase commands and mixed case commands that are otherwise difficult to distinguish from surrounding text
- Interface controls (check boxes, push buttons, radio buttons, spin buttons, fields, folders, icons, list boxes, items inside list boxes, multicolumn lists, containers, menu choices, menu names, tabs, property sheets), labels (such as **Tip:**, and **Operating system considerations:**)
- Keywords and parameters in text

Italic

- Citations (examples: titles of publications, diskettes, and CDs
- Words defined in text (example: a nonswitched line is called a *point-to-point line*)
- Emphasis of words and letters (words as words example: "Use the word *that* to introduce a restrictive clause."; letters as letters example: "The LUN address must start with the letter *L*.")
- New terms in text (except in a definition list): a *view* is a frame in a workspace that contains data.
- Variables and values you must provide: ... where *myname* represents....

`Monospace`

- Examples and code examples
- File names, programming keywords, and other elements that are difficult to distinguish from surrounding text
- Message text and prompts addressed to the user
- Text that the user must type
- Values for arguments or command options

Operating system-dependent variables and paths

This publication uses the UNIX convention for specifying environment variables and for directory notation.

When using the Windows command line, replace **$***variable* with **%***variable***%** for environment variables and replace each forward slash (**/**) with a backslash (\) in directory paths. The names of environment variables are not always the same in the Windows and UNIX environments. For example, %TEMP% in Windows environments is equivalent to $TMPDIR in UNIX environments.

Note: If you are using the bash shell on a Windows system, you can use the UNIX conventions.

Chapter 1. Getting started

This chapter contains information you need to get started with Netcool/Impact policies.

Policies overview

A policy contains a set of statements written in either the Impact Policy Language (IPL) or JavaScript.

Each statement is an instruction that describes a task to perform when certain events or status conditions occur in your environment. Instructions can be for high-level or low-level tasks. A single implementation of Netcool/Impact can have any number of associated policies.

An example of a high-level task is "Send an event to the Netcool/OMNIbus ObjectServer." An example of a low-level task is "Store a value in the internal MyDate variable."

For more information, see "Policy language" on page 3.

For an example of a policy, see "Policy example" on page 6.

Using policies

You use policies to instruct Netcool/Impact to perform a wide variety of actions.

Common actions include sending and retrieving Netcool/OMNIbus events, and adding, modifying or deleting data stored in external data sources. In addition, you can also use a policy to send and receive e-mail, to send and receive instant messages, and to communicate with external systems, devices and applications. For more information, see "Policy capabilities" on page 2.

Creating policies

You can create policies using the policy editor in the GUI.

You can also copy and paste policies from a web page or document file into a plain text editor to remove the rich text format. Then paste the policy into the policy editor. The policy editor tool provides a syntax checker, a policy tree view, and other utilities that help you create and manage policies more easily. For more information, see "Policy editor" on page 8.

Running policies

Policies are run by services that monitor the environment for events or changes in status.

These include the event reader and event listener services. You can also run a policy manually using the GUI or the `nci_trigger` command line utility. For more information, see "Policy triggers" on page 7.

Policy capabilities

Impact policies can perform a wide variety of tasks related to event management.

Event handling

Most Netcool/Impact policies provide instructions for parsing incoming event data.

Handling event data from event reader and event listener services is one of the primary tasks that you perform using Netcool/Impact. Policies also often update event data that is stored in the ObjectServer or another event source. This is particularly the case for event enrichment policies, which correlate events with information stored in other data sources and then update the events using that information.

You can use a policy to perform event-related tasks.

- Parse event data that originates with the Netcool/OMNIbus ObjectServer or another event source (for example, event data stored in an SQL database)
- Send new events to ObjectServer or another event source
- Update existing events in ObjectServer or other event sources
- Delete existing events in ObjectServer or other event sources

Data handling

You can use a policy to perform data-related tasks.

- Retrieve data from a data source (for example, the internal data repository, or an SQL database)
- Add new data to a data source
- Update data stored in a data source
- Delete data stored in a data source

Retrieving data from external data sources is a very common policy task, particularly for event enrichment policies, which operate as described in "Event handling."

E-mail

You can use a policy to perform e-mail-related tasks.

- Send new e-mail to an SMTP server using the e-mail sender service
- Parse e-mail retrieved from a IMAP or POP server using the e-mail reader service

Instant messages

You can use a policy to perform instant message-related tasks.

- Send instant messages to users of Yahoo! Messenger, AOL Instant Messenger, Microsoft Messenger, ICQ, and instant messaging clients that use a Jabber messaging service
- Parse instant messages sent to Netcool/Impact through a Jabber messaging service

Integration with external systems, applications, and devices

You can use a policy to integrate with external systems, applications, and devices that communicate.

Use the following methods:

- Web services API
- XML over HTTP
- JMS
- Custom socket protocols
- Proprietary third-party interfaces provided by data source adaptors (DSAs).

Exchanging data with using a Web services API, XML over HTTP, or JMS is a common way for a policy to communicate with external systems, applications, and devices. You can use these communication mechanisms to integrate Netcool/Impact with a very large variety of vendor software components.

Accessing Service-related information from a policy

You can also create a policy to access information related to Netcool/Impact services.

The following policy example can check if a service is running, and can start or stop a service.

```
GetByFilter("Service", "Name = 'OMNIbusEventReader'", false);
 Reader = OrgNode;
 log("Is Reader Running " + Reader.Running);

 // Starting the Reader
 Reader.Running = true;

 // Stop the Reader
 Reader.Running = false;
```

Policy language

You use the Impact Policy Language (IPL), or JavaScript to write the policies that you want Netcool/Impact to run.

The IPL is a scripting language similar in syntax to programming languages like C/C++ and Java. It provides a set of data types, built-in variables, control structures, and functions that you can use to perform a wide variety of event management tasks. It also allows you to create your own variables and functions, as in other programming languages.

JavaScript a scripting programming language commonly used to add interactivity to web pages. It can also be used in browser environments. JavaScript uses the same programming concepts that are used in IPL to write policies. For more information about JavaScript syntax, see http://www.w3schools.com/js/default.asp.

Data types

The policy-level data type is a different entity than the data types that are part of the data model.

For more information about policy data types, see “Policy-level data types” on page 11.

Variables

IPL and JavaScript have built-in variables, and user-defined variables.

For more information about variables, see “Variables” on page 15.

Operators

Operators are a special type of built-in function that modifies or compares a value or values.

For more information about operators, see “Operators” on page 19.

Control structures

Control structures specify which code statements are executed under which conditions, and at what times.

For more information about control structures, see “Control structures” on page 22.

Functions

The Impact Policy Language (IPL) and JavaScript support built-in functions and user-defined functions.

For more information about functions, see “Functions” on page 25.

External function libraries

Function libraries is a feature you can use to create a set of stored functions that can be called from any policy.

For more information about external function libraries, see “Function libraries” on page 30.

Exception handling

You can raise and handle policy-level exceptions.

For more information about exceptions, see “Exceptions” on page 32.

Clear cache syntax

The policy language provides a syntax that you can use to clear the cache associated with a particular data type.

Netcool/Impact stores information about data types in another, system level data type named `Types`. Each data item in `Types` represents a data type that is defined in the system.

To clear a data type cache, you first call the `GetByFilter` or the `GetByKey` function and retrieve the data item from `Types` that corresponds to the data type. The key value for data items in `Types` is the data type name. Then you set the `clearcache` member variable associated with the data item to `true`.

The following example shows how to clear the cache of a data type named `User`:

```
DataType = "Types";
Key = "User";
MyTypes = GetByKey(DataType, Key, 1);

MyTypes[0].clearcache = true;
```

Date/Time patterns

The policy language provides a date/time pattern syntax that you can use with the LocalTime function to format date/time strings and with the ParseDate function to convert formatted strings to the number of seconds in UNIX time.

The pattern syntax consists of a set of symbols that specify how the date/time is formatted. You use the symbols alone, or with other formatting characters, such as `:` and `/`.

Symbols

The following table shows the symbols used in date/time patterns.

Table 1. Date/Time pattern symbols

Symbol	Description
`G`	Era
`y`	Year
`M`	Month
`d`	Day
`h`	Hour
`H`	Hour
`m`	Minute
`s`	Second
`S`	Millisecond
`E`	Day in week
`D`	Day in year
`F`	Day of week in month (for example, if day is third Friday in a month, value is 3).
`w`	Week in year
`W`	Week in month
`a`	AM/PM marker
`k`	Hour in a day
`K`	Hour in AM/PM
`z`	Time zone

Examples

The following example shows how to return the given number of seconds in various formats using `LocalTime`.

```
Seconds = GetDate();

Time = LocalTime(Seconds, "MM/dd/yy");
```

```
Log(Time);

Time = LocalTime(Seconds, "HH:mm:ss");
Log(Time);
```

This example prints the following message to the policy log:

```
06/19/03
13:11:24
```

Policy example

The following policy is a complete example of a Netcool/Impact policy.

This policy accepts incoming event information from an event reader service and uses this information to look up affected internal customers in an external database. The policy then sends an e-mail to each customer and reports the event data to a third-party help desk system using the web services API provided by that application.

```
// Look up customers affected by the event in the external database.
// Customer is name of a Netcool/Impact data type associated with
// the database table that stores customer data.

DataType = "Customer";
Filter = "Server ='" + EventContainer.Node + "'";
CountOnly = false;

Customers = GetByFilter(DataType, Filter, CountOnly);

// Send an e-mail to each customer with notification of the event

Count = Length(Customers);

While (Count > 0) {

    Address = Customer.Email;
    Subject = "Change in server status";
    Message = "Netcool/Impact has detected a change in status for server " \
        + EventContainer.Node " + ". Summary is " + EventContainer.Summary;
        Sender = "Netcool/Impact";
        ExecuteOnQueue = false;

    Count = Count -1;
    SendEmail(Address, Subject, Message, Sender, ExecuteOnQueue);

}

// Report the event details to the third-party help desk system using
// the provided web services API.

WSSetDefaultPKGName("helpdesk");

WebServiceName = "HelpDeskService";
WebServicePort = "HelpDeskPort";
EndPoint = "http://helpdesk_01/webservice/";
Method = "submitTicket";

Params = { GetDate(), \
    "Netcool/Impact", \
    EventContainer.Node, \
    EventContainer.Summary };

Results = WSInvoke(WebServiceName, WebServicePort, EndPoint, Method, Params);
```

```
// Print results of the web services call to the policy log.

Log("Result of web services call: " + Results);
```

Policy triggers

A policy trigger is a component or feature of Netcool/Impact that is capable of starting a policy.

Some policy triggers, like event readers, are controlled by Netcool/Impact. You control other policy triggers, like the `nci_trigger` script and the graphical user interface (GUI).

Event readers as policy triggers

An event reader service queries a Netcool/OMNIbus ObjectServer or other event source at intervals and retrieves new and updated events.

It then determines whether any of the retrieved events are related to a policy that you have defined and then sends those events to the event processor service for handling. The event processor launches the appropriate policy in response to each incoming event and passes the event data to the policy in the form of an event container variable.

Database listeners as policy triggers

A database listener listens for incoming messages from an SQL database data source and then triggers policies based on the incoming message data.

Currently listener services for use with Oracle databases are supported.

E-Mail readers as policy triggers

The e-mail reader service queries an IMAP or POP e-mail server at intervals for new e-mail messages.

When the service retrieves a new e-mail, it converts the e-mail contents to the Impact event format and then determines whether the event matches one of the policies that you have defined. If the event matches a policy, the e-mail reader sends it to the event processor service for handling. The event processor launches the appropriate policy and passes the event data to the policy in the form of an event container variable.

Jabber readers as policy triggers

The Jabber reader service listens for incoming instant messages.

When the service receives a message, it converts the message contents to the event format and then sends the event to the event processor service for handling. The event processor launches the policy specified in the Jabber reader configuration and passes the event data to the policy in the form of an event container variable.

Web services listeners as policy triggers

The Web services listener listens at an HTTP port for incoming SOAP/XML messages.

When the service receives a message, it converts its contents to policy runtime parameters and passes them to the event processor service for handling. The event

processor launches the policy specified by the web services listener configuration and passes the event data in the form of policy runtime variables.

JMS listeners as policy triggers

The JMS listener service listens for incoming JMS messages.

When the service receives a message, it converts its contents to the event format and sends them to the event processor service for handling. The event processor launches the policy specified in the JMS listener configuration and passes the event data to the policy in the form of an event container variable.

nci_trigger

The nci_trigger script sends event data and a policy name to the event processor for handling.

The event processor launches the policy and passes the event data to the policy in the form of an event container variable.

Running policies in the graphical user interface

You can run policies from within the GUI.

You can use the GUI to pass runtime parameters to the policy, but you cannot pass event data. Use of the GUI for running policies is for testing purposes only.

Policy editor

The GUI provides a policy editor that you can use to create and edit policies.

The policy editor offers a text editor with syntax highlighting, a function browser, a syntax checker, a tree viewer, and other utilities to make it easy to manage policies. You can also write policies in an editor of your choice and then upload them into Netcool/Impact. After they are uploaded, you can edit them and check the syntax using the policy editor.

Note: If you create and edit a policy using an external editor of your choice, you must check its syntax using the `nci_policy` script before you run it. For more information about the `nci_policy` script, see the *Administration Guide*.

Chapter 2. Policy fundamentals

This chapter contains information about basic concepts needed to create policies using either the Impact Policy Language (IPL) or JavaScript.

Differences between IPL and JavaScript

When writing policies using IPL or JavaScript there are a number of differences. Use the following table as a reference.

Table 2. IPL and JavaScript differences

IPL	JavaScript	See
IPL is not case-sensitive.	JavaScript is case-sensitive, keep this in mind when you are creating variables, statements, objects, and, functions.	"RExtractAll" on page 136
When creating Arrays, in IPL you must use {} curly braces to assign array values.	In JavaScript, you must use [] square braces to assign array values.	• "Array" on page 12 • "JavaCall" on page 120
In IPL, the escape character can be either `\\s` or `\s`.	In JavaScript, the escape character must be `\\s`.	"RExtractAll" on page 136
In IPL, integers return as whole numbers, for example `1`.	In JavaScript, integers are `Float` as a result, numbers always display with decimals for example `1` becomes `1.0`.	• "Distinct" on page 95 • "EvalArray" on page 97 • "String operators" on page 21
`ClassOf` • If you pass an integer variable to `ClassOf` it returns as long in IPL. • If you pass a context variable to `ClassOf`, it returns as `BindingsVarGetSettable` in IPL. • If you pass an `OrgNode` variable to `ClassOf`, it returns as `OrgNode` in IPL.	• If you pass an integer variable to `ClassOf` it returns as double in JavaScript. • If you pass a context variable to `ClassOf`, it returns as `JavaScriptScriptableWrapper` in JavaScript. • If you pass an `OrgNode` variable to `ClassOf`, it returns as `VarGetSettable` in JavaScript.	"ClassOf" on page 81
`Event Container` If you are using IPL, you can optionally reference event field variables using the dot notation or the @ notation. The @ notation is a special shorthand that you can use to reference members of `EventContainer` instead of spelling out the full name `@Identifier`.	If you are using JavaScript, you must use the dot notation `EventContainer.Identifier`.	"EventContainer" on page 16

Table 2. IPL and JavaScript differences (continued)

IPL	JavaScript	See
`Exit` In IPL, when you use `Exit` in a user-defined function it exits that function, and the policy continues.	In JavaScript, when you use `Exit` in a user-defined function in a policy it exits the entire policy. If you want to stop a function in a JavaScript policy you must use the return command in the policy.	"Exit" on page 98.
`Float` In IPL, the float function converts an integer, string, or Boolean expression to a floating point number.	JavaScript does not add any decimal points to the results for integers and Boolean expressions that are used with the `Float` function.	"Float" on page 100.
`Float` In IPL, the `Float` function returns `10.695672`.	In JavaScript, the `Float` function returns extra precision, for example `10.695671999999998` instead of `10.695672` for the function `Eval`.	"Eval" on page 97.
`Eval` In IPL, Integer division of `10/5` is `2.0`.	In JavaScript, integer division of `10/5` is `2` for the function `Eval`.	"Eval" on page 97
`JavaCall` In IPL numbers are whole.	JavaScript uses doubles for the numbers, when using JavaScript, for a `JavaCall` that needs an integer argument, you must use the `Integer.parseInt` JavaCall to create an actual integer.	"JavaCall" on page 120

`Like`
 	
In IPL supports the `Like` operator. For example,
`teststring LIKE ".*abc.*";`
	JavaScript does not use the `Like` operator. The equivalent example is
	`/.*abc.*/.test(teststring);`
		"Comparison operators" on page 21

Customize data output to follow the JavaScript standard

Fix Pack 2

A property that is called `impact.featuretostringassource.enabled` can be added to the `NCI_server.props` file. When the property is set to true, it converts the data into a string to enable the data output by the Log function to be the same in JavaScript and IPL.

The default value of the property is true. When the property is set to false, the data follows the JavaScript standard and the data output by the Log function is not the same as in IPL. To make the output the same in JavaScript and IPL the same, customer must use the `Object.toSource()` function in JavaScript.

The customers, who use JavaScript and want to follow the JavaScript standard, can change the value of this property to false.

Add the following property to the `NCI_server.props` file.

To enable the data output by the Log function to be the same in JavaScript and IPL, make the value true.

```
impact.featuretostringassource.enabled=true
```

To follow the JavaScript standard, make the value false.

```
impact.featuretostringassource.enabled=false
```

Policy-level data types

The policy-level data type is a different entity than the data types that are part of the data model.

When you are writing policies there are two categories of data types: simple data types and complex data types. The simple data types are integer, float, string, boolean, and date. The complex data types are array, context, data item, and event container.

Simple data types

Simple data types represent a single value.

Simple data types used to create policies:

Integer

The integer data type represents a positive whole number or its negative value. Examples of integers are `0`, `1`, `2`, `3` and `4`.

Float

The float data type represents a floating-point or decimal number. Examples of floats are `0.1243` and `12.245`.

String

A string represents a sequence of characters, up to a length of 4 KB. In a policy, you enclose a string literal in either double or single quotation marks. An example of a string is `abcdefg`.

Boolean

Boolean represents a value of true or false. In a policy, you specify a boolean value using the `true` and `false` keywords.

Date

A date is a formatted string that represents a time or a calendar date. IPL and JavaScript use the following format for dates: *YYYY-MM-DD HH:MM:SS.n*, where *n* is a millisecond value. The date string can be enclosed in double or single quotation marks.

You specify the *n* millisecond value in date fields when you add a data item to a data type or when you update a data item. If the date does not have a millisecond value, specify `0`. You also specify the millisecond value when you work with date fields in the GUI.

The following policy example shows how to add a new data item to a data type that contains a date field:

```
MyContext = NewObject();
MyContext.FirstName = "Joe";
MyContext.LastName = "Example";
MyContext.Id = "1234";
MyContext.Created = "2006-07-31 11:12:13.4";

AddDataItem("Customer", MyContext);
```

Complex data types

Complex data types represent sets of values.

Context

Context is a data type that you can use to store sets of data.

Contexts are like the `struct` data type in C/C++. Contexts can be used to store elements of any combinations of data types, including other contexts and arrays. This data is stored in a set of variables called member variables that are "contained" inside the context. Member variables can be of any type, including other contexts.

You reference member variables using the dot notation. This is also the way that you reference member variables in a `struct` in languages like C and C++. In this notation, you specify the name of the context and the name of the member variable separated by a period (.). You use this notation when you assign values to member variables and when you reference the variables elsewhere in a policy.

Important: A built-in context is provided, called the policy context, that is created automatically whenever the policy is run. The policy context contains all of the variables used in the policy, including built-in variables.

Unlike arrays and scalar variables, you must explicitly create a context using the `NewObject` function before you can use it in a policy. You do not need to create the member variables in the context. Member variables are created automatically the first time you assign their value.

The following example shows how to create a new context, and how to assign and reference its member variables:

```
MyContext = NewObject();
MyContext.A = "Hello, World!";
MyContext.B = 12345;

Log(MyContext.A + ", " + MyContext.B);
```

This example prints the following message to the policy log:

```
Hello, World!, 12345
```

The following policy shows how to create a context called `MyContext` and assign a set of values to its member variables.

```
MyContext
= NewObject();

MyContext.One = "One";
MyContext.Two = 2;
MyContext.Three = 3.0;

String1 = MyContext.One + ", " + MyContext.Two + ", " + MyContext.Three;

Log(String1)
```

When you run this policy, it prints the following message to the policy log:

```
One, 2, 3.0
```

Array

The array is a native data type that you can use to store sets of related values.

An array in Netcool/Impact represents a heterogeneous set of data, which means that it can store elements of any combination of data types, including other arrays and contexts. The data in arrays is stored as unnamed elements rather than as member variables.

In IPL you assign values to arrays using the curly braces notation. This notation requires you to enclose a comma-separated list of the values to assign in curly braces. The values can be specified as literals or as variables whose values you have previously defined in the policy:

```
arrayname = {element1, element2, elementn}
```

Attention: Arrays in IPL and JavaScript are zero-based, which means that the first element in the array has an index value of `0`.

In JavaScript, use the square braces notation to assign array values as a comma-separated series of numeric, string, or boolean literals:

```
arrayname = [element1, element2, elementn]
```

Important: You can create an array of any size by manually defining its elements. You cannot import it from a file. You cannot have an array in an array unless it is a multi-dimensional array.

You access the value of arrays using the square bracket notation. This notation requires you to specify the name of the array followed by the index number of the element enclosed in square brackets. Use the following syntax to access the elements of a one-dimensional array and a multi-dimensional array:

```
arrayname[element index]
arrayname[first dimension element index][second dimension element index]
```

Examples

Here is an example of a one-dimensional array in IPL:

```
MyArray = {"Hello, World!", 12345};
Log(MyArray[0] + ", " + MyArray[1]);
```

Here is an example of a one-dimensional array in JavaScript:

```
MyArray = ["Hello, World!", 12345];
Log(MyArray[0] + ", " + MyArray[1]);
```

It prints the following text to the policy log:

```
Hello.World!, 12345
```

Here in an example of a two-dimensional array in IPL:

```
MyArray = {{"Hello, World!", 12345}, {"xyz", 78, 7, "etc"}};
Log(MyArray[0][0] + "." + MyArray[1][0]);
```

Here in an example of a two-dimensional array in JavaScript:

```
MyArray = [["Hello, World!", 12345], ["xyz", 78, 7, "etc"]];
Log(MyArray[0][0] + "." + MyArray[1][0]);
```

It prints the following text to the policy log:

```
Hello.World!.xyz
```

This example policy in IPL, uses the same two-dimensional array and prints the label and the value of an element to the parser log:

```
MyArray = {{"Hello, World!", 12345}, {"xyz", 78, 7, "etc"}};
log("MyArray is " + MyArray);
log("MyArray Length is " + length(MyArray));
ArrayA = MyArray[0];
log("ArrayA is " + ArrayA + " Length is " + length(ArrayA));
i = 0;
While(i < length(ArrayA)) {
    log("ArrayA["+i+"] = " + ArrayA[i]);
    i = i+1;
}
ArrayB = MyArray[1];
log("ArrayB is " + ArrayB + " Length is " + length(ArrayB));
i = 0;
While(i < length(ArrayB)) {
    log("ArrayB["+i+"] = " + ArrayB[i]);
    i = i+1;
}
```

This example policy in JavaScript, uses the same two-dimensional array and prints the label and the value of an element to the parser log:

```
MyArray = [["Hello, World!", 12345], ["xyz", 78, 7, "etc"]];
log("MyArray is " + MyArray);
log("MyArray Length is " + Length(MyArray));
ArrayA = MyArray[0];
Log("ArrayA is " + ArrayA + " Length is " + Length(ArrayA));
i = 0;
while(i < Length(ArrayA)) {
    Log("ArrayA["+i+"] = " + ArrayA[i]);
    i = i+1;
}
ArrayB = MyArray[1];
Log("ArrayB is " + ArrayB + " Length is " + Length(ArrayB));
i = 0;
while(i < length(ArrayB)) {
    Log("ArrayB["+i+"] = " + ArrayB[i]);
    i = i+1;
}
```

Here is the output in the parser log:

```
ArrayA[0] = Hello World!
ArrayA[1] = 12345
```

In the following policy, you assign a set of values to arrays and then print the values of their elements to the policy log.

```
Array1 = {"One", "Two", "Three", "Four",
"Five"};
Array2 = {1, 2, 3, 4, 5};
Array3 = {"One", 2, "Three", 4, "Five"};

String1 = "One";
String2 = "Two";
Array4 = {String1, String2};

Log(Array1[0]);
Log(Array2[2]);
Log(Array3[4]);
Log(Array4[1]);

Log(CurrentContext());
```

Here, you assign sets of values to four different arrays. In the first three arrays, you assign various string and integer literals. In the fourth array, you assign variables as the array elements.

When you run the policy, it prints the following message to the policy log:

```
One
3
4
Two
"Prepared with user supplied parameters "=(String2=Two, ActionType=1,
String1=One, EventContainer=(), ActionNodeName=TEMP, Escalation=6,
Array4={One, Two}, Array3={One, 2, Three, 4, Five}, Array2={1, 2,
3, 4, 5},
Array1={One, Two, Three, Four, Five})
```

Data item

The data item is a policy-level data type that is used to represent data items in the Netcool/Impact data model.

Like a context, a data item consists of a set of named member variables. In a data item, however, the member variables have a one-to-one correspondence with fields in the underlying data source. As with contexts, you reference the member variables using the dot notation.

Many built-in functions return an array of data items. These functions include `GetByFilter`, `GetByKey`, `GetByLinks`, `GetHibernatingPolicies`, and `GetScheduleMembers`.

The following example shows how to assign and reference values in a data item:

```
MyCustomers = GetByKey("Customer", 12345, 1);

Log(MyCustomers[0].Name + ", " + MyCustomer[0].Location);

MyCustomers[0].Location = "New Location";
```

Data items store data by reference, rather than by value. When you change the value of a member variable in a data item, Netcool/Impact immediately changes the value in the underlying data source. Either by directly accessing it from the internal data repository or by sending an SQL `UPDATE` statement to the data source (for SQL database data sources).

Event container

The event container is a policy-level data type that represents an event.

Like contexts and arrays, an event container consists of a set of named member variables. In an event container, the member variables have a one-to-one correspondence with fields in the associated event source. As with contexts and arrays, you reference the member variables using the dot notation.

You can create your own event container in a policy by calling the `NewEvent` function. The following example shows how to create an event container, and how to assign and reference its member variables.

```
MyEvent = NewEvent("OMNIbusEventReader");
MyEvent.Node = "ORACLE_01";
MyEvent.Summary = "System not responding to ping request";
```

Variables

IPL and JavaScript have built-in variables, and user-defined variables.

You use built-in variables to handle data retrieved from external data sources and to handle event data. The built-in variables are *DataItem*, *DataItems Num*, and *EventContainer*.

You can use user-defined variables to store values during the lifetime of a policy in the same way that you use variables in other programming languages.

You also use variables to pass values to functions. The variable is updated after the function is complete. As a result, you can only use variables to pass values to functions.

Built-in variables

Built-in variables during policy runtime are automatically populated and managed.

IPL and JavaScript have the following built-in variables:

- EventContainer
- DataItems
- DataItem
- Num

EventContainer

`EventContainer` is a built-in variable of the event container data type that represents an incoming event.

`EventContainer` has a set of member variables that correspond to fields in the incoming event. It also contains two predefined member variables. You can use these variables to specify the state of the event when you return it to the event source using the `ReturnEvent` function.

When a policy is triggered by an event reader, an email reader, or another mechanism, the event processor service creates a new `EventContainer` and populates its member variables with the field values in the event. The event processor creates one new member variable for each field in the event and assigns it the field value.

If you are using IPL, you can optionally reference event field variables using the dot notation or the @ notation. The @ notation is a special shorthand that you can use to reference members of `EventContainer` instead of spelling out the full name.

If you are using JavaScript you must use the dot notation `EventContainer.Identifier`.

The following example shows the use of the optional @ notation to reference event field variables for IPL.

```
Log(@ActionKey);
Log(@ActionKey + ":" + @Summary);
@Summary = @Summary + ": Updated by Netcool/Impact";
```

The following example shows the use of the dot notation `EventContainer.Identifier` for JavaScript or IPL

```
Log(EventContainer.ActionKey);
Log(EventContainer.ActionKey + ":" + EventContainer.Summary);
EventContainer.Summary = EventContainer.Summary + ": Updated by Netcool/Impact";
```

Event state variables are the two predefined member variables that you can use to specify the state of an event when you return it to the event source using the ReturnEvent function. The policy engine does not automatically populate these variables when the policy is triggered.

Table 3 shows the event state variables.

Table 3. Event state variables

Variable	Description
EventContainer.JournalEntry	Set this field to add a new journal entry when you return an updated event to the event source. You specify the contents of the journal entry in single quotation marks. If you want to include newline or tab characters, you concatenate them separately with the string and enclose them in double quotation marks. You can add only journal entries to events when you update them from within a policy. You cannot add journal entries to new events.
EventContainer.DeleteEvent	Set this field to true to delete the event when you return it to the event source.

The following example shows how to add a new journal entry when you return an event.

```
// Set the EventContainer.JournalEntry variable
EventContainer.JournalEntry = 'Modified on ' + LocalTime(GetDate()) \
    + "\r\n" + 'Modified by Netcool/Impact.';

// Return the event to the event source
ReturnEvent(EventContainer);
```

The following example shows how to delete an event when you return it to the event source.

```
// Set the EventContainer.DeleteEvent variable
EventContainer.DeleteEvent = true;

// Return the event to the event source
ReturnEvent(EventContainer);
```

DataItems

DataItems is a built-in variable that stores an array of data items.

The DataItems variable is populated automatically by the functions that return data item arrays, such as GetByFilter, GetByKey, GetByLinks, GetHibernatingPolicies, and GetScheduleMember. You can use these functions to assign the returned data item array to other variables.

The following example shows the use of DataItems in a policy.

```
// Call GetByFilter and pass the name of a data type,
// and a filter as input parameters. GetByFilter
// assigns the matching data items to the DataItems variable.

DataType = "Node";
Filter = "Location = 'New York'";
CountOnly = false;

GetByFilter(DataType, Filter, CountOnly);

// For each data item referenced by DataItems,
// print the value of the Name field to the policy
```

```
// log.

I = Num;
While (I > 0) {
    Log(DataItems[I-1]);
    I = I - 1;
}
```

DataItem

DataItem is a built-in variable that references the first element in the DataItems array.

You can use DataItem as shorthand in instances where you know the DataItems will contain only one element, or when you want to handle only the first element in the array. DataItem is equivalent to DataItems[0].

Num

Num is a built-in variable that stores the number of elements currently stored in the DataItems array.

You can use Num to count the number of data items returned by functions like GetByFilter, GetByKey, and GetByLinks, or you can use it to iterate through the DataItems array in a policy.

Note: The Num variable is supported for compatibility with earlier versions only. To retrieve the number of elements in an array returned by GetByFilter, GetByKey, or GetByLinks, use the Length function. For more information, see "Length" on page 124.

The following example shows how to use the Num variable in a policy.

```
// Call GetByFilter and pass the name of a data type,
// and a filter as input parameters. GetByFilter
// assigns the matching data items to the DataItems variable.

DataType = "Node";
Filter = "Location = 'New York'";
CountOnly = false;

GetByFilter(DataType, Filter, CountOnly);

// For each data item referenced by DataItems,
// print the value of the Name field to the policy
// log.

I = Num;
While (I > 0) {
    Log(DataItems[I-1]);
    I = I - 1;
}
```

User-defined variables

User-defined variables are variables that you define when you write a policy.

You can use any combination of letters and numbers as variable names as long as the first variable starts with a letter:

You do not need to initialize variables used to store single values, such as strings or integers. For context variables, you call the NewObject function, which returns a new context. For event container variables, you call NewEvent. You do not need to initialize the member variables in contexts and event containers.

The following example shows how to create and reference user-defined variables:

```
MyInteger = 1;
MyFloat = 123.4;
MyBoolean = True;
MyString = "Hello, World!";

MyContext = NewObject();
MyContext.Member = "1";

MyEvent = NewEvent();
MyEvent.Summary = "Event Summary";

Log(MyInteger + ", " + MyEvent.Summary);
```

In the example in this section, you create a set of variables and assign values to them. Then, you use the `Log` function in two different ways to print the value of the variables to the policy log.

The first way you use `Log` is to print out each of the values as a separate call to the function. The second way is to print out all the variables in the policy context at once, using the `CurrentContext` function. The `CurrentContext` function returns a string that contains the names and values of all the variables currently defined in the policy.

```
VarOne = "One";
VarTwo = 2;
VarThree = 3.0;
VarFour = VarOne + ", " + VarTwo + ", " + VarThree;

Log(VarOne);
Log(VarTwo);
Log(VarThree);
Log(VarFour);

Log(CurrentContext());
```

When you run this policy, it prints the following message to the policy log:

```
One
2
3.0
One, Two, Three
"Prepared with user supplied parameters "=(Escalation=5, EventContainer=(),
VarTwo=Two, VarOne=One, ActionNodeName=TEMP, VarFour=One, Two, Three,
VarThree=Three, ActionType=1)
```

As shown above, you do not have to declare variables before assigning their values in the way that you do in languages like C/C++ and Java. Arrays and scalar variables like integers or strings are created automatically the first time you assign a value to them. Contexts and event containers, however, must be explicitly created using the `NewObject` and `NewEvent` functions, as described later in this guide.

Operators

Operators are a special type of built-in function that modifies or compares a value or values.

IPL and JavaScript support a standard set of assignment, mathematical, comparison, boolean, and string operators. You use them to assign and retrieve values from variables, perform mathematical operations, compare values, and concatenate strings. IPL and JavaScript also define a set of operators that you can use to perform bitwise operations.

Assignment operator

The assignment operator is the equal sign =.

Use the single equal sign for assigning values to variables. Use the double equal sign == to compare to values in boolean expression.

The following examples show the use of the assignment operator:

```
a = 3;
b = "This is a test";
c = {"One", 2, true};
MyContext.a = "Test";
MyArray[0] = "Another test";
EventContainer.Summary = "Node " + EventContainer.Node + " not responding";
```

Bitwise operators

IPL and JavaScript support the use of the following Bitwise operators.

Table 4. Bitwise operators

Operator	Description	Example
&	AND: Returns a one in each bit position for which the corresponding bits of both operands are ones.	`a & b`
|	OR: Returns a one in each bit position for which the corresponding bits of either or both operands are ones.	`a | b`
^	XOR: Returns a one in each bit position for which the corresponding bits of either but not both operands are ones.	`a ^ b`

Boolean operators

IPL and JavaScript support a range of boolean operators.

Table 5. Boolean operators

Operator	Description
||	or
&&	and
!	not

The following examples show the use of the boolean operators.

```
If ((a == 4) || (b <= 3)) ...
If ((a = "Test") && (b != "Test")) ...
If (!MyBool) ...
```

Comparison operators

IPL and JavaScript support a range of comparison operators.

Table 6. Comparison operators

Operator	Description
<	Less than
>	Greater than
==	Equal to
<=	Less than or equal to
>=	Greater than or equal to
!=	Not equal to
LIKE	Performs a comparison using Perl 5 regular expressions. JavaScript does not have the LIKE operator.

The following examples show the use of the comparison operators in IPL.

```
If ((a >= 4) || (b <= 3)) ...
If ((a == "Test") && (b != "Test")) ...
If (a LIKE "New York.*") ...
```

Mathematic operators

IPL and JavaScript support a range of mathematic operators.

Table 7. Mathematic operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

Note: The division operator always returns a floating-point value. Even if the dividend and the divisor are integers and if the quotient returned by the operation has no fractional value.

The following examples show the use of the mathematic operators.

```
a = 1 + 2;
b = 4 - 3;
c = 6 * 4;
d = 6 / 4;
e = 8 % 5;
```

String operators

IPL and JavaScript support the use of the addition operator (+) for concatenating strings.

One common operation is to concatenate string values within an SQL filter. When you concatenate strings within a filter, make sure that all string literals are enclosed in single quotation marks within the resulting filter. For more information about SQL filters, see "SQL filters" on page 69.

Examples of using string operators

Note:

The following example shows how to concatenate strings using the addition operator.

```
MyString1 = "This is a";
MyString2 = "test.";
MyString3 = MyString1 + " " + MyString2;
Log(MyString3);
```

This example prints the following message to the policy log:

```
Parser log: This is a test.
```

The following example shows how to concatenate strings within an SQL filter.

```
// Call GetByFilter and pass a data type name
// and a filter as input parameters

DataType = "Node";
Filter = "NodeName = '" + EventContainer.Node + "'";
CountOnly = false;

MyNode = GetByFilter(DataType, Filter, CountOnly);
```

Yet another example demonstrates how to add a string, to an array:

```
MyArray = {};
MyString = 'test';
MyArray = MyArray + MyString;

This will add MyString as a new element to the Array.

Also in an Array you can remove an element using the Subtraction operator.
If you do the following:

MyArray = {test, test2};
MyString = 'test';
MyArray = MyArray - MyString;
log(MyArray);
```

The resulting array contains only the test2 string.

Control structures

Control structures specify which code statements are executed under which conditions, and at what times.

IPL and JavaScript use two control structures: If and While. You use the If structure to perform branching operations. You use the While structure to loop over a set of instructions until a certain condition is met.

If statements

You use the `if` statement to perform branching operations.

Use the `if` statement to control which statements in a policy are executed by testing the value of an expression to see if it is true. The `if` statement in the Impact Policy Language is the same as the one used in programming languages like C/C++ and Java.

The syntax for an `if` statement is the `if` keyword followed by a Boolean expression enclosed in parentheses. This expression is followed by a block of statements enclosed in curly braces. Optionally, the `if` statement can be followed by the `else` or `elseif` keywords, which are also followed by a block of statements.

```
if (condition){
    statements
} elseif (condition){
    statements
} else {
    statements
}
```

Where *condition* is a boolean expression and *statements* is a group of one or more statements. For example:

```
if (x == 0) {
    Log("x equals zero");
} elseif (x == 1){
    Log("x equals one");
} else {
    Log("x equals any other value.");
}
```

When the `if` keyword is encountered in a policy, the Boolean expression is evaluated to see if it is true. If the expression is true, the statement block that follows is executed. If it is not true, the statements is skipped in the block. If an `else` statement follows in the policy, the corresponding `else` statement block is executed.

In this example policy, you use the `if` statement to test the value of the `Integer1` variable. If the value of `Integer1` is `0`, the policy runs the statements in the statement block.

```
Integer1 = 0;

if (Integer1 == 0) {
    Log("The value of Integer1 is zero.");
}
```

When you run this policy, it prints the following message to the policy log:

```
The value of Integer1 is zero.
```

Another example shows how to use the `else` statement. Here, you set the value of the `Integer1` variable to `2`. Since the first test in the `if` statement fails, the statement block that follows the `else` statement is executed.

```
Integer1 = 2;

if (Integer1 == 1) {
    Log("The value of Integer1 is one.");
} else {
    Log("The value of Integer1 is not one.");
}
```

When you run this example, it prints the following message to the policy log:

```
The value of Integer1 is not one.
```

While statements

You use the `while` statement to loop over a set of instructions until a certain condition is met.

The `while` statement allows you to repeat a set of operations until a specified condition is true. The `while` statement in the Impact Policy Language is the same as the one used in programming languages like C, C++, and Java.

The syntax for the `while` statement is the `while` keyword followed by a Boolean expression enclosed in parentheses. This expression is followed by a block of statements enclosed in curly braces.

```
while (condition) { statements }
```

where `condition` is a boolean expression and `statements` is a group of one or more statements. For example:

```
I = 10;
while(I > 0) {
    Log("The value of I is: " + I);
    I = I - 1;
}
```

When the `while` keyword is encountered in a policy, the Boolean expression is evaluated to see if it is true. If the expression is true, the statements in the following block are executed. After the statements are executed, Netcool/Impact again tests the expression and continues executing the statement block repeatedly until the condition is false.

The most common way to use the `while` statement is to construct a loop that is executed a certain number of times depending on other factors in a policy. To use the `while` statement in this way, you use an integer variable as a counter. You set the value of the counter before the `while` loop begins and decrement it inside the loop. The While statement tests the value of the counter each time the loop is executed and exits when the value of the counter is zero.

The following example shows a simple use of the `while` statement:

```
Counter = 10;

while (Counter > 0) {
    Log("The value of Counter is " + Counter);
    Counter = Counter - 1;
}
```

Here, you assign the value of `10` to a variable named `Counter`. In the `while` statement, the policy tests the value of `Counter` to see if it is greater than zero. If `Counter` is greater than zero, the statements in the block that follows is executed. The final statement in the block decrements the value of `Counter` by one. The While loop in this example executes 10 times before exiting.

When you run this example, it prints the following message to the policy log:

```
The value of Counter is 10
The value of Counter is 9
The value of Counter is 8
The value of Counter is 7
The value of Counter is 6
The value of Counter is 5
The value of Counter is 4
The value of Counter is 3
The value of Counter is 2
The value of Counter is 1
```

The following example shows how to use the While statement to iterate through an array. You often use this technique when you handle data items retrieved from a data source.

```
MyArray = {"One", "Two", "Three", "Four"};

Counter = Length(MyArray);

while (Counter > 0) {
 Index = Counter - 1;
 Log(MyArray[Index]);
 Counter = Counter - 1;
}
```

Here, you set the value of `Counter` to the number of elements in the array. The While statement loops through the statement block once for each array element. You set the Index variable to the value of the Counter minus one. This is because arrays in IPL are zero-based. This means that the index value of the first element is `0`, rather than `1`.

When you run this example, it prints the following message to the policy log:

```
Four
Three
Two
One
```

In these examples, when you use this technique to iterate through the elements in an array, you access the elements in reverse order. To avoid doing this, you can increment the counter variable instead of decrementing it in the loop. This requires you to test whether the counter is less than the number of elements in the array inside the While statement.

The following example shows how to loop through an array while incrementing the value of the counter variable.

```
MyArray = {"One", "Two", "Three", "Four"};

ArrayLength = Length(MyArray);
Counter = 0;

while (Counter < ArrayLength) {
    Log(MyArray[Counter]);
    Counter = Counter + 1;
}
```

When you run this policy, it prints the following message to the policy log:

```
One
Two
Three
Four
```

Functions

The Impact Policy Language (IPL) and JavaScript support built-in functions and user-defined functions.

The built-in functions are the functions, that are immediately available after the product is installed. Built-in functions perform common high-level and low-level tasks such as, sending an event to the ObjectServer or using regular expressions to

extract a substring that matches this pattern. Examples of built-in functions are web services functions and SNMP functions. You can use the IPL or JavaScript to create and use function libraries.

User-defined functions are functions that you can use to organize the custom code in your policies.

Web services functions

Web services functions are functions that you use with the web services DSA.

You can use the following functions a to send messages to a web service provided by another application and to handle the message replies.

- `WSInvoke`
- `WSInvokeDL`
- `WSNewArray`
- `WSNewEnum`
- `WSNewObject`
- `WSNewSubObject`
- `WSSetDefaultPKGName`
- `WSDMGetResourceProperty`
- `WSDMInvoke`
- `WSDMUpdateResourceProperty`

For reference information about web services functions, see relevant sections in Chapter 6, "Functions," on page 73. For more information about web services DSA, see the *DSA Reference Guide*.

SNMP functions

SNMP functions are functions that you use with the SNMP DSA.

You use these functions to send data to and retrieve data from SNMP agents. You can also use SNMP functions to send traps and notifications to SNMP managers.

The following SNMP functions are provided:

- SnmpGetAction
- SnmpGetNextAction
- SnmpSetAction
- SnmpTrapAction

For reference information about provided SNMP functions, see relevant sections in Chapter 6, "Functions," on page 73. For more information about the SNMP DSA, see the *DSA Reference Guide*.

Java Policy functions

With Java Policy functions, a policy can run a Java program or call any Java method if their Java classes are available from the Netcool/Impact runtime class path.

Before you can use Java Policy functions, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $IMPACT_HOME/dsalib directory.
2. Restart the Impact Server to load the Java archive (JAR) files.

You must repeat this procedure for each Impact Server because the Java class files in the $IMPACT_HOME/dsalib directory are not replicated between servers.

Here are some of the most obvious scenarios where you would use these functions:

- To provide functionality that is not supported by the Impact Policy Language (IPL). By using the Java Policy functions you can access java.io.* library of Java API and all other functions the Java API provides.
- To use third-party libraries or APIs within policies. Some systems have a special communication protocols to access data and provide a Java library which implements its communication protocol. You can add this library JAR file to the runtime class path and start to communicate with the system by starting its Java classes from a policy through the Java Policy functions.
- To complement usage of other DSAs. The Java Policy functions can be used to resolve interoperability issues among different web services platforms, for example, WebSphere, Weblogic, Axis, and JAX-RPC. The Netcool/Impact Web Services DSA is often used to start the web services that are running on the other web services platforms. If the other platform is using certain complex types in their web services implementation, such as Hashtable or List, the call from Netcool/Impact fails because the Web Services DSA would not successfully compile the WSDL file due to these special types. To solve this problem, you can add the web services client jars that are generated by the other platform to the runtime class path. This enables policies to make the soap calls by using these client classes, thus resolving the interoperability issue.

The following Java Policy functions are provided:

- GetFieldValue
- JavaCall
- NewJavaObject
- SetFieldValue

For reference information about these Java Policy functions, see relevant sections in Chapter 6, “Functions,” on page 73.

User-defined functions

User-defined functions are functions that you use to organize your code in the body of a policy.

Once you define a function, you can call it in the same way as the built-in action and parser functions. Variables that are passed to a function are passed by reference, rather than by value. This means that changing the value of a variable within a function also changes the value of the variable in the general scope of the policy.

User-defined functions cannot return a value as a return parameter. You can return a value by defining an output parameter in the function declaration and then assigning a value to the variable in the body of the function. Output parameters are specified in the same way as any other parameter.

You can also declare your own functions and call them within a policy.
User-defined functions help you encapsulate and reuse functionality in your policy.

The syntax for a function declaration is the `Function` keyword followed by the name of the function and a comma-separated list of input parameters. The list of input parameters is followed by a statement block that is enclosed in curly braces.

Unlike action and parser functions, you cannot specify a return value for a user-defined function. However, because the scope of variables in IPL policy is global, you can approximate this functionality by setting the value of a return variable inside the function.

Function declarations must appear in a policy before any instance where the function is called. The best practice is to declare all functions at the beginning of a policy.

The following example shows how to declare a user-defined function called `GetNodeByHostname`. This function looks up a node in an external data source by using the supplied host name.

```
Function GetNodeByHostName(Hostname) {

 DataType = "Node";
 Filter = "Hostname ='" + Hostname + "'";
 CountOnly = False;

 MyNodes = GetByFilter(DataType, Filter, CountOnly);
 MyNode = MyNodes[0];

}
```

You call user-defined functions in the same way that you call other types of functions. The following example shows how to call the function declared in the previous example.

```
GetNodeByHostName("ORA_HOST_01");
```

Here, the name of the node that you want to look up is `ORA_HOST_01`. The function looks up the node in the external data source and returns a corresponding data item named `MyNode`. For more information about looking up data and on data items, see the next chapter in this book.

Important:

When you write an Impact function, check that you do not call the function within the function body as this might cause a recursive loop and cause a stack overflow error.

Function declarations

Function declarations are similar to those in scripting languages like JavaScript. Valid function names can include numbers, characters, and underscores, but cannot start with a number.

The following is an example of a user-defined function:

```
Function MyFunc(DataType, Filter, MyArray) {
    MyArray = GetByFilter(DataType, Filter, False);
}
```

Calling user-defined functions

You can call a user-defined function as follows:

```
Funcname([param1, param2 ...])
```

The following example shows a user-defined function call:

```
MyFunc("User", "Location = 'New York'", Users);
```

Examples of user-defined functions

The following example show how variables are passed to a function by reference:

```
// Example of vars by reference

Function IncrementByA(NumberA, NumberB) {
    NumberB = NumberB + NumberA;
}

SomeInteger = 10;
SomeFloat = 100.001;

IncrementByA(SomeInteger, SomeFloat);

Log("SomeInteger is now: " + SomeInteger);
// will return: IntegerA is now 10

Log("SomeFloat is now: " + SomeFloat);
// will return: FloatB is now 110.001
```

The following example shows how policies handle return values in user-defined functions:

```
// Example of no return output

Function LogTime(TimeToLog) {
    If (TimeToLog == NULL) {
        TimeToLog = getdate();
    }
    Log("At the tone the time will be: "+ localtime(TimeToLog));
}

LoggedTime = LogTime(getdate());

Log("LoggedTime = "+LoggedTime);

// will return: "LoggedTime = NULL" as nothing can be
// returned from user functions
```

Local transactions

You use local transactions in a policy if you want to use more than one SQL operation to be treated as a single unit of work.

This can be useful for cases where there is a group of related SQL commands that have to be committed to the database only when all of them are executed successfully. The following functions in the policy language enable the use of local transactions in an Impact policy:

- BeginTransaction
- CommitTransaction
- RollbackTransaction

For reference information about these functions, see “Local transactions template” on page 39 and relevant sections in Chapter 6, “Functions,” on page 73.

Function libraries

Function libraries is a feature you can use to create a set of stored functions that can be called from any policy.

Use external function libraries to encapsulate and reuse the custom code in your policies and are available for IPL and JavaScript policy languages.

Creating function libraries

A function library is a special type of policy that contains only user-defined functions.

You create this policy and define the functions that it contains in the same way you create standard policies. You can use parameters in the functions as both input and output variables.

- For an example of how to create function libraries that use JavaScript, refer to the example in the Load function, see “Load” on page 124.
- The following IPL example shows a function library. This library is a policy named `UTILS_LIBRARY`.

```
// NormalizeString trims whitespace, replaces the space character
// with an underscore and converts all characters to upper case

Function NormalizeString(StringToNormalize) {
    StringToNormalize = Trim(StringToNormalize);
    StringToNormalize = Replace(StringToNormalize, " ", "_");
    StringToNormalize = ToUpper(StringToNormalize);
}

// GetCustomersByNode returns an array of data items from
// a data source, where each data item represents a customer

Function GetCustomersByLocation(Location, Customers) {
    Type = "Customer";
    Filter = "Location = '" + Location + "'";
    CountOnly = false;
    Customers = GetByFilter(Type, Filter, CountOnly);
}
```

Calling functions in a library

To call a function in function library, you specify the library and function name.

- For an example of how to call functions in a library that uses JavaScript, refer to the example in the Load function, see “Load” on page 124.
- For IPL, use the following example which uses the following format:

  ```
  function_library.function_name(param, [param ...])
  ```

Where `function_library` is the name of the library policy, `function_name` is the name of the function, and `param` is the value of one or more parameters required by the function. You do not need to explicitly reference or include the library name before you call its functions, as is required in programming languages like C and C++. The following example shows how to call functions in the library named `UTILS_LIBRARY`. The functions are the same as those defined in the previous example.

```
// Normalize location string

UTILS_LIBRARY.NormalizeString(Location);

// Get customers at the specified location

UTILS_LIBRARY.GetCustomersByLocation(Location, Customers);

// Print customer info to the policy log

Log(Customers);
```

Synchronized statement blocks

You can use synchronized statement blocks to write thread-safe policies for use with a multi-threaded event processor.

You can use synchronized statement blocks in situations where more than one instance of a policy or different policies that access the same resource run simultaneously on different event processor threads.

Synchronized statement blocks consist of any set of IPL statements that are enclosed in curly braces and set apart with the `synchronized` keyword and a synchronization identifier.

The syntax for a synchronized statement block is as follows:

```
synchronized(identifier) { statements }
```

Where *identifier* is a unique name for the statement block and *statements* are any IPL programming statements.

The following example shows how to create a synchronized statement block for IPL:

```
synchronized(update_table) {

    DataType = "Customer";
    Filter = "Location = 'New York'";
    UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014'";

    BatchUpdate(DataType, Filter, UpdateExpression);

}
```

To create a synchronized statement block for JavaScript you create a function, and then call the function with a `Synchronizer`.

```
function update_table() {
DataType = "Customer";
Filter = "Location = 'New York'";
UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014'";
BatchUpdate(DataType, Filter, UpdateExpression);
}

syncmyFunc = new Packages.org.mozilla.javascript.Synchronizer(update_table);
syncmyFunc();
```

When Netcool/Impact processes a synchronized statement block, it registers the synchronization identifier and does not allow other synchronized statement blocks with that identifier to run until the registered block is finished running. Other statement blocks with the same identifier to run sequentially, rather than

simultaneously, with the first block. As a result, resources accessed in the synchronized portion of the policy are protected from simultaneous access by multiple threads.

Exceptions

You can raise and handle policy-level exceptions.

The Impact Policy Language and JavaScript can raise and catch exceptions within a policy and handle Java exceptions that are raised internally when a policy is run.

Raising exceptions

To raise an exception, you use the Raise keyword.

The following example shows the syntax for `Raise`:

```
Raise ExceptionName(ExceptionText);
```

where *ExceptionName* is a unique name for the exception and *ExceptionText* is the text output of the exception. This output is printed to the server log when the error is encountered. You can also access it inside an error handler using the `ErrorMessage` variable.

The following example shows how to raise an exception using the `Raise` keyword. In this example, the function raises an exception named `IntOutOfRangeException` if the value of the `Param1` parameter is less than 0.

```
Function MyFunction(Param1, Param2) {
    If (Param1 < 0) {
        Raise IntOutOfRangeException("Value of Param1 must be greater than 0");
    }
```

Handling exceptions

To handle an exception, you declare an exception handler.

The handler is a function that is called each time that a specific exception is raised. An exception is raised at the policy level, or a specific Java™ exception is raised by Netcool/Impact during the execution of a policy.

Declare exception handlers in advance of any position where they are triggered in a policy. Insert error handlers at the beginning of a policy before you specify any other operations.

The following example shows the syntax for exception handlers:

```
Handle ExceptionName {
    statements ...
}
```

Where *ExceptionName* is the name of the exception that is raised. The *ExceptionName* uses the `Raise` keyword, or the name of the Java exception class that is raised by Netcool/Impact during the execution of the policy.

The following example shows how to handle policy-level exceptions by using an exception handler.

```
Handle IntOutOfRangeException {
    Log("Error: Value of parameter submitted to MyFunction is less than 0");
}
```

```
Function MyFunction(Param1, Param2) {
    If (Param1 < 0) {
        Raise IntOutOfRangeException("Value of Param1 must be greater than 0");
    }
}
```

The following example shows how to handle Java exceptions by using exception handlers.

```
Handle java.lang.NullPointerException {
    Log("Null pointer exception in policy.");
}

Handle java.lang.Exception {
   log("ErrorMessage: " + ErrorMessage);
 MyException = javaCall("java.lang.Exception",ExceptionMessage, "getCause", null);
log("MyException is " + MyException);
MyException = javaCall("java.lang.Exception",MyException, "getCause", null);
log("MyException again is " + MyException);
}
```

The following examples show how to handle JavaScript exceptions by using exception handlers.

```
try
   {
   //Run some code here
   }
 catch(err)
   {
   //Handle errors here
   }

 Example 1:

 try {
   MyFunction(Param1, Param2);
 } catch(e)
   if (e == "IntOutOfRangeException") {
   Log("Error: Value of parameter submitted to MyFunction is less than 0");
   }
 }

  function MyFunction(Param1, Param2) {
     If (Param1 < 0) {
       throw "IntOutOfRangeException";
      }
    }

Example 2:

try {
  ...code that is running...
  } catch(e) {
    if (e.javaException instanceof java.lang.NullPointerException) {
      Log("Null pointer exception in policy.");
     }
     if (e.javaException instanceof java.lang.Exception) {
       log("ErrorMessage: " + ErrorMessage);
 MyException = javaCall("java.lang.Exception",ExceptionMessage, "getCause", null);
log("MyException is " + MyException);
MyException = javaCall("java.lang.Exception",MyException, "getCause", null);
log("MyException again is " + MyException);
      }
  }
```

Runtime parameters

You can define parameters that you pass to a policy when you run it either using the GUI or the `nci_trigger` script.

You can use these parameters when testing a policy in the GUI or when you want to automate a policy by external means from the command line (for example, using the UNIX cron tool).

Setting policy runtime parameters in the editor

Use this procedure to set the runtime parameters for your policy in the policy editor.

Procedure

1. In the policy editor toolbar, click the **Configure Runtime Parameters** icon to open the policy runtime parameter editor.
2. Click **New Runtime Parameter** to open the Create a New Policy Runtime Parameter window.

   Enter the information in the new runtime parameter configuration window. Required fields are marked with an asterisk (*).
3. To edit an existing runtime parameter, select the check box next to the parameter and select **edit** in in the corresponding cell of the **Edit** column.
4. Click **OK** to save the changes to the parameters and close the window.

Policy runtime parameter configuration window

Policy runtime parameters take a set of attributes.

Table 8. List of attributes that are used with a policy runtime parameters

Attribute	Description
Name	Type a name to describe the parameter.
Label	Type a label that will appear in the Policy Trigger window.
Format	Choose a format.
Default Value	Type a default value that will always display in the Policy Trigger window, to avoid entering it each time.
Description	Type some text to describe the parameter.

Running policies with parameters in the editor

If you specified any runtime parameters for the policy you can run the policy with these parameters in the policy editor.

Procedure

1. Click the **Run with Parameters** icon to open the Policy Runtime Parameters window.

   Note: The fields you see in the Policy Runtime Parameters window depend on the runtime parameters and values you specified for the policy. If you have not set a default value for a parameter you must provide it now, otherwise a NULL value will be passed.

For more information about setting runtime parameters, see “Setting policy runtime parameters in the editor” on page 34.

2. Click **Execute** to run the policy with parameters.

Running a policy using the nci_trigger script

Use this procedure to run a policy using the nci_trigger script.

Procedure

To run the policy, you start nci_trigger from the command line, as in the following example.
In this example, the name of the policy is `POLICY_01`. The value of `Value1` is `Testing1`, and the value of `Value2` is `Testing2`.

```
nci_trigger NCI tipadmin/netcool POLICY_01 Value1 Testing1 Value2 Testing2
```

In the following example, `Value1` and `Value2` are the runtime parameters that the policy handles.

```
// Value1 and Value2 are passed to the policy
Value3 = EventContainer.Value1 + " " + EventContainer.Value2;
Log(Value3);
```

Chained policies

Policy chaining is a feature where you chain multiple policies to run together sequentially when an event reader service triggers them.

Policies are run in series, rather than simultaneously, and each policy in the chain inherits the policy context from the previously run policy. This means that variables whose values were assigned in a previous policy in the chain maintain their values in subsequent policies.

When the event reader retrieves an event from the Netcool/OMNIbus ObjectServer, it compares the event data to each defined event mapping in the service configuration. If the event matches multiple chained mappings, it runs each of the mapped policies in sequence as they appear in the event mapping window.

Chaining policies

Follow this procedure to chain your policies.

Procedure

1. Expand **Event Automation** > **System Configuration**, click the **Services** link to open the **Services** tab.
2. Double click the name of the event reader service that you want to use to run the chained policies.
3. Select the **Event Mapping** tab in the window that opens.
4. For each policy in the chain, create an event mapping that associates a restriction filter with a policy name.

   When you create the event mapping, select the **Chain** option in the event mapping window.
5. Click **OK**.

Encrypted policies

An encrypted policy is a policy whose text content has been encrypted to a non-human readable format.

Encrypted policies can be run in the same way as non-encrypted policies.

You encrypt policies using the `nci_encryptpolicy` script. This script is located in the `$IMPACT_HOME/bin` directory and has the following syntax:

```
nci_encryptpolicy server_name
   password
   input_policy
   output_policy
```

where *server_name* is the name of the Impact Server, *password* is the encryption password, *input_policy* is the name of the policy you want to encrypt, and *output_policy* is the name of the resulting encrypted policy.

Before you can run the encrypted policy on the originating Impact Server or on another server, you first import it into the system using the GUI.

To import the policy, complete the following steps:

1. Open the **Policies** task pane in the Navigation panel.
2. Click the **Upload Local IPL File** button.
3. Click the **Browse** button in the window that opens and choose the encrypted policy file.
4. Click **OK**.

Once you have imported the policy, you can run it in the same way that you run any other policy.

Line continuation character

The line continuation character in IPL and JavaScript is the backslash (\).

You use this character to indicate that the code on a subsequent line is a continuation of the current statement. The line continuation character helps you format your policies so that they are easier to read and maintain.

Note: You cannot use the line continuation character inside a string literal as specified with enclosing quotation marks. This usage is not allowed and an error is reported during processing.

The following example shows the use of the line continuation character:

```
Log("You can use the line continuation character" + \
    "to format very long statements in a policy.");
```

Code commenting

IPL and JavaScript support both C-style comment blocks and C++-style single-line code commenting.

Comment blocks are single or multi-line blocks of comments enclosed by the forward slash (/) and asterisk (*) characters. The following example shows comment blocks.

```
/* This is a single-line comment block */
/* This is a multi-line comment block */
```

Single-line comments are prefixed by two forward slash characters.

The following example shows single-line comments

```
// These
// are
// single-line
// comments
```

Chapter 3. Local transactions

You use local transactions in a policy if you want to use more than one SQL operation to be treated as a single unit of work.

The following code is a typical template of a policy that uses local transactions. In this policy, `SQL_Operation_1()` will be executed first. As soon as the operation is completed successfully, the changes will be committed to the database. When the `BeginTransaction()` method is executed, all SQL operations following it will be executed using the same transaction. As a result any changes that they have made will not be committed to the database until the operations are executed successfully and `CommitTransaction()` is executed.

```
Handle com.micromuse.response.action.TransactionException {
Log(" Transaction Failed " + ErrorMessage);
RollbackTransaction();
}

SQL_Operation_1();
.....
BeginTransaction();
...
SQL_Operation_2();
SQL_Operation_3();
...
CommitTransaction();
...
SQL_Operation_4();
```

Local transactions template

Here is a typical template of a policy that uses local transactions.

In this policy, `SQL_Operation_1()` will be executed first. As soon as the operation is completed successfully, the changes will be committed to the database. When the `BeginTransaction()` method is executed, all SQL operations following it will be executed using the same transaction. As a result any changes that they have made will not be committed to the database until the operations are executed successfully and `CommitTransaction()` is executed.

```
Handle com.micromuse.response.action.TransactionException {
Log(" Transaction Failed " + ErrorMessage);
RollbackTransaction();
}

SQL_Operation_1();
.....
BeginTransaction();
...
SQL_Operation_2();
SQL_Operation_3();
...
CommitTransaction();
...
SQL_Operation_4();
```

The following examples examine various scenarios that may result after running the policy in the template.

SQL_Operation_2() and SQL_Operation_3() is executed successfully

In this scenario when the thread reaches the `CommitTransaction()` function, it will commit both SQL operations to the database and then enable the Auto Commit functionality so that as soon as SQL_Operation_4() is gets executed, the changes get committed to the database.

SQL_Operation_2() fails and SQL_Operation_3() is executed successfully

When SQL_Operation_2 () fails, an exception will be thrown that gets caught by the Handle block. In the Handle block, the `RollbackTransaction()` function is called which rolls back any changes done by SQL_Operation_2(). When the Handle block is finished, the execution goes back to the policy right after SQL_Operation_2(), which is SQL_Operation_3(). This SQL Operation will get executed but since it is being executed after a rollback has occurred, the changes will not get committed to the database. When the thread reaches the `CommitTransaction()` function, it would not commit anything to the database since a rollback had occurred. The only operation done by `CommitTransaction()` would be to enable auto-commit for any SQL operation following it. When the thread executes SQL_Operation_4(), any changes done will be committed to the database as soon the operation is completed as Netcool/Impact will be non-transactional after `CommitTransaction()`.

SQL_Operation_2() is executed successfully and SQL_Operation_3() fails

Let us assume that after `BeginTransaction()`, the SQL_Operation_2() gets executed successfully. The changes will not get committed to the database until all the operations between `BeginTransaction()` and `CommitTransaction()` get executed successfully. In this scenario, we have assumed that SQL_Operation_3() fails. As a result an exception will be thrown that will send the policy execution inside the Handle block where the `RollbackTransaction()` method gets called. This function will roll back any changes done since `BeginTransaction()` and, once the Handle block is completed, the execution is returned to the policy statement following SQL_Operation_3(). When the policy executes the `CommitTransaction()` method, it will not commit anything to the database because rollback has occurred. The only operation done by `CommitTransaction()` would be to enable auto-commit for any SQL operations following it. When the thread executes SQL_Operation_4(), any changes done will be committed to the database as soon the operation is completed as Netcool/Impact will be non-transactional after `CommitTransaction();`

Both SQL_Operation_2() and SQL_Operation_3() fails

When SQL_Operation_2() fails, the policy execution will enter the Handle block where `RollbackTransaction()` is executed, it will roll back any changes made since `BeginTransaction()` and transfer the policy execution to the statement following SQL_Operation_2(). When SQL_Operation_3() gets executed and fails, the `RollbackTransaction()` will be executed again in the Handle block and the same process repeats. When the policy execution reaches the `CommitTransaction()` function, it will not commit any changes and will enable auto-commit for any SQL operations following it.

Local transactions best practices

Here are some practical tips on the usage of local transactions.

- Local transactions should ideally combine SQL operations to a single database. Even though nothing is stopping you from using different data sources between `BeginTransaction()` and `CommitTransaction()`, you are recommended not to do that.
- Avoid doing operations inside the transaction block that are not related to the actual SQL operations.
- Do not rely solely on com.micromuse.response.action.TransactionException to be thrown for every possible failure in the transaction block. One way to get around this would be to also handle the general java.lang.Exception and call `RollbackTransaction()` in it. For example:

```
Handle com.micromuse.response.action.TransactionException {
Log(" Transaction Failed " + ErrorMessage);
RollbackTransaction();
}

Handle java.lang.Exception {
Log("Policy Execution Failed" + ErrorMessage);
// If there is no transaction, this won't do anything
RollbackTransaction();
}
```

Chapter 4. Stored procedures

You can call database stored procedures from within a policy using the CallStoredProcedure function.

You can use this function with Sybase, Microsoft SQL Server, DB2SQL, and Oracle databases.

Oracle stored procedures

The CallStoredProcedure function works with Oracle data sources.

The `CallStoredProcedure` function works in two ways:

- With automatic schema discovery
- Without automatic schema discovery

Automatic schema discovery is a feature of `CallStoredProcedure` with which Netcool/Impact automatically discovers the schema of the procedure before sending the procedure request to the database. This is the default behavior of the application.

Automatic schema discovery makes it easier to write stored procedure policies. This is because you do not have to explicitly declare the procedure schema in the policy body before you call the `CallStoredProcedure` function. However, because two database requests are made every time it calls the function is called, running the policy with automatic schema discovery creates an additional performance load on the database. This can also slow the performance, because the policy engine waits for the Oracle database to respond to both requests in sequence before continuing on to process the rest of the policy.

To avoid the extra processing load on the database and to minimize effects on performance, you can disable automatic schema discovery and explicitly specify the stored procedure schema in the body of the policy.

If you use multiple procedures that have the same name but are stored in different packages, you must disable automatic schema discovery. If automatic schema discovery is not disabled, the Netcool/Impact cannot resolve the procedure correctly and it displays an error. After you disable the automatic schema discovery, define the procedure name argument as *<packagename>.<procedurename>* in the policy where you invoke the `CallStoredProcedure` function.

To disable schema discovery globally for all policies, set the following property in the server properties file:`impact.storedprocedure.discoverprocedureschema=false`

The server properties file is named *servername*`_server.props`, where *servername* is the name of the Impact Server. The default value for this property is `true`. You can also disable schema discovery on a per-policy basis.

Writing policies with automatic schema discovery

You can call certain types of stored procedures from within a policy.

- Procedures that return scalar values.

- Procedures that return arrays. Oracle allows stored procedures to return an array of values as an output parameter. Typically, this array represents a row and each element in the array represents a row field.
- Procedures that return cursors. Oracle allows stored procedures to return a cursor as an output parameter. This cursor is an array of arrays that typically represents a set of rows in a database.

Calling procedures that return scalar values

Use this procedure to call an Oracle stored procedure that returns scalar values as output parameters.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  This example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

  ```
  Sp_Parameter.Hostname = "192.168.1.25";
  Sp_Parameter.Location = "New York";
  ```
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  Make sure that you use the data source name, not the name of a data type. In addition, the name of the stored procedure is case sensitive and has to appear exactly as it is defined in the database.

  The following example shows how to call the CallStoredProcedure function:

  ```
  CallStoredProcedure("ORA_01", "GetHostnameByIP", Sp_Parameter);
  ```

Creating the Sp_Parameter context:

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables:

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function:

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Example of an Oracle stored procedure that returns a scalar value:

The following example shows how to call an Oracle stored procedure that returns a scalar value.

Procedure

In this example, you call a procedure named GetHostnameByIP. This procedure has one input parameter named IPAddress and one output parameter named Hostname. The example calls the stored procedure and then prints the Hostname value to the policy log.

```
// Create the Sp_Parameter context

Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with
// input parameter values

Sp_Parameter.IPAddress = "192.168.1.25";

// Call CAllStoredProcedure and pass the name of the data
// source, the name of the stored procedure and Sp_Parameter

DataSource = "ORA_01";
StoredProc = "GetHostnameByIP";

CallStoredProcedure(DataSource, StoredProc, Sp_Parameter);

// Print the value of the Hostname output parameter
// to the policy log

Log(Sp_Parameter.Hostname);
```

The next example shows a shorter version:

```
Sp_Parameter = NewObject();
Sp_Parameter.IPAddress = "192.168.1.25";
CallStoredProcedure("ORA_01", "GetHostnameByIP", Sp_Parameter);
Log(Sp_Parameter.Hostname);
```

Calling procedures that return an array

Use this procedure to call an Oracle stored procedure that returns an array as an output parameter.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

  ```
  Sp_Parameter.CustType = "Premium";
  Sp_Parameter.Location = "New York";
  ```

  The following example shows how to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called `Name` and contains an Oracle `VARRAY`.

  ```
  Sp_Parameter.Name = NewObject();
  Sp_Parameter.Name.type = "CUST";
  ```

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  Make sure that you use the data source name, not the name of a data type. In addition, the name of the stored procedure is case sensitive and has to appear exactly as it is defined in the database.

  The following example shows how to call the CallStoredProcedure function:

  ```
  CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);
  ```

- You can now handle the returned array by accessing a member of the Sp_Parameter context that uses the same name as the underlying array type in the data source.

Creating the Sp_Parameter context:

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables:

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the `Sp_Parameter` member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called `Name` and contains an Oracle `VARRAY`.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `IPAddress` and `Location` input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function:

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is `Oracle_01` and the name of the stored procedure is `GetCustomerByID`.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Handling the returned array:

When you call the stored procedure, the underlying data source returns an array.

Procedure

Netcool/Impact assigns this array as a member variable in the Sp_Parameter context. The name of the member variable is the same name as the user-defined array data type returned from the underlying data source.
The following example shows how to handle the array returned by a stored procedure. In this example, the name of the array data type in the Oracle database is CUST.

```
CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);

Log("The name of the Customer is " + Sp_Parameter.Name.elements[0]);
Log("The location of the Customer is " + Sp_Parameter.Name.elements[1]);
```

Example of an Oracle stored procedure that returns an array:

The following complete example shows how to call an Oracle stored procedure that returns an array as an output parameter.

Procedure

In this example, the name of the data source is ORA_01, and the name of the stored procedure is GetCustomerByLocation. The stored procedure returns a CUST array with two fields. The first field stores the customer's name. The second field stores the customer's location.

```
// Create the Sp_Parameter context.

Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with values
// for the input parameters of the stored procedure

Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";

// Create an Sp_Parameter member variable that stores
// returned VARRAY

Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.Type = "CUST";

// Call CallStoredProcedure and pass the name of the data source,
// the name of the stored procedure and the Sp_Parameter context.

DataSource = "ORA_01";
```

```
ProcName = "GetCustomerByLocation";

CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Print the name and location of the customer to the policy log.

Log("The name of the Customer is " + Sp_Parameter.Name.elements[0]);
Log("The location of the Customer is " + Sp_Parameter.Name.elements[1]);
```

Calling procedures that return a cursor

Use this procedure to call an Oracle stored procedure that returns a cursor as an output parameter.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

  ```
  Sp_Parameter.CustType = "Basic";
  Sp_Parameter.Location = "Shanghai";
  ```
- Create the output parameter context using NewObject
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  The following example shows how to call the CallStoredProcedure function:

  ```
  CallStoredProcedure("ORA_01", "GetCustomersByLocation", Sp_Parameter);
  ```
- You can now handle the returned cursor by accessing the output parameter context that you created.

Creating the Sp_Parameter context:

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables:

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the Sp_Parameter member variables with values for the CustType and Location input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the Sp_Parameter member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called Name and contains an Oracle VARRAY.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the Sp_Parameter member variables with values for the IPAddress and Location input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function:

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Handling the returned cursor:

When you call the stored procedure, the underlying data source returns the cursor.

Procedure

Netcool/Impact converts the cursor as an array of arrays, where the first set of arrays represents the rows returned in the cursor, and the second set of arrays represents the fields in the rows.
The following example shows how to handle the cursor returned from a stored procedure.

```
Count = Length(Sp_Parameter.CUSTOMERS.elements);

While (Count > 0) {
    Index = Count - 1;
    Elements = Sp_Parameter.CUSTOMERS.elements[Index];
    Log("Customer name is: " + Elements.Name);
    Log("Customer ID is: " + Elements.ID);
    Count = Count - 1;
}
```

Example of an Oracle stored procedure that returns a cursor:

The following complete example shows how to call an Oracle stored procedure that returns a cursor as an output parameter.

Procedure

In this example, the name of the data source is `ORA_02` and the name of the stored procedure is `GetCustomerByLocation`. The stored procedure returns a cursor that consists of multiple rows from the database. Each row has multiple fields, among which are `Name` and `ID`.

```
// Create the Sp_Parameter context.

Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with
// values to pass as input parameters to the stored
// procedure.

Sp_Parameter.CustType = "Basic";
Sp_Parameter.Location = "Shanghai";

// Create the output parameter context

Sp_Parameter.CUSTOMERS = NewObject();

// Call CallStoredProcedure and pass the name of the
// data source, the stored procedure name and the
// Sp_Parameter context.

DataSource = "ORA_02";
ProcName = "GetCustomersByLocation";

CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Interate through the arrays in the output parameter
// context and print out the values of the Name and
// ID fields.

Count = Length(Sp_Parameter.CUSTOMERS);

While (Count > 0) {
    Index = Count - 1;
    Elements = Sp_Parameter.CUSTOMERS.elements[Index];
    Log("Customer name is: " + Elements.Name);
    Log("Customer ID is: " + Elements.ID);
    Count = Count - 1;
}
```

Writing policies without automatic schema discovery

Follow these steps if you want to write policies without automatic schema discovery.

Procedure

- Disable schema discovery globally for all policies.

  You do this by setting the `impact.storedprocedure.discoverprocedureschema=false` property in the server properties file:

  You can also disable schema discovery on a per-policy basis.
- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Create a new context for each parameter that you intend to pass to the procedure and assign this context to an Sp_Parameter member variable.
- Optional: Create a return parameter context.

  If you are calling a stored function that has a return parameter, you specify it as the first parameter in the Sp_Parameter context.
- Optional: Set the DiscoverProcedureSchema variable.

  If you do not want to globally disable schema discovery, you can disable it on a per policy basis.
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  The following example shows how to call the CallStoredProcedure function:

  ```
  CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
  ```

Disabling schema discovery globally

Use this procedure to disable schema discovery globally for all policies.

About this task

To avoid extra processing load on the database and to minimize performance issues, you can disable automatic schema discovery.

If you use multiple procedures that have the same name but that are stored in different packages, you must disable automatic discovery. If you do not disable automatic schema discovery, Netcool/Impact displays an error.

In both cases, you can specify the stored procedure schema directly in the policy as an alternative.

Procedure

Set the following property in the server properties file:

```
impact.storedprocedure.discoverprocedureschema=false
```

The server properties file is named *servername*`_server.props`, where *servername* is the name of the Impact Server. The default value for this property is `true`.

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Creating the parameter contexts

After you have created Sp_Parameter context, you create a new context for each parameter that you intend to pass to the procedure and assign this context to an Sp_Parameter member variable.

Procedure

To create a new parameter context, you call the `NewObject` function and assign it as a member of `Sp_Parameter` as follows:

```
Sp_Parameter["1"] = newObject();
```

where the name of the member variable is an index value starting with 1. Specify the index number as a string. Specify the parameters in the order in which they appear in the stored procedure call.
After you have created a parameter context, you then assign it a set of member variables that specify the parameter name, type, type name, direction, and value. Table 9 shows the valid parameter types and type names.

Table 9. Parameter types and type names

Type	Typename
3	Decimal, Number, Integer, Numeric, Smallint, Float
12	Varchar, Varchar2, String
93	Date
1111	Clob, Varray, Ref Cursor

You can assign the members variables in the parameter context as follows:

```
Sp_Parameter["1"].name = "firstName";
Sp_Parameter["1"].type = 12;
Sp_Parameter["1"].typename = "String";
Sp_Parameter["1"].direction = "INOUT";
Sp_Parameter["2"].value = "Muse";
```

Creating a return parameter context

If you are calling a stored function that has a return parameter, you specify it as the first parameter in the Sp_Parameter context.

Procedure

You specify the name and direction of this parameter as `RETURN` and the value as an empty string. Identify the return parameter with an index value of 1.
You can create the return parameter context and assign its member variables as follows:

```
Sp_Parameter["1"] = NewObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 3;
Sp_Parameter["1"].typename = "Integer";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";
```

Setting the DiscoverProcedureSchema variable

Use this procedure to set the DiscoverProcedureSchema variable.

Procedure

If you do not want to globally disable schema discovery as described previously in this section, you can disable it on a per policy basis by setting the DiscoverProcedureSchema variable to false as follows:

```
DiscoverProcedureSchema = false;
```

Calling the CallStoredProcedure function

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the Sp_Parameter variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Example of specifying schemas in a policy

The following example shows how to explicitly specify schemas in a policy when calling a variety of different stored procedure types.

```
/** Examples of CallStoredProcedure in new format*/

log("-------------Test standalone procedure combine_and_format_names------");

Sp_Name="combine_and_format_names";

Sp_Parameter = newObject();

Sp_Parameter["1"] = newObject();
Sp_Parameter["1"].name = "firstName";
Sp_Parameter["1"].type = 12;
Sp_Parameter["1"].typename = "String";
Sp_Parameter["1"].direction = "INOUT";
Sp_Parameter["1"].value = "Micro";

Sp_Parameter["2"] = newObject();
Sp_Parameter["2"].name = "lastName";
Sp_Parameter["2"].type = 12;
Sp_Parameter["2"].typename = "String";
Sp_Parameter["2"].direction = "INOUT";
Sp_Parameter["2"].value = "Muse";

Sp_Parameter["3"] = newObject();
Sp_Parameter["3"].name = "fullName";
```

```
Sp_Parameter["3"].type = 12;
Sp_Parameter["3"].typename = "String";
Sp_Parameter["3"].direction = "OUT";
Sp_Parameter["3"].value = "";

Sp_Parameter["4"] = newObject();
Sp_Parameter["4"].name = "nameFormat";
Sp_Parameter["4"].type = 12;
Sp_Parameter["4"].typename = "String";
Sp_Parameter["4"].direction = "IN";
Sp_Parameter["4"].value = "FIRST,LAST";

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);

log("Full name in given name format: " + Sp_Result.FULLNAME);

log("--------------End combine_and_format_names------------------");

log("--------------Test function returning NUMBER-------------------------");

Sp_Name = "returnNumber";

Sp_Parameter = newObject();

// Note: Return parameter has to be always first parameter

Sp_Parameter["1"] = newObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 3;
Sp_Parameter["1"].typename = "NUMBER";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";

Sp_Parameter["2"] = newObject();
Sp_Parameter["2"].name = "MAX_SALARY";
Sp_Parameter["2"].type = 3;
Sp_Parameter["2"].typename = "NUMBER";
Sp_Parameter["2"].direction = "IN";
Sp_Parameter["2"].value = 4;

Sp_Parameter["3"] = newObject();
Sp_Parameter["3"].name = "factor";
Sp_Parameter["3"].type = 3;
Sp_Parameter["3"].typename = "NUMBER";
Sp_Parameter["3"].direction = "IN";
Sp_Parameter["3"].value = 2;

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);

log( "power(max_salary , factor): " + Sp_Result.RETURN );

log("--------------End function returnNumber-------------------------------");

log("-------------Test procedure into package returning VARRAY--------------");

function printArray_ActionNode() {
    log("Array type : " + Sp_Result.pbooks.type);
    log("Array elements : " + Sp_Result.pbooks.elements[0]);
    log("Array elements : " + Sp_Result.pbooks.elements[1]);
}

Sp_Name = "procedureAndFunction.select_into_subject1";
```

```
Sp_Parameter = NewObject();

Sp_Parameter["1"] = newObject();
Sp_Parameter["1"].name = "psubject_id";
Sp_Parameter["1"].type = 12;
Sp_Parameter["1"].typename = "String";
Sp_Parameter["1"].direction = "IN";
Sp_Parameter["1"].value = "CS1";

Sp_Parameter["2"] = newObject();
Sp_Parameter["2"].name = "psubject_name";
Sp_Parameter["2"].type = 12;
Sp_Parameter["2"].typename = "String";
Sp_Parameter["2"].direction = "IN";
Sp_Parameter["2"].value = "Computer Science";

Sp_Parameter["3"] = newObject();
Sp_Parameter["3"].name = "pbooks";
Sp_Parameter["3"].type = 1111;
Sp_Parameter["3"].typename = "VARRAY";
Sp_Parameter["3"].direction = "OUT";
Sp_Parameter["3"].value = newObject();
Sp_Parameter["3"].value.type = "BOOKLIST1";

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1' , Sp_Name , Sp_Parameter);

printArray_ActionNode();

log("-------------End procedure procedureAndFunction.select_into_subject1---");

log("------Test function into package with no parameters returning ResultSet----");

log("Example -- Cursor as a OUT parameter");

function printCursorNum_ActionNode() {
    totalItems = Sp_Result.RETURN.Num;
    array = Sp_Result.RETURN.elements;
    log("Total Number of elements in cursor:  " + totalItems);
    getEachCursorElement_ActionNode(totalItems, array);
}

function getEachCursorElement_ActionNode(totalItems, array) {
    log("Printing Cursor Values");
    item = array[index];
    log("emp id: " + item.EMP_ID);  // column names have to be UpperCase /
    log("emp Name: " + item.EMP_NAME);
    index=index+1;
    runFunction0 = false;
    if (index < totalItems) {
        runFunction0=true;
    }
    if ( runFunction0 = true ) {
        printCursorNum_ActionNode();
    }
}

Sp_Name = "procedureAndFunction.returnResultSet";

Sp_Parameter = newObject();

Sp_Parameter["1"] = newObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 1111;
Sp_Parameter["1"].typename = "REF CURSOR";
Sp_Parameter["1"].direction = "RETURN";
```

```
Sp_Parameter["1"].value = "";

index = 0;

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1', Sp_Name, Sp_Parameter);

printCursorNum_ActionNode();

log("-----End function procedureAndFunction.returnResultSet-------------------");

log("-----DATE support-------------------------------------------------------------");

function printCursorNum_ActionNode() {
    totalItems = Sp_Result.RETURN.Num;
    array = Sp_Result.RETURN.elements;
    log("PP_SP Total Number of elements in cursor :  " + totalItems);
    getEachCursorElement_ActionNode(array);
    }

function getEachCursorElement_ActionNode(array) {
    item = array[index];
    log("StartTime: " + item.STARTTIME);  // column names have to be UpperCase
    log("StartTime: " + item.AGGINTERVAL);
    log("applicationName : " + item.APPLICATIONNAME);
    log("interfaceName :  " + item.INTERFACENAME);
    index=index+1;
    runFunction0 = false;
    if (index < totalItems) {
        runFunction0=true;
    }
    if ( runFunction0 = true ) {
        printCursorNum_ActionNode();
    }
}

Sp_Name = "procedureAndFunction.supportdate";

Sp_Parameter = newObject();

/// Following three date formats are supported /
//Sp_Parameter.STARTTIME_IN = "12-MAY-2003";
//Sp_Parameter.STARTTIME_IN = "2003-05-12";
//Sp_Parameter.STARTTIME_IN = "2003-05-12 02:03:04.5";

Sp_Parameter["1"] = newObject();
Sp_Parameter["1"].name = "RETURN";
Sp_Parameter["1"].type = 1111;
Sp_Parameter["1"].typename = "REF CURSOR";
Sp_Parameter["1"].direction = "RETURN";
Sp_Parameter["1"].value = "";

Sp_Parameter["2"] = newObject();
Sp_Parameter["2"].name = "STARTTIME_IN";
Sp_Parameter["2"].type = 93;
Sp_Parameter["2"].typename = "DATE";
Sp_Parameter["2"].direction = "IN";
Sp_Parameter["2"].value = "2003-05-12 00:00:00.0";

Sp_Parameter["3"] = newObject();
Sp_Parameter["3"].name = "INTERVAL";
Sp_Parameter["3"].type = 3;
Sp_Parameter["3"].typename = "NUMBER";
Sp_Parameter["3"].direction = "IN";
Sp_Parameter["3"].value = 5;
```

```
index = 0;

DiscoverProcedureSchema = false;

CallStoredProcedure('oracleOnOracle1' , Sp_Name, Sp_Parameter);

printCursorNum_ActionNode();

log("-------------End DATE support---------------------");
```

Sybase and Microsoft SQL Server stored procedures

You can use CallStoredProcedure to call the following types of procedures.

- Procedures that return a single value
- Procedures that return database rows

Automatic schema discovery is always used when Sybase or Microsoft SQL Server stored procedures are called. Unlike with Oracle stored procedures, you cannot disable automatic schema discovery.

Calling procedures that return a single value

Use this procedure to call a Sybase stored procedure that returns a single value.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  The following example shows how to populate the `Sp_Parameter` member variables with values for the `IPAddress` and `Location` input parameters:

  ```
  Sp_Parameter.IPAddress = "192.168.1.25";
  Sp_Parameter.Location = "Singapore";
  ```
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  The following example shows how to call `CallStoredProcedure`. In this example, the name of the data source is `SYB_01` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

  ```
  MyReturn = CallStoredProcedure("SYB_01", "GetCustomersByLocation", Sp_Parameter);
  ```
- You can now handle the returned value by accessing the first element of the array returned by `CallStoredProcedure`.

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the `Sp_Parameter` member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called `Name` and contains an Oracle `VARRAY`.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `IPAddress` and `Location` input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is `Oracle_01` and the name of the stored procedure is `GetCustomerByID`.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is `SYB_03` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is `DB2DS` and the name of the stored procedure is `GetHostnameByIP`.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Handling the returned value

When you call the stored procedure, the underlying data source returns a value as the result.

Procedure

Netcool/Impact assigns this value to the first element of an array and returns it from the `CallStoredProcedure` function.
The following example shows how to handle the results of a Sybase stored procedure that returns a single value.

```
MyReturn = CallStoredProcedure("SYB_01", "GetNodeByIPAddress", Sp_Parameter);

Log("The value returned by the procedure is: " + MyReturn[0]);
```

Example of a Sybase stored procedure that returns a single value

The following complete example shows how to call a Sybase stored procedure that returns a single value.

In this example, the data source name is `SYB_01` and the stored procedure name is `GetNodeByIPAddress`. The results of the stored procedure are assigned to the `MyReturn` array.

```
// Create the Sp_Parameter context.

Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with values for
// the stored procedure input parameters.

Sp_Parameter.IPAddress = "192.168.1.250";
Sp_Parameter.Location = "Melbourne";

// Call CallStoredProcedure and pass the name of the data source,
// the name of the stored procedure and Sp_Parameter as input
// parameters

DataSource = "SYB_01";
ProcName = "GetHostnameByIPAddress";

MyReturn = CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

Log("The hostname of the system is: " + MyReturn[0]);
```

Calling procedures that return database rows

Use this procedure to call a Sybase stored procedure that returns a set of database rows.

Procedure

- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

  The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters.

  ```
  Sp_Parameter.CustType = "Platinum";
  Sp_Parameter.Location = "Singapore";
  ```
- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  In this example, the name of the data source is `SYB_03` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

- You can now handle the returned rows by accessing the array of contexts returned by the CallStoredProcedure function.

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the `Sp_Parameter` member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called `Name` and contains an Oracle `VARRAY`.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `IPAddress` and `Location` input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is Oracle_01 and the name of the stored procedure is GetCustomerByID.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is SYB_03 and the name of the stored procedure is GetCustomersByLocation. The results of the stored procedure are assigned to the MyReturn variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is DB2DS and the name of the stored procedure is GetHostnameByIP.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Handling the returned rows

When you call the stored procedure, the underlying data source returns a set of database rows.

Procedure

Netcool/Impact creates an array of contexts and assigns each row to a context. Within each context, the member variables correspond to fields in the row. Netcool/Impact then returns the array from the CallStoredProcedure function. The following example shows how to handle the set of database rows returned from a Sybase stored procedure.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);

Count = Length(MyReturn);

While(Count > 0) {
    Index = Count - 1;
    Log("The customer name is: " + MyReturn[Index].Name);
    Log("The customer ID is: " + MyReturn[Index].ID);
    Count = Count - 1;
}
```

Example of a Sybase stored procedure that returns a set of database rows

The following complete example shows how to call a Sybase stored procedure that returns a set of database rows.

In this example, the data source is named SYB_03 and the stored procedure is named GetCustomersByLocation. The results of the stored procedure are stored as an array of contexts in the MyResults array.

```
// Create the Sp_Parameter context

Sp_Parameter = NewObject();

// Populate the Sp_Parameter member variables with values for
// the stored procedure input parameters.

Sp_Parameter.CustType = "Platinum";
Sp_Parameter.Location = "Mumbai";

// Call CallStoredProcedure and pass the data source name, the
// stored procedure name and Sp_Parameter as input parameters.

DataSource = "SYB_03";
ProcName = "GetCustomerByLocation";
```

```
MyResults = CallStoredProcedure(DataSource, ProcName, Sp_Parameter);

// Print the customer name and IDs to the policy log.

Count = Length(MyResults);

While(Count > 0) {
    Index = Count - 1;
    Log("The customer name is: " + MyReturn[Index].Name);
    Log("The customer ID is: " + MyReturn[Index].ID);
    Count = Count - 1;
}
```

DB2 SQL stored procedures

You can call DB2 SQL stored procedures from within a policy using the CallStoredProcedure function.

The `CallStoredProcedure` function sends a request to the database that contains the procedure name and its parameters. The results of the procedure are returned to the policy in a format that can be processed by Netcool/Impact. You can use the `CallStoredProcedure` function to call the following types of procedures:

- Procedures that accept **IN**, and **INOUT** parameters.
- Procedures that return values in **INOUT** and **OUT** parameters.
- Procedures that return Result Sets.

Parameters are useful in DB2 SQL procedures when implementing logic that is conditional on a particular input or set of input scalar values. You can also use the parameters when you want to return one or more output scalar values and you do not want to return a result set. DB2 SQL supports stored procedures with parameters that only accept an input value **IN**, that only return an output value **OUT**, or that accept an input value and return an output value **INOUT**. **IN** and **OUT** parameters are passed by value, and **INOUT** parameters are passed by reference.

A result set is the set of rows that a DB2 SQL procedure returns for a SELECT statement. You can either discover result set definitions by specifying values for the input values, or you manually define a result set and its columns.

Automatic schema discovery is always used when DB2 SQL stored procedures are called. Unlike Oracle stored procedures, you cannot disable automatic schema discovery for DB2 SQL stored procedures.

Calling procedures that return scalar values

Use the following steps to create a DB2 SQL stored procedure.

Procedure

- Create a DB2 SQL data source
- Create a new context called Sp_Parameter that is used to store input and output variables for the procedure.

  When you have an DB2 SQL data source, the next step before you call any DB2 SQL stored procedure is to create a context called `Sp_Parameter`. The Sp_Parameter context is used to store input and output variables for the DB2 SQL stored procedure.
- Populate the Sp_Parameter member variables with the input parameter values for the stored procedure.

Include the **IN** and **INOUT** parameters. There is no need to populate the Sp_Parameter with an **OUT** parameter because the value will be in the Sp_Parameter automatically after the CallStoredProcedure function runs successfully.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure:

```
Sp_Parameter.Hostname = '192.168.1.25';
Sp_Parameter.Location = 'New York';
```

- Call the CallStoredProcedure function and pass the name of the data source, the name of the stored procedure, and Sp_Parameter.

  When you call `CallStoredProcedure`, the function calls the procedure in the DB2 data source. The function then passes values for the input parameters from Netcool/Impact to the DB2 stored procedure.

  Then the function returns the output parameters as member variables in the `Sp_Parameter`. The names of the member variables correspond to the names of the output parameters.

  The following example shows how to call `CallStoredProcedure`. In this example, the name of the data source is `DB2DS` and the name of the stored procedure is `GetHostnameByIP`:

  ```
  CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
  ```

Creating a DB2 SQL data source

You must have a DB2 SQL data source for DB2 SQL stored procedures to work in Netcool/Impact.

Procedure

Information about how to create a DB2 SQL data source is documented in the *User Interface Guide*. Go to the chapter on *Data sources*, and refer to the section on *Creating SQL data sources*. You can use the following link to the information center website to access the *User Interface Guide*.
http://publib.boulder.ibm.com/infocenter/tivihelp/v8r1/topic/com.ibm.netcoolimpact.doc5.1.1/welcome.html

Creating the Sp_Parameter context

Before you call any stored procedure you need to create a new context called Sp_Parameter.

Procedure

To create a new Sp_Parameter context call the `NewObject` function as follows.

```
Sp_Parameter = NewObject();
```

Populating the Sp_Parameter member variables

Use these guidelines to populate the Sp_Parameter member variables.

Make sure that the name of the member variables is exactly the same as those of the input parameters in the procedure. Specify a value for each parameter in the stored procedure, even if you want to accept the default.

The following example shows how to populate the `Sp_Parameter` member variables with values for the `Hostname` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.Hostname = "192.168.1.25";
Sp_Parameter.Location = "New York";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `CustType` and `Location` input parameters in the stored procedure.

```
Sp_Parameter.CustType = "Premium";
Sp_Parameter.Location = "New York";
```

Use this code snippet to populate the `Sp_Parameter` member variable with a new context that will contain an output parameter returned from the stored procedure. In this example, the output parameter is called `Name` and contains an Oracle `VARRAY`.

```
Sp_Parameter.Name = NewObject();
Sp_Parameter.Name.type = "CUST";
```

The following example shows how to populate the `Sp_Parameter` member variables with values for the `IPAddress` and `Location` input parameters.

```
Sp_Parameter.IPAddress = "192.168.1.25";
Sp_Parameter.Location = "Singapore";
```

Calling the CallStoredProcedure function

When you call CallStoredProcedure, the function calls the procedure in the specified data source and returns the output parameters as member variables in Sp_Parameter.

You pass it the name of the data source, the name of the stored procedure, and the `Sp_Parameter` variable. Names of the member variables correspond to the names of the output parameters. Make sure that you use the data source name, not the name of a data type.

The following examples demonstrate how to call the CallStoredProcedure function.

In this example, the name of the data source is `Oracle_01` and the name of the stored procedure is `GetCustomerByID`.

```
CallStoredProcedure("Oracle_01", "GetCustomerByID", Sp_Parameter);
```

In this example, the name of the data source is `SYB_03` and the name of the stored procedure is `GetCustomersByLocation`. The results of the stored procedure are assigned to the `MyReturn` variable.

```
MyReturn = CallStoredProcedure("SYB_03", "GetCustomersByLocation", Sp_Parameter);
```

In this example, the name of the data source is `DB2DS` and the name of the stored procedure is `GetHostnameByIP`.

```
CallStoredProcedure('DB2DS', 'GetHostnameByIP', Sp_Parameter);
```

Examples of DB2 SQL stored procedures using parameters

Examples of DB2 SQL stored procedures using **`IN`**, **`OUT`**, and, **`INOUT`** parameters, and a stored procedure that returns a result set.

Example of inserting values to a table using IN parameters

The following example of a stored procedure, `insert_data_procedure` accepts two **`IN`** parameters and inserts the values to a table.

```
Data_Source = 'DB2DS';
Sp_Name = 'insert_data_procedure';
Sp_Parameter = NewObject();
```

```
Sp_Parameter.HostName = 'mycompany.com';
Sp_Parameter.IPAddress = '192.168.1.25';
CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
```

Example of a stored procedure that uses IN and OUT parameters

The following example of a stored procedure, `get_ip_info`, accepts an **`IN`** parameter, which is a `HostName` and returns the IP address using the **`OUT`** parameter `IPAddress`.

```
Data_Source = 'DB2DS';
Sp_Name = 'get_ip_info';
Sp_Parameter = NewObject();
Sp_Parameter.HostName = 'mycompany.com';
CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
Log("IP Address is: " + Sp_Parameter.IPAddress);
```

Example of a stored procedure that returns a result set

The following example shows a DB2 SQL stored procedure that returns a result set.

```
Data_Source = 'DB2DS';
Sp_Name = 'get_all_ip_data';
NetworkData = CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
Num_IP = Length(NetworkData);
log("First IP " + NetworkData[0].IPAddress);
```

In the example, `NetworkData` is an array with each entry representing a row of data.

- To get the number of rows in the result set, use the length function for example;

  ```
  Length(NetworkData);
  ```

- To access a specific field value, use the `Array[index].<fieldname>` syntax for example;

  ```
  NetworkData[0].IPAddress;.
  ```

  In this instance, `NetworkData[0]` contains the values for the first row of the result set. `NetworkData[0].IPAddress` returns the value for the column IPAddress for that first row.

Examples of DB2 SQL stored procedures that return an array

SQL procedures support parameters and variables of array types. Arrays are a convenient way of passing transient collections of data between an application and a stored procedure or between two stored procedures.

The following example shows a DB2 SQL stored procedure that returns an array.

```
Data_Source = 'DB2DS';
Sp_Name = 'get_host_names';
Sp_Parameter = NewObject();
```

In the following example, `HOST_LIST` is an **`OUT`** parameter containing an array of host names. When an **`OUT`** parameter contains an array, you must specify the value `TYPE` as an array `TYPE = ARRAY` before calling the stored procedure using the `CallStoredProcedure` function.

```
Sp_Parameter.HOST_LIST = NewObject();
Sp_Parameter.HOST_LIST.TYPE = "ARRAY";
CallStoredProcedure(Data_Source, Sp_Name, Sp_Parameter);
```

In this example myHosts stores the array returned from the **OUT** parameter:

```
myHosts = Sp_Parameter.HOST_LIST;

Num_Hosts = Length(myHosts);
i = 0;
while (i < Num_Hosts) {
  log("Host Name is " + myHosts[i]);
  i = i + 1;
}
```

Chapter 5. Filters

A filter is a text string that sets out the conditions under which Netcool/Impact retrieves the data items.

The use of filters with internal, SQL, LDAP, and some Mediator data types is supported. The format of the filter string varies depending on the category of the data type.

SQL filters

SQL filters are text strings that you use to specify a subset of the data items in an internal or SQL database data type.

For SQL database and internal data types, the filter is an SQL `WHERE` clause that provides a set of comparisons that must be true in order for a data item to be returned. These comparisons are typically between field names and their corresponding values.

Syntax

For SQL database data types, the syntax of the SQL filter is specified by the underlying data source. The SQL filter is the contents of an SQL `WHERE` clause specified in the format provided by the underlying database. When the data items are retrieved from the data source, this filter is passed directly to the underlying database for processing.

For internal data types, the SQL filter is processed internally by the policy engine. For internal data types, the syntax is as follows:

```
Field
    Operator
    Value [AND | OR | NOT (Field
    Operator
    Value) ...]
```

where *Field* is the name of a data type field, *Operator* is a comparative operator, and *Value* is the field value.

Attention: Note that for both internal and SQL data types, any string literals in an SQL filter must be enclosed in single quotation marks. The policy engine interprets double quotation marks before it processes the SQL filter. Using double quotation marks inside an SQL filter causes parsing errors.

Operators

The type of comparison is specified by one of the standard comparison operators. The SQL filter syntax supports the following comparative operators:

- >
- <
- =
- <=
- =>

- !=
- LIKE

  Restriction: You can use the LIKE operator with regular expressions as supported by the underlying data source.

The SQL filter syntax supports the AND, OR and NOT boolean operators.

Tip: Multiple comparisons can be used together with the AND, OR, and NOT operators.

Order of operation

You can specify the order in which expressions in the SQL are evaluated using parentheses.

Examples

Here is an example of an SQL filter:

```
Location = 'NYC'
Location LIKE 'NYC.*'
Facility = 'Wandsworth' AND Facility = 'Putney'
Facility = 'Wall St.' OR Facility = 'Midtown'
NodeID >= 123345
NodeID != 123234
```

You can use this filter to get all data items where the value of the `Location` field is `New York`:

```
Location = 'New York'
```

Using this filter you get all data items where the value of the `Location` field is `New York` or `New Jersey`:

```
Location = 'New York' OR Location = 'New Jersey'
```

To get all data items where the value of the Location field is `Chicago` or `Los Angeles` and the value of the `Level` field is `3`:

```
(Location = 'New York' OR Location = 'New Jersey') AND Level = 3
```

LDAP filters

LDAP filters are filter strings that you use to specify a subset of data items in an LDAP data type.

The underlying LDAP data source processes the LDAP filters. You use LDAP filters when you do the following tasks:

- Retrieve data items from an LDAP data type using `GetByFilter`.
- Retrieve a subset of linked LDAP data items using `GetByLinks`.
- Delete individual data items from an LDAP data type.
- Specify which data items appear when you browse an LDAP data type in the GUI.

Syntax

An LDAP filter consists of one or more boolean expressions, with logical operators prefixed to the expression list. The boolean expressions use the following format:

```
Attribute
    Operator
    Value
```

where *Attribute* is the LDAP attribute name and *Value* is the field value.

The filter syntax supports the =, ~=, <, <=, >, >=, and ! operators, and provides limited substring matching using the * operator. In addition, the syntax also supports calls to matching extensions defined in the LDAP data source. White space is not used as a separator between attribute, operator, and value, and those string values are not specified using quotation marks.

For more information on LDAP filter syntax, see Internet RFC 2254.

Operators

As with SQL filters, LDAP filters provide a set of comparisons that must be true in order for a data item to be returned. These comparisons are typically between field names and their corresponding values. The comparison operators supported in LDAP filters are:

- =
- ~=,
- <
- <=
- >
- >=
- !

One difference between LDAP filters and SQL filters is that any Boolean operators used to specify multiple comparisons must be prefixed to the expression. Another difference is that string literals are not specified using quotation marks.

Examples

Here is an example of an LDAP filter:

```
(cn=Mahatma Gandhi)
(!(location=NYC*))
(&(facility=Wandsworth)(facility=Putney))
(|(facility=Wall St.)(facility=Midtown)(facility=Jersey City))
(nodeid>=12345)
```

You can use this example to get all data items where the common name value is `Mahatma Gandhi`:

```
(cn=Mahatma Gandhi)
```

Using this example you get all data items where the value of the `location` attribute does not begin with the string `NYC`:

```
(!(location=NYC*))
```

To get all data items where the value of the `facility` attribute is `Wandsworth` or `Putney`:

```
(|(facility=Wandsworth)(facility=Putney))
```

Mediator filters

You use Mediator filters with the GetByFilter function to retrieve data items from some Mediator data types.

The syntax for Mediator filters varies depending on the underlying DSA. For more information about the Mediator syntax for a particular DSA, see the DSA documentation.

Chapter 6. Functions

The Impact Policy Language (IPL) and JavaScript support built-in functions and user-defined functions.

Unless stated otherwise the same functions can be used for IPL and JavaScript languages. There are differences in the syntax used in IPL and JavaScript.

You use variables to pass values to functions. The variable is updated after the function is complete. As a result, you can only use variables to pass values to functions.

Activate

The Activate function runs another policy.

After the policy finishes running, Netcool/Impact returns to the first policy and processes any subsequent statements that it contains.

You can run a policy by name or by data item.

To run a policy by name, call `Activate` and pass the name of the policy to the function as an input parameter.

To run a policy by data item, you first retrieve the item from the internal data repository and then call `Activate`, and pass it to the function as an input parameter. Policies are stored in the repository in the internal `Policy` data type. You can retrieve policy data items by calling the `GetByKey`, `GetByFilter`, or `GetByLinks` functions. For data items of type `Policy`, the value of the `KEY` field is the same as the policy name. You can use this value in a key expression when you call `GetByKey` or use it in an SQL filter string when you call `GetByFilter`.

When you call the `Activate` function, the secondary policy inherits the variables set in the original policy. These include the `EventContainer` variable and those variables that store data items retrieved from internal or external data types. These variables are not passed back to the original policy after the second policy finishes running.

Syntax

The `Activate` function has the following syntax:

```
Activate([DataItem], [PolicyName])
```

Parameters

The Activate function has the following parameters.

Table 10. Activate function parameters

Parameter	Type	Description
DataItem	Data item	Data item that stores the policy to be activated. Pass a null value for this parameter if you are activating the policy by name.
PolicyName	String	Name of the policy to trigger. Pass a null value for this parameter if you are activating the policy by data item.

Example

The following example shows how to use the Activate function to run a policy by name.

```
// Call Activate and pass the name of the policy as an input
// parameter

Activate(null, "POLICY_01");
```

The following example shows how to use the Activate function to run a policy by data item.

```
// Call GetByKey and pass the name of the data type and
// the key expression as input parameters

DataType = "Policy";
Key = "POLICY_01";
MaxNum = 1;

MyPolicy = GetByKey(DataType, Key, MaxNum);

// Call Activate and pass the policy data item as an input
// parameter

Activate(MyPolicy[0], null);
```

ActivateHibernation

The ActivateHibernation function continues running a policy that was previously put to sleep using the Hibernate function. You must also run the RemoveHibernation function to remove the policy from the hibernation queue and to free up memory resources.

The policy is continued at the statement that follows the Hibernate function call. After the policy finishes running, Netcool/Impact returns to the original policy and processes any remaining statements that it contains.

Before you run a hibernating policy, you must first retrieve it from the internal data repository. Hibernating policies are stored in the repository as data items in the internal Hibernation data type. You retrieve a hibernation data item by calling the GetHibernatingPolicies or the GetByFilter function. If you call GetByFilter, you use an SQL filter to specify the action key value that identifies the hibernating policy. After you have retrieved the data item, call ActivateHibernation and pass it to the function as an input parameter.

Syntax

The ActivateHibernation function has the following syntax:

```
ActivateHibernation(Hibernation)
```

Parameters

The ActivateHibernation function has the following parameters.

Table 11. ActivateHibernation function parameters

Parameter	Type	Description
Hibernation	Data Item	Data item that stores the hibernating policy.

Examples

The following example shows how to continue running a hibernating policy using the GetHibernatingPolicies and ActivateHibernation functions.

```
// Call GetHibernatingPolicies and pass the start and end action keys
// as input parameters

StartActionKey = "ActionKeyAAAA";
EndActionKey = "ActionKeyZZZZ";
MaxNum = 1;

MyHiber = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);

// Call ActivateHibernation and pass the Hibernation data item as an
// input parameter

ActivateHibernation(MyHiber[0]);
```

The following example shows how to continue running a hibernating policy using the GetByFilter and ActivateHibernation functions.

```
// Call GetByFilter and pass the name of the Hibernation data type,
// and a filter the specifies an action key that identifies the hibernation

DataType = "Hibernation";
Filter = "ActionKey = 'ActionKey0001'";
CountOnly = false;

MyHiber = GetByFilter(DataType, Filter, CountOnly);

// Call ActivateHibernation and pass the Hibernation data item as an
// input parameter

ActivateHibernation(MyHiber[0]);
```

AddDataItem

The AddDataItem function adds a data item to a data type.

You can use AddDataItem with internal, SQL database, and some Mediator data types.

Before you add a new data item, you must first create a new context using the NewObject function. Then you assign values to its member variables, where each variable corresponds to a field in the data type. After you have assigned these values, call AddDataItem and pass the data type name and the context to the

function as input parameters. When you call AddDataItem, the values of the variables are used to populate fields in the new item.

Syntax

The AddDataItem function has the following syntax:

```
[DataItem =] AddDataItem(DataType, ContextToCopy)
```

Parameters

The AddDataItem function has the following parameters.

Table 12. AddDataItem function parameters

Parameter	Type	Description
DataType	String	Name of the data type.
ContextToCopy	Context	Name of the context whose member variables contain initial field values for the data item.

Return value

The AddDataItem function can optionally return the new data item.

Example

The following example shows how to add a data item to a data type using the AddDataItem function. This example uses an internal or SQL database data type named Node. Data items in this type have three fields, Id, Name, and Location.

```
// Call NewObject and populate the member variables of the context with initial
// field values for the data item

MyContext = NewObject();
MyContext.Id = "000123";
MyContext.Name = "ORACLE_01";
MyContext.Location = "Raleigh";

// Call AddDataItem and pass the name of the data type and the context as
// input parameters

DataType = "Node";

AddDataItem(DataType, MyContext);
```

BatchDelete

The BatchDelete function deletes a set of data items from a data type.

You can use BatchDelete with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.

You can specify which items to delete using an SQL filter or by passing the items to the function in an array.

To delete data items using an SQL filter, call BatchDelete and pass the name of the data type and the filter string to the function as input parameters. The filter string specifies which data items to delete. It uses the SQL filter syntax, which is similar

to the syntax of the WHERE clause in an SQL SELECT statement. For more information about SQL filters, see "SQL filters" on page 69.

To delete data items by passing them in an array, you first retrieve them from the data type by calling GetByFilter, GetByKey, or GetByLinks. Then call BatchDelete and pass the name of the data type and the data item array to the function as input parameters.

Syntax

The BatchDelete function has the following syntax:

```
BatchDelete(DataType, [DeleteFilter], [DeleteDataItems])
```

Parameters

The BatchDelete function has the following parameters.

Table 13. BatchDelete function parameters

Parameter	Type	Description
DataType	String	Name of the data type.
DeleteFilter	String	SQL filter string that specifies which data items to delete. Optional.
DeleteDataItems	Array	Array of data items to delete. Optional.

Examples

The following example shows how to delete a set of data items using an SQL filter. In this example, you delete all of the data items in a data type named Customer where the value of the Location field is Raleigh.

```
// Call BatchDelete and pass the name of the data type and a filter string as
// input parameters

DataType = "Customer";
Filter = "Location = 'Raleigh'";

BatchDelete(DataType, Filter, null);
```

The following example shows how to delete a set of data items by passing them to the BatchDelete function in an array. In this example, you delete all of the data items in the data type Server where the value of the Facility field is SE_0014.

```
// Call GetByFilter and pass the name of the data type and a filter string as
// input parameters

DataType="Server";
Filter="Facility = 'SE_0014'";
CountOnly=false;

MyServers = GetByFilter(DataType, Filter, CountOnly);

// Call BatchDelete and pass the array of data items as an input parameter

BatchDelete(DataType, null, MyServers);
```

BatchUpdate

The BatchUpdate function updates field values in a set of data items in a data type.

You can use BatchUpdate with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.

To update the field values, call BatchUpdate and pass the name of the data type, a filter string, and an update expression to the function as input parameters. The filter string specifies which data items to update. It uses the SQL filter syntax, which is similar to the syntax of the WHERE clause in an SQL SELECT statement. The update expression is a comma-separated list of field assignments similar to the contents of the SET clause in an SQL UPDATE statement. For more information about SQL filters, see "SQL filters" on page 69.

Syntax

The BatchUpdate function has the following syntax:

```
NumberOfUpdates = BatchUpdate(DataType, Filter, UpdateExpression)
```

Parameters

The BatchUpdate function has the following parameters.

Table 14. BatchUpdate function parameters

Parameter	Type	Description
DataType	String	Name of the data type.
Filter	String	SQL filter string that specifies which data items to update.
UpdateExpression	String	Expression that specifies the fields to update and the corresponding updated values. The expression is a comma-separated list of field assignment similar to the SET clause in an SQL UPDATE statement.

Return value

This function returns a Num value, that is the number of rows that were updated.

Example

The following example shows how to update field values in a set of data items. In this example, you update the Location and Facility fields of items in a data type named Server.

```
// Call BatchDelete and pass the name of the data type, a filter string and
// an update expression as input parameters

DataType = "Server";
Filter = "Location = 'New York'";
UpdateExpression = "Location = 'Raleigh', Facility = 'SE_0014'";

BatchUpdate(DataType, Filter, UpdateExpression);
```

BeginTransaction

The BeginTransaction is a local transactions function that is used in SQL operations.

You use this function to start the transaction. This function is used a policy in conjunction with other local transactions functions.

For more information about the local transactions functions, see Chapter 3, "Local transactions," on page 39.

Arguments

The `BeginTransaction()` function takes no arguments.

Note: The ObjectServer does not support the use of the BeginTransaction function.

CallDBFunction

CallDBFunction calls an SQL database function.

You can use `CallDBFunction` with SQL database data types. You cannot use this function with internal, LDAP, or Mediator data types.

To call the function, call `CallDBFunction` and pass the name of a data type, a filter string, and the function expression as input parameters. The data type identifies the underlying SQL database where the function is to be run. The function expression is the function call that is to be run by the database. `CallDBFunction` returns the value that results from the function.

Using `CallDBFunction` is equivalent to running the following SQL statement against the database:

```
SELECT function FROM table WHERE filter
```

where *function* is the specified function expression, *table* is the data type name, and *filter* is the filter string.

Syntax

`CallDBFunction` has the following syntax:

```
Integer | Float | String | Boolean = CallDBFunction(DataType, Filter, Metric)
```

Parameters

The `CallDBFunction` function has the following parameters.

Table 15. CallDBFunction function parameters

Parameter	Type	Description
DataType	String	Name of a data type associated with the underlying SQL database.

Table 15. CallDBFunction function parameters (continued)

Parameter	Type	Description
Filter	String	Filter string that specifies which data items in the data type to run the function against. Not required for all types of database functions. To run the function without consideration for specific rows of data in the associated database table, enter a filter string such as `0=0` that always evaluates to `true`.
Metric	String	Database function expression.

Return value

`CallDBFunction` returns the value that resulted from the database function.

Examples

The following example shows how to call a database function named `NOW()` and return the results of the function for use in a policy.

```
// Call CallDBFunction and pass the name of a data type, a filter
// string and the function expression

DataType = "Server";
Filter = "0 = 0";
Metric = "NOW()";

DBTime = CallDBFunction(DataType, Filter, Metric);
```

CallStoredProcedure

The CallStoredProcedure function calls a database stored procedure.

You can use this function with Sybase, Microsoft SQL Server, DB2SQL, and Oracle databases.

For detailed instructions on calling stored procedures from within a policy, see Chapter 4, “Stored procedures,” on page 43.

Syntax

The `CallStoredProcedure` function has the following syntax:

```
[Array =] CallStoredProcedure(DataSource, ProcedureName, Sp_Parameter)
```

Parameters

The `CallStoredProcedure` function has the following parameters.

Table 16. CallStoredProcedure function parameters

Parameter	Format	Description
DataSource	String	Name of the data source associated with the database.

Table 16. CallStoredProcedure function parameters (continued)

Parameter	Format	Description
ProcedureName	String	The *ProcedureName* parameter can be just the stored procedure name or it can also be the fully qualified name using the following naming convention: *Catalog.Schema.ProcedureName*. For example, *AdventureWorks.HumanResources.uspUpdateEmployeePersonalInfo*. In instances where you try to call the procedure with just the procedure name, for example, *uspUpdateEmployeePersonalInfo* and you get an exception when you run the function; you can use the fully qualified name instead. For example, *AdventureWorks.HumanResources.uspUpdateEmployeePersonalInfo*.
`Sp_Parameter`	Context	Context named `Sp_Parameter` that contains the input parameters for the stored procedure as a set of name/value pairs. You cannot substitute any other name for this parameter.

Return value

Array of contexts that contain the output for the stored procedure. Returned for Sybase stored procedures only.

ClassOf

The `ClassOf` function returns the data type of a variable.

Syntax

The `ClassOf` function has the following syntax:

```
String = ClassOf(Var)
```

Parameters

The `ClassOf` function has the following parameter.

Table 17. ClassOf function parameters

Parameter	Format	Description
Var	Variable	Variable whose format you want to obtain.

Return value

Data type name for the variable.

Note:

- If you pass an integer variable to `ClassOf` it returns as long in IPL and returns as double in JavaScript.
- If you pass a context variable to `ClassOf`, it returns as `BindingsVarGetSettable` in IPL and returns as `JavaScriptScriptableWrapper` in JavaScript.
- If you pass an `OrgNode` variable to `ClassOf`, it returns as `OrgNode` in IPL and returns as `VarGetSettable` in JavaScript.

Example

The following example shows how to return the data type of a variable:

```
MyString = "This is a string.";
MyType = ClassOf(MyString);
Log(MyType);
```

This example prints the following message to the policy log:

```
Parser Log: String
```

CommandResponse

Use the CommandResponse function to run interactive and non-interactive programs on both local and remote systems.

The CommandResponse function sends a series of commands to a system using telnet, ssh, or tn3270 and then returns responses from the system to the policy for handling. The default communication protocol is telnet.

When you call the function, CommandResponse connects to a remote port on the system using the connection information that you specify and returns a new context that identifies the session. To send a command, you set the value of the context's SendCommand variable to the text of the command. Then you assign one or more substrings that match the expected response to the context's ExpectList variable. If the value of the system response matches one or more of the substrings in the ExpectList array, the value of the context's ResponseReceived variable is set to the full text of the response. To end the remote session, you set the value of the context's Disconnect variable to true.

The CommandResponse is similar to the functionality provided by the Expect utility. For information about concepts related to this tool, see the Expect Web site at http://expect.nist.gov.

Syntax

The CommandReponse function has the following syntax:

```
Session = CommandResponse(Host, UserName, Password|UserCredentials,
InitialPrompt|Options, Port, Timeout, Expiration)
```

Parameters

The CommandResponse function has the following parameters.

Table 18. CommandResponse function parameters

Parameter	Format	Description
Host	String	Host name or IP address of the remote system.
UserName	String	User name for a remote system account.
Password	String	Password for the remote system account. Not required if UserCredentials parameter is specified.
InitialPrompt	String	Substring that matches the last line in the initial response returned by the remote application. Using all or a subset of the expected service command line prompt generally provides the best results. If the prompt contains a trailing space, include it in the string passed as this parameter. Not required if *Options* parameter is specified.

Table 18. CommandResponse function parameters (continued)

Parameter	Format	Description
Options	Context	Context that specifies an initial response string and other settings specific to the remote system connection. See "Options"for more information.
Port	Integer	Port used by the remote application. 23 (telnet) is the default.
Timeout	Integer	Command timeout in seconds. If the remote system does not respond to a command in less than the timeout value, the session disconnects. Default is 60.
Expiration	Integer	Entire session expiration timeout in seconds. Must be greater than Timeout. Default is 600.

User credentials

You can specify user credentials and other options for the remote service using the *UserCredentials* context.

You can set the following member variables in this context.

Table 19. UserCredentials context member variables

Name	Description
Password	Password for the remote system account.
PassPhrase	An ssh passphrase. Required only if you are using public key authentication with ssh.
KeyFile	Location and name of the private key file located on the system where Netcool/Impact is running. Required only if you are using public key authentication with ssh.

Options

You can specify session initiation options for telnet, ssh, and tn3270 connections using the Options context.

You can set the following member variables in this context.

Table 20. Options context member variables

Name	Description
Service	Specifies the service running on the remote system. Possible values are telnet, ssh and tn3270. Default is telnet.

Table 20. Options context member variables (continued)

Name	Description
AutoInitiate	Specifies whether Netcool/Impact should automatically connect and log in using the supplied host, port, user name and password. Possible values are `true` or `false`. If set to `true`, Netcool/Impact automatically connects and logs into the remote application. If set to `false`, the policy must manually initiate a connection and login by setting the `CommandResponse` context variables `Connect` and `Login`, `successively`, to `true`. Note that Netcool/Impact ignores the actual values set to the `Connect` and `Login` variables. This syntax is used to provide consistency with other features of the policy language.
CommandTerminator	Specifies the syntax used to terminate a command sent to the to the remote system.
LoginPrompt	Specifies the login prompt sent by the service running on the remote system.
PasswordPrompt	Specifies the password prompt sent by the service running on the remote system.

Defaults

The `CommandResponse` function uses the following connection defaults.

Table 21. CommandResponse defaults

Parameter	Default
Port	`23`
Timeout	`60` (1 minute)
Expiration	`600` (10 minutes)
Options.AutoInitiate	`true`
Options.CmdTerminator	`\n`
Options.LoginPrompt	`ogin`
Options.PasswordPrompt	`ssword`

Return value

The `CommandResponse` function returns a context that identifies the remote session. This context has the following member variables.

Table 22. Session context member variables

Variable	Format	Description
SendCommand	String	Used to send a command to the remote system.
ExpectList	String[]	Array of substrings that match the possible expected responses for a command.
MatchedPrompt	String	Substring from the `ExpectList` array that matched the response from the remote system.
ResponseReceived	String	Response received from the remote system.

Telnet

To connect to a remote system using telnet, you call `CommandResponse` and pass the host name or IP address, a user name and password, an initial prompt substring, and the port number used by the telnet service on the remote host (default is 23). You can also pass a session timeout and expiration value in seconds. `CommandResponse` connects to the remote port on the system and returns a new context that identifies the session.

The following example shows how to send a command to a remote system using telnet. In this example, you connect to the telnet application running on port 23 on `localhost` and send an `ls -l /tmp` command.

```
// Call CommandResponse and pass the required values as
// input paramatersHost = "localhost";
UserName = "demouser1";
Password = "demouser";
Port = 23;
Timeout = 30;
Expiration = 60;
InitialPrompt = "[demouser1@localhost ~]$ ";

Session = CommandResponse(Host, UserName, Password, InitialPrompt, \
    Port, Timeout, Expiration);

// Set the value of the SendCommand variable to the text of the command
// that you want to execute on the remote system

Session.SendCommand = "ls -l /tmp";

// Set the value of the ExpectList variable to a substring that matches
// the expected value of the string returned by the remote host

Session.ExpectList = {"[demouser1@localhost ~]$ "};

// Print the response from the remote host to the policy log

Log(Session.ResponseReceived);

// Disconnect from the remote host

Session.Disconnect = true;
```

SSH

To connect to a remote system using ssh, you call `CommandResponse` and pass the host name or IP address, a `UserCredentials` context, an `Options` context, and the port number used by the ssh service on the remote host (default is 22). You can also pass a session timeout and expiration value in seconds.

For ssh connections, the `UserCredentials` context specifies a login password for the remote service and, optionally, an SSH passphrase, and the location of an RSA or DSA private key file on the system where the Netcool/Impact server is running. The `Options` context specifies the service to use (in this case, ssh) and the initial response options for the connection. Use of public key authentication is optional.

The `CommandResponse` function only supports the SSH-2 protocol. SSH-1 is not supported.

If you want to use public key authentication with ssh and the `CommandResponse` function, you must first generate a public/private key pair using the `ssh-keygen`

utility. The following command session example shows how to generate the key pair. In this example, the utility creates the key files in the $IMPACT_HOME/ssh directory.

```
cd $IMPACT_HOME
mkdir .ssh

chmod 700 .ssh

ssh-keygen -q -f ./ssh/id_rsa -t rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
```

Do not provide an empty passphrase at the prompt.

After you have generated the public/private key pair, you must copy the public key to the remote system where you plan to connect using ssh. You can identify the public key file generated by the ssh-keygen utility by its file name suffix, which is .pub. In the example above, ssh-keygen creates a public key file named id_rsa.pub in the $IMPACT_HOME/ssh directory.

To copy the public key to the remote system, first transfer the file to a temporary directory on the system and then append its contents to the authorized keys file located in the home directory of the user account on the system that you plan to use for authentication. This file is named authorized_keys and is located in the $HOME/.ssh directory.

When you run the ssh-keygen utility, you specify the name and location of the private key file as a command argument. In the example above, the name and location are ./ssh/id_rsa. You pass this name and location as part of the Options context when you call the CommandResponse function.

For more information about ssh and public key authentication, see the OpenSSH Web site at http://www.openssh.com.

The following example shows how to send a command to a remote system using ssh. In this example, you use a UserCredentials context to specify the password for the login user and an Options context to specify the service, which is ssh, the command prompt terminator, and an initial prompt string.

```
// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouser1";
Port = 22;
Timeout = 30;
Expiration = 60;

// Create new UserCredentials context to pass to CommandResponse
// and populate the Password member variable

UserCredentials = NewObject();
UserCredentials.Password = "p4ssw0rd";

// Create new Options context to pass to CommandResponse and
// populate the Service, CmdTerminator and InitialPrompt member
// variables

Options = NewObject();
Options.Service = "ssh";
Options.CmdTerminator = "\n";
Options.InitialPrompt = "[demouser1@localhost ~]$ ";
```

```
Session = CommandResponse(Host, UserName, UserCredentials, Options, Port,
Timeout, Expiration);

// Send a command to the ssh session

Session.SendCommand = "ls -l /tmp";


// Set the value of the ExpectList variable to a substring that
// matches the expected response

Session.ExpectList = {"[demouser1@localhost ~]$ "};


// Print the response to the policy log

Log(Session.ResponseReceived);

// Close the ssh session

Session.Disconnect = true;
```

The following example shows how to send a command to a remote system using ssh and public key authentication. In this example, you use a `UserCredentials` context to specify the password for the login user, the authentication passphrase, and the location of the private key file. You use an `Options` context to specify the service, which is `ssh` and an initial prompt string.

```
// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouser";

// Create new UserCredentials context to pass to CommandResponse
// and populate the Password, KeyFile and PassPhrase member variables

UserCredentials = NewObject();
UserCredentials.Password = "p4ssw0rd";
UserCredentials.KeyFile = "./.ssh/id_rsa";
UserCredentials.PassPhrase = "p4ssphr4se";

// Create new Options context to pass to CommandResponse and
// populate the Service and InitialPrompt member
// variables

Options = NewObject();
Options.Service = "ssh";
Options.InitialPrompt = "[demouser1@localhost ~]$ ";

Session = CommandResponse(Host, UserName, UserCredentials, \
    Options, null, null, null);

// Send a command to the ssh session

Session.SendCommand = "ls -l /tmp";

// Set the value of the ExpectList variable to a substring that
// matches the expected response

Session.ExpectList = {"[demouser1@localhost ~]$ "};

// Print the response to the policy log

Log(Session.ResponseReceived);
```

```
// Close the ssh session

Session.Disconnect = true;
```

tn3270

To connect to a remote system using tn3270, you call `CommandResponse` and pass the host name or IP address, a `UserCredentials` context, an `Options` context, and the port number used by the tn3270 service on the remote host (default is 23). You can also pass a session timeout and expiration value in seconds.

For tn3270 connections, the `UserCredentials` context specifies a login password for the remote service. The `Options` context specifies the service to use (in this case, `tn3270`) and the initial response options for the connection.

IPL and JavaScript provide a set of key strings that you can use to send special characters to the tn3270 session. You can use these key strings to move the input cursor and perform other actions when you send information using tn3270. For example, you can use key strings to pass a tab character at the end of the user name provided to `CommandResponse` for account login. If you are using tn3270 to interact with IBM Tivoli zNetview, this advances the cursor to the end of the user name string before it is entered.

You can use the following key strings with tn3270 and the `CommandResponse` function.

Table 23. CommandResponse key strings

Special character	Key string
Enter	`[enter]`
Tab	`[tab]`
F1	`[pf1]`
F2	`[pf2]`
F3	`[pf3]`
F4	`[pf4]`
F5	`[pf5]`
F6	`[pf6]`
F7	`[pf7]`
F8	`[pf8]`
F9	`[pf9]`

The following example shows how to interact with zNetview on a remote system using tn3270. In this example, you use a `UserCredentials` context to specify the password for the login user and an `Options` context to specify the service, which is `tn3270`, the login, password, and initial prompts, and the command prompt terminator.

```
// Call CommandResponse and set the required values as input parameters

Host = "localhost";
UserName = "demouserTAB_KEY";
Port = 23;
Timeout = 120;
Expiration = 240;
```

```
// Create new UserCredentials context to pass to CommandResponse
// and populate the Password member variable

UserCredentials = NewObject();
UserCredentials.Password = "Your password";

// Create new Options context to pass to CommandResponse and
// populate the Service, AutoInitiate, LoginPrompt, PasswordPrompt
// and CmdTerminator member variables

Options = NewObject();
Options.Service = "tn3270";
Options.AutoInitiate = false;
Options.LoginPrompt = "OPERATOR ID ==>";
Options.PasswordPrompt = "PASSWORD ==>";
Options.InitialPrompt = "Action===>";
Options.CmdTerminator = "ENTER_KEY";

Session = CommandResponse(Host, UserName, UserCredentials, Options, Port,
Timeout, Expiration);

Session.Connect = true;

Session.ExpectList = {"SELECTION ==>" };

Session.SendCommand = "netview";

Session.Login = true;

Session.SendCommand = "nldm list";
Session.ExpectList = {"CMD==>"};

Log(Session.ResponseReceived);

// Close the tn3270 session

Session.Disconnect = true;
```

CommitTransaction

The CommitTransaction function is a local transactions function that is used in SQL operations.

You use this function to commit the changes to the database. If the `RollbackTransaction()` function is not called, the `CommitTransaction()` function commits the changes to the database. If the `RollbackTransaction()` is called, the changes are undone and an internal flag is set to Auto Commit any future SQL operations.

You must always call the `CommitTransaction()` function to complete a transaction even if the `RollbackTransaction()` function is called.

For more information about the local transactions functions, see Chapter 3, “Local transactions,” on page 39.

Arguments

The `CommitTransaction()` function takes no arguments.

Note: The ObjectServer does not support the use of the CommitTransaction function.

CurrentContext

The CurrentContext function returns the current policy context.

The policy context consists of all of the currently defined variables in the policy, including EventContainer, DataItems, DataItem, and Num.

Important: This function must be used only in a Log statement and it cannot be assigned to a variable.

Syntax

The CurrentContext function has the following syntax:

```
CurrentContext()
```

Example

This example shows how to return the current policy context.

```
Log(CurrentContext());
```

This example prints the member variables of the context and their values to the policy log.

Decrypt

The Decrypt function decrypts a string that has been previously encrypted using Encrypt or the nci_crypt tool.

Syntax

The Decrypt function has the following syntax:

```
String = Decrypt(Expression)
```

Parameters

The Decrypt function has the following parameter.

Table 24. Decrypt function parameters

Parameter	Format	Description
Expression	String	String to decrypt.

Return value

Decrypted string.

Example

This example shows how to decrypt a string.

```
MyString = "AB953E4925B39218F390AD2E9242E81A";
MyDecrypt = Decrypt(MyString);
Log(MyDecrypt);
```

This example prints the following message to the policy log:

```
Parser Log: Password
```

DeleteDataItem

The DeleteDataItem function deletes a single data item from a data type.

To delete multiple data items, use the `BatchDelete` function. You can use `DeleteDataItem` with internal and SQL database data types.

Before you delete a data item, you must first retrieve it from the data type by calling `GetByFilter`, `GetByKey`, or `GetByLinks`. Then, call `DeleteDataItem` and pass it to the function as an input parameter.

Syntax

The `DeleteDataItem` function has the following syntax:

```
DeleteDataItem(DataItem)
```

Parameters

The `DeleteDataItem` function has the following parameter.

Table 25. DeleteDataItem function parameters

Parameter	Type	Description
DataItem	Data Item	Data item that you want to delete.

Examples

The following example shows how to delete a data item from a data type using the `DeleteDataItem` function. In this example, you retrieve the item from the data type using the `GetByFilter` function.

```
// Call GetByFilter and pass the name of the data type and a
// filter string as input parameters

DataType = "Server";
Filter = "Name = 'ORA_01'";
CountOnly = false;

MyServers = GetByFilter(DataType, Filter, CountOnly);
MyServer = MyServers[0];

// Call DeleteDataItem and pass the data type as an input parameter

DeleteDataItem(MyServer);
```

Deploy

The Deploy function copies data sources, data types, policies, and services between server clusters.

You can use this function to write automated deployment policies that copy Impact Server data between test and production environments.

Syntax

The `Deploy` function has the following syntax:

```
Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID)
```

Parameters

The Deploy function has the following parameters.

Table 26. Deploy function parameters

Parameter	Description
TargetCluster	Name of the destination server cluster.
Username	Valid Netcool/Impact user name.
Password	Valid Netcool/Impact password.
Elements	String or array of strings that specify which project components to copy between server clusters.
ElementsOfType	String that specifies which type of project components to copy between server clusters. You can specify Project, DataSource, DataType, Policy and Service.
CheckpointID	If you are using CVS and SVN version control system for Netcool/Impact, you can specify a checkpoint label. This label will be applied to all project components when checked into the version control system for the target cluster. If you are not using Subversion or you do not want to use a checkpoint label, use the null value for this parameter.

In addition to the parameters above, Deploy also optionally reads the following variables from the policy-level scope.

Table 27. Deploy function optional variables

Variable	Description
TargetNameserverHost	Host name or IP address of the system where the Name Server is running. This is typically a system where you are running the name server and GUI server as hosted applications in embedded version of WebSphere Application Server.
TargetNameserverPort	HTTP port used by the Name Server. The default is 9080.
TargetNameserverLocation	URL path on the Java application server where the name server is located. The default is /nameserver/services.
NameserverSslEnabled	Specifies whether the communication with the name server is realized over SSL. Value can be true or false. The default is false.

You use these variables when you want to deploy project data from a Netcool/Impact cluster that uses one name server to a cluster that uses a different Name Server.

Examples

The following example shows how to copy a project and all its data sources, data types, policies, and services to a server cluster named NCI_PROD_01. In this example, the name of the project is PROJECT_01. Both clusters use the same instance of the Name Server.

```
TargetCluster = "NCI_PROD_01";
Username = " tipadmin";
Password = "tippass";
Elements = "PROJECT_01";
ElementsOfType = "Project";
```

```
CheckpointID = null;

Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID);
```

The following example shows how to copy policies named POLICY_01, POLICY_02, and POLICY_03 to a server cluster named NCI_PROD_02. Both clusters use the same instance of the Name Server.

```
TargetCluster = "NCI_PROD_02";
Username = "tipadmin";
Password = "tippass";
Elements = {"POLICY_01", "POLICY_02", "POLICY_03"};
ElementTypes = "Policy";
CheckpointID = null;

Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID);
```

The following example shows how to copy a project and all its data sources, data types, policies, and services to a server cluster named NCI_PROD_03. In this example, the name of the project is PROJECT_01. The target cluster here uses a different instance of the Name Server.

```
TargetCluster = "NCI_PROD_01";
Username = "tipadmin";
Password = "tippass";
Elements = "PROJECT_01";
ElementsOfType = "Project";
CheckpointID = null;

// Specify the host and port where the nameserver for the target cluster is located

TargetNameserverHost = "192.168.1.1";
TargetNameserverPort = 9080;

Deploy(TargetCluster, Username, Password, Elements, ElementsOfType, CheckpointID);
```

DirectSQL

The DirectSQL function runs an SQL operation against the specified database and returns any resulting rows to the policy as data items.

You can use the DirectSQL function only with SQL database data types.

You use this function to perform SELECT queries with JOIN clauses and to perform other operations that cannot be carried out using GetByKey, GetByFilter, or GetByLinks. This function supports SELECT, UPDATE and DELETE statements.

Syntax

The DirectSQL function has the following syntax:

[Array =] DirectSQL(*DataSource*, *Query*, *CountOnly*)

Parameters

The DirectSQL function has the following parameters.

Table 28. DirectSQL function parameters

Parameter	Type	Description
DataSource	String	Name of the data source associated with the SQL database.

Table 28. DirectSQL function parameters (continued)

Parameter	Type	Description
Query	String	SQL operation to run against the database.
CountOnly	Boolean	Pass a `false` value for this parameter. Provided only for backwards compatibility.

Return value

The `DirectSQL` function returns an array of data items where each data item represents a row returned from the database by the SQL query. Fields in the data items have a one-to-one correspondence with fields in the returned rows.

Examples

The following example shows how to run an SQL `SELECT` operation with a `JOIN` clause against a database.

```
// Call DirectSQL and pass the name of the data source and the
// SQL SELECT statement as input parameters

DataSource = "MYSQL_01";
Query = "SELECT * FROM Customer LEFT JOIN Server ON " + \
    "Customer.Location = Server.Location";
CountOnly = false;

MyCustomers = DirectSQL(DataSource, Query, CountOnly);
```

The following example shows how to run an SQL `UPDATE` operation against a database.

```
// Call DirectSQL and pass the name of the data source and the
// SQL statement as input parameters

DataSource = "MYSQL_02";
Query = "UPDATE Customer SET Affected = true WHERE Location = 'New York'";
CountOnly = false;

DirectSQL(DataSource, Query, CountOnly);
```

The following example shows how to run an SQL `DELETE` operation against a database.

```
// Call DirectSQL and pass the name of the data source and the
// SQL statement as input parameters

DataSource = "MYSQL_03";
Query = "DELETE FROM Customer WHERE Location = 'New York'";
CountOnly = false;
DirectSQL(DataSource, Query, CountOnly);
num_rows_deleted = Num;
Log("Number of deleted rows: " + num_rows_deleted);
```

This query deletes the specified records and returns the number of rows deleted.

Caching on SELECT Statement

The `DirectSQL` function supports caching when used to run a `SELECT` statement that returns a result set from a database. Caching is configured separately for each data source.

To enable caching, you must edit the DirectSQL properties file. This file is named *servername_directsql.props*, where *servername* is the name of the Impact Server. The file is located in the `$IMPACT_HOME/etc` directory. Table 29 shows the properties in this file.

You must replace the string in each property with the data source number as represented in the data source list. The data source list is a file named *servername_datasourcelist* and is located in the `$IMPACT_HOME/etc` directory.

Table 29. DirectSQL Caching Properties

Property	Description
`impact.datasource.n.enablecaching`	Specifies whether caching is enabled for this data source. Value can be true or false.
`impact.datasource.n.cachesize`	Maximum number of rows per query to be cached.
`impact.datasource.n.querycachesize`	Maximum number of queries to be cached.
`impact.datasource.n.cacheinvalidation`	Amount of time in seconds before data items in the cache are considered stale and must be refreshed from the data source.
`impact.datasource.n.querycacheinvalidation`	Amount of time in seconds before queries in the cache are considered stale and must be refreshed from the data source.

Distinct

The `Distinct` function returns an array of distinct elements from another array.

Syntax

The `Distinct` function has the following syntax:

```
Array = Distinct(Array, [UniqueClause])
```

Parameters

The `Distinct` function has the following parameters.

Table 30. Distinct function parameters

Parameter	Format	Description
Array	Array	Array whose distinct elements you want to obtain.
UniqueClause	String	A string expression that specifies which fields must be different in all data items to return items in the array as part of the distinct result set. The format of this expression is one or more field names separated by the plus sign (+). Optional.

Return value

An array of distinct elements.

Note: In JavaScript, `Integers` return as `Float` instead of integers, for example `1` is `1.0`.

Examples

The following example shows how to return an array of distinct elements from another array using IPL.

```
MyArray = Distinct({"a", "a", "b", "b", "c"});
Log(MyArray);
```

This example prints the following message to the policy log:

```
Parser Log: {a, b, c}
```

The following example shows how to return an array of distinct elements from another array using JavaScript

```
"MyArray=Distinct(new Array["a", "a", "b", "b", "c"]);"
```

This example prints the following message to the policy log:

```
Parser Log: [a, b, c]
```

The following example shows how to return an array of distinct elements from an array of data items. In this example, you use the *UniqueClause* parameter to specify that all distinct elements returned must have different value in the `Node` and `Class` fields.

```
MyArray = Distinct(DataItems, "Node+Class");
```

Encrypt

The Encrypt function encrypts a string.

Syntax

The `Encrypt` function has the following syntax:

String = `Encrypt(`*Expression*`)`

Parameters

The `Encrypt` function has the following parameter.

Table 31. Encrypt function parameters

Parameter	Format	Description
Expression	String	String to encrypt.

Return value

An encrypted string.

Example

The following example shows how to encrypt a string.

```
MyString = Encrypt("Password");
Log(MyString);
```

This example prints the following message to the policy log:

```
Parser Log: AB953E4925B39218F390AD2E9242E81A
```

Eval

The Eval function evaluates an expression using the given context.

Syntax

The Eval function has the following syntax:

```
Integer | Float | String | Boolean = Eval(Expression, Context)
```

Note: In JavaScript, the Float variable returns extra precision, for example, 10.695671999999998 instead of 10.695672. In IPL, integer division of 10/5 is 2.0. In JavaScript, integer division of 10/5 is 2.

Parameters

The Eval function has the following parameters.

Table 32. Eval function parameters

Parameter	Format	Description
Expression	String	Expression to evaluate.
Context	Context	Context to use in evaluating the expression.

Return value

Result of the evaluated expression.

Example

The following example shows how to evaluate an expression using the given context.

```
MyContext = NewObject();

MyContext.a = 5;
MyContext.b = 10;

MyResult = Eval("a + b", MyContext);

Log(MyResult);
```

This example prints the following message to the policy log:

```
Parser Log: 15
```

EvalArray

The EvalArray function evaluates an expression using the given array.

Syntax

The EvalArray function has the following syntax:

```
Integer | Float | String | Boolean = EvalArray(Expression, Array)
```

Parameters

The EvalArray function has the following parameters.

Table 33. EvalArray function parameters

Parameter	Format	Description
Expression	String	Expression to evaluate.
Array	array	Array to use in evaluating the expression.

Return value

Result of the evaluated expression.

Note: In JavaScript, Integers return as Float instead of integers, for example 1 is 1.0.

Example

The following example shows how to evaluate an expression using the given array.

```
MyNode1 = NewObject();
MyNode1.Name = "ORA_01";
MyNode1.Location = "New York";

MyNode2 = NewObject();
MyNode2.Name = "ORA_02";
MyNode2.Location = "New York";

MyNodes = {MyNode1, MyNode2};

MyEval = EvalArray('"Name: " + Name + ", Location: " + Location', MyNodes);
Log(MyEval);
```

This example prints the following message to the policy log:

```
Parser Log: Name: ORA_01, Location: New York, Name: ORA_02, Location: New York
```

Exit

You use the Exit function to stop a function anywhere in a policy or to exit a policy.

The Exit function works differently in IPL and JavaScript. In IPL, when you use Exit in a user-defined function it exits that function, and the policy continues. In JavaScript, when you use Exit in a user-defined function in a policy it exits the entire policy. If you want to stop a function in a JavaScript policy you must use the return command in the policy.

Syntax

The Exit function has the following syntax:

```
Exit()
```

Examples

In this example, the value of the X variable is tested. If X is greater than ten, the policy terminates. If X is less than ten, it prints a message to the policy log. The following example is valid for IPL and JavaScript.

```
X = 15;
if (X > 10) {
   Log("Exiting if statement");
   Exit();
} else {
   Log("X is less than 10.");
}
Log("End of policy.");
```

The following example shows the use of the Exit function in IPL:

```
Log("Entering Policy TestExit...");
SetGlobalVar("exitFunction","false");
SetGlobalVar("exitPolicy","false");

function testExit(test){
   SetGlobalVar("exitPolicy",test);
   if (test = true){
      Log("Exiting function TestExit....");
      Exit();
   }else{
      Log("Staying in the Policy TestExit....");
   }
}

//Passing true will exit the function testExit and exit the policy
 on the second call(below)to Exit.
//Passing false will allow the function and policy to finish to the end.
testExit(false);
if(""+(GetGlobalVar("exitPolicy")) = "true"){
   log("Exiting policy...");
   Exit();
}
Log("If you see this message, the policy continued to the end....");
```

The following example shows the use of the Exit and return functions in JavaScript:

```
Log("Entering Policy TestExit...");
SetGlobalVar("exitFunction","false");
SetGlobalVar("exitPolicy","false");

function testExit(test){
   SetGlobalVar("exitPolicy",test);
   if (test == true){
      Log("Exiting function TestExit AND policy...");
      Exit();
   }else{
      Log("Staying in the Policy TestExit....");
      return;
      Log("I will not see this log statement as we have already returned");
   }
}

//Passing true will immediately exit the function testExit AND the policy.
//Passing false will allow the function and policy to finish to the end.
testExit(true);

Log("If you see this message, the policy continued to the end....");
```

Extract

The Extract function extracts a word from a string.

Syntax

The Extract function has the following syntax:

```
String = Extract(Expression, Index, [Delimiter])
```

Parameters

The Extract function has the following parameters.

Table 34. Extract function parameters

Parameter	Format	Description
Expression	String	String expression from which to extract substrings.
Index	Integer	Word position of the substring, where 0 indicates the first word.
Delimiter	array	Array of characters that separate words in the string. Default is an array with the space character as the single element.

Return value

The extracted substring.

Example

The following example shows how to extract a word from a string.

```
MyString = "This is a test.";
MyWord = Extract(MyString, 1, " ");
Log(MyWord);

MyString = "This|is|a|test.";
MyWord = Extract(MyString, 3, "|");
Log(MyWord);
```

This example prints the following message to the policy log:

```
Parser Log: is
Parser Log: test
```

Float

The Float function converts an integer, string, or Boolean expression to a floating point number.

Syntax

The Float function has the following syntax:

```
Float = Float(Expression)
```

Parameters

The `Float` function has the following parameter.

Table 35. Float function parameters

Parameter	Format	Description
Expression	Integer | String | Boolean	Expression to be converted.

Return value

The converted floating point number.

Example

The following example shows how to convert integers, strings, and Boolean expressions to float point numbers using IPL.

```
MyFloat = Float(25);
Log(MyFloat);

MyFloat = Float("25.12");
Log(MyFloat);

MyFloat = Float(true);
Log(MyFloat);
```

This example prints the following message to the policy log:

```
Parser Log: 25.0
Parser Log: 25.12
Parser Log: 1.0
```

The following example shows how to convert integers, strings, and Boolean expressions to float point numbers using JavaScript.

```
MyFloat = Float(25);
Log(MyFloat);

MyFloat = Float("25.12");
Log(MyFloat);

MyFloat = Float(true);
Log(MyFloat);
```

JavaScript does not add any decimal points to the results for integers and Boolean expressions that are used with the `Float` function.

This example prints the following message to the policy log:

```
Parser Log: 25
Parser Log: 25.12
Parser Log: 1
```

FormatDuration

The FormatDuration function converts a duration in seconds into a formatted date/time string.

Syntax

The FormatDuration function has the following syntax:

```
String = FormatDuration(Seconds)
```

Parameters

The FormatDuration function has the following parameter.

Table 36. FormatDuration function parameters

Parameter	Format	Description
Seconds	Integer	Number of seconds.

Return value

Formatted date/time string.

Example

The following example shows how to convert seconds into a formatted date/time strings.

```
Seconds = "41927";
Duration = FormatDuration(Seconds);
Log(Duration);
```

This example prints the following to the policy log:

```
Parser Log: 11:38:47s
```

GetByFilter

The GetByFilter function retrieves data items from a data type using a filter as the query condition.

To retrieve data items using a filter condition, you call GetByFilter and pass the data type name and the filter string as input parameters. The syntax for the filter string varies depending on whether the data type is an internal, SQL database, LDAP, or Mediator data type.

GetByFilter returns an array of references to the retrieved data items. If you do not assign the returned array to a variable, the function assigns it to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

You can use GetByFilter with internal, SQL database, and LDAP data types. You can also use GetByFilter with some Mediator data types.

Important: When data items are assigned to the built-in DataItem variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the DataItems variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the CommitChanges() function in your policy. The CommitChanges() function does not take any arguments.

Syntax

The GetByFilter function has the following syntax:

```
[Array =] GetByFilter(DataType, Filter, [CountOnly])
```

Parameters

The GetByFilter function has the following parameters.

Table 37. GetByFilter function parameters

Parameter	Format	Description
DataType	String	Name of the data type.
Filter	String	Filter expression that specifies which data items to retrieve from the data type.
CountOnly	Boolean	Pass a false value for this parameter. Provided for compatibility with earlier versions only.

Return value

Array of references to the retrieved data items. Optional.

Examples

The following example shows how to retrieve data items from an internal or SQL database data type.

```
// Call GetByFilter and pass the name of the data type
// and an SQL database filter expression

DataType = "Admin";
Filter = "Level = 'Supervisor' AND Location LIKE 'NYC.*'";
CountOnly = false;

MyAdmins = GetByFilter(DataType, Filter, CountOnly);
```

The following example shows how to retrieve data items from an LDAP data type.

```
// Call GetByFilter and pass the name of the data type
// and an LDAP filter expression

DataType = "Customer";
Filter = "(|(facility=NYC)(facility=NNJ))";
CountOnly = false;

MyCustomers = GetByFilter(DataType, Filter, CountOnly);
```

The following example shows how to retrieve data items from a Mediator data type.

```
// Call GetByFilter and pass the name of the data type
// and the Mediator filter exprssion

DataType = "SWNetworkElement";
Filter = "ne_name = 'DSX1 PNL-01 (ORP)'";
CountOnly = false;

MyElements = GetByFilter(DataType, Filter, CountOnly);
```

GetByKey

The GetByKey function retrieves data items from a data type using a key expression as the query condition.

To retrieve data items by key, you call GetByKey and pass the name of the data item and a key expression. The key expression varies depending on whether you want the data items to match a single key or multiple keys.

GetByKey returns an array of references to the retrieved data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

You can use GetByKey with internal, SQL database, and LDAP data types. You can also use GetByKey with some Mediator data types.

Important: When data items are assigned to the built-in DataItem variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the DataItems variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the CommitChanges() function in your policy. The CommitChanges() function does not take any arguments.

Syntax

The GetByKey function has the following syntax:

```
[Array =] GetByKey(DataType, Key, [MaxNum])
```

Parameters

The GetByKey function has the following parameters.

Table 38. GetByKey function parameters

Parameter	Format	Description
DataType	String	Name of the data type.
Key	Integer | Float | Boolean | String | Array	Key expression that specifies which data items to retrieve from the data type.
MaxNum	Integer	Maximum number of data items to retrieve. Default is 1. Optional.

Return value

Array of references to the retrieved data items. Optional.

Examples

The following example shows how to retrieve data items from a data type using a single key.

```
// Call GetByKey and pass the name of the data type,
// the key expression and the maximum number of data items
// to return.
```

```
DataType = "Customer";
Key = "12345";
MaxNum = 1;

MyCustomers = GetByKey(DataType, Key, MaxNum);
```

The following example shows how to retrieve data items from a data type using multiple keys.

Example using IPL.

```
// Call GetByKey and pass the name of the data type,
// the key expression and the maximum number of data
// items to return.

DataType = "Node";
Key = {"R12345", "D98776"};
MaxNum = 1;

MyCustomers = GetByKey(DataType, Key, MaxNum);
```

Example using JavaScript.

```
// Call GetByKey and pass the name of the data type,
// the key expression and the maximum number of data
// items to return.

DataType = "Node";
Key = ["R12345", "D98776"];
MaxNum = 1;

MyCustomers = GetByKey(DataType, Key, MaxNum);
```

GetByLinks

The GetByLinks function retrieves data items in target data types that are linked to one or more source data items.

To retrieve data items by link, you must first retrieve source data items using GetByFilter, GetByKey, or another call to GetByLinks. Then you call GetByLinks and pass an array of target data types and the sources. The function returns an array of data items in the target data types that are linked to the source data items.

GetByLinks returns an array of references to the retrieved data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

Important: When data items are assigned to the built-in DataItem variable, they are not immediately updated but are stored in a queue to optimize the number of calls to the database. So, for example, if you update multiple fields in the DataItems variable there will only be one call to update the underlying database, when a function call is made. To force all queued updates, call the CommitChanges() function in your policy. The CommitChanges() function does not take any arguments.

Syntax

The GetByLinks function has the following syntax:

```
[Array =] GetByLinks(DataTypes, [LinkFilter], [MaxNum], DataItems)
```

Parameters

The GetByLinks function has the following parameters.

Table 39. GetByLinks function parameters

Parameter	Format	Description
DataTypes	Array	Array of target data type names. The function returns data items of these data types that are linked to the source data items specified.
LinkFilter	String	Filter expression that specifies which linked data items to retrieve. Optional.
MaxNum	Integer	Maximum number of data items to retrieve. Default is 1. Optional.
DataItems	Array	Array of source data items.

Return value

Array of data items in the target data type that are linked to the source data item. Optional.

Examples

The following example shows how to retrieve data items linked to another data item.

Example using IPL.

```
// Call GetByLinks and pass the target data type,
// the maximum number of data items to retrieve and
// the source data item.

DataTypes = {"Location"};
Filter = "";
MaxNum = "10000";
DataItems = MyCustomers;

MyLocations = GetByLinks(DataTypes, Filter, MaxNum, DataItems);
```

Example using JavaScript.

```
// Call GetByLinks and pass the target data type,
// the maximum number of data items to retrieve and
// the source data item.

DataTypes = ["Location"];
Filter = "";
MaxNum = "10000";
DataItems = MyCustomers;

MyLocations = GetByLinks(DataTypes, Filter, MaxNum, DataItems);
```

The following example shows how to retrieve data items linked to another data item. In this example, the function filters the data items using the value of the Location field.

Example using IPL.

```
// Call GetByLinks and pass the target data type,
// the link filter, the maximum number of data items
// to retrieve and the source data item.
```

```
DataTypes = {"Operators"};
Filter = "Location = 'New York'";
MaxNum = "10000";
DataItems = MyCustomers;

MyOperators = GetByLinks(DataTypes, Filter, MaxNum, DataItems);
```

Example using JavaScript.

```
// Call GetByLinks and pass the target data type,
// the link filter, the maximum number of data items
// to retrieve and the source data item.

DataTypes = ["Operators"];
Filter = "Location = 'New York'";
MaxNum = "10000";
DataItems = MyCustomers;

MyOperators = GetByLinks(DataTypes, Filter, MaxNum, DataItems);
```

GetByXPath

The `GetByXPath` function provides a way to parse an XML string or get an XML string through an URL specified as parameter.

The data to be retrieved is specified as an XPath expression. When Netcool/Impact interacts with different systems there are scenarios where the data in Netcool/Impact is in XML format.

GetByXPath can be used in the following scenarios to retrieve the data for each source:

- The response from a Web services call. The GetByXPath function retrieves the data from a web service response.
- The response to a REST API call provides data in XML format. The GetByXPath function retrieves the information from the response.
- If the data read from a JMS DSA is an XML string, the GetByXPath function retrieves the data from the XML.

More information about XPath Expression is available on the W3C Web site and the W3Schools Web site:

- Go to http://www.w3.org, in the standards section, in the technology topic view, search for XPath, and then a topic titled, *XML Path Language (XPath) Version 1.0.*
- Go to http://www.w3schools.com, and search for XPath tutorial and select XPath Syntax.

Syntax

The GetByXPath function has the following syntax:

```
result=GetByXPath(inputString,namespacemapping,xPathExpression);
```

Parameters

The GetByXPath function has the following parameters.

Table 40. GetByXPath function parameters

Parameter	Format	Description
Input String	String	The input XML string from which the data must be retrieved. The input string can be a URL. The URL is identified as a string that starts with "file" or "http". If the URL is specified, the XML document is retrieved from the URL. The content is then used to extract the data. If authentication is required for the HTTP method, use the GetHTTP method to retrieve the XML string output, and use the output in the GetByXPath function.
Namespace Mapping	IPL Object	XPath expressions can contain namespace prefixes. This IPL object provides a way to specify the mapping between the prefix to the real namespace that corresponds to the prefix. The syntax for *Namespace Mapping* is `objectName.<prefixName>=<Namespace URI>` • `<prefixName>` is the prefix to be specified in the XPathExpression for an XML node. • `<Namespace URI>` is the corresponding URI in the XML document. For example, `nsMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";` • `xsi` is the prefix used in the XPath Expression. • `URI` is the URI corresponding to the prefix in the XML document.
XPathExpression	String	The XPath expression, that is applied against the input XML string and the data is retrieved.

Fix Pack 2

Return value

The return value is an Impact Object. The Impact object contains a field called `Result`.

For example:

```
result = GetByXPath(....);
val = result.Result;
```

The `val` object is another Impact object which contains the node names or the xml tags specified in the XPATH expressions as key. The values for the node names are returned as array of values.

For example:

```
<book>Book1,Book2</book>
```

The `val` Impact object has the attribute called "book"

```
val['book']
```

The values are represented as array of values in this object.

For example, {Book1,Book2} are the values that are stored in val['book']

The output is contained in a constant variable called Result that is contained in a variable called result:

For example:

```
result.Result.[NodeName] = Array of Values
```

where *NodeName* is the name of the node and *Array of Values* contains the actual values.

To access the values that are contained in the result variable, add val = result.Result.:

```
val = result.Result.
val.[NodeName] contains array of values;
```

If the return value is a string, the value will be String. If the return value is a number, the value will be of type Double. If it is a boolean, the type will be Boolean.

If a function is specified, only one function per execution is supported.

The functions can be specified as parameter inside the XPath expressions as well.

Example 1

Namespace mapping where no namespaces are defined in the *input string*.

```
nsMapping = NewObject();
```

Example 2

The following examples are of namespaces defined in the *input string*.

Example 2a:

```
nsMapping = NewObject();
nsMapping.tns="urn:www-collation-com:1.0";
nsMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";
nsMapping.col1="urn:www-collation-com:1.0"
```

Example 2b:

```
nsMapping= NewObject();
nsMapping.xsi="http://www.w3.org/2001/XMLSchema-instance";
nsMapping.soapenv="http://schemas.xmlsoap.org/soap/envelope/"
nsMapping.xsd="http://www.w3.org/2001/XMLSchema";
```

Example 3

An *input string*

```
<com:getCramerObjectDetailsResponse
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance";
xmlns:com="http://interfaces/sessions/ejb/ice/cramer/com";
xmlns:xsd="http://www.w3.org/2001/XMLSchema";
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/";
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">;
   <return>
```

```
      <cramerObjectDetails>
          <entry>
             <key xsi:type="xsd:string">AKEY</key>
             <value xsi:type="xsd:string">null</value>
          </entry>
          <entry>
             <key xsi:type="xsd:string">Class ID</key>
             <value xsi:type="xsd:string">4</value>
          </entry>
          <entry>
             <key xsi:type="xsd:string">DEVICE1</key>
             <value xsi:type="xsd:string">null</value>
          </entry>
          <entry>
              <key xsi:type="xsd:string">Model Type</key>
              <value xsi:type="xsd:string">1761493060</value>
          </entry>
          <entry>
             <key xsi:type="xsd:string">Class Name</key>
             <value xsi:type="xsd:string">CNAME</value>
          </entry>
          <entry>
             <key xsi:type="xsd:string">manufacturer</key>
             <value xsi:type="xsd:string">DEVICE2</value>
          </entry>
          <entry>
             <key xsi:type="xsd:string">ResourceId</key>
             <value xsi:type="xsd:string">81520</value>
          </entry>
       </cramerObjectDetails>
       <errorCode>0</errorCode>
       <errorMsg/>
    </return>
</com:getCramerObjectDetailsResponse>
```

Example 4

An *XPathExpression* inside a policy:

```
xPathExpr  = "//cramerObjectDetails/entry[key=\"Model Type\"]/value/text()";
xmlResult = GetByXPath(xmlStr, nsMapping, xPathExpr);
Log(xmlResult);

xPathExpr = "//cramerObjectDetails/entry/key/text() |
//cramerObjectDetails/entry/value/text() ";
xmlResult=GetByXPath(xmlStr, nsMapping, xPathExpr);
Log(xmlResult);

xPathExpr = "count(//entry)";
xmlResult=GetByXPath(xmlStr,nsMapping,xPathExpr);
Log(xmlResult);
```

Log result 1:

```
(value={1761493060})
```

Log result 2:

```
(key={AKEY, Class ID, DEVICE1, Model Type, Class Name, manufacturer, ResourceId},
value={null, 4, null, 1761493060, CNAME, DEVICE2, 81520})
```

Log result 3:

```
(FunctionResult=7.0)
```

In the log result 1, from the first expression, the text of the value node is retrieved when the text of the key node is Model Type.

In the log result 2, from the second expression, the text value of both key and value nodes is retrieved in XML.

GetClusterName

You use the GetClusterName function inside a policy to identify which cluster is running the policy.

Syntax

```
GetClusterName()
```

Parameters

This function does not take any parameters.

Return value

String type, the name of the cluster.

GetDate

The GetDate function returns the date/time as the number of seconds expired since the start of the UNIX epoch.

Syntax

The `GetDate` function has the following syntax:

```
Integer = GetDate()
```

Return value

Number of seconds since the start of the UNIX epoch.

Example

The following example shows how to return the date/time as the number of seconds in UNIX time.

```
Seconds = GetDate();
Time = LocalTime(Seconds, "MM/dd/yyyy HH:mm:ss zz");
Log(Seconds);
Log(Time);
```

This example prints the following to the policy log:

```
Parser Log: 1056042205
Parser Log: 06/19/2003 13:03:25 EDT
```

GetFieldValue

Use this function to get the value of static, or non-static fields. For non-static fields, use the variable *FieldName* for a Java class or *TargetObject* for a Java object. For a static Java class field, use the variable *ClassName*.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the `$IMPACT_HOME/dsalib` directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server because the Java class files in the `$IMPACT_HOME/dsalib` directory are not replicated between servers.

Syntax

GetFieldValue(ClassName, TargetObject, FieldName);

Parameters

Table 41. GetFieldValue function parameters

Parameter	Description
ClassName	Name of the Java class. For a non-static method call, this parameter would be set to null.
TargetObject	Name of the instantiated Java object. For a static method, this parameter would be set to null.
FieldName	Name of the field variable in the Java class to retrieve the value for.

Returns

Value of the field.

Example

Get the value of the static field named out of the Java class, *java.lang.System*:

```
fieldvalue = GetFieldValue("java.lang.System", null, "out");
```

Get the value of the non-static field *firstname* from a hypothetical Java class, *com.ibm.DeveloperAccount*. Since it is a non-static field, you are retrieving the field data off an instantiated object of the class. Assume that the constructor needs only a simple ID number to retrieve the account instance:

Example using IPL.

```
dev_acct = NewJavaObject("com.ibm.DeveloperAccount", {765224});
first_name = GetFieldValue(null, dev_acct, "firstname");
```

Example using JavaScript.

```
dev_acct = NewJavaObject("com.ibm.DeveloperAccount", [765224]);
first_name = GetFieldValue(null, dev_acct, "firstname");
```

GetGlobalVar

This function retrieves the global value saved by previous `SetGlobalVar` calls.

It should be used with a corresponding `SetGlobalVar()` call. The call only retrieves its own copy of the global variable.

Syntax

```
GetGlobalVar(variablename)
```

Parameters

Table 42. GetGlobalVar function parameters

Parameter	Description
variablename	Name of the variable whose value you want to retrieve.

Example

You can define in your policy a global flag to indicate if a particular exception happens when you run your policy. The flag is set to false originally. The exception handler in your policy sets the flag to true if an exception occurs. Your main policy can check the flag to decide if the exception happens after an action call such as `SendJMSMessage()`.

```
function GetGlobalVarTest(){
Log("\nrunTimeFlag in getglobalvartest: " + GetGlobalVar("runTimeFlag"));
}

function SetGlobalVarGetGlobalVarTest(){

Handle java.lang.NullPointerException {
    Log("\nNull pointer exception: runTimeFlag is " + GetGlobalVar("runTimeFlag"));
}
Handle java.lang.Exception {
    Log("\nException thrown: runTimeTest is " + GetGlobalVar("runTimeFlag"));
}

Date = 1235414139;
SetGlobalVar("runTimeFlag", Date);

SendJMSMessage(com.sun.appserv.naming.S1ASCtxFactory, "jms/ConnectionFactory",
"this is to test the exception handler");

Log("\nrunTimeFlag in saveglobalvar and getglobalvar test: " +
GetGlobalVar("runTimeFlag"));
GetGlobalVarTest();
}
```

GetHTTP

Fix Pack 2

You can use the `GetHTTP` function to retrieve any HTTP URL or to post content to a web page.

You can use it to:

- Retrieve web pages to get information
- Complete a form on a web page
- Run cgi, servlets, or other server scripts on the web server

Syntax

```
GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method,
AuthHandlerActionTreeName, FormParameters, FilesToSend,
HeadersToSend, HttpProperties)
```

Parameters

This function has the following parameters:

Table 43. GetHTTP function parameters

Parameter	Description
HTTPHost	IP address of the host to which the call is being made.
HTTPPort	Port number of the host to which the call is being made.
Protocol	Protocol used in the call.
Path	The remaining part of the URL being called that follows the port number.
ChannelKey	An arbitrary text.
Method	Method used with the function.
AuthHandlerActionTreeName	Name of an authorization handler action tree. This parameter is used for compatibility with earlier versions. The default is *DefaultAuthHandler*.
FormParameters	Name-value pairs that are URL encoded and added to the URL being accessed.
FilesToSend	Used in the POST method to send files.
HeaderToSend	Contains HTTP Header information in name-value pairs that needs to be added to the HTTP packet that is sent.

Table 43. GetHTTP function parameters (continued)

Parameter	Description
HttpProperties	A NewObject that contains name-value pairs. The valid variables are: • *ConnectionTimeout*: Sets the timeout until a connection is established. The default value of zero means the timeout is not used. • *ResponseTimeout*: Sets the default socket timeout (SO_TIMEOUT) in milliseconds which is the timeout for waiting for data. A timeout value of zero is interpreted as an infinite timeout. • *AuthenticationHost*: The host the credentials apply to. The host can be set to null if the credentials are applicable to any host. • *AuthenticationPort*: The port the credentials apply to. The port can be set to a negative value if the credentials are applicable to any port. • *AuthenticationScheme*: The authentication scheme the credentials apply to. The authentication scheme can be set to null if the credentials are applicable to any authentication scheme. – Basic: Basic authentication is the original and most compatible authentication scheme for HTTP. It is also the least secure as it sends the user name and password unencrypted to the server. – Digest: Digest authentication was added in the HTTP 1.1 protocol. Digest authentication is more secure than basic authentication as it never transfers the actual password across the network. Instead digest authentication uses it to encrypt a "nonce" value sent from the server. • *AuthenticationRealm*: The realm the credentials apply to. The realm can be set to null if the credentials are applicable to any realm. • *UserId*: The user name. • *Password*: The password. • *Content*: Content is generated by using methods such as creating an xml string or JSON string. You can use this property with the *HTTP POST* and *HTTP PUT* methods. • *ContentType*: The default `ContentType` is `text/xml`. You can use this property with the *HTTP POST* and *HTTP PUT* methods.

Example

The following piece of code logs the help page for the `Activate` function on an instance of Impact Server.

```
HTTPHost="9.15.165.115";
HTTPPort=9081;
Protocol="https";
Path="/nci/ActionFunctionBuilder?actionList=Activate";
ChannelKey="tom";
Method="";
AuthHandlerActionTreeName="";
FormParameters=newobject();
FilesToSend=newobject();
HeadersToSend=newobject();
```

```
//HttpProperties=null;
HttpProperties=newobject();

x=GetHTTP(HTTPHost, HTTPPort, Protocol, Path, ChannelKey, Method,
AuthHandlerActionTreeName,
FormParameters, FilesToSend, HeadersToSend, HttpProperties);

Log(x);
```

GetHibernatingPolicies

The GetHibernatingPolicies function retrieves data items from the Hibernation data type by performing a search of action key values.

To retrieve data items from the Hibernation data type, you call GetHibernatingPolicies and pass two action key values as input parameters. The function performs a search of action keys for all Hibernation data items and returns data items whose action keys fall between the two specified values.

GetHibernatingPolicies returns an array of retrieved hibernation data items. If you do not specify a return variable, the function assigns the array to the built-in DataItems variable and sets the value of the Num variable to the number of data items in the array.

Syntax

The GetHibernatingPolicies function has the following syntax:

```
[Array =] GetHibernatingPolicies(StartActionKey, EndActionKey, [MaxNum])
```

Parameters

The GetHibernatingPolicies function has the following parameters.

Table 44. GetHibernatingPolicies function parameters

Parameter	Format	Description
StartActionKey	String	Starting action key to be used in the lexicographical search.
EndActionKey	String	Ending action key to be used in the lexicographical search.
MaxNum	Integer	Maximum number of hibernations to return. Default is 1. Optional.

Return value

Array of matching Hibernation data items. Optional.

Example

The following example shows how to retrieve the first Hibernation data items whose action keys fall between the values ActionKey0001 and ActionKey1000.

```
// Call GetHibernatingPolicies and pass ActionKey0001
// and ActionKey1000 as input parameters

StartActionKey = "AlertKey0001";
EndActionKey = "AlertKey1000";
MaxNum = 1;
```

```
MyHibers = GetHibernatingPolicies(StartActionKey, EndActionKey, MaxNum);

// Call ActivateHibernation and pass the Hibernation
// as an input parameter

ActivateHibernation(MyHibers[0]);
```

GetScheduleMember

The GetScheduleMember function retrieves schedule members associated with a particular time range group and time.

To retrieve a schedule from a time range group, you call `GetScheduleMember` and pass the name of the time range group and the time in seconds. You can retrieve a specific member by position in the schedule, or retrieve all schedule members.

`GetScheduleMember` returns an array of retrieved schedule member data items. If you do not specify a return variable, the function assigns the array to the built-in `DataItems` variable and sets the value of the `Num` variable to the number of data items in the array.

Syntax

The `GetScheduleMember` function has the following syntax:

`[`*`Array`* `=] GetScheduleMember(`*`Schedule`*`, [`*`TimeToMatch`*`], [`*`ReturnAll`*`], Time)`

Parameters

The `GetScheduleMembers` function has the following parameters.

Table 45. GetScheduleMember function parameters

Parameter	Format	Description
Schedule	Data Item	Data item that contains the time range group to query for the schedule member.
TimeToMatch	Integer	Position of the schedule member in the schedule. Ignored if `ReturnAll` is specified. Optional.
ReturnAll	Boolean	If `true`, the function returns all matching schedule members. If `false`, the function returns the member specified by the `TimeToMatch` parameter. Optional.
Time	Integer	The time during which the schedule member is active. Expressed as the number of seconds since the beginning of the UNIX epoch. You can obtain this number by calling `GetDate`.

Return value

Array of the retrieved schedule member data items. Optional.

Examples

The following example shows how to retrieve the schedule member at position 0 of a time range group, using the current time.

```
// Call GetScheduleMember and pass the
// time range group, the member position and the current time

TimeRange = GetByKey("Schedule", "Weekday_Shift_01", 1);
Position = 0;
CurrentTime = GetDate();

Members = GetScheduleMember(TimeRange[0], Position, null, CurrentTime);
```

The following example shows how to retrieve all schedule members from time range group, using the current time.

```
// Call GetScheduleMember and pass the name of the
// time range group and the current time

TimeRange = GetByKey("Schedule", "Weekday_Shift_01", 1);
ReturnAll = true;
CurrentTime = GetDate();

Members = GetScheduleMember(TimeRange[0], null, ReturnAll, CurrentTime);
```

GetServerName

You use the GetServerName function inside a policy to identify which server is running the policy.

Syntax

```
GetServerName()
```

Parameters

This function does not take any parameters.

Return value

String type, the name of the Impact Server.

GetServerVar

You use this function to retrieve the global value saved by previous SetServerVar.

SetServerVar() and GetServerVar() functions can be used as a way to cache and share variable values across different policies or threads on the server. You have full control of these variables and are responsible for cleaning up global variables.

Syntax

```
GetServerVar(variablename)
```

Parameters

Table 46. GetServerVar function parameters

Parameter	Description
variablename	Name of the variable.

Example

For an example of using the `GetServerVar` function, see "SetServerVar" on page 145.

Hibernate

The Hibernate function causes a policy to hibernate.

To hibernate a policy, you call `Hibernate` and pass an action key and the number of seconds for the policy to hibernate as input parameters. The action key can be any unique key you want to use to identify the policy.

When a policy hibernates, it stops running and is stored as a data item of type `Hibernation`. At intervals, the hibernating policy activator queries the `Hibernation` data type and wakes those policies whose timeout value has expired. You can also wake a hibernating policy from within another policy using the `ActivateHibernation` function. JavaScript also supports hibernation functions.

Syntax

The `Hibernate` function has the following syntax:

```
Hibernate(ActionKey, [Reason], Timeout)
```

Parameters

The `Hibernate` function has the following parameters.

Table 47. Hibernate function parameters

Parameter	Format	Description
ActionKey	String	String that uniquely identifies the hibernating policy.
Reason	String	String that describes the reason for hibernating the policy. Optional.
TimeOut	Integer	Number of seconds before the hibernating policy is available to be wakened by the hibernating policy activator.

Example

The following example shows how to cause a policy to hibernate for one minute.

```
// Call Hibernate and pass an action key and the number of seconds
// to hibernate as runtime parameters

ActionKey = "ActionKey" + GetDate();
Timeout = 60;

Hibernate(ActionKey, null, Timeout);
RemoveHibernation(ActionKey);
```

Int

The Int function converts a float, string, or Boolean expression to an integer.

This function truncates any decimal fraction value associated with the number. For example, the value of `Int(1234.67)` is `1234`.

Syntax

The `Int` function has the following syntax:

```
Integer = Int(Expression)
```

Parameters

The Int function has the following parameter.

Table 48. Int function parameters

Parameter	Format	Description
Expression	Float | String | Boolean	Expression to be converted.

Return value

The converted integer number.

Important: A float value converted to an int value is truncated not rounded.

Example

The following example shows how to convert float, string, and boolean expressions to an integer.

```
MyInt = Int(123.45);
Log(MyInt);

MyInt = Int(123.54);
Log(MyInt);

MyInt = Int("456");
Log(MyInt);

MyInt = Int(false);
Log(MyInt);
```

This example prints the following message to the policy log:

```
Parser Log: 123
Parser Log: 123
Parser Log: 456
Parser Log: 0
```

JavaCall

You use this function to call the method MethodName in the Java object TargetObject with parameters, or, to call the static method MethodName in the Java class ClassName with parameters.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $IMPACT_HOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server because the Java class files in the $IMPACT_HOME/dsalib directory are not replicated between servers.

Syntax

JavaCall(ClassName, TargetObject, MethodName, Parameters)

Parameters

Table 49. JavaCall function parameters

Parameter	Description
ClassName	Name of the Java class. When you are using a non-static method call, this parameter is set to null.
TargetObject	Name of the instantiated Java object. When you are using a static method, this parameter is set to null.
MethodName	Name of the Java method in the Java class you are calling.
Parameters	An array of parameter values the method requires.

Returns

Value that the method returns, if any.

Examples

Call a method `println()` from the Java System object represented by variable out to print a line of text message to system stdout:

IPL example:

```
JavaCall(null, out, "println", {"Output from Impact"});
```

The same example in JavaScript using [] brackets:

```
JavaCall(null, out, "println", ["Output from Impact"]);
```

Get a system property named `app`. Call the `java.lang.System.getProperty( String key )` method with the following line:

IPL example:

```
propValue = JavaCall("java.lang.System", null, getProperty", { "app" } );
```

The same example in JavaScript:

```
propValue = JavaCall("java.lang.System", null, getProperty", [ "app" ] );
```

In this example, use this function to check JVM properties by calling methods on class `java.lang.System` from your policy. This IPL example, prints the value of a JRE system property named `app`:

```
propvalue = JavaCall("java.lang.System", null, "getProperty",
{ "app" } );
log("Property "app" is " + propvalue);
```

The same example in JavaScript:

```
propvalue = JavaCall("java.lang.System", null, "getProperty",
[ "app" ] );
Log("Property "app" is " + propvalue);
```

In the following IPL example, create a Java object of class `Vector` and call its methods:

```
// Create an new instance of Java class java.util.Vector
vector = NewJavaObject("java.util.Vector", null);
//Add "111111" to vector.
JavaCall(null, vector, "add", { "111111" });
// Retrieve element at position 0.
```

```
log("The first element is " + JavaCall(null, vector, "get", { 0 }) );
// Add element "22222" to position 0
JavaCall(null, vector, "add", { 0, "22222" });
// Print out the element at position 0.
// It should now be "22222", not "111111".
log("The first element is " + JavaCall(null, vector, "get", { 0 }) );
// Add element "33333" to vector.
JavaCall(null, vector, "add", { "33333" });
// Print out the current size of vector. The value should be 3.
log("Vector size is " + JavaCall(null, vector, "size", {}));
```

If you are using JavaScript, for a `JavaCall` that needs an integer argument you must use the `Integer.parseInt` JavaCall to create an actual integer.

```
// Create an new instance of Java class java.util.Vector
vector = NewJavaObject("java.util.Vector", null);
 index = JavaCall("java.lang.Integer", null, "parseInt", ["0"]);
//Add "111111" to vector.
JavaCall(null, vector, "add", ["111111"]);
// Retrieve element at position 0.
Log("The first element is " + JavaCall(null, vector, "get", [index]) );
// Add element "22222" to position 0
JavaCall(null, vector, "add", [index, "22222"]);
Log("The first element is " + JavaCall(null, vector, "get", [index]) );
```

JRExecAction

The JRExecAction function executes an external command using the JRExec server.

To run an external command, call `JRExecAction` and pass the name of the command, an array of strings that contain any command-line arguments and a timeout value.

Syntax

The `JRExecAction` function has the following syntax:

`JRExecAction(`*`Command`*`, `*`Parameters`*`, `*`ExecuteOnQueue`*`, `*`TimeOut`*`)`

Parameters

The `JRExecAction` function has the following parameters.

Table 50. JRExecAction function input parameters

Parameter	Format	Description
Command	String	Name of the external command, script, or program to run.
Parameters	Array	Array of parameters to pass to the command.
ExecuteOnQueue	Boolean	To place the command on the JRExec server queue, set this parameter to `true`. Commands on the queue are executed by the JRExec server in parallel mode. To run the command in parallel with any other commands that might currently be running, set to this parameter to`true`. Set this parameter to `false` for most uses of this function.
TimeOut	Integer	Number of seconds to wait after sending the command before timing out.

These parameters are only available after you execute the function:

Table 51. JRExecAction function output parameters

Parameter	Format	Description
ExecOutput	String	Return value of the script or command.
ExitCode	Integer	Exit code of the script or command.

Examples

The following example shows how to run an external command named `myscript` using the JRExec server using IPL.

```
// Call JRExecAction and pass the name of the command
// the input parameters and a timeout value
Command = "/usr/local/bin/myscript";
Parameters = {"param1", "param2", "param3"};
TimeOut = 5;
MyResult = JRExecAction(Command, Parameters, false, TimeOut);
// Output Values:
Log("ExecOutput: "+ExecOutput);
Log("ExitCode: "+ExitCode);
```

The following example shows how to run an external command named `myscript` using the JRExec server using JavaScript.

```
// Call JRExecAction and pass the name of the command
// the input parameters and a timeout value
Command = "/usr/local/bin/myscript";
Parameters = ["param1", "param2", "param3"];
TimeOut = 5;
MyResult = JRExecAction(Command, Parameters, false, TimeOut);
// Output Values:
Log("ExecOutput: "+ExecOutput);
Log("ExitCode: "+ExitCode);
```

Keys

The `Keys` function returns an array of strings that contain the field names of the given data item.

Syntax

The `Keys` function has the following syntax:

Array = Keys(*DataItem*)

Parameters

The `Keys` function has the following parameter.

Table 52. Keys function parameters

Parameter	Format	Description
DataItem	Data item	The data item whose keys you want to return.

Return value

Array of strings that contain the field names of the data item.

Example

The following example shows how to return an array of strings that contain the field names.

```
MyNodes = GetByFilter("Node", "0=0", false);
MyNode = MyNodes[0];
Fields = Keys(MyNode);
Log(Fields);
```

This example prints the names of all the fields in the `Node` data item.

Length

The Length function returns the number of elements or fields in an array or the number of characters in a string.

Syntax

The `Length` function has the following syntax:

`Integer` `= Length(`*`Array`* | *`String`*`)`

Parameters

The `Length` function has the following parameter.

Table 53. Length function parameters

Parameter	Format	Description
`Array` | *`String`*	Array or String	Array whose elements to count, or string whose characters to count.

Return value

Number of elements or characters.

Example

The following example shows how to return the number of elements in an array.

```
MyNodes = GetByFilter("Node", "Location = 'New York'", false);
NumNodes = Length(MyNodes);
Log(NumNodes);
```

This example prints the number of data items in the `MyNodes` array to the policy log.

Load

You use this function to load a JavaScript library into your JavaScript policy.

After you load a JavaScript library you can call its defined functions in your JavaScript policy.

The Load function has the following usage:

`Load(`*`libraryname`*`)`

where *libraryname* is the name of an Impact JavaScript policy, or a filename of an external JavaScript library, without the `.js` extension. To be able to load a library, you must first copy it over to the `$IMPACT_HOME/jslib` directory. After you load the library, you can call its functions by referencing their names.

Assume, for example, that your `MyLibrary.js` JavaScript policy has the following function defined:

```
function myfunc() {
  Log("Running myfunc");
}
```

You can load the `MyLibrary` policy into another JavaScript policy, and call its myfunc function using the following code:

```
Load("MyLibrary");
myfunc();
```

LocalTime

The LocalTime function returns the number of seconds since the beginning of the UNIX epoch as a formatted date/time string.

Syntax

The `LocalTime` function has the following syntax:

```
Date = LocalTime(Seconds, Pattern)
```

Parameters

The `LocalTime` function has the following parameters.

Table 54. LocalTime function parameters

Parameter	Format	Description
Seconds	Integer	Number of seconds.
Pattern	String	Date/time pattern. Optional. If not specified, the default date/time pattern is used.

Return value

A formatted date/time string.

Example

The following example shows how to return the given number of seconds in various formats.

```
// Return date/time string using default format

Seconds = GetDate();
Time = LocalTime(Seconds);
Log(Time);

// Return date/time strings using specified formats

Seconds = GetDate();
Time = LocalTime(Seconds, "MM/dd/yy");
Log(Time);
```

```
Seconds = GetDate();
Time = LocalTime(Seconds, "HH:mm:ss");
Log(Time);
```

This example prints the following message to the policy log:

```
Parser Log: Nov 11 2003, 15:44:38 EST
Parser Log: 06/19/03
Parser Log: 13:11:24
```

Log

The Log function prints a message to the policy log.

To print a message to the policy log, you call this function and pass the expression you want to print and, optionally, a log level.

The log level specifies the level of severity for the message, with 1 being the lowest and 3 being highest. The policy logger service configuration specifies the level of severity that the message must meet to be printed in the log. For example, if you configure the policy logger with a severity level of 2, only messages with a log level of 2 or less are printed. Messages with a log level of 0 are always logged.

Syntax

The Log function has the following syntax:

```
Log([LogLevel], Expression)
```

Parameters

The Log function has the following parameters.

Table 55. Log function parameters

Parameter	Format	Description
LogLevel	Integer	An Integer between 0 and 3 that specifies the level of severity for the message. Default is 0. Optional.
Expression	Integer | Float | String | Boolean	Message to print to the log.

Examples in IPL

The following example shows how to print a message to the policy log in IPL.

```
Log("This is a test.");
```

This example prints the following message to the policy log:

```
Parser Log: This is a test.
```

The following example shows how to print a message to the policy log with a severity level of 2 in IPL.

```
Log(2, "This is another test.");
```

This example prints the message to the policy log only if the policy logger service is configured with a log level of 2 or greater.

Examples in IPL and JavaScript

This is the format of a logged string in IPL:

```
log("Start with a string(IPL):"+MyContext);
Start with a string(IPL):
"Created by parser"=(Serial=POLICYADDED_01, Identifier=1314712147)
```

This is the format of a logged string in JavaScript:

```
Log("Start with a string:"+MyContext);
Start with a string:
{Identifier:1314712196, Serial:"POLICYADDED_01"}
```

Merge

The Merge function merges two contexts or event containers by adding the member variables of the source context or event container to the those of the target.

Syntax

The Merge function has the following syntax:

```
[Target] = Merge(Target, Source, [Exclude])
```

Parameters

The Merge function has the following parameters.

Table 56. Merge function parameters

Parameter	Format	Description
Target	Context | Event container	Target context or event container.
Source	Context | Event container	Source context or event container.
Exclude	Array	Array of strings that contain the names of member variables to exclude from the merge. Optional.

Return value

The merged contexts or event containers. Optional.

Examples

The following example shows how to merge two contexts.

```
MyContext1 = NewObject();
MyContext1.a = "This";
MyContext1.b = "is";

MyContext2 = NewObject();
MyContext2.c = "a";
MyContext2.d = "test.";

Merge(MyContext1, MyContext2, null);

Log(MyContext1);
```

This policy prints the following message to the policy log:

```
Parser Log: "Created by parser"=(a=This, b=is, c=a, d=test.)
```

The following example shows how to merge two contexts and exclude some member variables from the merge using IPL.

```
MyContext1 = NewObject();
MyContext1.a = "This";
MyContext1.b = "is";

MyContext2 = NewObject();
MyContext2.c = "a";
MyContext2.d = "test.";

Merge(MyContext1, MyContext2, {"c"});

Log(MyContext1);
```

This example prints the following message to the policy log:

```
Parser Log: "Created by parser"=(a=This, b=is, d=test.)
```

The following example shows how to merge two contexts and exclude some member variables from the merge using JavaScript.

```
MyContext1 = NewObject();
MyContext1.a = "This";
MyContext1.b = "is";
MyContext2 = NewObject();
MyContext2.c = "a";
MyContext2.d = "test.";
Merge(MyContext1, MyContext2, ["c"]);
Log(MyContext1);
```

This example prints the following message to the policy log:

```
Parser Log: "Created by parser"=(a=This, b=is, d=test.)
```

NewEvent

The NewEvent function creates a new event container.

Syntax

The `NewEvent` function has the following syntax:

`EventContainer` `= NewEvent(`*`EventReader`*`)`

Parameters

The `NewEvent` function has the following parameter.

Table 57. NewEvent function parameters

Parameter	Format	Description
`EventReader`	String	Name of the event reader associated with the event source.

Return value

Event container that stores the new event.

Example

The following example shows how to create a new event container, and how to populate its event field variables and the `EventReaderName` variable.

```
MyEvent = NewEvent("OMNIbusEventReader");

// Set the EventReaderName member variable, only required if the policy
// is run by a means other than an event reader service. This specifies the
// name of the event reader to use to send the event

MyEvent.EventReaderName = "OMNIbusEventReader";

// Set the event field variables

MyEvent.Identifier = "XYZ123";
MyEvent.Node = "DB_SERVER_01";
MyEvent.Class = "99999";
MyEvent.Manager = "Netcool/Impact";
MyEvent.Acknowledged = 0;
MyEvent.Severity = 5;
MyEvent.Type = 0;
```

NewJavaObject

The `NewJavaObject` function is used to call the constructor for a Java class.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the `$IMPACT_HOME/dsalib` directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server because the Java class files in the `$IMPACT_HOME/dsalib` directory are not replicated between servers.

Syntax

NewJavaObject(ClassName, Parameters)

Parameters

Table 58. NewJavaObject function parameters

Parameter	Description
ClassName	Name of the Java class you are instantiating a Java object for.
Parameters	An array of parameter values a constructor for this class requires.

Returns

The instantiated object.

Examples

To create a Java String Object `"This is a string!"` and assign it to variable `str`, then in an IPL policy, put in the following line:

```
str = NewJavaObject("String", {"This is a string!"});
```

This IPL example of code creates a Java object of class `java.util.HashTable` and then adds, retrieves, and removes data from it: Fix Pack 2

```
// Create a new instance of Java Hashtable class.
my_hash = NewJavaObject("java.util.Hashtable", null);
// Add table entry ( "one", "aaaa" } to my_hash.
JavaCall(null, my_hash, "put", { "one", "aaaa" } );
// Add entry ("two", "bbbb") to table my_hash.
JavaCall(null, my_hash, "put", { "two", "bbbb" } );
// Add entry ("three", "cccc") to table.
JavaCall(null, my_hash, "put", { "three", "cccc" });
// Print the table entry value indexed by the key "three"
log("Check hashtable value indexed by key \"three\". Value is " +
JavaCall(null, my_hash, "get", {"three" }));
// Remove the entry indexed by the key "one" from the table
JavaCall(null, my_hash, "remove", {"one"});
log("After remove call, my_hash becomes " + my_hash);
```

To create a Java String Object `"This is a string!"` and assign it to variable `str`, then in a JavaScript policy, put in the following line:

```
str = NewJavaObject("String", ["This is a string!"]);
```

This JavaScript example of code creates a Java object of class `java.util.HashTable` and then adds, retrieves, and removes data from it: Fix Pack 2

```
// Create a new instance of Java Hashtable class.
my_hash = NewJavaObject("java.util.Hashtable", null);
// Add table entry ("one", "aaaa") to my_hash.
JavaCall(null, my_hash, "put", ["one", "aaaa"]);
// Add entry ("two", "bbbb") to table my_hash.
JavaCall(null, my_hash, "put", ["two", "bbbb"] );
// Add entry ("three", "cccc") to table.
JavaCall(null, my_hash, "put", ["three", "cccc"]);
// Print the table entry value indexed by the key "three"
Log("Check hashtable value indexed by key \"three\". Value is " +
JavaCall(null, my_hash, "get", ["three"]));
// Remove the entry indexed by the key "one" from the table
JavaCall(null, my_hash, "remove", ["one"]);
Log("After remove call, my_hash becomes " + my_hash);
```

The Impact policy does not support file or directory operations. The Java API, however, supports these operations in its `java.io.*` library. You can access this library and all other functions the Java API provides by using the Java Policy functions. This piece of code, for example, calls `java.io.File` class, opens a directory, and outputs a list of the files in the directory:

This example applies to IPL.

```
homedir = NewJavaObject("java.io.File", {"/home/user/"});
file_list = JavaCall(null, homedir, "list", {});
Log("file_list is " + file_list);
```

This example applies to JavaScript.

```
homedir = NewJavaObject("java.io.File", {"/home/user/"});
file_list = JavaCall(null, homedir, "list", []);
Log("file_list is " + file_list);
```

NewObject

The NewObject function creates a new context.

Syntax

The `NewObject` function has the following syntax:

```
Context = NewObject()
```

Return value

The new context.

Example

The following example shows how to create a new context.

```
MyContext = NewObject();
```

ParseDate

The ParseDate function converts a formatted date/time string to the time in seconds since the beginning of the UNIX epoch.

Syntax

The ParseDate function has the following syntax:

```
Integer = ParseDate(Date, [Pattern])
```

Parameters

The ParseDate function has the following parameters.

Table 59. ParseDate function parameters

Parameter	Format	Description
Date	String	Formatted date/type string.
Pattern	String	String that contains the formatting pattern. Optional. If not specified, default format is used.

Return value

The time in seconds.

Example

The following example shows how to convert various formatted date/time strings to the time in seconds.

```
// Convert date/time string using default format

DateString = "Nov 11 2003, 15:44:38 EST";
Time = ParseDate(DateString);
Log(Time);

// Convert date/time strings using specified formats

DateString = "06/19/03";
Time = ParseDate(DateString, "MM/dd/yy");
Log(Time);

DateString = "13:11:24";
Time = ParseDate(DateString, "HH:mm:ss");
Log(Time);
```

This example prints the following message to the policy log:

```
Parser Log: 1068583478
Parser Log: 1056002400
Parser Log: 72684
```

Random

The Random function returns a random integer between zero and the given upper bound.

Syntax

The Random function has the following syntax:

```
Integer = Random(UpperBound)
```

Parameters

The Random function has the following parameter.

Table 60. Random function parameters

Parameter	Format	Description
UpperBound	Integer	Highest possible integer to be returned.

Return value

Random integer.

Example

The following example shows how to return a random integer between 1 and 10.

```
UpperBound = 9;
MyRandom = Random(UpperBound) + 1;
Log(MyRandom);
```

This example prints the following message to the policy log:

```
Parser Log: 6
```

ReceiveJMSMessage

The ReceiveJMSMessage function retrieves a message from the specified JMS destination.

To retrieve the message, you call this function and pass a JMSDataSource, and a message properties context as input parameters.

Syntax

The ReceiveJMSMessage function has the following syntax:

```
ReceiveJMSMessage(DataSource, MethodCallProperties)
```

Parameters

The ReceiveJMSMessage function has the following parameters:

Table 61. ReceiveJMSMessage function parameters

Parameter	Format	Description
`DataSource`	String	Existing, and valid JMS data source.
`MethodCallProperties`	Context	Context that contains optional `MessageSelector` and `Timeout`.

RemoveHibernation

The `RemoveHibernation` function deletes a data item from the Hibernation data type and removes it from the hibernation queue.

To remove a hibernation, you call `RemoveHibernation` and pass the action key for the data item as an input parameter.

Syntax

The `RemoveHibernation` function has the following syntax:

`RemoveHibernation(`*`ActionKey`*`)`

Parameters

The `RemoveHibernation` function has the following parameter.

Table 62. RemoveHibernation function parameters

Parameter	Format	Description
`ActionKey`	String	String that uniquely identifies the hibernating policy.

Example

The following example shows how to remove a hibernation whose action key is `ActionKey0001`.

```
// Call RemoveHibernation and pass the action key
// for the hibernation as an input parameter

ActionKey = "ActionKey0001";

RemoveHibernation(ActionKey);
```

Replace

The Replace function uses regular expressions to replace a substring of a given string.

Note: To replace a backslash character (\) in a string, you must escape the character twice in the expression, resulting in a string with four backslash characters (\\\\). For example, to replace the substring `first\second` in a string, you must specify it as `first\\\\second`.

Syntax

The Replace function has the following syntax:

```
String = Replace(Expression, Pattern, Substitution, MaxNum)
```

Parameters

The Replace function has the following parameters.

Table 63. Replace function parameters

Parameter	Format	Description
Expression	String	String that contains the substring to be replaced.
Pattern	String	Substring pattern to be replace.
Substitution	String	String to substitute for the substring.
MaxNum	Integer	Maximum number of replacements to perform.

Return value

The resulting string.

Example

The following example shows how to replace a substring in a string.

```
MyString = "New York";
Pattern = "York";
Substitution = "Jersey";
MyReplace = Replace(MyString, Pattern, Substitution, 1);
Log(MyReplace);
```

This example prints the following message to the policy log:

```
Parser Log: New Jersey
```

ReturnEvent

The ReturnEvent function inserts, updates, or deletes an event from an event source.

To insert a new event, you first create a new event container using NewEvent and then populate its member variables with the field values for the new event. Then you return the event to the event source using ReturnEvent.

Note that the container can contain a variable named EventReaderName that specifies the name of an event reader service. If the policy is to be run by another means besides an event reader (for example, using the GUI or nci_trigger command), you must assign a value to the EventReaderName variable. This value is the name of the event reader you want to use to send the new event.

To update an event, you change the values of the member variables in the event container as needed and then return the event to the event source using ReturnEvent.

To delete an event, you set the DeleteEvent member variable in the event container to true and then return the event to the event source using ReturnEvent.

Syntax

The ReturnEvent function has the following syntax:

```
ReturnEvent(Event)
```

Parameters

The ReturnEvent function has the following parameter.

Table 64. ReturnEvent function parameters

Parameter	Format	Description
Event	Event container	Event container that represents the event that you want to insert, update or delete.

Examples

The following example shows how to insert a new event using ReturnEvent.

```
// Set the EventReaderName variable only if policy is to be
// triggered using something other than an event reader service

MyEvent.EventReaderName = "OMNIbusEventReader";

// Create a new event container using NewEvent
// and populate its member variables

MyEvent = NewEvent("OMNIbusEventReader");
MyEvent.Node = "ACHILLES";
MyEvent.Summary = "Node not responding.";
MyEvent.AlertKey = MyEvent.Node + MyEvent.Summary;

// Return the new event to the event source

ReturnEvent(MyEvent);
```

The following example shows how to update an event in an event source.

```
// Update the values of the member variables
// in the event container

EventContainer.Summary = EventContainer.Summary + ": Updated by Netcool/Impact";

// Return the event to the event source

ReturnEvent(EventContainer);
```

The following example shows how to delete an event in an event source.

```
// Set the value of the DeleteEvent member variable
// to true

EventContainer.DeleteEvent = true;

// Return the event to the event source

ReturnEvent(EventContainer);
```

RExtract

The RExtract function uses regular expressions to extract a substring from a string.

This function supports Perl 5 regular expressions.

Syntax

The RExtract function has the following syntax:

```
String = RExtract(Expression, Pattern)
```

Parameters

The RExtract function has the following parameters.

Table 65. RExtract function parameters

Parameter	Format	Description
Expression	String	String that contains the substring to extract. The data to extract is within () in the Expression.
Pattern	String	Regular expression pattern that specifies the substring to extract.

Return value

The extracted string.

Example

The following example shows how to use the RExtract function.

```
Log(RExtract("Not responding to ping on host DB_01", "\s(DB_01).*"));
```

This example prints the following message to the policy log:

```
Parser Log: DB_01
```

RExtractAll

The RExtractAll function uses regular expression matching to extract multiple substrings from a string.

The resulting matches are returned as elements in an array. This function supports Perl 5 regular expressions.

To extract multiple substrings from a string, you call RExtractAll and pass a string expression and a regular expressions pattern. The pattern specifies the matching characters inside the expression and specifies which characters in the matching substring to extract. You identify the characters you want to extract by enclosing them in parentheses inside the pattern.

The default behavior for the RExtractAll function is to return the last match in a pattern. For example, if you have "<test1>;<test2>;<test3>" and the pattern returns everything between the parentheses <*>, by default it returns test3.

You can set the Boolean flag to false to return all the matches in a pattern, then this example "<test1>;<test2>;<test3>" returns all the matches in a pattern test1, test2, test3. If you set the Boolean flag to true the RExtractAll function returns the last match in a pattern, the same as the default behavior. The function assigns the extracted character strings as elements in an array and passes the array back to the policy.

Syntax

The RExtractAll function has the following syntax options and the use of the Boolean flag is optional:

```
Array = RExtractAll(Expression, Pattern);
Array = RExtractAll(Expression, Pattern, Flag);
```

Parameters

The RExtractAll function has the following parameters.

Table 66. RExtractAll function parameters

Parameter	Format	Description
Expression	String	String that contains the substring to extract.
Pattern	String	Regular expression pattern that specifies the substrings to extract. You specify the string to extract from the match using parentheses characters.
Boolean	true or false	Optional. Set the Boolean flag to false to return all the matches in a pattern. Set the Boolean flag to true to return the last match in a pattern.

Return value

An array of the resulting substrings.

Examples

The following examples show how to use the RExtractAll function without a Boolean flag.

Example 1

```
Expression = "Node is DB_02 on rack RK_419";
Pattern = "\\s.*(DB_02).*(RK_419).*";
Log(RExtractAll(Expression, Pattern));
```

This example prints the following message to the policy log:

```
Parser Log: {DB_02, RK_419}
```

Example 2 using IPL

```
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>");
testNum = Length(test);
Log ("Test Num= " + testNum);
Log ("Plant ID = " + test);
```

Example 2 using JavaScript

```
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
 test = RExtractAll(expression, "<plantId\\b[^>]*>(.*?)</plantId>");
 testNum = Length(test);
 Log ("Test Num= " + String(testNum));
 Log ("Plant ID = " + String(test));
```

This example prints the following message to the policy log:

```
06 Jul 2009 15:06:42,798: Parser Log: Test Num= 1
06 Jul 2009 15:06:42,799: Parser Log: Plant ID = {234}
```

The following example shows how to use the RExtractAll function with a Boolean flag set as false.

Example 3 using IPL

```
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
test = RExtractAll(expression, "<plantId\b[^>]*>(.*?)</plantId>",false);
testNum = Length(test);
Log ("Test Num= " + testNum);
Log ("Plant ID = " + (test));
```

Example 3 using JavaScript

```
expression="<plantId>122</plantId><plantId>204</plantId><plantId>234</plantId>";
 test = RExtractAll(expression, "<plantId\\b[^>]*>(.*?)</plantId>",false);
 testNum = Length(test);
 Log ("Test Num= " + String(testNum));
 Log ("Plant ID = " + String(test));
```

This example prints the following message to the policy log:

```
06 Jul 2009 15:06:42,798: Parser Log: Test Num= 3
06 Jul 2009 15:06:42,799: Parser Log: Plant ID = {122,204,234}
```

RollbackTransaction

The RollbackTransaction function rolls back any changes done by an SQL operation.

The RollbackTransaction function is a local transactions function that is used in SQL operations. The function causes all changes to the database to be undone when an exception occurs between the BeginTransaction() and CommitTransaction() functions. It is recommended to call the RollbackTransaction() function within the localized exception handler.

You must always call the CommitTransaction() function to complete a transaction even if the RollbackTransaction() function is called.

For more information about the local transactions functions, see Chapter 3, "Local transactions," on page 39.

Arguments

The RollbackTransaction() function takes no arguments.

Note: The ObjectServer does not support the use of the RollbackTransaction function.

SendEmail

The SendEmail function sends an email that uses the email sender service.

To send an email, you call the SendEmail function and pass the email address of the recipient and the text of the message. You can also pass text for the subject line and the sender field. You can send an email by passing a data item whose Email member variable contains a valid email address. This field must be named Email. You cannot use a data item that has another field that contains email addresses.

The SendEmail function uses UTF-8 encoding of the platform by default. You can customize the encoding by including the following syntax before calling SendEmail:

```
EncodingChar = "<type of charset>";
ex:
EncodingChar = "windows-1251";
```

Syntax

The SendEmail function has the following syntax:

```
SendEmail([User], [Address], [Subject], Message, [Sender], ExecuteOnQueue)
```

Parameters

The SendEmail function has the following parameters.

Table 67. SendEmail function parameters

Parameter	Format	Description
User	Data item	Data item of any data type whose Email field contains the email address of the recipient. Optional.
Address	String or a context object	Email address for the recipient of the email. If this parameter is specified, the *User* parameter is ignored. You can specify the To, CC, and BCC fields of the email separately. Multiple email addresses must be separated by a comma (,). Optional.
Subject	String	Subject for the message. Optional.
Message	String	Message body for the email.
Sender	String	Email address for the sender for the email. Optional
ExecuteOnQueue	Boolean	Specifies whether to place the outgoing message in the queue governed by the command execution manager service. If you specify true, the message is placed in the queue and sent asynchronously by this service. If you specify false, the message is sent directly by the Netcool/Impact. In this case, the policy engine waits for the message to be sent successfully before processing any subsequent instructions.

Example

The following example shows how to send an email from a policy that uses the email address of the recipient.

```
// Call SendEmail and pass the address, subject and message text
// as input parameters

Address = "srodriguez@example.com";
Subject = "Netcool/Impact Notification";
Message = EventContainer.Node + " has reported the following error condition: "
+ EventContainer.Summary;
Sender = "impact";
ExecuteOnQueue = false;

SendEmail(null, Address, Subject, Message, Sender, ExecuteOnQueue);
```

Here is an example of using the SendMail function if the Address parameter is a context object:

```
Addresses = NewObject();
Addresses.To = "to@example.com";
Addresses.Cc = "cc1@example.com,cc2@example.com";
Addresses.Bcc = "bcc@example.com";
Subject = "Netcool/Impact Notification";
Message = "Some problem was encountered";
Sender = "impact";
ExecuteOnQueue = false;
SendEmail(null, Addresses, Subject, Message, Sender, ExecuteOnQueue);
```

SendInstantMessage

The SendInstantMessage function sends an instant message using the Jabber service.

You must configure the Jabber service as described in the *Solutions Guide* before you use this function.

Fix Pack 2 To allow users to enter a password for a jabber group chat, you must add the `GroupChatPassword` variable to the policy before you call the `SendInstantMessage` function to send the group chat message.

Syntax

The `SendInstantMessage` function has the following syntax:

`SendInstantMessage(`*`To`*`, `*`Group`*`, `*`Subject`*`, `*`TextMessage`*`, `*`ExecuteOnQueue`*`)`

Parameters

The `SendInstantMessage` function has the following parameters.

Table 68. SendInstantMessage function parameters

Parameter	Format	Description
`To`	String	Screen name of the message recipient. To send multiple recipients, use a comma-separated list of names.
`Group`	String	String that identifies a chatroom (if any) to join. The format of this string is *`chatroom_name@server/nickname`*, where *`chatroom_name`* is the name of the chatroom, *`server`* is the name of a Jabber server and nickname is the name you want to use in the chat. Only available for use with Jabber servers. Optional.
`Subject`	String	Title for instant message. Only available for use with Jabber servers. Optional.
`TextMessage`	String	Content of the instant message.
`ExecuteOnQueue`	Boolean	Specifies whether to place the outgoing message in the queue governed by the command execution manager service. If you specify `true`, the message is placed in the queue and sent asynchronously by this service. If you specify `false`, the message is sent directly by the Netcool/Impact. In this case, the policy engine waits for the message to be sent successfully before processing any subsequent instructions. Optional.

Recipient ID formats

When you call `SendInstantMessage`, you specify the message recipient using the `To` parameter. The recipient ID is typically a combination of the messaging system user name and service ID. The service ID is defined in the configuration properties for the Jabber service. A set of abbreviations is also provided that you can use instead of the service ID. The format of the message recipient ID varies, depending on the instant messaging system you are using.

The Jabber interface supports the following messaging systems:

- Jabber
- ICQ
- AIM
- Yahoo!
- MSN

You can use the following recipient ID formats for the Jabber messaging service.

Table 69. Recipient IDs for Jabber messaging service

Format	Example
Jabber ID and fully-qualified service ID	`NetcoolAdmin@jabber.example.com`
Jabber ID and fully-qualified service ID with resource	`NetcoolAdmin@jabber.example.com/ops1`
Jabber ID and service abbreviation	`NetcoolAdmin@jabber`

You can use the following recipient ID formats for the ICQ messaging service.

Table 70. Recipient IDs for ICQ messaging service

Format	Example
ICQ ID and fully-qualified service ID	`137463829@icq.example.com`
ICQ ID and service abbreviation	`137463829@icq`

You can use the following recipient ID formats for the AIM messaging service.

Table 71. Recipient IDs for AIM messaging service

Format	Example
AIM ID and fully-qualified service ID	`NetcoolAdmin@aim.example.com`
AIM ID and service abbreviation	`NetcoolAdmin@aim`

You can use the following recipient ID formats for the Yahoo! messaging service.

Table 72. Recipient IDs for Yahoo! messaging service

Format	Example
Yahoo! ID and fully-qualified service ID	`NetcoolAdmin@yahoo.example.com`
Yahoo! ID and service abbreviation	`NetcoolAdmin@yahoo`

The Jabber interface provides three sets of recipient ID formats for use with the MSN messaging service. The first format is for MSN subscribers. The second format is for Hotmail members. The third format is for Passport members.

You can use the following recipient ID formats when sending messages to an MSN subscriber using the MSN messaging service.

Table 73. Recipient IDs for MSN subscribers

Format	Example
MSN ID and fully-qualified service ID	`NetcoolAdmin%msn.com@msn.example.com`
MSN ID and service abbreviation	`NetcoolAdmin%msn.com@msn`

You can use the following recipient ID formats when sending messages to a Hotmail member using the MSN messaging service. Two types of abbreviation are available for Hotmail.

Table 74. Recipient IDs for Hotmail members

Format	Example
Hotmail ID and fully-qualified service ID	`NetcoolAdmin%hotmail.com@msn.example.com`
Hotmail ID and service abbreviation	`NetcoolAdmin@msn`
Hotmail ID and service abbreviation	`NetcoolAdmin%hotmail.com@msn`

You can use the following recipient ID formats when sending messages to a Passport member using the MSN messaging service.

Table 75. Recipient IDs for Passport members

Format	Example
Passport ID and fully-qualified service ID	`NetcoolAdmin%passport.com@msn.example.com`
Passport ID and service abbreviation	`NetcoolAdmin%passport.com@msn`

Examples

The following example shows how to send an instant message to a user named `NetcoolAdmin`.

```
// Call SendInstantMessage and pass the name of the recipient
// and the content of the message as input parameters

To = "NetcoolAdmin@jabber.example.com";
TextMessage = "Node_0456 is not responding to ping.";

SendInstantMessage(To, null, null, TextMessage, false);
```

The following example shows hot to send instant messages to multiple recipients.

```
// Call SendInstantMessage and pass a comma-separated list of recipients
// and the content of the message as input parameters
```

```
To = "NetcoolOperator@jabber.example.com, \
    NetcoolAdmin%hotmail.com@msn.jabber.example.com";
TextMessage = "Node_0123 is not responding to ping.";

SendInstantMessage(To, null, null, TextMessage, false);
```

The following example shows how to send a message to a chatroom hosted on a Jabber server.

```
// Call SendInstantMessage and pass the chatroom information,
// a subject and the content of the message as input parameters
Group = "netcoolchat@jabber.example.com/NetcoolAdmin";
Subject = "Alert: Node_0123 status changed.";
TextMessage = "Node_0123 has been restored.";

SendInstantMessage(null, Group, Subject, TextMessage, false);
```

SendJMSMessage

The SendJMSMessage function sends a message to the specified destination using the JMS DSA.

To send the message, you call this function and pass the JMSDataSource, a message properties context, and the message body as input parameters.

Syntax

The SendJMSMessage function has the following syntax:

```
SendJMSMessage(DataSource, MethodCallProperties, Message)
```

Parameters

The SendJMSMessage function has the following parameters.

Table 76. SendJMSMessage function parameters

Parameter	Format	Description
`DataSource`	String	Valid, and existing JMS data source.
`MethodCallProperties`	Context	Context that contains message header, and other JMS properties for the message. Custom message properties are supported.
`Message`	String | Context	String or context that contains the body of the message.

SetFieldValue

Use the SetFieldValue function to set the field variable in the Java class to some value.

If it is a static field, then you specify the Java class ClassName. If it is a non-static value, then you provide the instance at TargetObject.

Adding Java archive (JAR) files to the shared library directory

Before you can use this policy function, you must make the Java classes available to Netcool/Impact during run time. To make the Java classes available, complete the following steps:

1. Copy the Java classes to the $IMPACT_HOME/dsalib directory.
2. Restart the Impact Server to load the JAR files.

You must repeat this procedure for each Impact Server. This is necessary because the Java class files in the $IMPACT_HOME/dsalib directory are not replicated between servers.

Syntax

SetFieldValue(ClassName, TargetObject , FieldName, FieldValue);

Parameters

Table 77. SetFieldValue function parameters

Parameter	Description
ClassName	Name of the Java class. When using a non-static method call, this parameter would be set to null.
TargetObject	Name of the instantiated Java object. When using a static method, this parameter would be set to null.
FieldName	Name of the field variable in the Java class that you are setting the value for.
FieldValue	The value you are setting the field to.

Returns

N/A

Examples

Reversing the example for "GetFieldValue" on page 111, if you want to set the non-static firstname field on an instance of the DeveloperAccount class using IPL:

```
dev_acct = NewJavaObject("com.ibm.DeveloperAccount", {65224});
SetFieldValue(null, dev_acct, "firstname", "Sam");
```

Reversing the example for "GetFieldValue" on page 111, if you want to set the non-static firstname field on an instance of the DeveloperAccount class using JavaScript:

```
dev_acct = NewJavaObject("com.ibm.DeveloperAccount", [65224]);
SetFieldValue(null, dev_acct, "firstname", "Sam");
```

Let us assume there is a static counter variable, disconnects, in a hypothetical Java class, com.ibm.tivoli.EventStats, which we want to increment through an Impact policy:

```
counter = GetFieldValue("com.ibm.tivoli.EventStats", "disconnects");
counter += 1;
SetFieldValue("com.ibm.tivoli.EventStats", null, "disconnects", counter);
```

SetGlobalVar

The SetGlobalVar function creates in a policy a global variable which can be accessed from any local functions, library functions, and exception handlers in a policy.

The word “global” refers to the thread scope, which means that any policy code will access its own copy of the global variable from its own thread. Different threads that run the same policy will not interfere with one another, that is if the policy value is changed by one such running thread, the change does not affect the value of the global variable in other threads.

Syntax

`SetGlobalVar(variablename, variablevalue)`

Parameters

Table 78. SetGlobalVar function parameters

Parameter	Description
variablename	Name of the variable.
variablevalue	Initial value of the variable.

Example

This piece of code creates a global variable "MyAge" and sets its initial value to 33:

```
SetGlobalVar("MyAge", 33)
```

This piece of code clears the entry for the `variable1` variable, by passing null value to SetGlobalVar() call:

```
SetGlobalVar(variable1, null)
```

SetServerVar

The SetServerVar function creates a server-wide global variable in a policy.

It can be accessed by any functions and exception handlers, like a global variable created by SetGlobalVar(). Unlike in SetGlobalVar() calls, however, all threads running the same policy will share the same copy of the global variable. So if one thread running the same policy changes the variable value, the change is visible to all other threads running the same policy.

Syntax

`SetServerVar(variablename, variablevalue)`

Parameters

Table 79. SetServerVar function parameters

Parameter	Description
variablename	Name of the variable.
variablevalue	Initial value of the variable.

Examples

Here are examples of policies that use the `SetServerVar` and `GetServerVar` functions:

```
//Policy 1
function SaveServerVarTest(){
flag = "THIS FLAG DENOTES A SERVERVAR";
SetServerVar("runTimeFlag", flag);
```

```
Activate(null, 'Policy2');
}

function GetServerVarTest(){
Log("runTimeFlag = " + GetServerVar("runTimeFlag"));
}
//Policy2
Log("runTimeFlag = " + GetServerVar("runTimeFlag"));
```

SnmpGetAction

The SnmpGetAction function retrieves a set of SNMP variables from the specified agent

The values are then stored in a variable named ValueList and any error messages in a variable named ErrorString. This function operates by sending an SNMP `GET` command to the specified agent.

When you call `SnmpGetAction`, you pass an SNMP data type and, for SNMP v3, any authorization parameters that are required. To override the agent and variable information specified in the SNMP data type, you can also optionally pass a host name, a port number, a list of OIDs, and other information needed to retrieve the data.

Syntax

The following is the syntax for `SnmpGetAction`:

`SnmpGetAction(`*`TypeName`*`, `*`[HostId]`*`, `*`[Port]`*`, `*`[VarIdList]`*`, `*`[Community]`*`, `*`[Timeout]`*`, `*`[Version]`*`, `*`[UserId]`*`, `*`[AuthProtocol]`*`, `*`[AuthPassword]`*`, `*`[PrivPassword]`*`, `*`[ContextId]`*`, `*`[ContextName]`*`)`

Parameters

The `SnmpGetAction` function has the following parameters.

Table 80. SnmpGetAction function parameters

Parameter	Format	Description
`TypeName`	String	Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to retrieve the SNMP data.
`HostId`	String	Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.
`Port`	Integer	Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.
`VarIdList`	Array	Optional. Array of strings containing the OIDs of SNMP variables to retrieve from the agent. Overrides values specified in the SNMP data type.
`Community`	String	Optional. Name of the SNMP write community string. Default is `public`.
`Timeout`	Integer	Optional. Number of seconds to wait for a response from the SNMP agent before timing out.
`Version`	Integer	Optional. SNMP version number. Possible values are `1`, `2` and `3`. Default is `1`.

Table 80. SnmpGetAction function parameters (continued)

Parameter	Format	Description
UserId	String	Required for SNMP v3 authentication. If using SNMP v1 or v2, or using v3 without authentication, pass a `null` value for this parameter.
AuthProtocol	String	Optional. For use with SNMP v3 authentication only. Possible values are. `MD5_AUTH`, `NO_AUTH`, `SHA_AUTH`. `NO_AUTH` is the default.
AuthPassword	String	Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.
PrivPassword	String	Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.
ContextId	String	Optional. For use with SNMP v3 authentication only. Authentication context ID.
ContextName	String	Optional. For use with SNMP v3 authentication only. Authentication context name.

Return Values

When you call `SnmpGetAction`, it sets the following variables in the policy context: `ValueList` and `ErrorString`.

The `ValueList` variable is an array of strings, each of which stores the value of one variable retrieved from the SNMP agent. The strings in the array are assigned in the order that the variable OIDs are specified in the SNMP data type or the `VarIdList` parameter.

`ErrorString` is a string variable that contains any error messages generated while attempting to perform the SNMP `GET` command.

Example 1

The following example shows how to retrieve a set of SNMP variables by calling `SnmpGetAction` and passing the name of an SNMP data type as an input parameter. In this example, the SNMP data type is named `SNMP_PACKED`. The data type configuration specifies the host name and port where the SNMP agent is running and the OIDs of the variables you want to retrieve.

```
// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";

SnmpGetAction(TypeName, "192.168.1.1", 161, null, null, null, \
    null, null, null, null, null, null, null);

// Print the results of the SNMP GET to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(ValueList[Index]);
 Count = Count + 1;
}
```

Example 2

The following example shows how to retrieve a set of SNMP variables by calling SnmpGetAction and explicitly overriding the default host name, port, and other configuration values set in the SNMP data type.

Example 2 using IPL.

```
// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList =  {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;

SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, \
    Timeout, null, null, null, null, null, null,null);

// Print the results of the SNMP GET to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(ValueList[Index]);
 Count = Count + 1;
}
```

Example 2 using JavaScript.

```
// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];
Community = "private";
Timeout = 15;
SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, \
Timeout, null, null, null, null, null, null,null);
// Print the results of the SNMP GET to the policy log
Count = 0;
While (Count < Length(ValueList)) {
Log(ValueList[Index]);
Count = Count + 1;
}
```

Example 3

The following example shows how to retrieve a set of SNMP variables using SNMP v3 authentication.

Example 3 using IPL.

```
// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList =  {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;
Version = 3;
```

```
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, \
    Timeout, Version, UserId, AuthProtocol, AuthPassword, null, ContextId, null);

// Print the results of the SNMP GET to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(ValueList[Index]);
 Count = Count + 1;
}
```

Example 3 using JavaScript.

```
// Call SnmpGetAction and pass the name of the SNMP data type that contains
// configuration information required to perform the SNMP GET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";
SnmpGetAction(TypeName, HostId, Port, VarIdList, Community, \
Timeout, Version, UserId, AuthProtocol, AuthPassword, null, ContextId, null);
// Print the results of the SNMP GET to the policy log
Count = 0;
While (Count < Length(ValueList)) {
Log(ValueList[Index]);
Count = Count + 1;
}
```

SnmpGetNextAction

The `SnmpGetNextAction` function retrieves the next SNMP variables in the variable tree from the specified agent.

It stores the resulting OIDs in a variable named `VarIdList`, the resulting values in a variable named `ValueList`, and any error messages in a variable named `ErrorString`. The function sends a series of SNMP `GETNEXT` commands to the specified agent where each command specifies a single OID for which the next variable in the tree is to be retrieved.

When you call `SnmpGetNextAction`, you pass an SNMP data type and, for SNMP v3, any authorization parameters that are required. To override the agent and variable information specified in the SNMP data type, you can also optionally pass a host name, a port number, a list of OIDs, and other information needed to retrieve the data.

Syntax

The following is the syntax for `SnmpGetNextAction`:

```
SnmpGetNextAction(TypeName, [HostId], [Port], [VarIdList], [Community],
    [Timeout], [Version], [UserId], [AuthProtocol], [AuthPassword],
    [PrivPassword], [ContextId], [ContextName])
```

Parameters

The SnmpGetNextAction function has the following parameters.

Table 81. SnmpGetNextAction function parameters

Parameter	Format	Description
TypeName	String	Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to retrieve the SNMP data.
HostId	String	Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.
Port	Integer	Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.
VarIdList	Array	Optional. Array of strings containing the OIDs of SNMP variables to retrieve from the agent. Overrides values specified in the SNMP data type.
Community	String	Optional. Name of the SNMP write community string. Default is public.
Timeout	Integer	Optional. Number of seconds to wait for a response from the SNMP agent before timing out.
Version	Integer	Optional. SNMP version number. Possible values are 1, 2 and 3. Default is 1.
UserId	String	Required for SNMP v3 authentication. If using SNMP v1 or v2, or v3 without authentication, pass a null value for this parameter.
AuthProtocol	String	Optional. For use with SNMP v3 authentication only. Possible values are. MD5_AUTH, NO_AUTH, SHA_AUTH. NO_AUTH is the default.
AuthPassword	String	Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.
PrivPassword	String	Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.
ContextId	String	Optional. For use with SNMP v3 authentication only. Authentication context ID.
ContextName	String	Optional. For use with SNMP v3 authentication only. Authentication context name.

Example 1

The following example shows how to retrieve SNMP variables in the variable tree by calling SnmpGetNextAction and passing the name of an SNMP data type as an input parameter. In this example, the SNMP data type is named SNMP_PACKED. The data type configuration specifies the host name and port where the SNMP agent is running and the OIDs of the variables whose subsequent values in the tree you want to retrieve.

```
// Call SnmpGetNextAction and pass the name of the SNMP
// data type that contains configuration information required
// to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";

SnmpGetNextAction(TypeName, "192.168.1.1", 161, null, null, \
    null, null, null, null, null, null, null, null);

// Print the results of the SNMP GETNEXT to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(VarIdList + ": " + ValueList[Index]);
 Count = Count + 1;
}
```

Example 2

The following example shows how to retrieve SNMP variables in the variable tree by calling SnmpGetNextAction and explicitly overriding the default host name, port, and other configuration values set in the SNMP data type.

Example 2 using IPL.

```
// Call SnmpGetNextAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList =  {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;

SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, \
    Timeout, null, null, null, null, null, null, null);

// Print the results of the SNMP GETNEXT to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(VarIdList + ": " + ValueList[Index]);
 Count = Count + 1;
}
```

Example 2 using JavaScript.

```
// Call SnmpGetNextAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP GETNEXT
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];
Community = "private";
Timeout = 15;
SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, \
Timeout, null, null, null, null, null, null, null);
// Print the results of the SNMP GETNEXT to the policy log
Count = 0;
```

```
While (Count < Length(ValueList)) {
Log(VarIdList + ": " + ValueList[Index]);
Count = Count + 1;
}
```

Example 3

The following example shows how to retrieve subsequent SNMP variables in the variable tree using SNMP v3 authentication.

Example 3 using IPL.

```
// Call SnmpGetNextAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP GETNEXT

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList =  {".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"};
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, \
    Timeout, Version, UserId, AuthProtocol, AuthPassword, null, \
    ContextId, null);

// Print the results of the SNMP GET to the policy log

Count = 0;

While (Count < Length(ValueList)) {
 Log(VarIdList + ": " + ValueList[Index]);
 Count = Count + 1;
}
```

Example 3 using JavaScript.

```
// Call SnmpGetNextAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP GETNEXT
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [".1.3.6.1.2.1.1.5.0", ".1.3.6.1.2.1.1.6.0"];
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";
SnmpGetNextAction(TypeName, HostId, Port, VarIdList, Community, \
Timeout, Version, UserId, AuthProtocol, AuthPassword, null, \
ContextId, null);
// Print the results of the SNMP GET to the policy log
Count = 0;
While (Count < Length(ValueList)) {
Log(VarIdList + ": " + ValueList[Index]);
Count = Count + 1;
}
```

SnmpSetAction

The SnmpSetAction function sets variable values on the specified SNMP agent.

If the attempt to set variable fails, it stores the resulting error message in a variable named `ErrorString`. This function operates by sending an SNMP `SET` command to the specified agent.

When you call `SnmpSetAction`, you pass an SNMP data type, the host name, and port of the agent, an array of OIDs, and the array of values that you want to set. If you are using SNMP v3, you can also include information required to authenticate as an SNMP user.

Syntax

The following is the syntax for `SnmpSetAction`:

```
SnmpSetAction(TypeName, [HostId], [Port], [VarIdList], \
ValueList, [Community], [Timeout], [Version], [UserId], [AuthProtocol],\
[AuthPassword], [PrivPassword], [ContextId], [ContextName])
```

Parameters

The `SnmpSetAction` function has the following parameters.

Table 82. SnmpSetAction function parameters

Parameter	Format	Description
TypeName	String	Name of the SNMP data type that specifies the host name, port, OIDs, and other information needed to set the SNMP data.
HostId	String	Optional. Host name or IP address of the system where the SNMP agent is running. Overrides value specified in the SNMP data type.
Port	Integer	Optional. Port where the SNMP agent is running. Overrides value specified in the SNMP data type.
VarIdList	Array	Array of strings containing the OIDs of SNMP variables to set on the agent. Overrides values specified in the SNMP data type.
ValueList	Array	Array of strings containing the values you want to set. You must specify these values in the same order that the OIDs appear either in the SNMP data type or in the *VarIdList* variable.
Community	String	Optional. Name of the SNMP write community string. Default is `public`.
Timeout	Integer	Optional. Number of seconds to wait for a response from the SNMP agent before timing out.
Version	Integer	Optional. SNMP version number. Possible values are `1`, `2` and `3`. Default is `1`.
UserId	String	Required for SNMP v3 authentication. If using SNMP v1 or v2, or using v3 without authentication, pass a `null` value for this parameter.
AuthProtocol	String	Optional. For use with SNMP v3 authentication only. Possible values are. `MD5_AUTH`, `NO_AUTH`, `SHA_AUTH`. `NO_AUTH` is the default.

Table 82. SnmpSetAction function parameters (continued)

Parameter	Format	Description
AuthPassword	String	Optional. For use with SNMP v3 authentication only. Authentication password associated with the specified SNMP User ID.
PrivPassword	String	Optional. For use with SNMP v3 authentication only. Privacy password associated with the specified SNMP User ID.
ContextId	String	Optional. For use with SNMP v3 authentication only. Authentication context ID.
ContextName	String	Optional. For use with SNMP v3 authentication only. Authentication context name.

Example 1

The following example shows how to set SNMP variables by calling `SnmpSetAction` and passing the name of an SNMP data type, an array of OIDs, and an array of values as input parameters. In this example, the SNMP data type is named `SNMP_PACKED`.

Example 1 using IPL.

```
// Call SnmpSetAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP SET

TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = {" .1.3.6.1.2.1.1.4.0", "  .1.3.6.1.2.1.1.5.0"};
ValueList = {"Value_01", "Value_02"};

SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, \
    null, null, null, null, null, null, null, null, null);
```

Example 1 using JavaScript.

```
// Call SnmpSetAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP SET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [" .1.3.6.1.2.1.1.4.0", " .1.3.6.1.2.1.1.5.0"];
ValueList = ["Value_01", "Value_02"];
SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, \
null, null, null, null, null, null, null, null, null);
```

Example 2

The following example shows how to set SNMP variables using SNMP v3 authentication.

Example 2 using IPL.

```
// Call SnmpSetAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP SET

TypeName = "SNMP_PACKED";
```

```
HostId = "192.168.1.1";
Port = "161";
VarIdList = { ".1.3.6.1.2.1.1.4.0",  " .1.3.6.1.2.1.1.5.0"};
ValueList = {"Value_01", "Value_02"};
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";

SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, \
    Community, Timeout, Version, UserId, AuthProtocol, \
    AuthPassword, null, ContextId, null);
```

Example 2 using JavaScript.

```
// Call SnmpSetAction and pass the name of the
// SNMP data type that contains configuration information
// required to perform the SNMP SET
TypeName = "SNMP_PACKED";
HostId = "192.168.1.1";
Port = "161";
VarIdList = [ ".1.3.6.1.2.1.1.4.0", " .1.3.6.1.2.1.1.5.0"];
ValueList = ["Value_01", "Value_02"];
Community = "private";
Timeout = 15;
Version = 3;
UserId = "snmpusr";
AuthProtocol = "MD5_AUTH";
AuthPassword = "snmppwd";
ContextId = "ctx";
SnmpSetAction(TypeName, HostId, Port, VarIdList, ValueList, \
Community, Timeout, Version, UserId, AuthProtocol, \
AuthPassword, null, ContextId, null);
```

SnmpTrapAction

The SnmpTrapAction function sends a trap (for SNMP v1) or a notification (for SNMP v2) to an SNMP manager.

Sending traps or notifications is not supported for SNMP v3.

Syntax

The following is the syntax for `SnmpTrapAction`:

```
SnmpTrapAction(HostId, Port, [VarIdList], [ValueList], \
[Community], [Timeout], [Version], [SysUpTime], [Enterprise], \
[GenericTrap], [SpecificTrap], [SnmpTrapOid])
```

Parameters

The `SnmpTrapAction` function has the following parameters.

Table 83. SnmpTrapAction function parameters

Parameter	Format	Description
HostId	String	Host name or IP address of the system where the SNMP manager is running.
Port	Integer	Port where the SNMP manager is running.

Table 83. SnmpTrapAction function parameters (continued)

Parameter	Format	Description
VarIdList	Array	Optional. Array of strings containing the OIDs of SNMP variables to send to the manager.
ValueList	Array	Optional. Array of strings containing the values you want to send to the manager. You must specify these values in the same order that the OIDs appear in the *VarIdList* variable.
Community	String	Optional. Name of the SNMP write community string. Default is `public`.
Timeout	Integer	Optional. Number of seconds to wait for a response from the SNMP agent before timing out.
Version	Integer	Optional. SNMP version number. Possible values are 1 and 2. Default is 1.
SysUpTime	Integer	Optional. Number of milliseconds since the system started. Default is the current system time in milliseconds.
Enterprise	String	Required for SNMP v1 only. Enterprise ID.
GenericTrap	String	Required for SNMP v1 only. Generic trap ID.
SpecificTrap	String	Required for SNMP v1 only. Specific trap ID.
SnmpTrapOid	String	Optional for SNMP v1. Required for SNMP v2. SNMP trap OID.

Example 1

The following example shows how to send an SNMP v1 trap to a manager using `SnmpTrapAction`.

Example 1 using IPL.

```
// Call SnmpTrapAction

HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
VarIdList = {".1.3.6.1.2.1.2.2.1.1.0", "sysDescr"};
ValueList = {"2", "My system"};

SnmpTrapAction(HostId, Port, VarIdList, ValueList, \
    Community, 15, Version, SysUpTime, Enterprise, GenericTrap, \
    SpecificTrap, null);
```

Example 1 using JavaScript.

```
// Call SnmpTrapAction
HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
```

```
VarIdList = [".1.3.6.1.2.1.2.2.1.1.0", "sysDescr"];
ValueList = ["2", "My system"];
SnmpTrapAction(HostId, Port, VarIdList, ValueList, \
Community, 15, Version, SysUpTime, Enterprise, GenericTrap, \
SpecificTrap, null);
```

Example 2

The following example shows how to send an SNMP v2 notification to a manager using `SnmpTrapAction`. SNMP v2 requires that you specify an SNMP trap OID when you call this function.

Example 2 using IPL.

```
// Call SnmpTrapAction

HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
VarIdList = {".1.3.6.1.2.1.2.2.1.1.0", "sysDescr"};
ValueList = {"2", "My system"};
SnmpTrapOid = ".1.3.6.1.2.4.1.11";

SnmpTrapAction(HostId, Port, VarIdList, ValueList, \
    Community, 15, Version, SysUpTime, Enterprise, \
    GenericTrap, SpecificTrap, SnmpTrapOid);
```

Example 2 using JavaScript.

```
// Call SnmpTrapAction
HostId = "localhost";
Port = 162;
Version = 1;
Community = "public";
SysUpTime = 1001;
Enterprise = ".1.3.6.1.2.1.11";
GenericTrap = 3;
SpecificTrap = 0;
VarIdList = [".1.3.6.1.2.1.2.2.1.1.0", "sysDescr"];
ValueList = ["2", "My system"];
SnmpTrapOid = ".1.3.6.1.2.4.1.11";
SnmpTrapAction(HostId, Port, VarIdList, ValueList, \
Community, 15, Version, SysUpTime, Enterprise, \
GenericTrap, SpecificTrap, SnmpTrapOid);
```

Split

The Split function returns an array of substrings from a string using the given delimiters.

Syntax

The `Split` function has the following syntax:

Array = Split(*Expression*, *Delimiters*)

Parameters

The Split function has the following parameters.

Table 84. Split function parameters

Parameter	Format	Description
Expression	String	String to split into substrings
Delimiters	String	String that contains the delimiter characters.

Return value

Array of substrings.

Example

The following example shows how to split a string into an array of substrings.

```
MyString = "This is a test.";
Delimiters = " ";
MyArray = Split(MyString, Delimiters);
Log(MyArray);
```

This example prints the following message to the policy log:

```
Parser Log: {This, is, a, test.}
```

String

The String function converts an integer, float, or boolean expression to a string.

Syntax

The String function has the following syntax:

```
String = String(Expression)
```

Parameters

The String function has the following parameters.

Table 85. String function parameters

Parameter	Format	Description
Expression	Integer | Float | Boolean	Expression to be converted to a string.

Return value

Converted string value.

Example

The following example shows how to convert integers, floats, and boolean expressions to a string.

```
MyString = String(123);
Log(MyString);

MyString = String(123.54);
```

```
Log(MyString);

MyString = String(true);
Log(MyString);
```

This example prints the following message to the policy log for IPL:

```
Parser Log: 123
Parser Log: 123.54
Parser Log: true
```

This example prints the following message to the policy log for JavaScript. JavaScript treats numbers as doubles, as a result, numbers display using decimals:

```
Parser Log: 123.0
Parser Log: 123.54
Parser Log: true
```

Strip

The Strip function strips all instances of the given substring from a string.

The order in which you supply the characters is not significant.

Syntax

The `Strip` function has the following syntax:

`String` = `Strip(`*`Expression`*`, `*`Characters`*`)`

Parameters

The `Strip` function has the following parameters.

Table 86. Strip function parameters

Parameter	Format	Description
`Expression`	String	String to strip.
`Characters`	String	String that contains characters to strip from the string.

Return value

The string with the specified characters stripped out.

Example

The following example shows how to strip a characters from a string.

```
MyString = "abccababc.";
MyCharacters = "ab";
MyStrip = Strip(MyString, MyCharacters);
Log(MyStrip);
```

This example prints the following message to the policy log:

```
Parser Log: ccc.
```

Substring

The Substring function returns a substring from a given string using index positions.

Index positions start at 0.

Syntax

The `Substring` function has the following syntax:

```
String = Substring(Expression, Start, End)
```

Parameters

The `Substring` function has the following parameters.

Table 87. Substring function parameters

Parameter	Format	Description
Expression	String	String to search for the substring.
Start	Integer	Starting character position of the substring.
End	Integer	Ending character position of the substring, plus one.

Return value

The substring returned by index positions.

Example

The following example shows how to return a substring using index positions.

```
MyString = "This is a test.";
MySubstring = Substring(MyString, 0, 4);
Log(MySubstring);
```

This example prints the following message to the policy log:

```
Parser Log: This
```

Synchronized

Use the Synchronized function to write thread-safe policies for use with a multi-threaded event processor using IPL or JavaScript.

Synchronized for IPL

The syntax and parameters for the `Synchronized` function differ for IPL and JavaScript. Refer to the information in the following sections for the details.

The `Synchronized` function has the following syntax for IPL:

```
synchronized(identifier) { statements }
```

Parameters

The Synchronized function has the following parameters for IPL.

Table 88. Synchronized function parameters for IPL

Parameter	Format	Description
identifier	String	The unique name for the statement block.
statements	String	Programming statements.

Example

The following example shows how to insert an item into a custom data type using IPL.

```
insert_table = NewObject();

synchronized (insert_table) {
 MyContext = NewObject();
 MyContext.Cust_ID = 5;
 MyContext.Cust_Loc = "New York";
 AddDataItem("Cust", MyContext);
}
```

Synchronized for JavaScript

The Synchronized function has the following syntax for JavaScript:

```
synchronized(<function name>, <identifier> );
```

Parameters

The Synchronized function has the following parameters for JavaScript.

Table 89. Synchronized function parameters for JavaScript

Parameter	Format	Description
<function name>	String	The name for the function.
<identifier>	String	The unique name for the statement block.

Example

The following example shows how to insert an item into a custom data type using JavaScript.

```
insert_table = NewObject();

function mySyncFunc() {
 MyContext = NewObject();
 MyContext.Cust_ID = 5;
 MyContext.Cust_Loc = "New York";
 AddDataItem("Cust", MyContext);
}

Synchronized(mySyncFunc, insert_table);
```

ToLower

The ToLower function converts a string to lower case characters.

Syntax

The ToLower function has the following syntax:

```
String = ToLower(Expression)
```

Parameters

The ToLower function has the following parameter.

Table 90. ToLower function parameters

Parameter	Format	Description
Expression	String	String to convert to lower case.

Return value

The lower case string.

Example

The following example shows how to convert a string to lower case.

```
MyString = "tHiS iS a TeSt.";
MyLower = ToLower(MyString);
Log(MyLower);
```

This example prints the following message to the policy log:

```
Parser Log: this is a test.
```

ToUpper

The ToUpper function converts a string to upper case characters.

Syntax

The ToUpper function has the following syntax:

```
String = ToUpper(Expression)
```

Parameters

The ToUpper function has the following parameter.

Table 91. ToUpper function parameters

Parameter	Format	Description
Expression	String	String to convert to upper case.

Return value

The upper case string.

Example

The following example shows how to convert a string to upper case.

```
MyString = "tHiS iS a TeSt.";
MyUpper = ToUpper(MyString);
Log(MyUpper);
```

This example prints the following message to the policy log:

```
Parser Log: THIS IS A TEST.
```

Trim

The Trim function trims leading and trailing white space from a string.

Syntax

The `Trim` function has the following syntax:

```
String = Trim(Expression)
```

Parameters

The `Trim` function has the following parameter.

Table 92. Trim function parameters

Parameter	Format	Description
Expression	String	String that you want to trim.

Return value

The string with white space trimmed.

Example

The following example shows how to trim white space from a string.

```
MyString = " This is a test. ";
MyTrim = Trim(MyString);
Log(MyTrim);
```

This example prints the following message to the policy log:

```
Parser Log: This is a test.
```

TBSM functions

An overview of functions that are used specifically with TBSM.

Netcool/Impact has the three Tivoli Business Service Manager-specific policy functions.

- **PassToTBSM**: Used to send events from Netcool/Impact to Tivoli Business Service Manager. In an Netcool/Impact policy, you can add the `PassToTBSM` function to the policy. When you activate the policy using an Netcool/Impact service, the event information is sent to TBSM.
- **RemoteTBSMShell**: Used to send `RadShell` commands from a policy using a remote Impact Server. The Impact Server and the Tivoli Business Service Manager server must be configured for Name Server clustering. The clustered Name Server must contain clusters of the Impact Server and the Tivoli Business Service Manager server.

- **TBSMShell**: Used to add RadShell commands to a policy. This function is Tivoli Business Service Manager-specific and is available only in a Tivoli Business Service Manager installation.

PassToTBSM

Use the PasstoTBSM function to send event information from Netcool/Impact to TBSM. In a Netcool/Impact policy, you can invoke the PassToTBSM function which gets read as an event by Tivoli Business Service Manager.

PassToTBSM takes one argument, EventContainer. The EventContainer has a set of member variables that correspond to fields in the incoming event.

Examples

An example of a JMSSend policy that sends messages to a JMS data source using the SendJMSMessage function.

```
// Set JMSDataSource to a valid and existing JMSDataSource in Impact.
// The destination where the message is sent is obtained from the JMSDataSource.

Log("\nSetting up props before calling SendJMSMessage action function");
JMSDataSource = 'myJMS_ds';

// Create a message properties object and populate its
// member variables with message header properties and custom properties

MsgProps = NewObject();
MsgProps.TimeToLive = 0;
MsgProps.Priority = 9;
MsgProps.Expiration = 2000;
MsgProps.DeliveryMode = "PERSISTENT";
MsgProps.ReplyTo="queue2";


// Create a map message content and populate its member
// variables where each variable and value represent a name/
// value pair for the resulting map

MsgMapBody = NewObject();
MsgMapBody.branch = "ATM1";
MsgMapBody.status = "Marginal";

// Call SendJMSMessage and pass the JNDI properties
// context, the message properties context, the message
// map context and other parameters

Log("\n\nCalling SendJMSMessage action function now");
SendJMSMessage(JMSDataSource, MsgProps, MsgMapBody);

log("Received a Message : " + currentContext());
log("Message : " + EventContainer);;
```

An example of a JMSReceive policy run by the **JMSMessageListener** service that assigns fields to the received message and passes the information to Tivoli Business Service Manager:

```
// branch = @JMSMessage.branch;
// status = @JMSMessage.status;

ec=NewEvent();
ec.branch = @JMSMessage.branch;
```

```
ec.status = @JMSMessage.status;

Log(" Branch : " + ec.branch + "  Status : " + ec.status);
PassToTBSM(ec);
```

RemoteTBSMShell

A stand-alone implementation of Netcool/Impact can run RADShell commands from a policy in Tivoli Business Service Manager.

The stand-alone Impact Server and the Tivoli Business Service Manager server must be configured to use Name Server clustering. The clustered Name Server must contain the clusters of the Impact Server and the Tivoli Business Service Manager server. The RemoteTBSMShell command can be run from any policy using the following syntax:

Syntax

```
Result = RemoteTBSMShell('<command>');:
```

Where *command* is the *radshell* command to run and Result is the string returned by the *radshell* command.

This example creates a template called DBFarm.

```
Result = RemoteTBSMShell ('createTemplate ("DBFarm","man_svg.gif")');
```

TBSMShell

This topic describes the TBSMShell function which lets you put RADshell commands in a policy. With the TBSMShell function, you can change the TBSM configuration in a policy. Only use this function if your Data server is **not** configured for failover. If your Data server is configured for failover, use the RemoteTBSMShell function.

Syntax

```
Result = TBSMShell('command');
```

Where *command* is a string that you execute in RADShell, and Result is the string output by RADShell after running the command. The command can be a sequence of RADShell commands separated by semicolons.

This example creates a template called DBFarm.

```
Result = TBSMShell ('createTemplate ("DBFarm","man_svg.gif")');
```

UpdateEventQueue

The UpdateEventQueue function updates or deletes events in the event reader event queue.

Use UpdateEventQueue for situations in which a policy modifies an incoming event that is expected to have other related events in the event queue at the same time.

Note: The UpdateEventQueue does not access events in the EventProcessor queue. It works only with the events in the EventReader queue. If you update or delete events in the EventReader queue, it modifies only the event containers within

Netcool/Impact and does not affect the events in Netcool® OMNIbus. To modify events in Netcool OMNIbus, you need to use the `ReturnEvent` function.

To update events, you call `UpdateEventQueue` and pass the name of the event reader, a filter string, and an update expression as input values. The filter string specifies which events to update. It uses the SQL filter syntax, which is similar to the syntax of the `WHERE` clause in an SQL `SELECT` statement. The update expression is a comma-separated list of field assignments similar to the contents of the `SET` clause in an SQL `UPDATE` statement. For more information about SQL filters, see "SQL filters" on page 69.

To delete events, you call `UpdateEventQueue` and pass the name of the event reader, a filter string, and a boolean value that indicates that you want to perform a delete operation. As with the update operation above, the filter string uses the SQL filter syntax and specifies which events you want to delete.

Syntax

The `UpdateEventQueue` function has the following syntax:

```
[Integer = ] UpdateEventQueue(EventReaderName, Filter, UpdateExpression, IsDelete)
```

Parameters

The `UpdateEventQueue` function has the following parameters.

Table 93. UpdateEventQueue function parameters

Parameter	Type	Description
EventReaderName	String	Name of the event reader whose queue you want to update or delete.
Filter	String	SQL filter expression that specifies which events in the queue to update or delete.
UpdateExpression	String	Update expression that specifies which fields and corresponding values to update. If you want to delete events, pass a `null` value for this parameter.
IsDelete	Boolean	Boolean value that indicates whether to delete the specified events. Possible values are `true` and `false`.

Return value

Number of events updated or deleted. Optional.

Examples

The following example shows how to update events in the event queue:

```
EventReaderName = "OMNIbusEventReader";
Filter = "Node = 'Node Name'";
UpdateExpression = "Node = 'New Node Name'";
IsDelete = false;

NumUpdatedEvents = UpdateEventQueue(EventReaderName, Filter, \
    UpdateExpression, IsDelete);

Log("Number of updated events: " + NumUpdatedEvents);
```

The following example shows how to delete events in the event queue:

```
EventReaderName = "OMNIbusEventReader";
Filter = "Node = 'ORA_01'";
IsDelete = true;

NumDeletedEvents = UpdateEventQueue(EventReaderName, Filter, null, IsDelete);
```

URLDecode

The URLDecode function returns a URL encoded string to its original representation.

This function parallels the Java function `java.net.URLDecoder.decode()`.

Syntax

The `URLDecode` function has the following syntax:

```
String = URLDecode(Expression,[Encoding])
```

Parameters

The `URLDecode` function has the following parameters.

Table 94. URLDecode function parameters

Parameter	Format	Description
Expression	String	The string that you want to decode.
Encoding	String	The encoding scheme you want to use. This is optional. The recommended and default encoding is `UTF-8`.

Return value

The decoded string.

Example

The following example shows how to decode a URL encoded string back to its original representation.

```
ReceivedString = "System.out.println%28%22Hello+World%21%22%29%3B";
OriginalString = URLDecode(ReceivedString, "UTF-8");
Log(OriginalString);
```

This example prints the following message to the policy log:

```
Parser Log:
System.out.printIn("Hello world!");
```

URLEncode

The URLEncode function converts a string to a URL encoded format.

This function parallels the Java function `java.net.URLEncoder.encode()`.

Syntax

The URLEncode function has the following syntax:

```
String = URLEncode(Expression, [Encoding])
```

Parameters

The URLEncode function has the following parameters.

Table 95. URLEncode function parameters

Parameter	Format	Description
Expression	String	String that you want to encode.
Encoding	String	The encoding scheme you want to use. This is optional. The recommended and default encoding is UTF-8.

Return value

The URL encoded string.

Example

The following example shows how to encode the query string of a URL and form a valid URL.

```
BaseURL = "http://hostname:port/query";
QName1 = "filter";
QVal1 = URLEncode("key='42ITA'");
QName2 = "comment";
QVal2 = URLEncode("#$&@^%$!!","UTF-8");
Querystring = "?" + QName1 + "=" + QVal1 + "&" + QName2 + "=" + QVal2;

FullURL = BaseURL + Querystring;
Log(FullURL);
```

This example prints the following message to the policy log:

```
Parser Log:
http://hostname:port/query?filter=key%3D%2742ITA%27&comment
=%23%24%26%40%5E%25%24%21%21
```

WSDMGetResourceProperty

The WSDMGetResourceProperty function retrieves the value of a management property associated with a WSDM (Web Services Distributed Management) managed resource.

You can use this function to retrieve information about the state of a WSDM-enabled system, application or device.

To retrieve the property value, you call WSDMGetResourceProperty and pass the URI of the WSDM endpoint reference and a flattened XML QName that specifies which property to retrieve.

Syntax

```
Array = WSDMGetResourceProperty(endPointRef, methodName, propQName)
```

Parameters

The WSDMGetResourceProperty function has the following parameters.

Table 96. WSDMGetResourceProperty function parameters

Parameter	Format	Description
endPointRef	String	URI that specifies the endpoint where the WSDM resource is located.
UserName	String	Optional. User name required by the Web service for SOAP authentication, if any. If no username is required omit this parameter.
Password	String	Optional. Password required by the Web service for SOAP authentication, if any. If no password is required, omit this parameter.
propQName	String	Flattened XML QName that specifies the management property to retrieve. The format for the flattened QName is *namespace*:*localname* [*URI*], where *namespace* is the XML namespace where the property is defined, *localname* is the name of the XML element that contains the property and *URI* is the endpoint where the WSDM resource is located. For more information about QNames, see the XML specifications at http://www.w3.org.

Return Value

The WSDMGetResourceProperty function returns the property value to the policy as an array. For properties that consist of a single value, the value is stored in the first array element. For properties that consist of more than one value, the values are stored in the array in the order that they are retrieved from the WSDM resource. In most cases, this function returns an array that contains a single property value.

Example

The following example shows how to use WSDMGetResourceProperty to retrieve a management property named MemoryInUse from the endpoint http://www.example.com/wsdm-endpoint.

```
// Specify endpoint URI and flattened QName

MyEndPoint = "http://www.example.com/wsdm-endpoint";
MyQName = "wsrl:MyProperty [http://www.example.com/wsdm-endpoint]";

// Call WSDMGetResourceProperty and pass the endpoint
// and QName and input parameters

MyResult = WSDMGetResourceProperty(MyEndPoint, MyQName);

// Print the value of the property to the policy log

Log("Value of MyProperty is " + MyResult[0]);
```

WSDMInvoke

The WSDMInvoke function sends a web services message to a WSDM (Web Services Distributed Management) managed resource.

The structure and content of this message is defined by the receiving WSDM entity. You can use this function to send other kinds of messages to a WSDM resource besides those that retrieve or update a management property.

To retrieve the property value, you call `WSDMInvoke` and pass the URI of the WSDM endpoint reference, the method name and a Java QName object that specifies which property to retrieve.

Syntax

Array = WSDMInvoke(*endPointRef*, *methodName*, *propQName*)

Parameters

The `WSDMInvoke` function has the following parameters.

Table 97. WSDMInvoke function parameters

Parameter	Format	Description
endPointRef	String	URI that specifies the endpoint where the WSDM resource is located.
Method	String	Name of the method exposed by the API located at the WSDM resource endpoint.
propQName	Object	Java QName object that specifies the management property to retrieve. You can create a new instance of this object in the policy using a call to the `NewJavaObject` function provided by the Java DSA.
UserName	String	Optional. User name required by the Web service for SOAP authentication, if any. If no username is required omit this parameter.
Password	String	Optional. Password required by the Web service for SOAP authentication, if any. If no password is required, omit this parameter.

Return Value

The `WSDMInvoke` function returns any values sent in the WSDM reply as an array. For properties that consist of a single value, the value is stored in the first array element. For properties that consist of more than one value, the values are stored in the array in the order that they are retrieved from the WSDM resource. In most cases, this function returns an array that contains a single property value.

Example

The following example shows how to use `WSDMInvoke` to remotely invoke a web services method named `GetResourceProperty`. This method is exposed by the API located at the specified WSDM endpoint.

Example using IPL.

```
// Specify endpoint URI, method name and QName

MyEndPoint = "http://www.example.com/wsdm-endpoint";
MyMethodName = "GetResourceProperty";
MyQNameParams = {"http://docs.oasis-open.org/wsrf/rl-2", "CurrentTime", "wsrl"};
MyQName = NewJavaObject("javax.xml.namespace.QName", qnameParams);

// Call WSDMInvoke and pass the endpoint, the method name
```

```
// and the QName object

MyResult = WSDMInvoke(MyEndPoint, MyMethodName, MyQName);

// Print the value of the property to the policy log

Log("Value of MyProperty is " + MyResult[0]);
```

Example using JavaScript.

```
// Specify endpoint URI, method name and QName
MyEndPoint = "http://www.example.com/wsdm-endpoint";
MyMethodName = "GetResourceProperty";
MyQNameParams = ["http://docs.oasis-open.org/wsrf/rl-2", "CurrentTime", "wsrl"];
MyQName = NewJavaObject("javax.xml.namespace.QName", qnameParams);

// Call WSDMInvoke and pass the endpoint, the method name
// and the QName object

MyResult = WSDMInvoke(MyEndPoint, MyMethodName, MyQName);

// Print the value of the property to the policy log

Log("Value of MyProperty is " + MyResult[0]);
```

WSDMUpdateResourceProperty

The WSDMUpdatetResourceProperty function updates the value or values of a management property associated with a WSDM (Web Services Distributed Management) managed resource.

You can use this function to set information about the state of a WSDM-enabled system, application, or device.

To update the property value, call `WSDMUpdateResourceProperty` and pass the URI of the WSDM endpoint reference, a flattened XML QName that specifies the property and an array of new property values.

Syntax

`WSDMUpdateResourceProperty(`*`endPointRef`*`, `*`propQName`*`, `*`params`*`)`

Parameters

The `WSDMUpdateResourceProperty` function has the following parameters.

Table 98. WSDMUpdateResourceProperty function parameters

Parameter	Format	Description
endPointRef	String	URI that specifies the endpoint where the WSDM resource is located.
propQName	String	Flattened XML QName that specifies the management property to update. The format for the flattened QName is *namespace*:*localname* [*URI*], where *namespace* is the XML namespace where the property is defined, *localname* is the name of the XML element that contains the property and *URI* is the endpoint where the WSDM resource is located. For more information about QNames, see the XML specifications at `http://www.w3.org`.

Table 98. WSDMUpdateResourceProperty function parameters (continued)

Parameter	Format	Description
ArrayOfValues	Array	An array that contains the value or values of the property. For properties that consist of a single value, you must store the value in the first array element. For properties that consist of more than one value, you must store the values in the array in the order that they are managed by the WSDM resource. In most cases, the property consists of a single value.
UserName	String	Optional. User name required by the Web service for SOAP authentication, if any. If no user name is required, omit this parameter.
Password	String	Optional. Password required by the Web service for SOAP authentication, if any. If no password is required, omit this parameter.

Example

The following example shows how to use `WSDMUpdateResourceProperty` to update a management property named `MemoryInUse` from the endpoint `http://www.example.com/wsdm-endpoint`.

Example using IPL.

```
// Specify endpoint URI, flattened QName and property value

MyEndPoint = "http://www.example.com/wsdm-endpoint";
MyQName = "wsrl:MyProperty [http://www.example.com/wsdm-endpoint]";
Params = {"256"};

// Call WSDMUpdateResourceProperty and pass the endpoint
// and QName and property value

WSDMUpdateResourceProperty(MyEndPoint, MyQName, Params);
```

Example using JavaScript.

```
// Specify endpoint URI, flattened QName and property value

MyEndPoint = "http://www.example.com/wsdm-endpoint";
MyQName = "wsrl:MyProperty [http://www.example.com/wsdm-endpoint]";
Params = ["256"];

// Call WSDMUpdateResourceProperty and pass the endpoint
// and QName and property value

WSDMUpdateResourceProperty(MyEndPoint, MyQName, Params);
```

WSInvokeDL

The `WSInvokeDL` function is used to make Web services calls when a WSDL file is compiled with nci_compilewsdl, or when a Web services DSA policy wizard is configured.

Syntax

This function has the following syntax:

```
[Return] = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSParams, [callProps])
```

This function returns the value of your target Web services call.

Parameters

The WSInvokeDL function has the following parameters

Table 99. WSInvokeDL function parameters

Parameter	Format	Description
WSService	String	Web service name. This name is defined in the /definitions/service element of the WSDL file.
WSEndPoint	String	The endpoint URL of the target Web service.
WSMethod	String	Defines which method you would like to call in WSInvokeDL().
WSParams	Array	An array that contains all of the parameters required by the specified Web service operation. The operation parameters are defined by /definitions/message/part elements in the WSDL file.
callProps	String, boolean, integer	Optional container in which you can set any of the following properties listed after this table.

callProps properties

Remember: Any options set in callProps have to precede the actual call to WSInvokeDL.

- **Chunked** specifies whether the request can be chunked or not.
- **MTOM** enables or disables the Message Optimization for the SOAP message.
- **CharSet** use it to set the encoding other than UTF-8.
- **HTTP** the default HTTP version is 1.1. You can use this property to set the protocol version to 1.0.
- **ReuseHttpClient** enables the underlying infrastructure to reuse the HTTP client if one is available. The **ReuseHttpClient** is useful if the client is using HTTPS to communicate with the server. The ssl handshake is not repeated for each request. The parameter must be set to true or false.
- **EnableWSS** enables Web Service Security. If you specify **EnableWSS** you must also specify the following properties:
  - **WSSRepository** specifies the path location of WSS Repository.
  - **WSSConfigFile** specifies configuration file for **EnableWSS**.
- **Username** specifies the username for basic authentication.
- **Password** specifies password for basic authentication.
- **PreemptiveAuth** enables Preemptive Authentication.
- **Timeout** this property is used in a blocking scenario. The client system times out after waiting the specified amount of time.

  You can optionally set a global Web Service DSA call timeout property called, impact.server.dsainvoke.timeout. The property must be added to the Netcool/Impact server property file, <servername>_server.props.

  The value is set in milliseconds. For example, impact.server.dsainvoke.timeout=30000 (30 seconds).

  When you set the properties in any of the .props files, restart theNetcool/Impact server to implement the changes.

If the impact.server.dsainvoke.timeout property is set, all WSInvokeDL calls will use the same timeout setting.

- **MaintainSession** sets the session management to enabled status. When session management is enabled, the system maintains the session-related objects across the different requests. The parameter must be set to true or false.
- **CacheStub** caches generated stubs. This value must be set to **true** if either or both of the following properties are enabled, **ReuseHttpClient**, **MaintainSession**. Examples of usage:

```
callProps.CacheStub=true;
callProps.ReuseHttpClient = true;
```

Examples

Remember: Any options set in callProps have to precede the actual call to WSInvokeDL.

Apart from its primary usage, the callProps container can be used to enable security. For example, if the basic authentication is enabled through the wizard, the sample policy contains the following lines:

```
callProps.Username="username";
callProps.Password="password";
```

The following example shows how to use the WSInvokeDL function to send a message to the target Web service.

Example using IPL.

```
ServiceName = "StockQuote";
EndPointURL = "http://www.webservicex.net/stockquote.asmx"
MethodName = "GetQuote";
ParameterArray = { "IBM" }

[Return] = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSParams, [callProps])
```

Example using JavaScript.

```
ServiceName = "StockQuote";
EndPointURL = "http://www.webservicex.net/stockquote.asmx";
MethodName = "GetQuote";
ParameterArray = [ "IBM" ];

Results = WSInvokeDL(WSService, WSEndPoint, WSMethod, WSParams, [callProps])
```

WSNewArray

The WSNewArray function creates a new array of complex data type objects or primitive values, as defined in the WSDL file for the Web service.

You use this function in cases where you are required to pass an array of complex objects or primitives to a Web service as message parameters.

Syntax

This function has the following syntax:

```
Array = WSNewArray(ElementType, ArrayLength)
```

Parameters

The WSNewArray function has the following parameters.

Table 100. WSNewArray function parameters

Parameter	Format	Description
ElementType	String	Name of the complex object or primitive data type defined in the WSDL file. The name format is [*Package*.]*TypeName*, where *Package* is the name of the package you created when you compiled the WSDL file, without the .jar suffix. The package name is required only if you did not previously call the WSSetDefaultPKGName function in the policy.
ArrayLength	Integer	Number of elements in the new array.

Return Value

A new Web services array.

Examples

The following example shows how to use WSNewArray to create a new Web services array, where you have previously called WSSetDefaultPKGName in the policy. This example creates a new array of the data type String as defined in the mompkg.jar file compiled from a WSDL file.

```
// Call WSSetDefaultPKGName

WSSetDefaultPKGName("mompkg");

// Call WSNewArray

MyArray = WSNewArray("String", 4);
```

The following example shows how to use WSNewArray to create a new Web services array, where you have not previously called WSSetDefaultPKGName in the policy.

```
// Call WSNewArray

MyArray = WSNewArray("mompkg.String", 4);
```

WSNewEnum

The WSNewEnum function returns an enumeration value to a target Web service.

Syntax

This function has the following syntax:

```
[Return] = WSNewEnum(EnumType, EnumValue);
```

Parameters

The WSNewEnum function has the following parameters.

Table 101. WSNewEnum function parameters

Parameter	Format	Description
EnumType	String	The enumeration class name that exists in the package that is created by nci_compilewsdl.
EnumValue	String	The enumeration value to return.

Return Value

A new enumeration type and value.

Example

The following example shows how to use the WSNewEnum function to send a message to the target Web service.

```
euro = WSNewEnum("net.webservicex.www.Currency", "EUR");
usd = WSNewEnum("net.webservicex.www.Currency", "USD");
```

WSNewObject

The WSNewObject function creates a new object of a complex data type as defined in the WSDL file for the Web service.

You use this function in cases where you are required to pass data of a complex type to a Web service as a message parameter.

Syntax

This function has the following syntax:

```
Object = WSNewObject(ElementType)
```

Parameters

This WSNewObject function has the following parameter.

Table 102. WSNewObject function parameter

Parameter	Format	Description
ElementType	String	Name of the complex data type defined in the WSDL file. The name format is *[Package.]TypeName*, where *Package* is the name of the package you created when you compiled the WSDL file, without the .jar suffix.

Return Value

A new Web services object.

Examples

The following example shows how to use WSNewObject to create a new Web services object, where you have previously called WSSetDefaultPKGName in the

policy. This example creates a new object of the data type ForwardeeInfo as defined in the mompkg.jar file compiled from the corresponding WSDL.

```
// Call WSSetDefaultPKGName
WSSetDefaultPKGName("mompkg");

// Call WSNewObject

MyObject = WSNewObject("ForwardeeInfo");
```

The following example shows how to use WSNewObject to create a new Web services object, where you have not previously called WSSetDefaultPKGName in the policy.

```
// Call WSNewObject

MyObject = WSNewObject("mompkg.ForwardeeInfo");
```

WSNewSubObject

The WSNewSubObject function creates a new child object that is part of its parent object and has a field or attribute name of ChildName.

Syntax

This function has the following syntax:

```
Object = WSNewSubObject(ParentObject, ChildName)
```

Parameters

This WSNewSubObject function has the following parameter.

Table 103. WSNewSubObject function parameters

Parameter	Format	Description
ParentObject	String	Name of the parent object.
ChildName	String	Name of the new child object.

Return Value

A new Web services child object.

Examples

The following example shows how to use WSNewSubObject to create a new Web services child object:

```
// Call WSNewSubObject

ticketId=WSNewSubobject(incident, "TICKETID");
```

WSSetDefaultPKGName

The WSSetDefaultPKGName function sets the default package used by WSNewObject and WSNewArray.

The package name is the name you supplied to the nci_compilewsdl script when you compiled the WSDL file for the Web service. This is also the name of the jar file created by this script, without the .jar suffix.

Syntax

This function has the following syntax:

```
WSSetDefaultPKGName(PackageName)
```

Parameters

The WSSetDefaultPKGName function has the following parameter.

Table 104. WSSetDefaultPKGName function parameter

Parameter	Format	Description
PackageName	String	Name of the default WSDL package used by WSNewObject and WSNewArray.

Example

The following example sets the default package used by subsequent calls to WSNewObject and WSNewArray to google.

```
WSSetDefaultPKGName("google");
```

Appendix A. Accessibility

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. These are the major accessibility features you can use with *Netcool/Impact* when accessing it on the *IBM Personal Communications* terminal emulator:

- You can operate all features using the keyboard instead of the mouse.
- You can read text through interaction with assistive technology.
- You can use system settings for font, size, and color for all user interface controls.
- You can magnify what is displayed on your screen.

For more information about viewing PDFs from Adobe, go to the following web site: http://www.adobe.com/enterprise/accessibility/main.html

Appendix B. Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785 U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan Ltd.
1623-14, Shimotsuruma, Yamato-shi
Kanagawa 242-8502 Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement might not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
2Z4A/101
11400 Burnet Road
Austin, TX 78758 U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to

IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

If you are viewing this information in softcopy form, the photographs and color illustrations might not be displayed.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

Adobe, Acrobat, PostScript and all Adobe-based trademarks are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other product and service names might be trademarks of IBM or other companies.

Glossary

This glossary includes terms and definitions for Netcool/Impact.

The following cross-references are used in this glossary:

- See refers you from a term to a preferred synonym, or from an acronym or abbreviation to the defined full form.
- See also refers you to a related or contrasting term.

To view glossaries for other IBM products, go to www.ibm.com/software/globalization/terminology (opens in new window).

A

assignment operator
An operator that sets or resets a value to a variable. See also operator.

B

Boolean operator
A built-in function that specifies a logical operation of AND, OR or NOT when sets of operations are evaluated. The Boolean operators are &&, || and !. See also operator.

C

command execution manager
The service that manages remote command execution through a function in the policies.

command line manager
The service that manages the command-line interface.

Common Object Request Broker Architecture (CORBA)
An architecture and a specification for distributed object-oriented computing that separates client and server programs with a formal interface definition.

comparison operator
A built-in function that is used to compare two values. The comparison operators are ==, !=, <, >, <= and >=. See also operator.

control structure
A statement block in the policy that is executed when the terms of the control condition are satisfied.

CORBA
See Common Object Request Broker Architecture.

D

database (DB)
A collection of interrelated or independent data items that are stored together to serve one or more applications. See also database server.

database event listener
A service that listens for incoming messages from an SQL database data source and then triggers policies based on the incoming message data.

database event reader
An event reader that monitors an SQL database event source for new and modified events and triggers policies based on the event information. See also event reader.

database server
A software program that uses a database manager to provide database services to other software programs or computers. See also database.

data item
A unit of information to be processed.

data model
An abstract representation of the business data and metadata used in an installation. A data model contains data sources, data types, links, and event sources.

data source
A repository of data to which a federated server can connect and then retrieve data by using wrappers. A data source can contain relational databases, XML files, Excel spreadsheets, table-structured files, or other objects. In a federated system, data sources seem to be a single collective database.

data source adapter (DSA)
A component that allows the application to access data stored in an external source.

data type
An element of a data model that represents a set of data stored in a data source, for example, a table or view in a relational database.

DB See database.

DSA See data source adapter.

dynamic link
An element of a data model that represents a dynamic relationship between data items in data types. See also link.

E

email reader
A service that polls a Post Office Protocol (POP) mail server at intervals for incoming email and then triggers policies based on the incoming email data.

email sender
A service that sends email through an Simple Mail Transfer Protocol (SMTP) mail server.

event An occurrence of significance to a task or system. Events can include completion or failure of an operation, a user action, or the change in state of a process.

event processor
The service responsible for managing events through event reader, event

listener and email reader services. The event processor manages the incoming event queue and is responsible for sending queued events to the policy engine for processing.

event reader
A service that monitors an event source for new, updated, and deleted events, and triggers policies based on the event data. See also database event reader, standard event reader.

event source
A data source that stores and manages events.

exception
A condition or event that cannot be handled by a normal process.

F

field A set of one or more adjacent characters comprising a unit of data in an event or data item.

filter A device or program that separates data, signals, or material in accordance with specified criteria. See also LDAP filter, SQL filter.

function
Any instruction or set of related instructions that performs a specific operation. See also user-defined function.

G

generic event listener
A service that listens to an external data source for incoming events and triggers policies based on the event data.

graphical user interface (GUI)
A computer interface that presents a visual metaphor of a real-world scene, often of a desktop, by combining high-resolution graphics, pointing devices, menu bars and other menus, overlapping windows, icons and the object-action relationship. See also graphical user interface server.

graphical user interface server (GUI server)
A component that serves the web-based graphical user interface to web browsers through HTTP. See also graphical user interface.

GUI See graphical user interface.

GUI server
See graphical user interface server.

H

hibernating policy activator
A service that is responsible for waking hibernating policies.

I

instant messaging reader
A service that listens to external instant messaging servers for messages and triggers policies based on the incoming message data.

instant messaging service
A service that sends instant messages to instant messaging clients through a Jabber server.

IPL See Netcool/Impact policy language.

J

Java Database Connectivity (JDBC)
An industry standard for database-independent connectivity between the Java platform and a wide range of databases. The JDBC interface provides a call level interface for SQL-based and XQuery-based database access.

Java Message Service (JMS)
An application programming interface that provides Java language functions for handling messages.

JDBC See Java Database Connectivity.

JMS See Java Message Service.

JMS data source adapter (JMS DSA)
A data source adapter that sends and receives Java Message Service (JMS) messages.

JMS DSA
See JMS data source adapter.

K

key expression
An expression that specifies the value that one or more key fields in a data item must have in order to be retrieved in the IPL.

key field
A field that uniquely identifies a data item in a data type.

L

LDAP See Lightweight Directory Access Protocol.

LDAP data source adapter (LDAP DSA)
A data source adapter that reads directory data managed by an LDAP server. See also Lightweight Directory Access Protocol.

LDAP DSA
See LDAP data source adapter.

LDAP filter
An expression that is used to select data elements located at a point in an LDAP directory tree. See also filter.

Lightweight Directory Access Protocol (LDAP)
An open protocol that uses TCP/IP to provide access to directories that support an X.500 model and that does not incur the resource requirements of the more complex X.500 Directory Access Protocol (DAP). For example, LDAP can be used to locate people, organizations, and other resources in an Internet or intranet directory. See also LDAP data source adapter.

link An element of a data model that defines a relationship between data types and data items. See also dynamic link, static link.

M

mathematic operator
A built-in function that performs a mathematic operation on two values. The mathematic operators are +, -, *, / and %. See also operator.

mediator DSA
A type of data source adaptor that allows data provided by third-party systems, devices, and applications to be accessed.

N

Netcool/Impact policy language (IPL)
A programming language used to write policies.

O

operator
A built-in function that assigns a value to a variable, performs an operation on a value, or specifies how two values are to be compared in a policy. See also assignment operator, Boolean operator, comparison operator, mathematic operator, string operator.

P

policy A set of rules and actions that are required to be performed when certain events or status conditions occur in an environment.

policy activator
A service that runs a specified policy at intervals that the user defines.

policy engine
A feature that automates the tasks that the user specifies in the policy scripting language.

policy logger
The service that writes messages to the policy log.

POP See Post Office Protocol.

Post Office Protocol (POP)
A protocol that is used for exchanging network mail and accessing mailboxes.

precision event listener
A service that listens to the application for incoming messages and triggers policies based on the message data.

S

security manager
A component that is responsible for authenticating user logins.

self-monitoring service
A service that monitors memory and other status conditions and reports them as events.

server A component that is responsible for maintaining the data model, managing services, and running policies.

service
A runnable sub-component that the user controls from within the graphical user interface (GUI).

Simple Mail Transfer Protocol (SMTP)
An Internet application protocol for transferring mail among users of the Internet.

Simple Network Management Protocol (SNMP)
A set of protocols for monitoring systems and devices in complex networks. Information about managed devices is defined and stored in a Management Information Base (MIB). See also SNMP data source adapter.

SMTP See Simple Mail Transfer Protocol.

SNMP
See Simple Network Management Protocol.

SNMP data source adapter (SNMP DSA)
A data source adapter that allows management information stored by SNMP agents to be set and retrieved. It also allows SNMP traps and notifications to be sent to SNMP managers. See also Simple Network Management Protocol.

SNMP DSA
See SNMP data source adapter.

socket DSA
A data source adaptor that allows information to be exchanged with external applications using a socket server as the brokering agent.

SQL database DSA
A data source adaptor that retrieves information from relational databases and other data sources that provide a public interface through Java Database Connectivity (JDBC). SQL database DSAs also add, modify and delete information stored in these data sources.

SQL filter
An expression that is used to select rows in a database table. The syntax for the filter is similar to the contents of an SQL WHERE clause. See also filter.

standard event reader
A service that monitors a database for new, updated, and deleted events and triggers policies based on the event data. See also event reader.

static link
An element of a data model that defines a static relationship between data items in internal data types. See also link.

string concatenation
In REXX, an operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two characters or strings.

string operator
A built-in function that performs an operation on two strings. See also operator.

U

user-defined function
A custom function that can be used to organize code in a policy. See also function.

V

variable
A representation of a changeable value.

W

web services DSA
A data source adapter that exchanges information with external applications that provide a web services application programming interface (API).

X

XML data source adapter
A data source adapter that reads XML data from strings and files, and reads XML data from web servers over HTTP.

Index

Special characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Printed in USA

SC14-7553-00