TIBCO® Object Service Broker

Programming in Rules

Software Release 6.0
July 2012

two-second advantage™

TIBCO®
The Power of Now®

Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE LICENSE FILE) OR IF THERE IS NO SUCH SOFTWARE LICENSE AGREEMENT OR CLICKWRAP END USER LICENSE AGREEMENT, THE LICENSE(S) LOCATED IN THE "LICENSE" FILE(S) OF THE SOFTWARE. USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.

This document contains confidential information that is subject to U.S. and international copyright laws and treaties. No part of this document may be reproduced in any form without the written authorization of TIBCO Software Inc.

TIBCO, The Power of Now, TIBCO Object Service Broker, and and TIBCO Service Gateway are either registered trademarks or trademarks of TIBCO Software Inc. in the United States and/or other countries.

All other product and company names and marks mentioned in this document are the property of their respective owners and are mentioned for identification purposes only.

THIS SOFTWARE MAY BE AVAILABLE ON MULTIPLE OPERATING SYSTEMS. HOWEVER, NOT ALL OPERATING SYSTEM PLATFORMS FOR A SPECIFIC SOFTWARE VERSION ARE RELEASED AT THE SAME TIME. SEE THE README FILE FOR THE AVAILABILITY OF THIS SOFTWARE VERSION ON A SPECIFIC OPERATING SYSTEM PLATFORM.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

THE CONTENTS OF THIS DOCUMENT MAY BE MODIFIED AND/OR QUALIFIED, DIRECTLY OR INDIRECTLY, BY OTHER DOCUMENTATION WHICH ACCOMPANIES THIS SOFTWARE, INCLUDING BUT NOT LIMITED TO ANY RELEASE NOTES AND "READ ME" FILES.

The TIBCO Object Service Broker technologies described herein are protected under the following patent numbers:

Australia:	-	-	671137	671138	673682	646408
Canada:	2284250	-	-	2284245	2284248	2066724
Europe:	-	-	0588446	0588445	0588447	0489861
Japan:	-	-	-	-	-	2-513420
USA:	5584026	5586329	5586330	5594899	5596752	5682535

Contents

Preface

TIBCO® Object Service Broker is an application development environment and integration broker that bridges legacy and non-legacy applications and data.

This manual explains how to use the rules language to create and modify application code. It also explains how to edit, execute, and debug rules.

Topics

Related Documentation

This section lists documentation resources you may find useful.

TIBCO Object Service Broker Documentation

The following documents form the TIBCO Object Service Broker documentation set:

Fundamental Information

The following manuals provide fundamental information about TIBCO Object Service Broker:

- *TIBCO Object Service Broker Getting Started* Provides the basic concepts and principles of TIBCO Object Service Broker and introduces its components and capabilities. It also describes how to use the default developer's workbench and includes a basic tutorial of how to build an application using the product. A product glossary is also included in the manual.
- *TIBCO Object Service Broker Messages with Identifiers* Provides a listing of the TIBCO Object Service Broker messages that are issued with alphanumeric identifiers. The description of each message includes the source and explanation of the message and recommended action to take.
- *TIBCO Object Service Broker Messages without Identifiers* Provides a listing of the TIBCO Object Service Broker messages that are issued without a message identifier. These messages use the percent symbol (%) or the number symbol (#) to represent such variable information as a rules name or the number of occurrences in a table. The description of each message includes the source and explanation of the message and recommended action to take.
- *TIBCO Object Service Broker Quick Reference* Presents summary information for use in the TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Shareable Tools* Lists and describes the TIBCO Object Service Broker shareable tools. Shareable tools are programs supplied with TIBCO Object Service Broker that facilitate rules language programming and application development.
- *TIBCO Object Service Broker Release Notes* Read the release notes for a list of new and changed features. This document also contains lists of known issues and closed issues for this release.

Application Development and Management

The following manuals provide information about application development and management:

- *TIBCO Object Service Broker Application Administration* Provides information required to administer the TIBCO Object Service Broker application development environment. It describes how to use the administrator's workbench, set up the development environment, and optimize access to the database. It also describes how to manage the Pagestore, which is the native TIBCO Object Service Broker data store.
- *TIBCO Object Service Broker Managing Data* Describes how to define, manipulate, and manage data required for a TIBCO Object Service Broker application.
- *TIBCO Object Service Broker Managing External Data* Describes the TIBCO Object Service Broker interface to external files (not data in external databases) and describes how to define TIBCO Object Service Broker tables based on these files and how to access their data.
- *TIBCO Object Service Broker National Language Support* Provides information about implementing the National Language Support in a TIBCO Object Service Broker environment.
- *TIBCO Object Service Broker Object Integration Gateway* Provides information about installing and using the Object Integration Gateway which is the interface for TIBCO Object Service Broker to XML, J2EE, .NET and COM.
- *TIBCO Object Service Broker for Open Systems External Environments* Provides information on interfacing TIBCO Object Service Broker with the Windows and Solaris environments. It includes how to use SDK (C/C++) and SDK (Java) to access TIBCO Object Service Broker data, how to interface to TIBCO Enterprise Messaging Service (EMS), how to use the TIBCO Service Gateway for WMQ, how to use the Adapter for JDBC-ODBC, and how to access programs written in external programming languages from within TIBCO Object Service Broker.
- *TIBCO Object Service Broker for z/OS External Environments* Provides information on interfacing TIBCO Object Service Broker to various external environments within a TIBCO Object Service Broker z/OS environment. It also includes information on how to access TIBCO Object Service Broker from different terminal managers, how to write programs in external programming languages to access TIBCO Object Service Broker data, how to interface to TIBCO Enterprise Messaging Service (EMS), how to use the TIBCO Service Gateway for WMQ, and how to access programs written in external programming languages from within TIBCO Object Service Broker.

- *TIBCO Object Service Broker Parameters* Lists the TIBCO Object Service Broker Execution Environment and Data Object Broker parameters and describes their usage.
- *TIBCO Object Service Broker Programming in Rules* Explains how to use the TIBCO Object Service Broker rules language to create and modify application code. The rules language is the programming language used to access the TIBCO Object Service Broker database and create applications. The manual also explains how to edit, execute, and debug rules.
- *TIBCO Object Service Broker Managing Deployment* Describes how to submit, maintain, and manage promotion requests in the TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Defining Reports* Explains how to create both simple and complex reports using the reporting tools provided with TIBCO Object Service Broker. It explains how to create reports with simple features using the Report Generator and how to create reports with more complex features using the Report Definer.
- *TIBCO Object Service Broker Managing Security* Describes how to set up, use, and administer the security required for an TIBCO Object Service Broker application development environment.
- *TIBCO Object Service Broker Defining Screens and Menus* Provides the basic information to define screens, screen tables, and menus using TIBCO Object Service Broker facilities.
- *TIBCO Service Gateway for Files SDK* Describes how to use the SDK provided with the TIBCO Service Gateway for Files to create applications to access Adabas, CA Datacom, and VSAM LDS data.

System Administration on the z/OS Platform

The following manuals describe system administration on the z/OS platform:

- *TIBCO Object Service Broker for z/OS Installing and Operating* Describes how to install, migrate, update, maintain, and operate TIBCO Object Service Broker in a z/OS environment. It also describes the Execution Environment and Data Object Broker parameters used by TIBCO Object Service Broker.
- *TIBCO Object Service Broker for z/OS Managing Backup and Recovery* Explains the backup and recovery features of OSB for z/OS. It describes the key components of TIBCO Object Service Broker systems and describes how you can back up your data and recover from errors. You can use this information, along with assistance from TIBCO Support, to develop the best customized solution for your unique backup and recovery requirements.

- *TIBCO Object Service Broker for z/OS Monitoring Performance* Explains how to obtain and analyze performance statistics using TIBCO Object Service Broker tools and SMF records
- *TIBCO Object Service Broker for z/OS Utilities* Contains an alphabetically ordered listing of TIBCO Object Service Broker utilities for z/OS systems. These are TIBCO Object Service Broker administrator utilities that are typically run with JCL.

System Administration on Open Systems

The following manuals describe system administration on open systems such as Windows or UNIX:

- *TIBCO Object Service Broker for Open Systems Installing and Operating* Describes how to install, migrate, update, maintain, and operate TIBCO Object Service Broker in Windows and Solaris environments.
- *TIBCO Object Service Broker for Open Systems Managing Backup and Recovery* Explains the backup and recovery features of TIBCO Object Service Broker for Open Systems. It describes the key components of a TIBCO Object Service Broker system and describes how to back up your data and recover from errors. Use this information to develop a customized solution for your unique backup and recovery requirements.
- *TIBCO Object Service Broker for Open Systems Utilities* Contains an alphabetically ordered listing of TIBCO Object Service Broker utilities for Windows and Solaris systems. These TIBCO Object Service Broker administrator utilities are typically executed from the command line.

External Database Gateways

The following manuals describe external database gateways:

- *TIBCO Service Gateway for DB2 Installing and Operating* Describes the TIBCO Object Service Broker interface to DB2 data. Using this interface, you can access external DB2 data and define TIBCO Object Service Broker tables based on this data.
- *TIBCO Service Gateway for IDMS/DB Installing and Operating* Describes the TIBCO Object Service Broker interface to CA-IDMS data. Using this interface, you can access external CA-IDMS data and define TIBCO Object Service Broker tables based on this data.
- *TIBCO Service Gateway for IMS/DB Installing and Operating* Describes the TIBCO Object Service Broker interface to IMS/DB and DB2 data. Using this interface, you can access external IMS data and define TIBCO Object Service Broker tables based on it.

- *TIBCO Service Gateway for ODBC and for Oracle Installing and Operating*
  Describes the TIBCO Object Service Broker ODBC Gateway and the TIBCO Object Service Broker Oracle Gateway interfaces to external DBMS data. Using this interface, you can access external DBMS data and define TIBCO Object Service Broker tables based on this data.

Typographical Conventions

The following typographical conventions are used in this manual.

Table 1 General Typographical Conventions

Convention	Use
TIBCO_HOME *OSB_HOME*	By default, all TIBCO products are installed into a folder referenced in the documentation as *TIBCO_HOME*. On open systems, TIBCO Object Service Broker installs by default into a directory within *TIBCO_HOME*. This directory is referenced in documentation as *OSB_HOME*. The default value of *OSB_HOME* depends on the operating system. For example on Windows systems, the default value is `C:\tibco\OSB`. Similarly, all TIBCO Service Gateways on open systems install by default into a directory in *TIBCO_HOME*. For example on Windows systems, the default value is `C:\tibco\OSBgateways\6.0`. On z/OS, no default installation directories exist.
`code font`	Code font identifies commands, code examples, filenames, pathnames, and output displayed in a command window. For example: Use `MyCommand` to start the foo process.
`bold code font`	Bold code font is used in the following ways: • In procedures, to indicate what a user types. For example: Type **`admin`**. • In large code samples, to indicate the parts of the sample that are of particular interest. • In command syntax, to indicate the default parameter for a command. For example, if no parameter is specified, `MyCommand` is enabled: `MyCommand [enable | disable]`
italic font	Italic font is used in the following ways: • To indicate a document title. For example: See *TIBCO ActiveMatrix BusinessWorks Concepts*. • To introduce new terms For example: A portal page may contain several portlets. *Portlets* are mini-applications that run in a portal. • To indicate a variable in a command or code syntax that you must replace. For example: `MyCommand` *PathName*

Table 1 General Typographical Conventions (Cont'd)

Convention	Use
Key combinations	Key name separated by a plus sign indicate keys pressed simultaneously. For example: Ctrl+C. Key names separated by a comma and space indicate keys pressed one after the other. For example: Esc, Ctrl+Q.
	The note icon indicates information that is of special interest or importance, for example, an additional action required only in certain circumstances.
	The tip icon indicates an idea that could be useful, for example, a way to apply the information provided in the current section to achieve a specific result.
	The warning icon indicates the potential for a damaging situation, for example, data loss or corruption if certain steps are taken or not taken.

Table 2 Syntax Typographical Conventions

Convention	Use
`[ ]`	An optional item in a command or code syntax. For example: `MyCommand [optional_parameter] required_parameter`
`|`	A logical `OR` that separates multiple items of which only one may be chosen. For example, you can select only one of the following parameters: `MyCommand para1 | param2 | param3`

Table 2 Syntax Typographical Conventions

Convention	Use

`{ }`	A logical group of items in a command. Other syntax notations may appear within each logical group.
	
	For example, the following command requires two parameters, which can be either the pair `param1` and `param2`, or the pair `param3` and `param4`.
	
	`MyCommand {param1 param2} | {param3 param4}`
	
	In the next example, the command requires two parameters. The first parameter can be either `param1` or `param2` and the second can be either `param3` or `param4`:
	
	`MyCommand {param1 | param2} {param3 | param4}`
	
	In the next example, the command can accept either two or three parameters. The first parameter must be `param1`. You can optionally include `param2` as the second parameter. And the last parameter is either `param3` or `param4`.
	
	`MyCommand param1 [param2] {param3 | param4}`

Connecting with TIBCO Resources

How to Join TIBCOmmunity

TIBCOmmunity is an online destination for TIBCO customers, partners, and resident experts, a place to share and access the collective experience of the TIBCO community. TIBCOmmunity offers forums, blogs, and access to a variety of resources. To register, go to http://www.tibcommunity.com.

How to Access All TIBCO Documentation

You can access TIBCO documentation here:

http://docs.tibco.com

How to Contact TIBCO Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support as follows.

- For an overview of TIBCO Support, and information about getting started with TIBCO Support, visit this site:

  http://www.tibco.com/services/support

- If you already have a valid maintenance or support contract, visit this site:

  https://support.tibco.com

  Entry to this site requires a user name and password. If you do not have a user name, you can request one.

Chapter 1 Introduction to TIBCO Object Service Broker Rules

This chapter introduces the TIBCO Object Service Broker rules.

Topics

Use of TIBCO Object Service Broker Rules

What is the Rules Language?

The rules language is the programming language that you use within TIBCO Object Service Broker to create and modify your applications. It consists of:

- Database access commands, for example, GET or DELETE statements
- High-level constructs such as loops, for example, FORALL and UNTIL statements
- High-level constructs such as exception signaling and handling, for example, SIGNAL and ON statements
- Pseudo-conversational constructs, for example, the DISPLAY & TRANSFERCALL statement
- Language statements that are characteristic of procedural programming, for example, CALL and EXECUTE statements

What is a Rule?

A rule is a series of one or more rules language statements that contains a procedure or returns a value. A rule must have at least:

- A unique name within a rules library
- One or more action statements

It can also contain:

- Variables
- Conditional processing
- Calls to other rules
- Transference to another transaction
- Error handling

How are Rules Defined?

You define rules using the Rule Editor. This tool, which is described in this manual, is available to you from the developer workbench shipped with TIBCO Object Service Broker. The Rule Editor displays either an empty rule template for creating a new rule or the source code for an existing rule.

How Do You Execute a Rule?

As soon as your rules pass syntax checking, you can execute them in any of the following ways:

- In standard execution mode, using the Rule Executor from the workbench. This is described in Chapter 16, Processing in Standard Execution Mode, on page 175.
- In batch mode, using asynchronous processing. This is described in Chapter 17, Processing Asynchronously in Batch Mode, on page 191.
- In debug mode, using the Rule Debugger. This facility is described in Chapter 18, Processing in Debug Mode, on page 199.

What Tools are Available for Use with Rules?

The following table lists the tools available to you to write, debug, and execute your rules and indicates where you can get information about each one:

Tool or Function	Refer to …
Rule Editor, the primary tool when writing rules.	Chapter 13, Using Rules Libraries, on page 141 to Chapter 15, Editing Rules, on page 157.
Rule Executor.	Chapter 16, Processing in Standard Execution Mode, on page 175.
Rule scheduling.	SCHEDULE Statement on page 53 and Chapter 17, Processing Asynchronously in Batch Mode, on page 191.
Rule Debugger.	Chapter 18, Processing in Debug Mode, on page 199.
Library Definer.	Chapter 13, Using Rules Libraries, on page 141.
Rule Printer—this produces a tree of rules and associated objects based on an entry rule.	RULEPRINTER in *TIBCO Object Service Broker Shareable Tools*.

Tool or Function	Refer to …
Search Utility—in addition to other functionality, this utility searches the installation library for specified rules, exceptions, and exception handlers.	CROSSREFSEARCH in *TIBCO Object Service Broker Shareable Tools*.
Copy definition—this copies the definition of a rule from one library to another. The source and destination libraries can be located on the same node or on a node remote to each other.	COPYDEFN and COPY_DEFN in *TIBCO Object Service Broker Shareable Tools*.
Compare definition—this compares the definition of one rule to another. The source and destination rules can be located on the same node or on nodes remote to each other.	DIFFDEFN and DIFF_DEFN in *TIBCO Object Service Broker Shareable Tools*.

Sample Set of Rules

Description

The EMPLOYEES_RAISE rule assigns raises to employees based on their job title. To avoid commit limits, it calls in the REPLACE_SALARY rule shown in REPLACE_SALARY Rule on page 6 to actually replace the occurrences in the EMPLOYEES table. After the REPLACE_SALARY rule is processed, control is passed back to the EMPLOYEES_RAISE rule, which then completes processing.

EMPLOYEES_RAISE Rule

```
RULE EDITOR ===>                                                    SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------------
_ JOBTITLE = 'SENIOR ANALYST';                                    | Y N N
_ JOBTITLE = 'ANALYST';                                           |   Y N
_ ----------------------------------------------------------------+-------------
_ RATE = 0.1;                                                     | 1
_ RATE = 0.05;                                                    |   1
_ RATE = 0.02;                                                    |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;                |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:             | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                             |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;                 |
_    CALL REPLACE_SALARY(REGION);                                 |
_    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||              |
_      EMPLOYEES.SALARY);                                         |
_    END;                                                         |
_ ------------------------------------------------------------------------------
  ON GETFAIL:
     CALL ENDMSG('POSITION IS INVALID');
```

REPLACE_SALARY Rule

Chapter 2 Composition of a Rule

This chapter describes the composition of a rule.

Topics

Components of a Rule

What are the Components?

A rule is divided into four components:

- The rules declaration, including optional arguments and local variables
- Conditions, including a quadrant of Y/N (yes/no) values
- Actions, including columns of action sequence numbers
- Exception handlers

Illustration of the Component Parts of a Rule

Using the following sample rule, this chapter explains the components of a TIBCO Object Service Broker rule.

```
RULE EDITOR ===>                                                  SCROLL: P
 EMPLOYEES_RAISE(JOBTITLE, REGION);              Declaration
 _ LOCAL RAISE, RATE;
 _ ------------------------------------------------------------------------
 _ JOBTITLE = 'SENIOR ANALYST';                  Conditions      | Y N N
 _ JOBTITLE = 'ANALYST';                                         |   Y N
 _ ------------------------------------------------------------+----------
 _ RATE = 0.1;                                                   | 1
 _ RATE = 0.05;                                   Actions        |   1
 _ RATE = 0.02;                                                  |     1
 _ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;              |     2
 _ FORALL EMPLOYEES.SALARY * RATE;                               | 2 2 3
 _   EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;                |
 _   CALL REPLACE_SALARY(REGION);                                |
 _   CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||            |
 _     EMPLOYEES.SALARY);                                        |
 _ END;                                                          |
 _ ------------------------------------------------------------------------
   ON GETFAIL:                                   Exception Handlers
      CALL ENDMSG('POSITION IS INVALID');


 PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Information about Syntax

For an explanation of the syntax of these components, refer to Chapter 8, Using Expressions and Operators, on page 77 and Syntax of Rules on page 215.

Identifying a Rule

Declaring a Rule Name and Its Arguments

Use the rules header to name the rule and declare any arguments. For example, in the sample rule, the rule name is EMPLOYEES_RAISE, and two arguments, JOBTITLE and REGION, are declared. The declaration ends with a semicolon (;) and arguments are enclosed in parentheses () and separated by commas (,).

Sample Rules Declaration

The following example shows the rules declaration containing the rule name and argument list:

```
    RULE EDITOR ===>
EMPLOYEES_RAISE(JOBTITLE, REGION);
.....;
_ ---------------------------------------------------
```

Behavior of Rules Arguments

The scope of an argument is the rule where the argument is declared. The data representation of an argument is dynamic; it conforms to the semantic data type and syntax of the value passed to it. Argument values are passed to the rule when the rule is invoked. Passing arguments between rules is done by value; no rule can alter the value of an argument.

Handling Dynamic Data Values

Declaring Local Variables

You use local variables to hold dynamic data values. If your rule makes use of dynamic data values, declare the local variables that you require below the rules header.

Sample Local Variable

The following example shows the rules declaration containing the rule name and argument list, the local variables RAISE and RATE, and the assignment of a value to the local variable RATE:

```
    RULE EDITOR ===>
  EMPLOYEES_RAISE(JOBTITLE, REGION);
  LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------------
- .....                                                          |
- ---------------------------------------------------------------+----------
- RATE = 0.1;                                                    |
  .....
```

Valid Values

The declaration begins with the reserved word LOCAL and ends with a semicolon (;). If you have more than one local variable, they are separated with commas.

A local variable can be assigned an arithmetic value or a string. The maximum value or string length that can be assigned to local variables during a session is determined by the session attributes.

Scope of Local Variables

The scope of a local variable is the rule where it is declared and any descendant rules (rules that are below the rule in the calling hierarchy). They can be used anywhere in an action, except to supply a value for an indirect reference. For more information about indirect references, refer to Indirect Referencing on page 95.

Data Representation of Local Variables

The following rules apply to the data representation of a local variable:

- If the variable is assigned a value from a table, it conforms to the semantic data type and syntax of the value assigned to it.
- If the variable is assigned a literal value in a rule (for example, *variable*=12;), no semantic type is assigned, only a syntax.
- If the variable is assigned a literal value in a rule and the concatenation operator is used (for example, *variable* = 'a' || 'b';), a string (S) semantic data type is assigned.

For more information about syntax and semantic data types, refer to Syntax of Data Elements on page 80 and Semantic Data Types on page 84.

Initial Behavior of a Local Variable

Before a value is assigned to a local variable, it behaves as:

- An empty string in a character context
- A zero in a numeric context
- An N in a logical context

Providing Conditional Processing

Conditional Processing

If you require conditional processing based on truth value, use:

1. The conditions section to specify what conditions is to be evaluated
2. The action section of the rule to specify how the conditions is to be processed

The values in the condition section are evaluated sequentially for their truth value. An Y/N quadrant is then used to regulate the processing of the specified actions. For more information about conditional processing, refer to Chapter 10, Conditional Processing, on page 115.

What Comprises a Condition?

A condition is comprised of one of the following that evaluates to a Y/N value. It ends in a semicolon(;).

- An expression with a comparison operator
- An argument name or local name
- A functional rule and its argument values
- A *table.field* reference

Sample Condition Segment and Associated Actions

The following example shows a condition segment containing an expression and the action segment showing the initial actions to take place based on the values determined by the Y/N quadrant:

```
_ ------------------------------------------------------------+--------------
_ JOBTITLE = 'SENIOR ANALYST';                                | Y N N
- JOBTITLE = 'ANALYST';                                       |   Y N
_ ------------------------------------------------------------+--------------
_ RATE = 0.1;                                                 | 1
- RATE = 0.05;                                                |   1
```

Coding Actions

Action Statements

An action is an executable statement and a rule must contain at least one action. You code the actions in the main body of the rule using action statements. Each action ends with a semicolon (;), except for an action starting with a FORALL or UNTIL statement that ends with a colon (:). For more information about action statements refer to Chapter 4, Types of Action Statements, on page 27.

Action Sequence Numbers

Action sequence numbers determine which actions are executed for each particular condition; the same action can be executed for different conditions. An action must have an action sequence number to be executable. If an action occupies more than one line, the action sequence number is associated with the first line of the action only.

Restrictions

Action sequence numbers are not permitted within a FORALL or UNTIL loop. Since a FORALL or UNTIL loop constitutes a single compound statement, all the actions within it are executed whenever the FORALL or UNTIL is executed. Refer to Looping Statements on page 29 for more information.

Editing Action Sequence Numbers

If the rule contains no conditions, the Rule Editor supplies consecutive numbers. The statements are executed in order from top to bottom. You can edit the numbers, and when you save the rule the Rule Editor renumbers them sequentially. If you delete an action sequence number, the corresponding action is not executed. If the rule contains conditions, you must supply values for the action sequence numbers.

Behavior of Action Sequence Numbers

You can type in an alphanumeric character for each executable action and all the characters are converted into sequenced numbers when you save the rule or press Enter. For example, if you enter 1, 2, A, B, they are changed to 1, 2, 3, 4.

Sample of the Rules Actions

```
_ ------------------------------------------------------------+--------------
_ ....;                                                       | Y N N
- ....;                                                       |   Y N
_ ------------------------------------------------------------+--------------
_ RATE = 0.1;                                                 | 1
_ RATE = 0.05;                                                |   1
_ RATE = 0.02;                                                |     1
_  GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;           |     2
_  FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:        | 2 2 3
_     RAISE = EMPLOYEES.SALARY * RATE;                        |
_     EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;            |
_     CALL REPLACE_SALARY(REGION);                            |
_     CALL MSGLOG(EMPLOYEES.LNAME || ‘ NOW EARNS ‘ ||         |
_        EMPLOYEES.SALARY;                                    |
_     END;                                                    |
_ ------------------------------------------------------------------------------
```

Handling Exceptions

Exception Handlers

If you require exception handling, use the exceptions segment of a rule to intercept user-defined or system-defined exceptions. An exception handler is comprised of:

- The ON statement
- The name of the exception that is trapped
- A colon (:), which ends the statement
- Optionally, statements executed when the exception is trapped

For more information on exception handling refer to Chapter 6, Exception Handling, on page 59.

Behavior of Exception Handlers

There are no action sequence numbers associated with exception handlers. Every statement of an exception handler is executed when the exception is trapped.

Sample Exception Handling Statement

The following shows the exception handler ON GETFAIL. In this example, the shareable tool ENDMSG is issued if the value provided for the JOBNAME argument is not a valid name.

```
_ ------------------------------------------------------------+--------------
 _  ON GETFAIL:
 _     CALL ENDMSG('POSITION IS INVALID');
```

See Also *TIBCO Object Service Broker Shareable Tools* about tools.

Chapter 3 Supported Characters

This chapter describes the supported characters within a rule.

Topics

Lexical Elements

A rule is a sequence of lexical elements. Lexical elements are:

- Delimiters
- Hexadecimal literals (zero or more characters enclosed between an “X'” and a single quotation mark)
- Identifiers (including reserved words)
- Numeric literals
- Raw-data literals (zero or more characters enclosed between an “R'” and a single quotation mark)
- String literals (zero or more characters enclosed in single quotation marks)
- Unicode literals (zero or more characters enclosed between a “U'” and a single quotation mark)

Delimiters

A delimiter is one of the following special characters:

```
| & ¬ * ( ) - + = : ; ' , / < >
```

or one of the following compound symbols:

```
** <= >= ¬= || R' r' U' u' X' x'
```

Tokens

Each item that belongs to the group of lexical elements is called a token. For example, **, LOCAL, and '123 abc' are all tokens. Adjacent tokens can be separated by spaces, the delimiters previously listed, or a new line. An identifier or numeric literal must be separated in this way from an adjacent identifier or numeric literal. The only lexical elements that can contain a space are the string literal and the Unicode literal. String literals, hexadecimal literals, raw data literals, and Unicode literals can span more than one line; all other lexical elements must fit on a single line.

Hexadecimal literals are shown as X'xx...', raw data literals as R'xx...', and Unicode literals as U'xx...'.

In these examples, items 123 and abc are separate tokens when not enclosed in single quotation marks.

Reserved Words

The following names are reserved by the system as keywords in the rules language:

AND	ASCENDING	BROWSE
CALL	COMMIT	CONTINUE [a]
DELETE	DESCENDING	DISPLAY
END	EXECUTE	FORALL
GET	IN	INSERT
LIKE	LOCAL	NOT
NULL	ON	OR
ORDERED	PRINT	REPLACE
RETURN	ROLLBACK	SCHEDULE
SIGNAL	TO	TRANSFERCALL
UNTIL	UPDATE	WHERE

a. CONTINUE, while a reserved word, is not currently used in the rules language.

Character Set

All rules language constructs are represented with a character set that is subdivided as follows:

- Uppercase and lowercase letters

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  a b c d e f g h i j k l m n o p q r s t u v w x y z
- Digits

  0 1 2 3 4 5 6 7 8 9
- Special characters

  @ # $ | ¬ & * ( ) - _ = + ; : ' , . / < >
- The space character

If your system is NLS-enabled these characters are mapped to the appropriate code page for your operating environment. Refer to National Character Set Support on page 21 for more information.

National Character Set Support

Allowable Usage

If your hardware is designed and configured to handle other national character sets, you can use them in TIBCO Object Service Broker as follows:

- In tables where a field's syntax is fixed length character (C), variable length character (V), or single-byte or double-byte character (W)
- On screens and reports as literals
- In quoted strings

The alphabetic characters in TIBCO Object Service Broker names must consist of only the letters described in Character Set on page 19.

Behavior of National Character Sets

The following behavior applies to the use of national character sets:

- The characters are sorted in EBCDIC order.
- When printing, control characters are filtered out.
- Uppercase and lowercase characters are mapped to each other.

The use of a specific national character set does not affect the decimal separator character. This is set by the DECIMALSEPARATOR session parameter.

Specifying a National Character Set

If your TIBCO Object Service Broker system is not NLS-enabled, to use a specific character set specify one of the values shown in the following table in your user profile.

TIBCO Object Service Broker Name	Character Set	English Name [a]
CDNB	Canadian Bilingual	Canadian Bilingual
DANS	Dansk	Danish/Norwegian
DEUT	Deutsch	Austrian/German

TIBCO Object Service Broker Name	Character Set	English Name [a]
ENGB	English	English (Great Britain)
ENGL	English	English (US)
ESPA	Español	Spanish
FRAN	Français	French
ITAL	Italiano	Italian
NORS	Norsk	Danish/Norwegian
PORT	Portugues	Portuguese
SCHW	Schweiz	Swiss/French and Swiss/German
SUOM	Suomi	Finnish/Swedish
SVEN	Svenska	Finnish/Swedish

a. You can use the names listed in this column to find more information about the character set in the appropriate IBM manual.

See Also

TIBCO Object Service Broker National Language Support about NLS-enabled operating environments.

TIBCO Object Service Broker Parameters about session parameters.

TIBCO Object Service Broker Managing Security about user profiles.

Double-byte Character String Support

Allowable Usage

If your hardware is designed and configured to handle double-byte characters and you are using the z/OS version of TIBCO Object Service Broker, you can use double-byte characters in TIBCO Object Service Broker as follows:

- In tables where a field has double-byte and single-byte character (W) syntax defined
- As arguments to a rule, as long as the arguments do not require a specific syntax other than syntax W

You cannot use double-byte characters as literals on report tables or screen tables. You can create syntax W protected fields and assign a value to the fields through your rules.

Behavior of Double-byte Characters

The following behavior applies to the use of double-byte character sets:

- When used in an assignment statement, the double-byte character is converted to its single-byte equivalent. Usual assignment rules apply.
- The characters are sorted in EBCDIC order and comparison is based on the EBCDIC collating sequence.
- When printing to a double-byte character string device, the control characters SI, SO (shift in, shift out) are not printed. When printing to a device that does not support a double-byte character string, the control characters are printed using a printable character.

Use of Quotation Marks

The type of data that you are processing determines whether you must use single quotation marks (' ') to delimit the data. In general, string literals require quotation marks and numeric literals do not. The following sections describe the usage of quotation marks and compound delimiters containing the quotation mark.

String Literals

If your processing operation uses a string literal, you must delimit the value with opening and closing single quotation marks, for example, when specifying a parameter value for a table, as in:

```
GET EMPLOYEES WHERE REGION='MIDWEST';
```

Placing Quotation Marks Within String Literals and Unicode Literals

To place a single quotation mark within a string literal or a Unicode literal, use two quotation marks. For example: `'I''M'` represents the value I'M.

Placing Slashes Within Unicode Literals

Because in Unicode literals the slash (/) is treated as an escape character to enable exact specification of a Unicode character (for example, `U'/20AC'` is the euro sign), you need a way to place a displaying slash within a Unicode literal. To do this, use two slashes. For example: `U'I//O'` represents the value I/O.

TIBCO Object Service Broker Names

TIBCO Object Service Broker objects, rules arguments, and local variables are not considered to be string literals and therefore are not enclosed in quotation marks. The only exception to this is when an object is being passed as an absolute value. For example:

```
INSERT EMPLOYEE_INFO WHERE SCREEN='NEW_EMPLOYEE';
```

Numeric Literals

Except for some special cases with primary keys described in Exceptions when Accessing Primary Key Fields on page 25, numeric literals do not require quotation marks. Since TIBCO Object Service Broker does not permit identifier names to begin with a digit, these literals can be recognized by their totally numeric composition.

Exceptions when Accessing Primary Key Fields

Because primary key values are not stored in a compressed state, when referring to a primary key with syntax C or V and with a value that has leading zeros, you must specify the value within quotation marks. For example, if a primary key field of MONTH is defined with syntax C or V and has a value of 01, a GET statement with `MONTH='01'` is correct but `MONTH=1` or `MONTH=01` is incorrect.

Chapter 4 Types of Action Statements

This chapter list the different categories of action statements.

Topics

Overview

What Types of Actions Can a Rule Perform?

When you write your rules, you can include statements that perform the following types of actions:

- Assignment
- Database synchronization
- Exceptions
- Looping
- Output
- Rules invocation
- Table access

These action statements are then executed in a defined order, as described in Coding Actions on page 13.

Additional Information

Topic	Refer to…
Syntax of action statements in Backus-Naur format.	Syntax of Rules on page 215.
Exception handling.	Chapter 6, Exception Handling, on page 59 and Chapter 7, The Exception Statements, on page 71.
Expressions and operators, many of which are used in the examples of this chapter.	Chapter 8, Using Expressions and Operators, on page 77 and Chapter 12, Arithmetic Processing, on page 133.

Categories of Action Statements

Assignment Statements

Use assignment statements to assign values to fields or to local variables. There are two kinds of assignment statements:

- Simple assignment
- Assignment-by-name

The rules language uses the equal sign (=) as the assignment operator. Refer to Assignment Operator on page 92, for additional information about assignment statements.

Database Synchronization Statements

Database synchronization statements can be used to synchronize the data in the database after changes are made to it. For more information about synchronization, refer to Establishing Synchronization Points on page 106. Refer to the following topics for a full description of each database synchronization statement and an example of its usage:

Statement	Go to…
COMMIT	COMMIT Statement on page 36.
ROLLBACK	ROLLBACK Statement on page 52.

Looping Statements

Looping statements allow repetitive processes to take place. Refer to the following topics for a full description of each looping statement and an example of its usage:

Statement	Go to…
UNTIL This is also an exception handler.	UNTIL Statement on page 74.
UNTIL…DISPLAY This is also an output statement and an exception handler.	UNTIL … DISPLAY Statement on page 76.

Statement	Go to…
FORALL This is also a table access statement.	FORALL Statement on page 42.

Output Statements

Output statements present data to a screen or print reports. Refer to the following topics for a full description of each output statement and an example of its usage:

Statement	Go to…
DISPLAY	DISPLAY Statement on page 39.
DISPLAY & TRANSFERCALL This is also a rules invocation statement.	DISPLAY & TRANSFERCALL Statement on page 40.
UNTIL…DISPLAY This is also a looping statement and an exception handler.	UNTIL … DISPLAY Statement on page 76.
PRINT	PRINT Statement on page 49.

Rules Invocation Statements

A rule can invoke another rule as a function through a logical or arithmetic expression, or as a procedure with a CALL statement. A functional rule returns a value that can be used as an operand in a rules statement. You can use a functional rule anywhere you can use an expression. A procedural rule is called from within another rule to perform actions that do not return a value. Refer to the following topics for a full description of each rules invocation statement and an example of its usage:

Statement	Go to…
CALL	CALL Statement on page 34.
EXECUTE	EXECUTE Statement on page 41.
RETURN	RETURN Statement on page 51.
SCHEDULE	SCHEDULE Statement on page 53.

Statement	Go to…
TRANSFERCALL	TRANSFERCALL Statement on page 55.
DISPLAY & TRANSFERCALL This is also an output statement.	DISPLAY & TRANSFERCALL Statement on page 40.

Table Access Statements

Rules use table access statements to retrieve and manipulate data in the tables. When a rule refers to a table, a table buffer is created, which serves as a workspace for the table. Refer to the following topics for a full description of each output statement and an example of its usage:

Statement	Go to…
DELETE	DELETE Statement on page 38.
FORALL	FORALL Statement on page 42.
GET	GET Statement on page 45.
INSERT	INSERT Statement on page 47.
ORDERED	ORDERED Clause on page 48.
REPLACE	REPLACE Statement on page 50.
WHERE clause	WHERE Clause on page 56.

Behavior of Rules when Using Table Access Statements

Using table access statements, rules do the following:

- Enter new information into the table by first placing the new data in the appropriate fields of a table buffer and then executing either a REPLACE or an INSERT statement.
- Retrieve information from the table and place the information in a table buffer with a GET or FORALL statement.
- Use the information in the table buffer to determine the occurrence to delete from the table using the DELETE statement.

Selecting Data

To qualify the selection of occurrences, use the WHERE clause. You can also use the WHERE clause to select the table instances of a parameterized table, or you can specify the table instance in parentheses () after the table name. For example:

```
FORALL EMPLOYEES WHERE REGION = 'MIDWEST':
```

is the same as

```
FORALL EMPLOYEES('MIDWEST'):
```

Chapter 5 The Action Statements

This chapter lists and describes the action statements.

Topics

CALL Statement

The CALL statement invokes a rule within the scope of the current transaction. Upon completion, control passes to the next action in the calling rule. Arguments in CALL statements are passed by an argument list. All arguments must be specified when a rule is called.

Usage Notes

- The called rule can be a rule in the installation library, the system library, or the local library that you use for your session when you execute the rule, or it can be a shareable tool. Refer to Chapter 13, Using Rules Libraries, on page 141 for more information about rules libraries. Refer to Chapter 16, Processing in Standard Execution Mode, on page 175 for more information about rules execution.
- You can invoke the rule directly or indirectly. Refer to Indirect Referencing on page 95 for additional information about indirect referencing.
- You cannot trap for the following: a non-existent rule, whether the rule name is not semantic data type I (for identifier) or typeless, or an incorrect number of arguments.

Exceptions

ROUTINEFAIL	Signaled if the called routine fails and the cause of the error cannot be signaled by other system exceptions
RULEFAIL	Signaled if a table access resulting from the call is incorrect

Examples

```
1. CALL EMPLOYEE_COUNT(10);
2. CALL ENDMSG(COUNT || ' EMPLOYEES IN DEPARTMENT#' || DEPT);
3. CALL FCNKEYS.ROUTINE;
```

About the Examples

- Example 1 invokes a rule that contains one argument; the argument value is within the parentheses.
- Example 2 invokes the shareable tool ENDMSG, which contains one argument, *message*. The value for *message* is a concatenation of the value for the local variable COUNT, the text string ‘EMPLOYEES IN DEPARTMENT#’, and the value for the argument DEPT passed in from a higher-level calling rule.

- Example 3 indirectly invokes a rule. In this example the value of the field ROUTINE, in the table FCNKEYS, is the name of the rule to be called. For more information, refer to Indirect Referencing on page 95.

See Also *TIBCO Object Service Broker Shareable Tools* about the use of shareable tools.

COMMIT Statement

The COMMIT statement applies all changes made to TDS and external data since the last synchronization point. You do not need to explicitly issue a COMMIT at the end of a transaction. Data is implicitly committed for you at the end of the transaction.

Usage Notes

- The FORALL statement operates only on data committed to the database.
- A GET statement that does not specify a primary key value operates only on committed data.
- You can use the COMMIT statement before the commit limit is reached or when the exception ON COMMITLIMIT is signaled.
- The COMMIT statement does not release locks. The locks are released on the affected table at transaction end.
- When issuing a COMMIT, you can get an unusual and untrappable SYNC error resulting from the COMMIT updating too many page buffers in the Data Object Broker. The number of page buffers is determined by the WORKINGSET Data Object Broker parameter.

To avoid SYNC errors against updates to a parameterized table, issue a COMMIT at the end of the updates to each instance of the table.

Exceptions

COMMITLIMIT	Signaled if the maximum number of updates (INSERTs, REPLACEs, or DELETEs) between synchronization points is reached

Examples

```
1. INSERT EMPLOYEES WHERE REGION = 'MIDWEST';
   COMMIT;
   FORALL EMPLOYEES WHERE REGION = 'MIDWEST':
2. FORALL $EMPLOYEES:
      FORALL EMPLOYEES($EMPLOYEES.REGION):
      EMPLOYEES.SALARY = EMPLOYEES.SALARY + 100;
      REPLACE EMPLOYEES($EMPLOYEES.REGION);
      END;
      COMMIT;
   END;
3. INSERT EMPLOYEES('MIDWEST');
   _____________________________________
   ON COMMITLIMIT:
   COMMIT;
```

About the Examples

- In example 1, the COMMIT statement commits the insertion to the EMPLOYEES table so that the inserted occurrence can be retrieved in the FORALL.
- In example 2, a commit is made at the end of the updates to each instance of the EMPLOYEES table.
- In example 3, a COMMIT is issued when the exception COMMITLIMIT is signaled.

See Also *TIBCO Object Service Broker Parameters* about the Data Object Broker parameters.

DELETE Statement

The DELETE statement removes an occurrence from a table in the database. This statement requires the primary key to be available in one of the following ways:

- Via a previous GET or FORALL on the table
- By explicit selection of the primary key
- By assigning a value to the primary key

Field Selection

You can specify which occurrence to remove by basing your selection criteria on the primary key field and using the equality relational operator (=). No other field selection is allowed.

Retrieving Uncommitted Data

If you delete an occurrence using a DELETE statement, you must commit the update to the table before you can retrieve the updated data. Otherwise, previously committed data that does not reflect your update is retrieved from the database. Refer to COMMIT Statement on page 36 for more information about committing data.

Exceptions

DATAREFERENCE	Signaled if an error is detected in the specification of the selection criteria
DELETEFAIL	Signaled if the occurrence does not exist in the table
INTEGRITYFAIL	Signaled if an attempt is made to violate data integrity

Exceptions lower in the hierarchy of the INTEGRITYFAIL group can also be signaled.

Examples

```
1. DELETE STUDENT WHERE ID = '810883';
2. GET STUDENT;
   DELETE STUDENT;
```

About the Examples

- If a specific primary key value is selected using a WHERE clause, as shown in example 1, the selected occurrence is deleted.
- If a GET or FORALL is performed without selection before the DELETE, as shown in example 2, the first occurrence in the table is deleted.

DISPLAY Statement

The DISPLAY statement shows a specified screen. You can access data that is entered on a screen using GET and FORALL statements on the screen tables of the screen. The DISPLAY statement can also be combined with the TRANSFERCALL statement and the UNTIL statement. Refer to the DISPLAY & TRANSFERCALL Statement on page 40 and UNTIL … DISPLAY Statement on page 76 for more information.

Usage Notes

To insert data from an application into a screen table, use the INSERT statement before the DISPLAY statement.

Exceptions

DISPLAYFAIL	Signaled if an error is detected during an attempt to display a screen
DEFINITIONFAIL	Signaled if the screen does not exist

Example

```
FORALL MANAGERS:
  MANAGER_DATA.* = MANAGERS.*;
  INSERT MANAGER_DATA('MANAGER_SCR');
  END;
DISPLAY MANAGER_SCR;
```

About the Example

In this example, data is retrieved from the MANAGERS table, assigned and then inserted into the screen fields of a screen table, and then the screen appears.

See Also

TIBCO Object Service Broker Defining Screens and Menus about defining and presenting TIBCO Object Service Broker screens.

DISPLAY & TRANSFERCALL Statement

In a text application environment, you can use the DISPLAY & TRANSFERCALL statement to improve concurrent access to the resources required by an application.

You could use this statement if:

- You are creating an update application to be used by more than one user at a time.
- You are accessing occurrences that could be required by other users.

Usage Notes

- The DISPLAY & TRANSFERCALL statement allows pseudo-conversational processing. This processing controls resource utilization only within TIBCO Object Service Broker. You can display a screen after ending the transaction, which frees the tables that the application uses but still gives you access to the information that is input to the screen.
- Your screen environment is preserved across the transaction boundary. Anything associated with the screen, such as data occurrences, attributes set by screen tools, or cursor positioning, persists between displays unless you change the data or attributes.
- You can specify whether the rule runs in browse or update mode. For more information, refer to Setting the Mode of the Transaction on page 111.

Exceptions

DISPLAYFAIL	Signaled if an error is detected during an attempt to display a screen
DEFINITIONFAIL	Signaled if the screen does not exist

Example

```
DISPLAY QUERY_SCREEN & TRANSFERCALL PROCESS_QUERY;
```

About the Example

In the example, QUERY_SCREEN is the screen that appears and PROCESS_QUERY is the rule that can access input from QUERY_SCREEN. When you press a function key or the Enter key, PROCESS_QUERY begins as a new transaction.

See Also

TIBCO Object Service Broker Defining Screens and Menus about defining and presenting TIBCO Object Service Broker screens.

EXECUTE Statement

The EXECUTE statement starts a new transaction to invoke a rule. Upon completion, control passes to the next action in the original transaction. Like the CALL statement, the EXECUTE statement can invoke a rule directly or indirectly.

Usage Notes

- The executed rule can be a rule in the installation library, system library, or the local library you are using for your session when you execute it, or it can be a shareable tool. Refer to Chapter 13, Using Rules Libraries, on page 141 for more information about rules libraries. Refer to Chapter 16, Processing in Standard Execution Mode, on page 175 for more information about rules execution.
- You can invoke the rule directly or indirectly. Refer to Indirect Referencing on page 95 for additional information.
- You can nest transactions using the EXECUTE statement.
- You can specify whether the transaction runs in browse or update mode.
- For additional information about this statement and transactions in particular, refer to Chapter 9, Transaction Processing, on page 103.

Exceptions

EXECUTEFAIL	Signaled if an error is detected in the child transaction

Examples

```
1. EXECUTE EMPLOYEE_COUNT(10);
2. EXECUTE FCNKEYS.ROUTINE;
```

About the Examples

- Example 1 directly invokes the EXECUTE EMPLOYEE_COUNT rule and passes in the argument value 10.
- Example 2 indirectly invokes the rule whose name is the value of the field ROUTINE of the table FCNKEYS.

FORALL Statement

The FORALL statement is a looping construct that processes a set of occurrences retrieved from the database. The body of the loop consists of the statements to be executed for each occurrence that satisfies the selection criteria. FORALL statements can be nested.

Usage of FORALL

A FORALL statement contains:

1. A table name, an optional WHERE clause, optional ORDERED clauses, an optional UNTIL clause, and actions.
2. A colon (:), which follows the optional clauses (or the table name, if there are no clauses).
3. Actions, which comprise the body of the loop and follow the colon. Each action starts on a separate line.

   Action sequence numbers are not permitted within a FORALL loop; since a FORALL loop constitutes a single statement, all actions within it are executed whenever a FORALL is executed.
4. An END statement on a separate line that marks the end of the FORALL.

Usage Notes

- If you make updates to a table, you must commit the updates to the database before you can retrieve them with a FORALL statement. Refer to the COMMIT Statement on page 36 for details about committing data.
- You cannot issue a FORALL within a FORALL on the same table.

Selection within a FORALL

As with all table access statements, parameters and selection on fields are specified in a WHERE clause, as shown in the second example in Examples on page 43.

Ordering in Selection

When a FORALL statement is executed, table occurrences are selected in the order in which they are stored, unless a different order is specified by one of the following:

- One or more ORDERED clauses
- The ORD field in the table definition

The following shows an example of the ORDERED clause. In this example, the occurrences are ordered by descending values of the field PRICE, then by ascending values of the field MODEL, and then by ascending values of the primary key LICENSE# (the default for ordering is ascending).

```
FORALL CARS WHERE CITY = INVOICE.CITY ORDERED DESCENDING
 PRICE & ORDERED ASCENDING MODEL :
 CALL $PRINTLINE('CAR ID ' || CARS.LICENSE# ||
 ' MODEL ' || CARS.MODEL || ' RETAIL PRICE:$' ||
 CARS.PRICE);
END;
```

Exception Handling

- No exceptions are raised if occurrences are not selected by the FORALL statement. The actions in the body of the FORALL statement are not executed and processing continues for statements following the END statement.
- A FORALL loop cannot contain a SIGNAL statement. For information about SIGNAL statements, refer to SIGNAL Statement on page 73.
- Any exception in the INTEGRITYFAIL group, except for COMMITLIMIT, can be signaled.
- The DATAREFERENCE exception is signaled if an error is detected in the specification of the selection criteria.

Refer to Chapter 6, Exception Handling, on page 59 for a description of how rules handle exceptions.

Termination of a FORALL Statement

FORALL statement execution terminates under either of these circumstances:

- All occurrences satisfying the FORALL selection criteria were processed.
- An exception is detected (and not handled by rules inside the FORALL loop) during the execution of the statements comprising the loop.

The table buffer is undefined after all occurrences satisfying the FORALL selection criteria are processed. Accessing CARS.MODEL after the FORALL statement in Selection within a FORALL on page 42 would not provide the model of the last car but would raise the UNASSIGNED exception.

Examples

```
1. FORALL MANAGER:
      CALL PRINT_MANAGER;
   END;
2. FORALL EMPLOYEES WHERE REGION = 'MIDWEST'
      & HIREDATE > *.BIRTHDATE + 40 :
      ....
   END;
3. FORALL MANAGER WHERE MANAGER_NAME LIKE '*SON' :
      ....
   END;
4. FORALL MANAGER UNTIL GETFAIL:
      ....
   END;
```

About the Examples

- Example 1 retrieves all occurrences in the MANAGER table and calls PRINT_MANAGER to print the occurrences.
- Example 2 retrieves all occurrences in the MIDWEST table instance of the EMPLOYEES table where the value of the HIREDATE field is greater than the value of the BIRTHDATE field plus 40. The asterisk (*) represents the current table.
- Example 3 retrieves all occurrences in the MANAGER table where the MANAGER_NAME field ends in 'SON'. To get all the manager names that do not end in 'SON', write the FORALL statement like this:

  ```
  FORALL MANAGER WHERE ¬ (MANAGER_NAME LIKE '*SON') :
  ```

- Example 4 retrieves all occurrences in the MANAGER table until the GETFAIL exception is signaled. Refer to UNTIL Statement on page 74 for a description of how an UNTIL clause can be handled.

See Also *TIBCO Object Service Broker Managing Data* about TIBCO Object Service Broker tables.

GET Statement

The GET statement retrieves the first occurrence in a table satisfying the specified selection criteria. You can specify the retrieval order of the occurrences by using the ORDERED clause.

Retrieving Uncommitted or Committed Data

If you retrieve an occurrence that you updated in the same transaction by specifying a unique primary key value in a GET statement, the uncommitted data is retrieved. Otherwise, previously committed data that does not reflect your update is retrieved from the database. Refer to Database Synchronization Statements on page 29 for information about committing data.

Locking Considerations

If a GET statement specifies a row that is unique by primary key, then a share lock is taken upon that row.

If a GET is not unique by primary key, then a share lock is taken on the table (or table instance in the case of a parameterized table).

If the keywords WITH MINLOCK appear at the end of a GET statement, and either the GET is ordered by anything but the primary key, or the GET includes selection that is not unique by primary key, then a share lock will be taken on the table (or table instance in the case of a parameterized table) only during GET processing. Once the row to be returned has been determined, the lock will be reduced to a share lock on only that table row.

Exceptions

DATAREFERENCE	Signaled if an error is detected in the specification of selection criteria
GETFAIL	Signaled if there are no occurrences that meet the selection criteria
INTEGRITYFAIL	Signaled if an attempt is made to violate data integrity

Any exception lower in the hierarchy of the INTEGRITYFAIL group can also be signaled.

Examples

1. GET STUDENTS;
2. GET STUDENTS WHERE STUDENT# = '810883';
3. GET MONTHS WHERE MONTH = MM & DAYS >= DD;

```
4. GET EMPLOYEES WHERE REGION='MIDWEST' & DEPTNO =
      INPUT.DEPT ORDERED
   DESCENDING LNAME;
```

About the Examples

- Example 1 retrieves the first occurrence in the STUDENTS table.
- Example 2 retrieves the first occurrence in the STUDENTS table, where the value of the STUDENT# field is equal to 810883.
- Example 3 retrieves the first occurrence in the MONTHS table where the value of the MONTH field is equal to MM and the DAYS field has a value greater than or equal to DD. MM and DD are local variables that are assigned a value in the parent rule.
- Example 4 orders the occurrences in the MIDWEST instance of the EMPLOYEES table in descending order according to the values in the LNAME field, and then retrieves the first occurrence whose DEPTNO equals the value of the field DEPT of the INPUT table.

See Also *TIBCO Object Service Broker Managing Data* about TIBCO Object Service Broker tables.

INSERT Statement

The INSERT statement adds a new occurrence to a table. Occurrences within a table must have unique primary keys. No field selection is possible; the WHERE clause can specify only parameter values.

Exceptions

DATAREFERENCE	Signaled if an error is detected in the specification of selection criteria
INSERTFAIL	Signaled if an attempt is made to insert an occurrence with a primary key that already exists
INTEGRITYFAIL	Signaled if an attempt is made to violate data integrity

Any exception lower in the hierarchy of the INTEGRITYFAIL group can also be signaled.

Examples

```
1. INSERT STUDENT;
2. INSERT CARS WHERE CITY = INPUT.CITY;
3. INSERT EXPENSE_DATA WHERE SCREEN ='EMPLOYEE_EXPENSE';
```

About the Examples

- Example 1 inserts data into the STUDENT table.
- Example 2 inserts data into the parameterized CARS table. The values for the parameter CITY are provided by INPUT.CITY.
- Example 3 inserts data into the EXPENSE_DATA screen table in the EMPLOYEE_EXPENSE screen.

ORDERED Clause

An ORDERED clause can be used in conjunction with a WHERE clause to retrieve occurrences in a specific order using the operators ASCENDING or DESCENDING.

Default Order for Tables

By default a table is ordered in ascending order by primary key, although this can be changed in the table definition.

Usage Notes

- If an ASCENDING or DESCENDING operator is not specific, the order returned is ascending.
- Multiple ORDERED clauses can be used in one access statement. Use the AND (&) operator to join the clauses.
- If seven or more ORDERED clauses are used in a table access statement, the external sort program setup for your TIBCO Object Service Broker installation is used to sort the values. In this case, the total number of bytes occupied by all control fields must not exceed 4088 bytes.

Examples

```
1. GET EMPLOYEES('MIDWEST') & DEPTNO = INPUT.DEPT ORDERED
   DESCENDING LNAME;
2. FORALL CARS WHERE CITY = INVOICE.CITY ORDERED DESCENDING
      PRICE & ORDERED ASCENDING MODEL :
```

About the Examples

- Example 1 orders the occurrences of the MIDWEST instance of the EMPLOYEES table in descending order of LNAME field, and then retrieves the first occurrence whose DEPTNO equals the value of the DEPT field of the INPUT table.
- Example 2 retrieves the occurrences in descending values of the field PRICE, then by ascending values of the field MODEL, and then by ascending values of the primary key (the default for ordering is ascending).

See Also *TIBCO Object Service Broker Managing Data* about TIBCO Object Service Broker tables.

PRINT Statement

The PRINT statement is used to send a report to a predetermined destination: the message log, printer or file. Your session options determine the destination; you can override the print destination options by a call to the $SETRPTMEDIUM tool. The PRINT statement can print a report directly by referring to the report name, indirectly by referring to a field of a table, or to an argument name that represents the report name.

Exceptions

DEFINITIONFAIL	Signaled if the report does not exist

Examples

```
1. PRINT DEPT_SALARY;
2. PRINT(REPORTNAME);
```

About the Examples

- Example 1 prints the report DEPT_SALARY directly.
- Example 2 prints the report whose name is the value of the argument for a rule.

See Also

TIBCO Object Service Broker Defining Reports about defining and producing reports.

TIBCO Object Service Broker Shareable Tools about shareable tools.

REPLACE Statement

The REPLACE statement updates an occurrence in the database. You should first retrieve the data that you want to modify using a GET or FORALL statement.

Usage Notes

- To alter the primary key value of an occurrence, you must DELETE the old occurrence and INSERT the new one.
- No field selection is possible; the WHERE clause can specify only parameter values.

Exceptions

DATAREFERENCE	Signaled if an error is detected in the specification of selection criteria
INTEGRITYFAIL	Signaled if an attempt is made to violate data integrity Any exception lower in the hierarchy of the INTEGRITYFAIL group can also be signaled.
REPLACEFAIL	Signaled if the primary key does not exist or if an attempt is being made to update a table from a rule running in browse mode

Examples

```
1. REPLACE STUDENTS;
2. REPLACE CARS(INPUT.CITY);
```

About the Examples

- Example 1 replaces data in the STUDENTS table.
- Example 2 replaces data in the parameterized table CARS; it is parameterized by CITY. The values for the CITY parameter are provided by INPUT.CITY.

RETURN Statement

The RETURN statement returns a specified result. The result can be an identifier or a constant. A rule that contains a RETURN statement is a function.

Exceptions None

Examples

```
1. RETURN('Y');
2. RETURN(EMPLOYEE_SALARY.SALARY + INCREASE);
```

About the Examples

- Example 1 sets the return value to Y.
- Example 2 returns the salary of the employee plus an increase.

ROLLBACK Statement

The ROLLBACK statement discards all changes made to TDS and external data since the last synchronization point. Locks are not released on the affected tables until the transaction ends.

Exceptions None

Example

```
FORALL HIREDATE:
GET EMPLOYEES('MIDWEST') WHERE EMPNO=HIREDATE.EMPNO;
  EMPLOYEES.HIREDATE = HIREDATE.DATE;
  REPLACE EMPLOYEES('MIDWEST');
  END;
-----------------------------------------------------------
ON GETFAIL:
  ROLLBACK;
```

About the Example

In this example, a ROLLBACK statement is issued so that none of the changes made to the database since the last synchronization point are applied.

SCHEDULE Statement

The SCHEDULE statement enables asynchronous processing to take place. The SCHEDULE statement uses an instance of the @SCHEDULEMODEL table to run a named rule using z/OS JCL, Windows batch programs, or Solaris or scripts.

Usage Notes

- You can specify whether the transaction runs in browse or update mode. For more information, refer to Setting the Mode of the Transaction on page 111.
- If you are using TIBCO Object Service Broker for z/OS, you can use the TO clause to schedule a rule to be sent to a queue for later processing.
- By not including the TO clause, you can submit an instance of the @SCHEDULEMODEL table for immediate processing by the operating system.
- Refer to Chapter 17, Processing Asynchronously in Batch Mode, on page 191 for information about the @SCHEDULEMODEL table and batch processing.
- For the Open Systems platforms, the SCHEDULE statement automatically wraps, within single quotation marks, values that contain spaces or that have a length of zero. The substitution value for a zero-length string continues to be two single quotation marks.

Exceptions

LOCKFAIL	Signaled if a lock is held on the @SCHEDULEMODEL table (z/OS only)
SECURITYFAIL	Signaled if there is a security violation on the @SCHEDULEMODEL table (z/OS only)

Examples

```
1. SCHEDULE PRINT_INVOICE(INPUT.INVOICE#);
2. SCHEDULE TO 'WEEKEND' CLEANUP WHERE LOCATION = INPUT.CITY;
```

About the Examples

- Example 1 submits the PRINT_INVOICE rule immediately for batch processing. The value for its one argument is provided by the INVOICE# field of the INPUT table.
- Example 2 send the CLEANUP rule to a queue called WEEKEND. This rule has one argument called LOCATION, and its value is provided by the CITY field of the INPUT table.

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about the use of queues.

TIBCO Object Service Broker for Open Systems Installing and Operating about post installation steps on Solaris for the SCHEDULE statement.

TRANSFERCALL Statement

The TRANSFERCALL statement terminates the current transaction and starts a new one. When the called rule is finished executing, the transaction is complete. If the transaction is called from a nested transaction, control passes to the next action in the parent transaction.

Usage Notes

- You can specify whether the rule runs in browse or update mode. For more information refer to Setting the Mode of the Transaction on page 111.
- The TRANSFERCALL statement can invoke a rule directly by referring to the rule name, or indirectly by referring to a field of a table, or to an argument name that represents the rule name.
- Refer to Chapter 9, Transaction Processing, on page 103 for additional information about TRANSFERCALL and transactions.

Exceptions

None

Examples

```
1. TRANSFERCALL SELECT_RENTAL(NEAREST_CAR(LOCATION));
2. TRANSFERCALL IN BROWSE SELECT_RENTAL WHERE
      RENTAL_LOC = NEAREST_CAR(LOCATION);
3. TRANSFERCALL FCNKEYS.ROUTINE;
```

About the Examples

- Example 1 invokes the SELECT_RENTAL rule directly.
- Example 2 invokes the SELECT_RENTAL rule in browse mode.
- Example 3 invokes the rule whose name is the value of the field ROUTINE of the table FCNKEYS.

WHERE Clause

The WHERE clause is used to select:

- The occurrences of a field
- The table instance of a parameterized table (optional method)
- The arguments of a rule (optional method)

You can use it in conjunction with the ORDERED clause.

Usage Notes

- You can select the occurrences of a field that match a pattern using the pattern match operator LIKE. The pattern can contain question marks (?) representing one character of any kind, asterisks (*) representing zero or more unspecified characters, and any other character that must be present in the field.
- You can refer to the current occurrence of the current table with an asterisk (*).
- You can use comparisons with relational and logical operators joined together by the AND (&) or OR (|) operators. The operators are listed in Chapter 3, Supported Characters, on page 17. The number of conditions that you can code in a WHERE clause is dependent on a number of factors. For example:
  - The size of the data in the right hand side of any conditions
  - The lengths of the fields on any condition
  - Whether the fields being used in the conditions are key fields or not

Examples

```
1. GET EMPLOYEES WHERE REGION ='MIDWEST' & ¬
   (STATE_PROV='ONT'| STATE_PROV='MINN');
2. DELETE MANAGER WHERE MANAGER_NUM = 80002;
3. DELETE EMPLOYEES WHERE REGION = 'MIDWEST';
4. GET MANAGER WHERE MANAGER_NUM = *.USERID;
```

About the Examples

- Example 1 gets the first occurrence from the EMPLOYEES table where the parameter equals MIDWEST and the STATE_PROV field does not equal ONT or MINN.
- Example 2 shows how you can also specify selection criteria for a DELETE statement. The selection criteria must consist of an equality between a value and the primary key. In the example, MANAGER_NUM is the primary key for the MANAGER table.

- Example 3 shows how you can use the WHERE clause with any of the table access statements to select table instances. In the example, REGION is the parameter name and MIDWEST is the parameter value.
- Example 4 shows how you can use the asterisk (*) symbol to retrieve an occurrence that has the same value in two fields of the same table.

See Also *TIBCO Object Service Broker Managing Data* about TIBCO Object Service Broker tables.

Chapter 6 Exception Handling

This chapter describes exception handling.

Topics

Functionality of Exception Handling

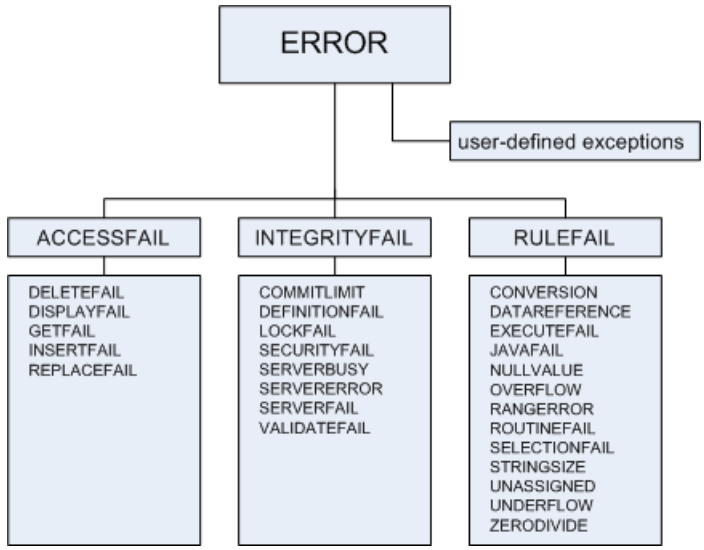
What is an Exception and How is It Handled?

An exception is an indication that an exceptional or unusual circumstance has arisen during rules processing, for example if you have inadequate security on a table when attempting to access it. When an exception is encountered during rules processing, a signal can be raised by TIBCO Object Service Broker or by a user-written rule to change the flow of a program. If TIBCO Object Service Broker handles the exception, a system exception is issued. If the user-written rule handles the exception, either a system exception or a user-defined exception can be issued.

Hierarchy of System Exceptions

As previously stated, the TIBCO Object Service Broker runtime environment signals system exceptions to permit an application to recover from an error. System exceptions form a hierarchy of names as shown in the following illustration.

Three levels of system exceptions are defined, and an exception traps any of the exception names that are below it in the hierarchy. The ERROR exception is at the highest level and traps all detectable errors that can be handled by an exception handler, whereas an exception such as GETFAIL traps only an error that occurs when a table occurrence is not found on the execution of a GET statement.



Exceptions Raised by Exception Handler Statements

In some cases, statements comprising an exception handler could raise the same exception. In these cases, if the second exception is not handled somewhere lower in the calling hierarchy, the currently executing handler does not handle the second exception. The rules executor detects a possible infinite loop condition and aborts the transaction.

Examples of Untrappable Exceptions

The following are examples of untrappable exceptions:

- A rule called from a library that is not specified in the user's search path
- A rule called that does not exist
- A rule called with the wrong number of arguments
- Memory or storage limits exceeded

System Exceptions and Their Conditions

System Exceptions

ACCESSFAIL	A table access error is detected or a rule running in browse mode is attempting to update a table.
COMMITLIMIT	The limit on the number of updates between synchronization points has been reached.
CONVERSION	A value has invalid syntax or cannot be converted to the target syntax.
DATAREFERENCE	An error is detected in selection criteria.
DEFINITIONFAIL	An error is detected in the definition of a table.
DELETEFAIL	The primary key for a DELETE statement does not exist or a rule running in browse mode is attempting to update a table.
DISPLAYFAIL	An error is detected when trying to display a screen.
ERROR	An error is detected.
EXECUTEFAIL	An error is detected in the child transaction.
GETFAIL	No occurrence satisfies the selection criteria.
INSERTFAIL	The primary key provided for an INSERT statement already exists or a rule running in browse mode is attempting to update a table.
INTEGRITYFAIL	Attempt to violate data integrity detected.
JAVAFAIL	A Java exception is raised by a called Java external routine.
LOCKFAIL	Another transaction is accessing this data in a way that prevents you from accessing the data.
NULLVALUE	An arithmetic operator is being applied to a numeric null or a numeric null is being passed as an argument value when a numeric value is required.

OVERFLOW	A value is too large to be assigned to the target syntax. The maximum value for the syntax is inserted into the receiving field. All tables are limited to the defined dictionary length. As well, screen and report tables are also limited to the display length.
RANGERROR	An argument to a given TIBCO Object Service Broker routine is out of the allowable range.
REPLACEFAIL	A primary key provided for a REPLACE statement does not exist or a rule running in browse mode is attempting to update a table.
ROUTINEFAIL	A call to a TIBCO Object Service Broker routine cannot complete successfully and a more specific system exception cannot be signaled.
RULEFAIL	An error results from incorrect rules coding, given that the dictionary definition of the database is correct.
SECURITYFAIL	Permission for the requested action is denied.
SELECTIONFAIL	This exception traps failures during the conversion process of data in a character field to a numeric syntax or date so as to enable comparison with that in a WHERE clause.
SERVERBUSY	The requested external database server is not available to process the transaction.
SERVERERROR	External database server error detected.
SERVERFAIL	The connection to an external database server broke during a transaction or the external database server failed.
STRINGSIZE	The receiving string field is too short to contain the full length of the string value being assigned to it. The value is truncated to the length of the receiving field and inserted into that field.
UNASSIGNED	Reference is being made to a field of a table not assigned a value.
UNDERFLOW	A value is too small to be represented in the target syntax (usually exponent errors). The minimum value for the syntax is inserted into the receiving field.

VALIDATEFAIL	An attempt is being made to update a screen or table with invalid data. For example: • An attempt is being made to insert data into a table that failed a reference check or a non-Y value is being returned from a validation rule. • Invalid data is being inserted into a screen table from a rule (that is, the data failed the screen table reference check). • Invalid data existed on the screen when the user left the screen by using the Validation Exit key.
ZERODIVIDE	Attempt to divide by zero detected.

See Also

TIBCO Object Service Broker Defining Screens and Menus about TIBCO Object Service Broker screens.

TIBCO Object Service Broker Defining Reports about TIBCO Object Service Broker reports

TIBCO Service Gateway manuals about accessing external databases.

Coding of Exception Handlers

Signaling Exceptions

You use the SIGNAL statement from within the body of your rule or within the exception handler segment to issue a user-defined exception. This statement is described in SIGNAL Statement on page 73.

You must explicitly handle user-defined exceptions. System exceptions are signaled for you. You can handle system exceptions explicitly or TIBCO Object Service Broker can handle them on your behalf.

Handling Exceptions

When coding your rules, you can use:

- UNTIL, UNTIL … DISPLAY, or FORALL statements from within the body of your rule to handle one or more exceptions
- ON statements in the exception handler segment of your rule to handle individual exceptions

These statements are known as exception handlers. Except for FORALL … UNTIL, which is described as part of the FORALL Statement on page 42, they are described in Chapter 7, The Exception Statements, on page 71.

Maximum Number of Exceptions

You can handle up to 32 exceptions in a rule, using the ON statements, and UNTIL statements, including UNTIL … DISPLAY and FORALL … UNTIL.

Sample Rule With Exception Handling

The following sample rule signals and handles the user-defined exception DATE_INVALID and explicitly handles the system exception GETFAIL.

```
   RULE EDITOR ===>                                                SCROLL: P
MGRINFO(DATE,ID);
_
_ ------------------------------------------------------------------------
_ VALID_DATE(DATE);                                          |  Y N
_ ------------------------------------------------------------+-----------
_ GET MANAGER WHERE MANAGER_NUM = ID;                        |  1
_ SIGNAL DATE_INVALID;                                       |    1
_ ------------------------------------------------------------------------
  ON GETFAIL MANAGER :
     CALL ENTER_MANAGER;
  ON DATE_INVALID :
     CALL ENDMSG('Invalid date entered.');




PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Scope of Exception Handlers

Scope of An Exception Handler

An exception handler is in effect only during the execution of the actions of the rule in which it occurs. It traps exceptions generated both in the rule and in any of its descendant rules—rules that are below it in the calling hierarchy. If an exception is not trapped within the calling hierarchy, the transaction terminates with an error condition and the message log shows that the exception was signaled.

Scope of Multiple Exception Handlers

If an exception handler in two or more rules at different levels in the calling hierarchy can handle the same exception, the handler in the lowest rule at or above the point where the exception is raised handles the exception.

If more than one handler within a rule can handle an exception, the most specific handler is executed. For example:

- If a rule has both a GETFAIL handler and an ACCESSFAIL handler and a GETFAIL exception occurs, the GETFAIL handler is invoked.
- If the rule has no GETFAIL handler but does have an ACCESSFAIL handler, the ACCESSFAIL handler is invoked.

Limiting the Scope of Data Access Exceptions

You can limit the scope of data access exception handlers by specifying a table name. Data access exception handlers are:

- ACCESSFAIL
- DEFINITIONFAIL
- DELETEFAIL
- GETFAIL
- INSERTFAIL
- LOCKFAIL
- REPLACEFAIL
- SECURITYFAIL
- SERVERBUSY

- SERVERERROR
- SERVERFAIL
- VALIDATEFAIL

If a table name is specified, the handler traps the corresponding exception only if it is detected while accessing that table. If no table is specified, the handler traps the exception regardless of which table is being accessed.

VALIDATEFAIL Exception for Screens

When is VALIDATEFAIL Issued?

VALIDATEFAIL is issued on a DISPLAY statement when a screen contains invalid data and a user uses the Validation Exit key. VALIDATEFAIL is also issued when data inserted into a screen table fails reference checking validation.

Handling of Invalid Data

A VALIDATEFAIL exception can manage an exception issued for invalid data, but the data occurrences containing invalid values are still present in the set of occurrences available to the rule. The invalid values can be read within the exception handler or a subsequent rule in the same transaction. If the data entered is completely incompatible with the field definitions (for example, alphabetic characters in numeric fields) the invalid values are changed to null.

Example Rule

An example of a VALIDATEFAIL exception accessing screen data follows. The rule displays a screen called NEW_EMPLOYEE for the user to add information about a new employee. If the user exits from the screen while a field has invalid data, the VALIDATEFAIL exception sends the data on the screen to an audit trail.

```
   RULE EDITOR ===>                                                SCROLL: P
EMPADD(EMPNO);
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ EMPLOYEE_INFO.EMPNO = EMPNO;                                | 1
_ INSERT EMPLOYEE_INFO('NEW_EMPLOYEE');                       | 2
_ UNTIL EXIT_DISPLAY DISPLAY NEW_EMPLOYEE :                   | 3
_    CALL PROCESS_PFKEY('NEW_EMPLOYEE');                      |
_    END;                                                     |
_ ------------------------------------------------------------------------------
_ ON VALIDATEFAIL :
_    GET EMPLOYEE_INFO('NEW_EMPLOYEE') WHERE EMPNO = EMPNO;
_    CALL ENDMSG('SEE AUDIT TRAIL FOR INVALID DATA.');
_    CALL @OPENDSN('AUDIT.TRAIL');
_    CALL @WRITEDSN(EMPLOYEE_INFO.EMPNO);
_    CALL @WRITEDSN(EMPLOYEE_INFO.LNAME);
_    CALL @WRITEDSN(EMPLOYEE_INFO.POSITION);
_    CALL @WRITEDSN(EMPLOYEE_INFO.HIREDATE);
_    CALL @WRITEDSN(EMPLOYEE_INFO.DEPTNO);
_    CALL @WRITEDSN(EMPLOYEE_INFO.SALARY);
_    CALL @CLOSEDSN;

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

See Also *TIBCO Object Service Broker Defining Screens and Menus* about TIBCO Object Service Broker screens.

Chapter 7 The Exception Statements

This chapter describes the exception statements.

Topics

ON Statement

The ON statement begins an exception handler. It includes the exception name and is coded in the exception handler portion of the rule. It can be followed by a sequence of actions that are executed if the exception is detected.

Usage of ON

The ON statement consists of:

1. The keyword ON
2. The name of either a user-defined or system exception
3. A colon (:)
4. Actions, if coded. Each action starts on a separate line.

Usage Notes

Action sequence numbers are not permitted within an ON statement or its following action statements.

Examples

```
1. ON GETFAIL MANAGER:
      CALL ENTER_MANAGER;
2. ON DEFINITIONFAIL:
      CALL ENDMSG ('The report does not exist.');
```

About the Examples

- Example 1 traps the exception GETFAIL on the MANAGER table. If a GETFAIL occurs on the MANAGER table, the ENTER_MANAGER rule is called. If a GETFAIL occurs on another table, an error message is issued.
- Example 2 traps the exception DEFINITIONFAIL. If a DEFINITIONFAIL occurs, the shareable tool ENDMSG is called and a message is returned to the screen.

SIGNAL Statement

The SIGNAL statement raises the exception specified within the statement. You use the SIGNAL statement to raise user-defined exceptions within your rules.

Usage of SIGNAL

A SIGNAL statement contains:

1. The keyword SIGNAL
2. The name of a user-defined exception
3. A semicolon (;)

Usage Notes

- An ON or UNTIL statement can detect an exception raised by the SIGNAL statement, and subsequently issue actions.
- When using the ON or UNTIL statements, you must provide the exception name given in the SIGNAL statement; you cannot use an indirect reference in the SIGNAL statement. To use an indirect reference, use the $SIGNAL tool.
- If you issue a SIGNAL statement from within a trigger or validation rule, the trigger or validation rule must explicitly handle the exception within its calling hierarchy. If it is not explicitly handled, the transaction that caused the trigger or validation terminates.
- You cannot issue a SIGNAL statement within a FORALL statement.

Examples

```
1. SIGNAL MISSING_INVOICE;
2. ON GETFAIL MANAGER :
      SIGNAL UNKNOWN_NAME;
```

About the Examples

- Example 1 signals the user-defined exception MISSING_INVOICE.
- In example 2, the GETFAIL exception signals the user-defined exception UNKNOWN_NAME if the GET fails on the MANAGER table.

UNTIL Statement

The UNTIL statement specifies an exception or a list of exceptions, each separated by the keyword OR. It allows looping to take place in rules execution. Looping terminates if an exception is detected. An END statement, on a separate line, marks the end of the UNTIL statement.

You can use the UNTIL statement on its own, or in conjunction with a FORALL statement and as part of the UNTIL … DISPLAY statement. For more information about the FORALL statement refer to FORALL Statement on page 42 and for the UNTIL … DISPLAY statement, refer to UNTIL … DISPLAY Statement on page 76.

Usage of UNTIL

An UNTIL statement contains:

1. The keyword UNTIL
2. An exception name, or a list of exceptions separated by the keyword OR
3. A colon (:)
4. The actions, which comprise the body of the loop. Each action starts on a separate line.

   Action sequence numbers are not permitted within an UNTIL loop; since an UNTIL loop constitutes a single statement, all the actions within it are executed whenever the UNTIL is executed.
5. An END statement, on a separate line

Usage Notes

If a loop terminates because of an exception, control passes to new actions as follows:

- If the exception is specified in an UNTIL statement for the loop, the actions executed next are those following the END statement of the loop (control passes to those actions even if there is an ON statement for that exception in the exception handler part of the rule). Upon completion of those actions, the rule is finished executing and control passes to the caller.
- If the exception is not handled by the UNTIL statement for the loop but is handled by an ON statement in the exception handler part of the rule, the actions executed next are those listed in the ON statement. Refer to ON Statement on page 72 for more information.
- If the exception is not specified in an UNTIL statement for the loop or in an ON statement in the exception handler part of the rule, either the exception is trapped by an exception handler in a rule higher in the calling hierarchy, or the transaction terminates with an error condition.

Examples

```
1. UNTIL NO_MORE_NUMBERS :
      MULTIPLIER = MULTIPLIER + 1;
      CALL CHECKDIGIT(LENGTH($NUMBER));
   END;
2. FORALL SOURCETAB1 UNTIL GETFAIL:
      GET SOURCETAB2 WHERE LINE_NUM = SOURCETAB1.LINE_NUM;
      CALL COMPARELINES(SOURCETAB1.TEXT, SOURCETAB2.TEXT);
   END;
```

About the Examples

- Example 1 performs the operations within the UNTIL up to the point the user-defined exception NO_MORE_NUMBERS is issued.
- Example 2 performs the FORALL loop operations up to the point the system defined exception GETFAIL is issued.

UNTIL … DISPLAY Statement

The UNTIL … DISPLAY statement, which is a looping construct with a display, displays a screen repetitively until an exception is encountered.

Usage of UNTIL … DISPLAY

An UNTIL … DISPLAY statement contains:

1. The keyword UNTIL
2. An exception name or a list of exceptions separated by the keyword OR
3. A DISPLAY statement followed by a screen name
4. A colon (:)
5. The actions, which comprise the body of the loop, follow the colon. Each action starts on a separate line.

   Action sequence numbers are not permitted within an UNTIL loop; since an UNTIL loop constitutes a single statement, all the actions within it are executed whenever the UNTIL is executed.
6. An END statement, on a separate line

Usage Notes

Execution of an UNTIL … DISPLAY statement terminates when an exception on the list is detected (and not handled by rules inside the UNTIL loop) during the execution of the statements comprising the loop. Control passes to the actions that follow the END statement of the loop.

Example

```
UNTIL DONE DISPLAY QUERY_SCREEN:
CALL PROCESS_FCNKEYS('QUERY_SCREEN');
  END;
```

About the Example

In this example, the QUERY_SCREEN screen appears up to the point the DONE exception is signaled during the display.

See Also

TIBCO Object Service Broker Defining Screens and Menus about defining and displaying TIBCO Object Service Broker screens.

Chapter 8 Using Expressions and Operators

This chapter describes how to use expressions and operators.

Topics

Overview

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

What Comprises an Expression?

An expression is any combination of values and operators (such as + or -), which results in a single value when evaluated. Conditions, actions, and exception handlers can contain expressions.

Valid Values for Expressions

The values of an expression can consist of a:

- Local variable
- Field of a table
- Rules argument name
- Function call
- Literal
- Indirect reference

Examples of Expressions

Expression	Evaluated Result or Assignment
3+2*3	9
10*(1+4)**2	250

Expression	Evaluated Result or Assignment
SUM + (TABLEREF).(FIELDREF) where SUM=123 TABLEDEF=EmpTable FIELDREF=Commission EmpTable.Commission=1700	1823

Operations That Can be Performed With Expressions

Expressions can be compared to each other using relational operators—these are called compound expressions. Compound expressions can be combined using logical operators.

The logical operators AND (&) OR (|) are allowed only within selection criteria (WHERE clause). The NOT (¬) operator is allowed within rules conditions and within selection criteria. Expressions can also be assigned to local variables or fields of tables with the assignment operator.

The Reserved Word NULL

The reserved word NULL is used to represent a null value. It can be used in many of the same places that an expression can be used. For more information about nulls, refer to Chapter 11, Null Processing, on page 123.

Syntax of Data Elements

Each data element of an expression has a syntax. The syntax of a value, local variable, or field describes how the data is stored. The syntax and length of a field are specified in the table definition. The syntax for a local variable is determined when a value is assigned to it. The syntax of a literal is determined by its content.

Valid Syntaxes

The following table describes the valid syntaxes, their permitted lengths, and additional information to help you in their use:

Syntax	Name	Valid Lengths (bytes)	Notes
B	Binary	2 – 12 [a]	Signed integer values. Supports display masks. A length of 2 supports values -32,768 to 32,767. A length of 4 supports values of -2,147,483,648 to 2,147,483,647. A length of 12 supports values of -39,614,081,257,132,168,796,771,975,168 to 39,614,081,257,132,168,796,771,975,167
C	Fixed-length character string	1 to the maximum length for an occurrence - 2. [b] 1 – 127 (key field or parameter). [b] Maximum total length for up to 16 fields in a composite primary key: 127. Maximum total length for all data parameters: 240. 1 – 254 (display fields). [b]	Uppercase alphanumerics. Trailing blanks not significant.
F	Floating point	4, 8, and 16. 24, 56, and 112 binary digits of precision for the mantissa. Approximate range: 5.4 x 10 -79 to 7.2 x 1075. [c]	Not valid for key fields or parameters. Should not be used when exact comparisons or exact values are required. Does not support ordering.
P	Packed decimal	1 – 16 (holds 1 – 31 decimal digits). [a, d]	Supports display masks.

Syntax	Name	Valid Lengths (bytes)	Notes
RD	Raw data	5 to the maximum length for an occurrence- 2. [b]	Consists of a 4-byte length followed by data. Not valid for key fields. Not valid for parameters except for SUB tables with a source table of DB2 or SLK type. Does not support ordering. Does not support default values in the Table Definer. Not valid for screens or reports. Not valid for Service Gateway for Datacom, Service Gateway for IDMS/DB, and Service Gateway for IMS/DB.
UN	Unicode	2 to the maximum length for an occurrence - 2. [b] Must be even.	Consists of UTF16 characters. Not valid for key fields. Not valid for parameters except for SUB tables with a source table of DB2 or SLK type. Does not support ordering. Does not support default values in the Table Definer. Not valid for screens or reports. Not valid for Service Gateway for Datacom, Service Gateway for IDMS/DB, and Service Gateway for IMS/DB.
V	Variable-length character string	1 to the maximum length for an occurrence - 2.[b] 1 – 127 (key field). [b] Maximum total length for up to 16 fields in a composite primary key: 127. 1 – 254 (display fields). [b]	Uppercase and lowercase alphanumerics. Trailing blanks significant. Not valid for parameters except for SUB tables with a source table of DB2 or SLK type.
W	Double-byte or single-byte character string	4 to the maximum length for an occurrence - 2. [b, e]	Uppercase and lowercase alphanumerics. Trailing blanks significant. Not valid for parameters. Valid for z/OS only.

a. Table parameters are limited to binaries of length 2 and 4 and packed decimals of length 1 to 8.

b. The maximum length of a field is limited by the maximum length of an occurrence for a table type. Refer to Maximum Occurrence Length on page 82. The length calculation for a table occurrence includes all of the table fields and parameters and is 17 + sum(short field length + 1) + sum(long field length + 2) + sum(data parameter +1). A short field length is <= 127 and a long field length is > 127.

c. In Open Systems, if you need to get results from arithmetic on floating-point data that are identical to what you would obtain with pre-Release 4.0 TIBCO Object Service Broker, use the NOIBMFLOAT Execution Environment parameter. For more detail, refer to *TIBCO Object Service Broker Parameters*.

d. Byte length=half-byte for each decimal digit plus half-byte for sign. Round up to the nearest whole byte. Specify the number of digits after the decimal separator in the Table Definer.

e. A single display position can display a single-byte character; however, you need two display positions to display a double-byte character.

Maximum Occurrence Length

Table Type	Description	Maximum Occurrence Length in Bytes
ADA	TIBCO Object Service Broker ADABAS table.	31744
CLC	Calculation table.	3915
DAT	TIBCO Object Service Broker CA-Datacom table.	3915
DB2	TIBCO Object Service Broker DB2 table.	31744
EXP	Export table.	31744
IDM	TIBCO Object Service Broker CA-IDMS table.	31744
IMP	Import table.	31744
IMS	TIBCO Object Service Broker IMS table.	31744
MAP	MAP table.	31744
PRM	Parameter table.	3915

Table Type	Description	Maximum Occurrence Length in Bytes
RPT	Report table.	3915
SCR	Screen table.	3915
SES	Session table.	31744
SLK	TIBCO Object Service Broker SLK table.	31744
SUB	Subview table.	31744
TDS	Table data store table.	3915
TEM	Temporary table.	31744
VSM	TIBCO Object Service Broker VSAM table.	31744

See Also *TIBCO Object Service Broker Managing Data* about defining TIBCO Object Service Broker tables.

Semantic Data Types

Each data element of an expression has a semantic data type. The semantic data type describes how the expression element can be used. The semantic data type of a field is defined in the table definition and determines what operations and conversions can be performed on values of the field. Operators in the rules language are defined only for meaningful semantic data types. For example, negating a string or adding a number to an identifier are invalid operations (for example, adding 3 to a license plate number).

Valid Semantic Data Types

The following table describes the semantic data types, the valid syntax for each data type, and the operations that can be performed on each data type:

Semantic Data Type	Name	Description	Possible Operators	Valid Syntax
	Typeless	Literals defined in TIBCO Object Service Broker rules and fields defined without a semantic data type. Example: name = 'Analyst'.	Relational Concatenation Assignment Arithmetic Logical	B, C, F, P, RD, UN, V, W
C	Count	Only integer numbers (no fractions) used for integral count. Example: An inventory count (zero items or more items; there is never half an item).	Relational Concatenation Assignment Arithmetic	B, C, P, UN, V
D	Date	Used for dates; you must include at least a year. Valid dates range from 0000/01/01 to 9999/12/31 inclusive. Example: The hire date of an employee: Jan 05, 1998.	Relational Concatenation Assignment Arithmetic (addition and subtraction)	B
I	Identifier	Unique identifier Example: The employee identifier: A1082.	Relational Concatenation Assignment	B, C, P, UN, V

Semantic Data Type	Name	Description	Possible Operators	Valid Syntax
L	Logical	Information meeting Yes or No conditions. Example: Used as a flag: Is this employee current?	Relational Concatenation Assignment Logical	C
Q	Quantity	Real numbers used for measurement. Example: The salary of an employee: $950.57.	Relational Concatenation Assignment Arithmetic	B, C, F, P, UN, V
S	String	Character string data. Example: The position of an employee: Analyst.	Relational Concatenation Assignment	C, RD, UN, V, W

See Also *TIBCO Object Service Broker Managing Data* about defining TIBCO Object Service Broker tables.

Operators to Combine Expressions

An expression can be combined with one or more other expressions. The rules language supports the use of arithmetic operators and the concatenation operator (||) to combine expressions.

Arithmetic Operators

The rules language contains the following operators for doing arithmetic:

Operator	Usage
**	Exponentiation.
*	Multiplication.
/	Division.
+	Addition.
-	Subtraction.
-	Unary minus.
+	Unary plus.

The arithmetic operators allow four types of operands: count, quantity, date, and typeless. The only operands allowed for unary - and unary + are count, quantity, and typeless. For a complete description of arithmetic operators and their behavior, refer to What are the Arithmetic Operators? on page 134.

Concatenation Operator

The rules language uses a double vertical bar (||) as the concatenation operator. Concatenation is valid between any two semantic data types and always has a result with semantic type string. For its syntax, the result follows these rules in order:

1. Concatenation is forbidden between a field of syntax UN and one of syntax W.
2. If one of the operands has syntax RD, the result has syntax RD.
3. If one of the operands has syntax UN, the result has syntax UN.
4. If one of the operands has syntax W, the result has syntax W.

5. The result has syntax V.

Operators Within Expressions

Operators within an expression conform to conventional notation and obey the precedence that follows. Exponentiation has highest precedence and addition, subtraction, and string-concatenation have the same lowest precedence. In cases where more than one operator has the same precedence, such as addition and string-concatenation, the operators are evaluated strictly from left to right, unless explicitly overridden using parentheses.

Operator	Usage
**	Exponentiation.
*, /	Multiplication, division.
+, -	Unary plus, unary minus.
+, -, ||	Addition, subtraction, string-concatenation.

Examples

Some examples of operators within expressions are:

```
CARS.PRICES = (PRICES.BASE + PRICES.SHIPPING) * TAXES.RETAIL
AMOUNT = PRINCIPAL * (1 + INTEREST) ** YEARS
```

You must use parentheses if you intend to do two or more exponentiation operations consecutively. Two correct examples are:

```
(A ** B) ** C
A ** (B ** C)
```

Relational Operators

Comparison Operators

This table shows the operators that the rules language uses for making comparisons:

Operator	Description	Notes
=	Equality	
¬=	Not equal	On Open Systems, this can display as ^=. You cannot substitute the keyword NOT for the not sign (¬) symbol.
<	Less than	
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
LIKE	Pattern matching	Use only in selection criteria statements, not in condition statements. The only permissible syntax for use with NOT (¬) is: `WHERE NOT(fieldname LIKE 'value').`

Semantic Data Type and Syntax Validations

Note the following points about semantic data types and syntax in expressions containing these operators:

- The relational operators for equality and inequality (=, ¬=) can be used with any two operands of the same semantic data type. They can also be used for two operands of different semantic data types if one of the operands is typeless or if the types are identifier and string, or identifier and count.
- The relational operators for ordering (<, <=, >, >=) cannot be used with an operand of type logical. Otherwise, they can be used with two operands with the same semantic data type or if one operand is typeless.

- The relational operator for pattern matching (LIKE) can be used with any semantic data types and always returns a logical value.
- Trailing blanks are significant for variable-length strings, but not for fixed-length strings. For example, comparing two fixed-length strings with lengths 12 and 16 gives the same result as if the shorter string were extended to length 16 and padded with four blanks on the right.
- If syntax differs, operands are converted to a common syntax. Refer to Chapter 12, Arithmetic Processing, on page 133 for a table of syntax conversions.
- The result of a comparison is always a logical value (Y or N).
- The LIKE relational operator is not case sensitive if you are matching syntax C data. In all other cases, the other relational operators distinguish between upper and lowercase.

Equality Relational Operators

The relational operators for equality and inequality are = and ¬=. The following chart shows the semantic types on which you can use these operators. In each case, the result of the relational operation is a logical value. Y indicates a valid operation.

	COUNT	DATE	IDENTIFIER	LOGICAL	QUANTITY	STRING	Typeless
COUNT	Y		Y				Y
DATE		Y					Y
IDENTIFIER	Y		Y			Y	Y
LOGICAL				Y			Y
QUANTITY					Y		Y
STRING			Y			Y	Y
Typeless	Y	Y	Y	Y	Y	Y	Y

Ordering Relational Operators

The relational operators for ordering are <, >, <=, and >=. The following chart shows the semantic data types on which you can use these operators. In each case the result of the relational operation is a logical value. Y indicates a valid operation.

	COUNT	DATE	IDENTIFIER	LOGICAL	QUANTITY	STRING	Typeless
COUNT	Y						Y
DATE		Y					Y
IDENTIFIER			Y				Y
LOGICAL							
QUANTITY					Y		Y
STRING						Y	Y
Typeless	Y	Y	Y		Y	Y	Y

Logical Operators

Logical operators join or negate expressions in rules language statements.

Valid Operators

The following table lists the logical operators that you can use with the rules language (both the word and the symbol are valid). The logical operator OR (|) takes precedence over the logical operator AND (&).

Word	Symbol	Meaning
AND	&	The expression evaluates to true (Y) if and only if both logical operands are true. This operator can be used only in selection criteria.
OR	|	The expression evaluates to true (Y) if either logical operand is true, or if both are true. This operator can be used only in selection criteria.
NOT	¬	The expression evaluates to true (Y) if and only if the logical operand is false (N). This operator can be used in the conditions part of a rule as well as in selection criteria. The only permissible syntax for use with LIKE is: WHERE NOT(fieldname LIKE 'value').

Notes on the NOT Symbol

- On Open Systems, the NOT symbol could display as a caret symbol (^).
- In ostty, you can enter the NOT operator by typing the word “not”, or by typing a caret symbol (^) (Shift+6 on some keyboards).

Assignment Operator

The rules language uses the equal sign (=) as the assignment operator.

Valid Assignments

The following chart shows which fields and values with the following semantic data types can be assigned to each other. In each case, the semantic data type of the result is the semantic data type of the left operand, unless the left operand is a local variable. If the left operand is a local variable, the semantic data type of the result is the semantic type of the right operand. Y indicates a valid operation.

	COUNT	DATE	IDENTIFIER	LOGICAL	QUANTITY	STRING	Typeless
COUNT	Y		Y			Y	Y
DATE		Y				Y	Y
IDENTIFIER	Y		Y			Y	Y
LOGICAL				Y			Y
QUANTITY					Y	Y	Y
STRING	Y	Y	Y		Y	Y	Y
Typeless	Y	Y	Y	Y	Y	Y	Y

An assignment of a string field to a date field can cause unexpected results if the default date format of your installation contains a two-digit year, for example, YYDDD or YYWW.

Types of Assignment Statements

Use assignment statements to assign values to fields or to local variables. There are two kinds of assignment statements:

- Simple assignment
- Assignment-by-name

Syntax of Assignment Statements

```
<assignment target> = <expression>
```

```
<assign by name>
```

Simple Assignment of a Value

In simple assignment, a single value is assigned to a field of a table or to a local variable.

Syntax

```
<assignment target> ::=
   <field of a table>
   <local name>
```

Examples

Two examples of simple assignment statements are:

```
CARS.PRICE = (PRICES.BASE + PRICES.SHIPPING)* TAXES.RETAIL;
```

```
AMOUNT = PRINCIPAL * (1 + INTEREST) ** YEARS;
```

Assigning Values by Name

In assignment-by-name, the field values of the table on the right are assigned to identically named fields of the table on the left. The field names are replaced with the asterisk symbol (*) for both the source and target tables.

Syntax

```
<table reference>.* =<table reference>.*
```

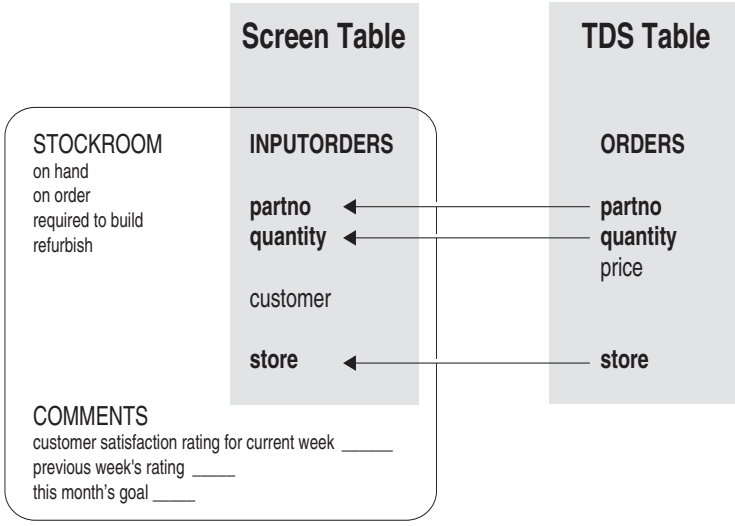
```
<table reference>.* = NULL
```

Example

```
INPUTORDERS.* = ORDERS.* ;
```

Assignment Relationship

One use of assignment-by-name assigns the values of fields of a screen table to fields of a data table, or vice versa. The following illustrates the assignment relationship between the screen fields in the screen table INPUTORDERS and the fields in the ORDERS table, as viewed in a user's application:



Initializing All the Fields of a Table

Assignment-by-name is a convenient way to initialize all the fields of a table. For example:

```
ORDERS.* = NULL;
```

This statement initializes all the fields in the ORDERS table to null values.

Indirect Referencing

Uses of Indirect Referencing

To facilitate the coding of generic rules and functions, the rules language permits indirect referencing. You can refer indirectly to:

- Tables
- Fields
- Screens
- Reports
- Rules

You can use indirect references in the conditions or actions part of a rule.

Restrictions

You cannot use:

- Indirect references in the ON statement itself but you can use them in the statements that form the body of the exception handler
- A local variable or return a value from a function to supply a value for an indirect reference

Providing Values

Values for an indirect reference (that is, names of tables, fields, screens, reports, or rules) are provided by:

- An argument to the rule that does the indirect referencing
- A reference to a *table.field*

The value you provide, either through an argument or in a *table.field* reference, must have a semantic type of identifier or be typeless.

Using a Rule Argument for Indirect Referencing

Example of an Argument to a Rule

The COUNT rule in the following figure is a generic rule that determines the sum of a field over all occurrences. The rule is general enough to sum any field of any table (without parameters). The rule receives the name of the table in the argument TABLEREF and it receives the name of the field in the argument FIELDREF.

```
     RULE EDITOR ===>                                                SCROLL: P
COUNT(TABLEREF, FIELDREF);
_ LOCAL SUM;
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ SUM = 0;                                                    | 1
_ FORALL TABLEREF :                                           | 2
_    SUM = SUM +(TABLEREF).(FIELDREF);                        |
_    END;                                                     |
_ CALL MSGLOG('THE SUM OF ' || TABLEREF ||'.' || FIELDREF ||  | 3
_   ' IS ' || SUM);                                          |
_ ------------------------------------------------------------------------------

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

About the Example

- The parentheses around the names TABLEREF and FIELDREF in the assignment statement signify indirection.
- The FORALL statement loops over the table specified by the indirect table reference (TABLEREF), and the action in the body of the loop involves the field specified by the indirect field reference (FIELDREF).
- If the COUNT rule is called as:

  ```
  CALL COUNT('EMPLOYEE_VIEW','SALARY');
  ```

The value of TABLEREF is EMPLOYEE_VIEW, and the value of FIELDREF is SALARY. The called rule finds the sum of the salaries of all employees.

Use of Parentheses

Parentheses must enclose the indirect reference when one of the following occurs:

- A data element is referenced in an expression.
- The indirect reference is used on the left side of an assignment statement.
- The indirect reference represents the name of a field in a WHERE or ORDERED clause.

Using Table.Field for Indirect Referencing

Examples of Table.Field

The following series of figures show:

1. An example of a rule that uses indirect referencing
2. The *table.field* form of indirect reference

   The example in Reference to a Parameterized Table on page 102 generalizes this rule for use with parameterized tables.
3. An example of a table instance used for indirect referencing

About the Examples

In the following examples:

1. When a PF key is used while viewing the DELETE_EMPLOYEE screen, the screen name DELETE_EMPLOYEE is passed to the PROCESS_FCNKEY tool.
2. The occurrence for that PF key in the table instance FCNKEYS(DELETE_EMPLOYEE) is then retrieved.
3. The CALL statement invokes the rule that corresponds to the PF key used.

   Looking at the FCNKEYS(DELETE_EMPLOYEE) table instance, if you press PF22 the DEL_EMP rule is invoked.

Calling a Rule that Uses Indirect Reference

The rule shown in the following figure initializes the function keys for the screen DELETE_EMPLOYEE, displays the screen, and calls the PROCESS_FCNKEY rule, which has *screen* as its argument.

```
   RULE EDITOR ===>                                                SCROLL: P
DELETE_EMPLOYEE;
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ FCNKEY_SPECS.FCNKEYS = FCNKEY_MSG('DELETE_EMPLOYEE');       | 1
_ INSERT FCNKEY_SPECS('DELETE_EMPLOYEE');                     | 2
_ UNTIL EXIT_DISPLAY DISPLAY DELETE_EMPLOYEE :                | 3
_    CALL PROCESS_FCNKEY('DELETE_EMPLOYEE');                  |
_    END;                                                     |
_ ------------------------------------------------------------------------------
```

```
PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Table.Field Form of Indirect Reference

The PROCESS_FCNKEY tool is called to get the FCNKEYS table, which is parameterized by *screen*, and then uses the *table.field* indirect reference in the statement: `CALL FCNKEYS.ROUTINE`.

```
 RULE EDITOR ===>                                                  SCROLL: P
PROCESS_FCNKEY(SCREEN);
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ GET FCNKEYS(SCREEN) WHERE PF_KEY = ENTERKEY(SCREEN);        | 1
_ CALL FCNKEYS.ROUTINE;                                       | 2
_ ------------------------------------------------------------------------------
_ ON GETFAIL FCNKEYS :
_    CALL SCREENMSG(SCREEN, MESSAGE('GENERAL', 3, ENTERKEY(SCREEN)));









PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Example of a Table Instance Used for Indirect Referencing

The ROUTINE field in this table, FCNKEYS, contains the name of the rule to be invoked when a PF key is used.

```
BROWSING TABLE    :  FCNKEYS(DELETE_EMPLOYEE)
COMMAND ==>
                                                                    SCROLL: P
   PF_KEY        NAME            COMMAND           ROUTINE
 _ -----  ---------------- ---------------- -----------------
 _ PF1    HELP                              DISPLAY_HELP
 _ PF22   DELETE                            DEL_EMP
 _ PF3    EXIT                              EXIT_DISPLAY




 PFKEYS: 1=HELP 5=FIND NEXT 9=RECALL 18=EXCLUDE 13=PRINT 3=END 14=EXPAND
```

Reference to a Parameterized Table

To generalize the sample rule shown in Example of an Argument to a Rule on page 96 to handle tables that have up to two parameters, or none at all, you can expand the rule as shown in the following figure:

```
    RULE EDITOR ===>                                                SCROLL: P
COUNT1(TABLEREF, FIELDREF, PARM1, PARM2);
_ LOCAL SUM;
_ ------------------------------------------------------------------------------
_ PARM1 = NULL;                                                  | Y N N
_ PARM2 = NULL;                                                  |   Y N
_ ---------------------------------------------------------------+--------------
_ FORALL TABLEREF :                                              | 1
_    SUM = SUM +(TABLEREF).(FIELDREF);                           |
_    END;                                                        |
_ FORALL TABLEREF(PARM1) :                                       |   1
_    SUM = SUM +(TABLEREF).(FIELDREF);                           |
_    END;                                                        |
_ FORALL TABLEREF(PARM1, PARM2) :                                |     1
_    SUM = SUM +(TABLEREF).(FIELDREF);                           |
_    END;                                                        |
_ CALL MSGLOG('THE SUM OF ' || TABLEREF ||'.' || FIELDREF ||     | 2 2 2
_   ' IS ' || SUM);                                              |
_ ------------------------------------------------------------------------------


PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Chapter 9 Transaction Processing

This chapter describes how to use transactions.

Topics

Overview

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

The Objective of a Transaction

The objective of a transaction is to move the database from one consistent state to another consistent state. This includes transactions that:

- Do not attempt to update the database
- Attempt to update the database, and during the course of the process, determine that the updates are invalid
- Successfully update data

Example of a Transaction

An example of a transaction is adding an employee to the database. The business rules for this type of transaction state that an employee must be assigned a valid department number. Before committing the addition of the employee to the database, the transaction must satisfy one of the following conditions:

1. Reference the department table to guarantee that the department number assigned to the employee is valid
2. Not process the addition of the employee if the department number is invalid
3. Process the addition of the employee, if the department number is valid

Based upon the business rules under which it is operating, this transaction guarantees that the database is in a consistent state upon completion of the transaction.

What Starts a Transaction?

Transaction Statements

The following statements can be used within a rule to start a transaction. The statement that you use determines what happens to an existing transaction when the statement is issued.

EXECUTE	Starts a new transaction but does not end the current transaction—the current transaction is suspended. It is used to nest transactions.
TRANSFERCALL	Ends the current transaction and begins a new one
DISPLAY & TRANSFERCALL	Maintains all the data displayed for the screen and the screen context, such as the displayed data, the cursor position, the display attributes, but ends the current transaction and begins a new one
SCHEDULE	Maintains the current transaction and submits a batch job or file that could start up a new Execution Environment and session, and initiates a separate batch transaction for processing in the new session.

For more information about these statements, refer to the SCHEDULE Statement on page 53; also refer to Chapter 17, Processing Asynchronously in Batch Mode, on page 191.

Course of a Transaction

The rules language statements encountered when the transaction is run determine the specifics of what takes place during the course of a transaction. In general, however, basic similarities exist in all transactions:

1. A synchronization point is established.
2. The transaction ends and a final synchronization point is established.
3. Locks are released.

Fail Safe processing ensures that the updates made to multiple databases in a single transaction are synchronized at all times.

Establishing Synchronization Points

TIBCO Object Service Broker creates implicit synchronization points at the beginning and end of every transaction. You can also establish explicit synchronization points within your transaction using COMMIT and ROLLBACK statements.

- The COMMIT statement commits the changes to the database that were made since the last synchronization point.
- The ROLLBACK statement discards pending changes and keeps the database in the same state it was at as of the last synchronization point.

TIBCO Object Service Broker automatically commits all changes made since the last synchronization point when the transaction terminates normally and discards all pending changes made after the last synchronization point when the transaction terminates abnormally.

Actions within Synchronization Points

When a synchronizing point is established, the following actions could take place, depending on the coding of the transaction:

- Data is retrieved.
- Locks are taken on the definitions and depending on the mode, on the data.
- Updates are made to the data.
- Explicit synchronization points are processed.

See Also *Managing Backup and Recovery* for your operating environment about Fail Safe processing.

Nesting Transactions

How are Transactions Nested?

When you EXECUTE a rule from within a current transaction, it suspends the current transaction—known as the parent transaction—and starts a child transaction. The transactions are now nested.

Behavior of Nested Transactions

When a transaction is nested, the following occurs:

- A child transaction starts a new transaction level.
- At the completion of the child transaction, control passes back to the parent transaction.

A parent transaction can handle failures of a child transaction with the EXECUTEFAIL exception.

What Determines the Transaction Level?

The first transaction started in a user session starts at transaction level one. The session parameter TRANMAXNUM determines the maximum number of transactions that can be nested for a user session.

Finding the Name of a Rule in a Transaction

You can use the $RULENAME shareable tool to find out the name of a rule in the current transaction or in a parent transaction.

See Also *TIBCO Object Service Broker Parameters* about session parameters.

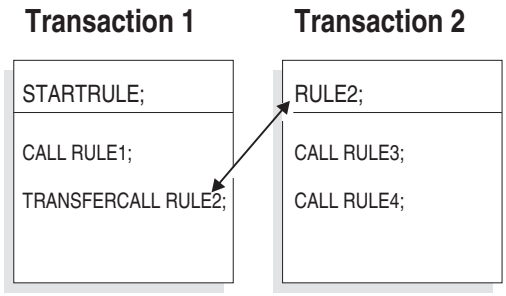
TIBCO Object Service Broker Shareable Tools about the $RULENAME tool.

Changing the Flow of a Transaction

From within a transaction you can change the flow of a transaction. The following series of diagrams illustrate how the flow changes from transaction to transaction depending upon the statements that you use.

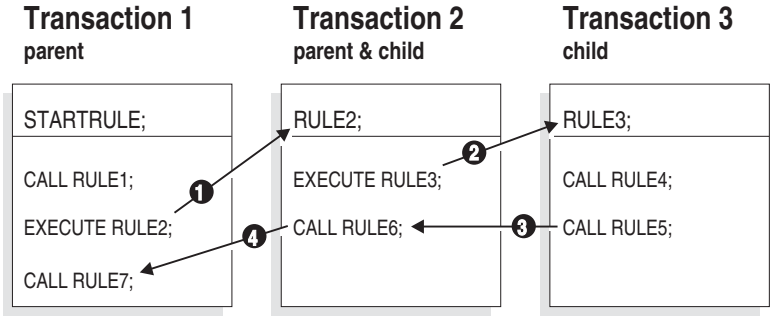
Starting a New Transaction

The following diagram shows Transaction 1 issuing a TRANSFERCALL statement to start Transaction 2 as a new transaction.



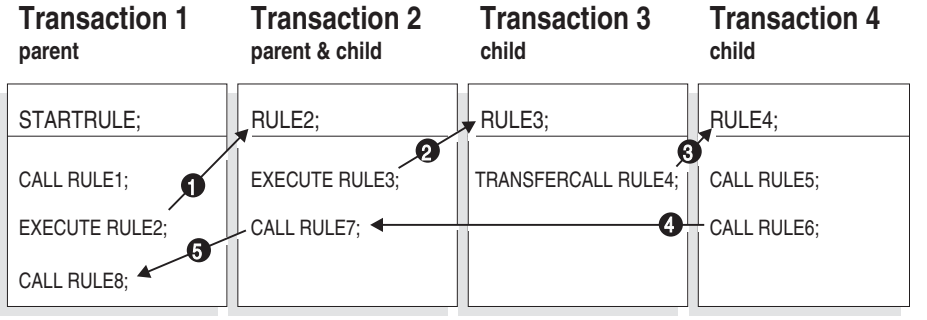
Starting a New Nested Transaction

The following diagram shows the flow of nested transactions. In this diagram, Transaction 1 is the parent of Transaction 2 (child), and Transaction 2 is the parent of Transaction 3 (child). The numbered arrows indicate the sequence of actions from one transaction to another.



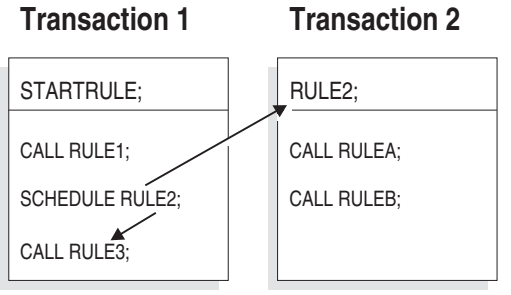
Starting a New Transaction Within a Nested Transaction

The following diagram shows the flow of a nested transaction that also starts a new transaction using a TRANSFERCALL statement. In this diagram Transaction 1 is the parent of Transaction 2 (child), and Transaction 2 is the parent of both Transaction 3 (child) and Transaction 4 (child). The numbered arrows indicate the sequence of actions from one transaction to another.



Starting a Batch Transaction

The following diagram shows a scheduled batch transaction. Transaction 2 can run at any time and Transaction 1 continues to run with control passing to RULE3 after the SCHEDULE statement is issued. RULE3 does not wait for RULE2 to start and complete.



Setting the Mode of the Transaction

How Do You Set The Mode?

Use the IN clause to specify whether the transaction should be operating in browse or update mode. You can freely intermingle browse and update mode transactions. Unless it is explicitly set, a transaction inherits its mode from the transaction that initiated it[1].

Example

The following statement starts a transaction in browse mode:

```
TRANSFERCALL IN BROWSE FIND_DEPT;
```

In this example, no updates can be made to the persistent data used by the transaction FIND_DEPT.

Mode Determines Locks on Data

The mode determines the locks taken on TDS data only. Locks taken on external data are governed by the external database. Locks on table definitions are always taken regardless of the mode. For more information about locking, refer to Locks Taken on the Data on page 112.

Exception Raised

If an attempt is made to update data that is persistent while executing in browse mode, the ACCESSFAIL exception is raised. For more information about exceptions, refer to Chapter 6, Exception Handling, on page 59. However, within a transaction executing in browse mode, you can update tables with temporary data (TEM, SCR, RPT, and SES tables) and EXP files.

See Also

TIBCO Object Service Broker Managing Data and *TIBCO Object Service Broker Managing External Data* about TIBCO Object Service Broker and external data tables.

TIBCO Object Service Broker Parameters about session parameters.

1. The session parameters BROWSE and NOBROWSE determine the mode of the first transaction started in TIBCO Object Service Broker.

Locks Taken on the Data

What Determines the Type of Locking?

The types of accesses that your transaction makes to your data and the mode in which your transaction operates determine the types and levels of locks that are taken on the tables you are accessing. Locks on data are taken only if the transaction is operating in update mode. Locks on table definitions are always taken, regardless of mode.

Locking behavior can also be modified using the WITH MINLOCK modifier for the GET statement in the rules language. For details, see GET Statement on page 45.

Types of Locks

A lock is either a shared lock or an exclusive lock, as follows:

Shared	The definition of the table cannot be modified but other transactions operating in update mode can read the data.
Exclusive	The definition of the table cannot be modified and other transactions cannot read or modify the data.

Exception Handling

You can use the LOCKFAIL exception to handle situations where you are unable to obtain a lock. For example, the transaction can require an exclusive lock but another transaction is holding a shared lock. Refer to Chapter 6, Exception Handling, on page 59 for more information about exceptions.

Data Accesses and Types of Locking

The following table describes the locks taken depending on your data access:

Type of Access	Mode of Transaction	Type of Lock	Level of Lock
GET or FORALL	Browse	Shared	Table definition only.

GET or FORALL using unique (that is, equality) selection on the primary key fields, for example:
`GET EMPLOYEES where EMPNO=79912;`
`GET TESTDATA where PKF1=A & PKF2=B;`
	Update	Shared	Table definition.
			Occurrence.

GET or FORALL using non-unique selection, for example:
`FORALL EMPLOYEES ordered EMPNO:`
`GET EMPLOYEES where EMPNO>79912;`
`FORALL EMPLOYEES:`
	Update	Shared	Table definition.
			Table instance (if parameterized).
			Table (if not parameterized).

GET using non-unique Selection and WITH MINLOCK, for example: `FORALL EMPLOYEES WHERE SALARY<100000 WITH MINLOCK:`	Update	Shared	Table definition. If parameterized: Table instance during GET processing, Occurrence thereafter. If non-parameterized: Table during GET processing, Occurrence thereafter.
DELETE, INSERT, REPLACE	Update	Shared	Table definition.
		Upgrade shared to exclusive	Occurrence.
Viewing the table definition	Browse and/or update	Shared	Table definition.

Type of Access	Mode of Transaction	Type of Lock	Level of Lock
Updating the table definition	Update	Exclusive	Table definition.
Building a secondary index	Update	Shared	Table definition. Table instance (if parameterized). Table (if not parameterized).
Browsing a CLC (calculation) table (read access on the underlying table)	Browse	Shared	Source table definition.
Accessing a CLC (calculation) table	Update	Shared	Source table data. Source table instance (if parameterized).
Accessing a SUB (subview) table defined with MODE=D	Update	Shared	Source table definition. Table definition. Table instance (if parameterized). Table (if not parameterized). Occurrence.
Accessing a SUB (subview) table defined with MODE=B	Browse	Shared	Source table definition. Table definition.

See Also *TIBCO Service Gateway* manuals about table locking as it relates to external databases.

TIBCO Object Service Broker Managing Data about TIBCO Object Service Broker tables and table types.

Chapter 10 Conditional Processing

This chapter describes how to use conditional processing.

Topics

TIBCO Object Service Broker Conditional Processing

What is Conditional Processing?

In TIBCO Object Service Broker, you use conditional processing to determine whether a given expression is true or false. Based on the result, you control which action statements in the rule are subsequently executed.

Uses of Conditional Processing

Using conditional processing you can, for example:

- Use a relational operator to determine the actions to be performed by a rule.

  In conditions with relational comparisons, only one comparison is permitted per condition. For a complete list of comparisons that you can make in conditions, refer to Comparison Operators on page 88.

- Use a Y or N value passed in from a calling rule to determine the actions to be performed by a rule.

  The value can be passed in as an argument value in the calling rule; or a local variable value passed in from a rule higher up in the calling hierarchy.

- Use the value of a field of type logical using a *table.field* reference to determine the actions to be performed.

  The value must be passed in from a rule higher up in the calling hierarchy.

- Code a functional rule to return a logical value.

If you want to provide conditional processing, you can specify conditions to your rule in the conditions section of the rule.

What are Conditions?

Conditions are logical expressions evaluated sequentially for their truth value. The sequencing of actions following this evaluation is regulated by the Y/N quadrant and the action numbers that are specified for the rules statements. If one of the conditions is satisfied, the actions corresponding to it are executed and no further conditions are evaluated. If no conditions are given, all the numbered action statements in the body of the rule are executed.

Example of a Condition Segment and Associated Actions

The following example shows the condition segment containing an expression and the action segment showing the first action to take place based on the values determined by the Y/N quadrant:

```
_ ------------------------------------------------------------+--------------
_ JOBTITLE = 'SENIOR ANALYST';                                | Y N N
_ JOBTITLE='ANALYST';                                         |   Y N
_ ------------------------------------------------------------+--------------
_ RATE = 0.1;                                                 | 1
_ RATE = 0.05;                                                |   1
- RATE = 0.002;                                              |     1
```

Adding Conditions

To add conditions, type an `I` into the line command field of the condition section and press Enter to insert a line. Add each condition, ending each one with a semicolon(;).

This part of the rule also contains the Y/N quadrant, which contains values that associate conditions with actions. The Rule Editor manages the Y/N quadrant; you cannot edit it.

Maximum Number of Conditions

You can have up to six conditions in a rule.

Examples of Conditions

Types of Examples

The following are examples of conditions as:

- An expression
- An argument to a rule
- A *table.field* reference
- A functional rule

Expression as a Condition

In the following example:

- Actions under the Y column are performed only when the argument DEPT has a value greater than 0.
- Actions under the N column are performed only when the DEPT is less than or equal to 0.

```
      RULE EDITOR ===>                                                SCROLL: P
EMPLOYEE_COUNT(DEPT);
_ LOCAL COUNT;
_ ------------------------------------------------------------------------------
_ DEPT > 0;                                                    | Y N
_ -------------------------------------------------------------+--------------
_ COUNT = 0;                                                   | 1 1
_ FORALL EMPLOYEES WHERE REGION = 'MIDWEST' & DEPTNO = DEPT    | 2
_     ORDERED DESCENDING LNAME :                               |
_     CALL MSGLOG(EMPLOYEES.LNAME);                            |
_     COUNT = COUNT + 1;                                       |
_     END;                                                     |
_ CALL ENDMSG(COUNT || ' EMPLOYEES IN DEPARTMENT# ' || DEPT)   | 3 2
_   ;                                                          |
_ ------------------------------------------------------------------------------
_  ON CONVERSION:
_    CALL ENDMSG('Value for DEPT must be numeric');




PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Argument as a Condition

In the following example, actions under the Y column are performed only when the argument NEW has a value of Y. If the value is N, actions under the N column are performed.

```
      RULE EDITOR ===>                                                SCROLL: P
EMPLOYEE_UPDATE(NEW);
_
_ ------------------------------------------------------------------------------
_ NEW ;                                                        | Y N
_ -------------------------------------------------------------+--------------
_ CALL ENDMSG('NEW EMPLOYEE');                                 | 1
_ CALL ENDMSG('EXISTING EMPLOYEE');                            |   1
_--------------------------------------------------------------+--------------


PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

When using an argument as a condition, any character value passed into the argument other than Y is evaluated as if it is N. An error occurs if a numeric value is passed in.

table.field as a Condition

In the following example:

1. GET_EXEMPT1 retrieves an occurrence from the MANAGER table.
2. GET_EXEMPT calls GET_EXEMPT2.
3. GET_EXEMPT2 evaluates the MANAGER.EXEMPT field for its Y/N value.

4. GET_EXEMPT2 is processed based on the value in the EXEMPT field.

```
      RULE EDITOR ===>                                                SCROLL: P
GET_EXEMPT1(NAME);
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ GET MANAGER WHERE MANAGER_NAME = NAME;                      | 1
_ CALL GET_EXEMPT2(NAME);                                     | 2
_ ------------------------------------------------------------------------------
_ ON GETFAIL :
_    CALL ENDMSG(NAME || ' IS AN INVALID NAME');




PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

```
      RULE EDITOR ===>                                                SCROLL: P
GET_EXEMPT2(NAME);
_
_ ------------------------------------------------------------------------------
_ MANAGER.EXEMPT;                                             | Y N
_ ------------------------------------------------------------+---------------
_ CALL ENDMSG(NAME || ' IS EXEMPT');                          | 1
_ CALL ENDMSG(NAME || ' IS NOT EXEMPT');                      |   1
_------------------------------------------------------------------------------




PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Functional Rule as a Condition

To handle more complicated situations, you can invoke a functional rule as a condition. The following rule illustrates the use of a function, in this case the PATTERN_MATCH tool. In the example:

- The rule checks to see if the commands **SELECT** or **DELETE**, or an abbreviation of these commands, such as **SEL** or **DEL**, are valid.

Normally, if you have several commands, you would store them in a table and use LIKE as your selection operator.

- If the command is a valid **SELECT** or **DELETE** command, the appropriate rule is called.
- If the command is not valid, the user gets a message on the screen explaining that the command is not acceptable. The following illustrates a rule using PATTERN_MATCH:

```
      RULE EDITOR ===>                                                SCROLL: P
PROCESS_CMD(USER_COMMAND);
_
_ ------------------------------------------------------------------------------
_ PATTERN_MATCH(COMMAND, 's*');                                  | Y N N
_ PATTERN_MATCH(COMMAND, 'd*');                                  |   Y N
_ ---------------------------------------------------------------+--------------
_ CALL SELECT_COMMAND;                                           | 1
_ CALL DELETE_COMMAND;                                           |   1
_ CALL SCREENMSG(USERCOMMAND ||' IS AN INVALID COMMAND');        |     1
_---------------------------------------------------------------+--------------


PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

When using a functional rule as a condition, any character value returned other than Y is evaluated as if it is N. An error occurs if the functional rule returns a numeric value.

Chapter 11 Null Processing

This chapter describes how to use null processing.

Topics

TIBCO Object Service Broker Nulls

What is a Null Value?

A null represents data of no value. A null value occurs because no data exists in a field, or data of no value is explicitly assigned to a field. In TIBCO Object Service Broker, null has a value less than any other value.

Syntax and Behavior of Nulls

As with all other types of data, a null can assume either a character or numeric syntax. A field that is defined with a syntax of C, UN, V, or W holds a character null. A field that is defined with a syntax of B, P, or F holds a numeric null. The behavior of nulls is determined by their syntax and whether the data is character or numeric. A null field of syntax RD can act as either one.

The syntax within which a null is being used determines how the null value is processed:

- When a null is represented by character syntaxes (C, UN, V, or W), its behavior is identical to an empty string ('').
- When a null is represented by numeric syntaxes (B, P, or F), its behavior is different from any other numeric value.
- When an RD null is used as a character syntax, it acts as a character null; when it is used as a number syntax, it acts as a numeric null.

In TIBCO Object Service Broker, the keyword NULL represents a null value of syntax B.

How Can A Field Contain Null Values?

There are three ways that a field can contain a null value:

- An occurrence is inserted into a table, no assignment is made to the field, and no default value is specified.
- An occurrence is inserted or replaced into a table and the field is assigned the keyword NULL or an empty string ('') and no default value is specified.
- A new field is added to a table that already contains data. All occurrences for this new field contain null values.

Manipulation of Nulls

Allowable Manipulation

Nulls can only be manipulated logically. They cannot be manipulated arithmetically. Attempts to add, subtract, multiply, divide, and so on, result in the NULLVALUE exception.

Logical Manipulation

Nulls can be manipulated through the use of the reserved word NULL, which represents a null value. The following table describes where the keyword NULL can be used and the syntax that each context requires:

Allowable Usage	Syntax
Rule condition.	<expression> < relational operator> < expression>
Assignment.	<assignment target> = <NULL>
Passing rules arguments by name.	<argument> = <NULL>
Function return.	return (<NULL>)
Selection.	<field or parameter name> <relational operator> <expression>
Passing rules arguments by position.	(<NULL> {, expression})
Initialize all fields of an occurrence.	<table reference>.*=NULL

Restrictions

The reserved word NULL cannot be used:

- As an operand in an expression. For example: `NULL+1`, or `NULL ||'text'`
- To replace the empty string for an argument when a rule is executed from the workbench

Behavior of Nulls

The context in which a null is used determines its behavior. The following sections describe the behavior of a null depending on its context.

Conversion

TIBCO Object Service Broker provides automatic conversion from one syntax to another when required:

Value	Converted To	Result
Numeric null	Character syntax (C, UN, V, or W)	Empty string ('')
Character null or empty string ('')	Numeric syntax (B, P, or F)	Zero (0)
RD null	Character syntax (C, UN, V, or W)	Empty string ('')
RD null	Numeric syntax (B, P, or F)	Numeric null
Character null or empty string (''), and numeric null	RD syntax	RD null

Assignment

When an assignment is made to a field, the value being assigned is converted to the syntax of the field. When an assignment of a null is made to a local variable, the type and syntax of the local variable becomes the same as the value being assigned to it:

Assigned Value	Assigned To	Result
Character null or empty string ('')	Character field	Empty string ('')
Character null or empty string ('')	Date field	Numeric null
Character null or empty string ('')	Local variable	Empty string ('')
Character null or empty string ('')	Numeric field	Zero (0)

Assigned Value	Assigned To	Result
Numeric null	Character field	Empty string ('')
Numeric null	Date field	Numeric null
Numeric null	Local variable	Numeric null
Numeric null	Numeric field	Numeric null
RD null	Character field	Empty String ('')
RD null	Date field	Numeric null
RD null	Local variable	RD null
RD null	Numeric field	Numeric null

Table Parameters

When a parameter of a table is specified, the value provided is converted to the syntax of the parameter:

Specified Value	Parameter Syntax	Result
Character null	Character syntax	Empty string ('') This is an invalid value and the DATAREFERENCE exception is raised.
Numeric null	Numeric syntax	Numeric null This is an invalid value and the DATAREFERENCE exception is raised.
Empty string ('')	Numeric syntax	Zero (0)
RD null	Character syntax	Empty string ('') This is an invalid value and the DATAREFERENCE exception is raised.
RD null	Numeric syntax	Numeric null This is an invalid value and the DATAREFERENCE exception is raised.

Rules Arguments

When a null value is passed to the argument of a rule, the type and syntax of the argument becomes the same as the null value being passed:

Passed Value	Result
Character null	Character syntax
Numeric null	Numeric syntax
RD null	RD syntax

Routine Arguments

When a value is passed to the argument of a TIBCO Object Service Broker routine (a tool written in non-rules code), it is converted to the syntax of the formal argument:

Passed Value	Syntax of the Formal Argument	Result
Numeric null	Character syntax	Empty string ('')
Numeric null	Numeric syntax	NULLVALUE exception raised
Empty string ('')	Numeric syntax	Zero (0)
RD null	Character syntax	Empty string ('')
RD null	Numeric syntax	NULLVALUE exception raised

Expressions

Expressions in TIBCO Object Service Broker are evaluated at two different points: when a selection is encountered (for example, `field1 = *.field2 + 40`), and when the rule is being executed.

Evaluation Point	Operation	Result
Selection	Arithmetic on a numeric null or RD null.	The current occurrence is discarded and evaluation of other occurrences continues.

Evaluation Point	Operation	Result
Execution	Arithmetic on a numeric null or RD null.	NULLVALUE exception raised.
Selection or execution	Arithmetic on a character null.	Zero (0).
Selection or execution	Concatenation on a numeric null or RD null.	Empty string ('').

Relational Expressions

The following relational expressions involving null are true:

Left Operand Value	Operator	Right Operand Value
Numeric null.	=	Numeric null.
Numeric null.	=	Empty string ('').
Numeric null.	<	Any non-null value.
Character null.	=	Empty string ('').
Character null.	=	Zero (0).
RD null.	=	RD null.
RD null.	=	Character null.
RD null.	=	Numeric null.
RD null.	<	Any non-null value.

The relational expressions that use the form:

<expression> <relational operator> <expression>

can appear in a rules condition, or as part of selection within a WHERE clause. Within selection, an empty string can be passed into selection statements that use the LIKE operator.

Ordering

For the purpose of ordering, null is considered to be less than any value. Occurrences with a null field appear first when they are ORDERED ASCENDING on that field.

Primary and Secondary Keys

A primary key field cannot contain a null value. Nulls are permitted for secondary key fields that have a character semantic type.

Key Type	Null Key Result
Primary key	INSERT results in the INSERTFAIL exception. REPLACE results in the REPLACEFAIL exception. Insertion of a null primary key value to an IDgen table is allowed.
Secondary key	Numeric null is not allowed.

Required Fields

In general, fields that are defined as required cannot contain null values. An INSERT results in an INSERTFAIL exception and a REPLACE results in a REPLACEFAIL exception.

When a populated table has its definition modified, exceptions can arise as a result of:

- Adding a new field that is defined as required
- Making a previously non-required field required

Subsequent updates to the table cause the appropriate exceptions to be raised.

Default Values

Default values replace assigned null values. Fields that have a default value specified and no assignment made to them, or that are assigned a null value, have the default value assigned to them immediately before an occurrence insertion or replacement.

Field	Assignment	Before an INSERT or REPLACE
Default value specified.	No assignment made.	Default value assigned.
Default value specified.	Assigned to a null value.	Null value replaced by the default value.

Unassigned Fields

An unassigned field can be referenced by giving it a reference value of null.

Example	Resulting Action
LOCAL1=*table.field;*	UNASSIGNED exception is raised.
table.field=NULL; LOCAL1=*table.field*;	The value is passed and no exception is raised.

Indirect Names

Nulls are not allowed as values for an indirect reference. If, during processing, an indirect reference evaluates to a null value, the transaction terminates and an ERROR exception is raised. The exception is trapped with ON ERROR.

Initialization of Local Variables

When a local variable is allocated, its initial value is an empty string ('').

Chapter 12 Arithmetic Processing

This chapter describes how to use arithmetic processing.

Topics

Overview

What are the Arithmetic Operators?

The arithmetic operators and their symbols are:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Exponentiation (**)

Permitted Syntaxes for Arithmetic Operations

You can perform arithmetic operations with any of the syntax types, provided the semantic data type permits it. Refer to Permissible Operations on page 137 for further information. Arithmetic operations are performed on the following syntax types:

- Binary (B)
- Floating point (F)
- Packed decimal (P)

Formatting Numeric Results

You can format the numeric result with a display mask using the $PIC tool.

Conversion of Strings

Data is converted to a numeric data type before the arithmetic operation when the syntax is one of the following:

- Fixed-length character string (C)
- Raw data (RD)
- Unicode (UN)
- Variable-length character string (V)

- Double-byte and single-byte character string (W)

- Arithmetic involving an RD field is allowed only if the fields semantic data type is typeless and if the other field in the operation is not also an RD field.
- Arithmetic involving an RD field that contains non-numeric data uses the numeric value of that data; for example, an operation with an RD field that is equal to “A” (R'C1') in combination with a field of syntax B would use the decimal value -63.

See Also *TIBCO Object Service Broker Shareable Tools* about the $PIC tool.

Strings as Operands

Converting Strings to Numeric Syntax

If an operand is of syntax C, UN, V, or W, the string is converted to a numeric syntax based on the value of the string. TIBCO Object Service Broker applies the following conversion rules to string data used in arithmetic operations:

- Binary is assigned for a string that is all digits.
- Packed decimal is assigned if the string contains a period (.).
- Floating point is assigned if the string has a numeric to the right and left of an exponent sign (E). An optional period (.) is also allowed before the exponent sign.

If a binary or packed field cannot hold the value, float is used.

If one operand of an arithmetic operation is of syntax RD, it is converted to the syntax of the other operand prior to the operation. Arithmetic operations are not permitted between two RD operands.

Conversion of Numbers to Strings

When a number is converted to a variable length string, non-significant zeros are removed. For example:

The number	Becomes…
00010	10
040.0170	40.017
00500.0200E-013 [a]	500.02E-13

a. This can also be expressed as 00500.0200e-013.

Permissible Operations

The following charts show the arithmetic operations that can be performed and the result returned. The empty square symbol (❒) indicates that the operation is invalid.

Addition (+) Operator

	COUNT	DATE	QUANTITY	Typeless
COUNT	Count	Date (if COUNT is binary)	❒	Count
DATE	Date (if COUNT is binary)	❒	❒	Date
QUANTITY	❒	❒	Quantity	Quantity
Typeless	Count	Date	Quantity	Typeless

Subtraction (-) Operator

	COUNT	DATE	QUANTITY	Typeless
COUNT	Count	❒	❒	Count
DATE	Date	Count	❒	Date
QUANTITY	❒	❒	Quantity	Quantity
Typeless	Count	❒	Quantity	Typeless

Multiplication, Division, and Exponentiation (*, /, **) Operators

	COUNT	QUANTITY	Typeless
COUNT	Count	Quantity	Count
QUANTITY	Quantity	Quantity	Quantity
Typeless	Count	Quantity	Typeless

These operations are subject to the syntax restrictions of the operation result. For example, the following is invalid: the addition of a DATE and a packed COUNT gives a packed result, while the intended result is a date, which must be binary. Refer to the table in Resultant Syntax from Arithmetic Operations on page 138.

Resultant Syntax from Arithmetic Operations

The following table describes the syntax that results when operations are performed on numeric fields with a variety of syntax types. Where the number of decimal places or the length of the result can vary, more information is given for each arithmetic operation.

If an arithmetic operation causes an overflow, the computation is attempted in floating point syntax.

Resultant Syntax

1st Operator Syntax	2nd Operator Syntax	Result Syntax	Decimal Places or Length of Result for:			
			+ and or -	*	/	**
B(L1)	B(L2)	B(4)				
B(L1)	P(L2,D2)	P(16,D)	D=D2	D=D2	D=DMAX	D=DMAX
B(L1)	F(L2)	F(L2)				
P(L1,D1)	B(L2)	P(16,D)	D=D1	D=D1	D=DMAX	D=DMAX
P(L1,D1)	P(L2,D2)	P(16,D)	D=MIN(MAX(D1, D2), DMAX))	D=MIN(D1+ D2, DMAX)	D=DMAX	
P(L1,D1)	F(L2)	F(L2)				
F(L1)	B(L2)	F(L1)				
F(L1)	P(L2,D2)	F(L1)	L=L1	L=L1	L=L1	L=L1
F(L1)	F(L2)	F(L)	L=MIN(L1,L2)	L=MIN(L1, L2)	L=MIN(L1,L2)	L=MIN(L1,L2)

Table Key

The values for the table are as follows:

D	Number of decimal places in the result.
DMAX	Maximum number of decimal places that the result can hold without overflowing the integer portion.

D1	Number of decimal places in the first operand.
D2	Number of decimal places in the second operand.
L	Length of the result.
L1	Length of the first operand.
L2	Length of the second operand.
OP2	Value of the second operand.

Chapter 13 Using Rules Libraries

This chapter describes how to use rules libraries.

Topics

Organization of Rules Libraries

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

Overview

Before using the Rule Editor, you must understand the organization and usage of TIBCO Object Service Broker rules libraries. Rules that you create and edit are placed in the local library that you are currently accessing. Rules that you are executing can be obtained from a number of different libraries.

Types of Rules Libraries

TIBCO Object Service Broker stores rules in a three-tiered library system. The three tiers are:

- Local libraries
- Installation library
- System library

Local Libraries

Your local library contains rules you created, copied, or modified for your own use. Your default local library for use during your session is defined as part of your session options. You can also define additional local libraries using the DEFINE_LIBRARY tool. Refer to Defining a Library on page 147 for more information.

Provided you have the proper access, you can access local libraries other than your default. Refer to Changing Local Libraries on page 145 for more information.

Installation Library

The installation library contains rules available to all users of a particular TIBCO Object Service Broker installation. The content of this library is controlled by the TIBCO Object Service Broker system administrator. It is usually called SITE.

System Library

The system library contains rules shipped as part of the TIBCO Object Service Broker system. It is usually called COMMON.

Viewing the Listing of Libraries

To view the listing of libraries, position your cursor on the option:

```
DL define library ==>
```

and press Enter. A screen similar to the following appears:

```
                        List of Defined Libraries
Command ==>                                                      Scroll P

     NAME     AUTHOR                                                SUMMARY*
   -------- -------- ------------------------------------------------------------
 _ AABB00   AAA000   HOME library of AAA000
 _ AATEST   AAA000   Test library of AAA000
 _ BBAA00   BBB000   HOME library of BBB000
 _ BBTEST   BBB000   Test library of BBB000
 _ DOCSAMP  USR40    Library of sample rules for doc
 _ DOCTOOLS USR40    Library of sample rules for tools




 D-Delete  S-Select
 PFKEYS: 12=EXIT 13=PRINT 3=END 5=FIND NEXT 9=RECALL
```

Available Commands

D	Deletes a library from the system. You are prompted to confirm the deletion.
S	Displays a listing of all the rules contained in the selected library.

Changing the Search Path for Rules Execution

The search path used for your rules execution is determined by the Search specification in your user profile, your session options, or the menu definition used to list the rule. TIBCO Object Service Broker provides the following default search order: your local library is searched first, followed by the installation library, followed by the system library. You can change the search order used for your session as required. The first rule encountered with the appropriate rule name is executed.

See Also *TIBCO Object Service Broker Managing Security*, *TIBCO Object Service Broker Parameters*, and *TIBCO Object Service Broker Defining Screens and Menus* about changing the search order for rules execution.

TIBCO Object Service Broker Managing Security about defining user profiles.

Changing Local Libraries

Login Library

The default or login library that you access when you log in to TIBCO Object Service Broker is normally the local library assigned to your user ID. The name of your login library is shown in the **Library** field at the top of the workbench.

Modifying Your Login Library

You can modify the login library that you use via:

- Your user profile
- Command line arguments to the osBatch and S6BBATCH utilities
- Your session options

Accessing a Different Local Library

If you require rules from a different local library after starting a session, type the name of the library where the rules are stored in the **Library** field of the workbench. For example, you can change from the local library USR40 to the local library EXAMP by overtyping USR40 with EXAMP. Changing the library name changes the local library you are accessing. The change stays in effect until you enter a different name in the **Library** field, or until you exit from TIBCO Object Service Broker.

You must have security access to the local library to access rules stored within it. To determine who owns a library, use the Define Library option from the workbench.

Copying a Rule to a Different Library

Using the workbench or rules, you can copy a rule from one library to another. To copy a rule using the workbench, do one of the following:

- Position your cursor on the menu option CD copy defn and press Enter

  This invokes the COPYDEFN tool. You can use this tool to copy one or more rules to a library on your local node or to a remote node.

- Use the **C** line command from within the Object Manager screen of the Rule Editor.

  From the displayed screen, type the name of the library you want the rule to be copied to in the **DEST_LIB** field and the name you want the rule to be called in the **DEST_RULE** field.

You can also call or execute the COPY_DEFN tool. As with the COPYDEFN tool, you can copy one or more rules to a library on your local node or to a remote node.

See Also

TIBCO Object Service Broker Managing Security about your user profile and about security accesses.

TIBCO Object Service Broker Parameters about specifying session options.

TIBCO Object Service Broker for z/OS External Environments about using S6BBATCH.

TIBCO Object Service Broker for Open Systems Utilities about using osBatch.

TIBCO Object Service Broker Shareable Tools about the copy definition tools, COPY_DEFN and COPYDEFN.

Defining a Library

Steps to Define a Library:

To define a rules library, complete the following steps from the workbench:

1. Position your cursor on the option DL define library or type DEFINE_LIBRARY in the EX execute rule option, and press Enter.
2. Type in the name of a new library, and press Enter.
3. Enter relevant information into the Description screen.
4. Press PF3 to save the definition.

Example Definition

The example:

```
DL define library ==> EXAMP
```

displays the following screen:

```
DESCRIPTION OF LIBRARY            EXAMP                              UNIT=DOCSAMP

MODIFIED ON                   BY                  CREATED ON 23 MAR 1998 BY USR40

KEYWORDS:
SUMMARY :

                                  DESCRIPTION
 _ ------------------------------------------------------------------------
 _
 _




 PFKEYS: 3=END 5=VIEW DOCUMENT 13=PRINT  12=EXIT
```

Available PF Keys

PF3	Save changes and return to the workbench.
PF5	Script the description. Press PF5 again to return to edit mode.
PF12	Cancel changes and return to the workbench.
PF13	Print the description.

Providing the Description for a Rules Library

The Library Definer updates some of the fields on the Library Definer screen, but you maintain the **KEYWORDS**, **SUMMARY**, and **DESCRIPTION** fields. You can specify the following information in the **KEYWORDS**, **SUMMARY**, and **DESCRIPTION** fields:

KEYWORDS	Type individual words that briefly describe the library. These words are used by the Keyword Search facility of TIBCO Object Service Broker. The field is one line long and can contain multiple entries, separated by commas or blanks, for example, `EXAMPLE RULES, LIBRARY`.
SUMMARY	Type a one line summary of the DESCRIPTION, for example, `Library of example rules.`
DESCRIPTION	Type information about the library, for example, what its role is, or why it is used. The information is entered in text format using TIBCO Object Service Broker SCRIPT commands, and there is no limit on the amount of information you can enter.

See Also *TIBCO Object Service Broker Shareable Tools* about SCRIPT and its commands.

Chapter 14 Using the Rule Editor

This chapter describes how to use the rule editor.

Topics

Invoking the Rule Editor

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

Steps to Invoke the Rule Editor

To invoke the Rule Editor for a new rule or an existing rule, do one of the following from the workbench:

- Type the name of your rule beside the Rule Editor option and press Enter:

  ```
  ER edit rule ==> rulename
  ```

- Enter the tool name EDITRULE, and the name of the rule in parentheses, next to the Execute Rule option and press Enter:

  ```
  EX execute rule ==> EDITRULE(rulename)
  ```

- Enter the value ER and the name of the rule next to the COMMAND option and press Enter:

  ```
  COMMAND==> ER rulename
  ```

Using any of these methods causes the Rule Editor screen to appear. If you do not enter a rule name and you are invoking the Rule Editor, the Object Manager screen for the Rule Editor appears instead. Refer toScreen Layout of the Rule Editor on page 153 for information about this screen. Refer toValid Values for Rule Names on page 151 for information about rule names.

Example Rule Editor Screen

```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------
_ JOBTITLE = 'SENIOR ANALYST';                                  | Y N N
_ JOBTITLE = 'ANALYST';                                         |   Y N
_ ------------------------------------------------------------+----------
_ RATE = 0.1;                                                   | 1
_ RATE = 0.05;                                                  |   1
_ RATE = 0.02;                                                  |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;              |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:           | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                           |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;               |
_    CALL REPLACE_SALARY(REGION);                               |
_    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||            |
_      EMPLOYEES.SALARY);                                       |
_    END;                                                       |
_ ------------------------------------------------------------------------
  ON GETFAIL:
     CALL ENDMSG('POSITION IS INVALID');

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Rule Displayed

The Rule Editor displays the source code of the rule for you to edit after searching the rules libraries in the following order:

1. Local library
2. Installation library
3. System library

This is the default order, but the search order can be modified by you or your system administrator as part of your user profile or session options.

If the rule does not exist in one of the three libraries, the Rule Editor displays a template for creating a new rule.

Valid Values for Rule Names

The following conditions apply to naming a rule. The name:

- Can be a character string of up to sixteen characters, beginning with a letter (A-Z) or a special character ($ or #), continuing with more letters, special characters, or digits (0-9), or underscore characters

The special character @ (at sign) is used by TIBCO Object Service Broker supplied objects.

- Must not contain spaces
- Must be unique to the rules library

See Also *TIBCO Object Service Broker Managing Security* and *TIBCO Object Service Broker Parameters* about modifying the search order for rules libraries.

TIBCO Object Service Broker Shareable Tools about using user exits with the Rule Editor.

Screen Layout of the Rule Editor

When a rule appears for editing it is divided into four parts. The parts, which are separated by horizontal lines, contain:

- The rules definition, including the optionally defined arguments and local variables
- Conditions, including a quadrant of yes/no (Y/N) values
- Actions, including columns of action sequence numbers
- Exception handlers

A PF key line appears at the bottom of the screen.

General Format of the Rule Editor Screen

```
RULE EDITOR ===>

           Rules Definition
     _
     _ ------------------------------------------------------
     _                                      |
     _     Conditions                       |   Y/N Quadrant
     _                                      |
     _ -------------------------------------+----------------
     _                                      |
     _     Actions                          |  Action Sequence
     _                                      |       Numbers
     _ ------------------------------------------------------
     _
     _     Exception Handlers
     _
     _


 PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Modifiable Sections

All the displayed areas can be modified except the Y/N Quadrant, which is managed entirely by the Rule Editor. You make direct modifications to a rule by overtyping or filling in the blanks in the displayed area. You make indirect modifications by issuing editing commands.

Scrolling within a Rule Editor Screen

Since a rule is restricted to the width of the screen, you need to scroll only vertically within a rule. Use the following keys to scroll:

PF7	Up
PF8	Down

The default scroll amount is one page.

Modifying the Scroll Value

To modify the scroll value, type one of the following characters into the **Scroll** field before using the scroll function keys:

H	Scroll half a page.
P	Scroll a page.
M	Scroll the maximum amount.
C	Scroll from the line where the cursor is positioned.
nnn	Scroll *nnn* lines.

To change the scroll amount temporarily, type one of the valid values in the primary command field and use a scroll key.

Accessing a Listing of Rules to Edit

Using the Object Manager Screen

If you invoke the Rule Editor without specifying a rule name, an Object Manager screen similar to the following one appears. This screen contains a listing of all the rules in your local library. You can select the rule from this listing.

```
                    List of rules to edit     LIBRARY: DOCEXMPL
Command ==>                                                      Scroll P
           Enter a primary command or one or more line commands
         NAME          DATE       TIME   UNIT                        DESCRIPTION*
   ---------------- ---------- ---- -------- ---------------------------------
  _ ABC              2000-03-20 1154 USR      Sample rule
  _ ABEND_CODES      1999-11-02 1146 USR40    GENERATED TO PRINT REPORT ABEND_C
  _ ABS_1            2000-02-13 1626 DOCUMENT Sample rule for Shareable Tools C
  _ ADDOBJSEL        1997-06-09 1130 OBJ      Add a selected object to the list
  _ CHANGE_LOCATION  1997-07-09 0838 USR40    sample rule to change @session ta
  _ CHECK_DONE       1998-03-25 0850 ACC      Sample rule for Report Writer
  _ CHECK_PARMVALU   1997-06-20 1134 TEST
  _ CHK_OVRWRITE     1999-06-16 1615 TEST     ensure no overwrite when SAVE
  _ COMMIT50         1997-12-14 1106 USR      COMMITs are applied periodically
  _ COPYCODES        1998-03-26 0830 DOCMSG   copy abend_codes to CODES table
  _ COPYCOMPONENTS   1997-11-11 1203 DOCMSG   copy COMPONENTS to UTIL table
  _ COPYUTILITIES    1997-12-06 0955 DOCMSG
  _ COUNT            1997-09-01 1453 DOCSAMP  Sample for Processing manual
  _ DEPARTMENTS      1997-06-23 0958 USR      Lists the employees in a departme
  _ EMPLOYEE_EXPENSE 1998-01-16 1405 USR      Display the employee_expense scre
  _ EMPLOYEES_RAISE  1998-02-16 0945 USR      Provides raises for selected empl
C-Copy  D-Delete  G-Debug  P-Print  S-Select  X-eXecute

PFKEYS: 12=EXIT 13=PRINT 3=END 5=FIND NEXT 9=RECALL
```

Available Commands

You can use the primary commands **SELECT**, **APPLY**, **ORDERED**, and **FIND** in the primary command field to narrow down your selection. You can also use the following line commands:

C	Copy a rule from one library to another.
D	Delete a rule from your local library. You are prompted for confirmation.
G	Invoke the debugging tool. Refer to Chapter 18, Processing in Debug Mode, on page 199 for more information.

P	Print a hardcopy of the rule.
S	Select a rule from your local library.
X	Execute the rule.

Chapter 15 Editing Rules

This chapter describes how to edit rules.

Topics

Functional Overview

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

Types of Editing Allowed

When editing rules you can do any of the following:

- Overtype text.
- Use line commands to make selective changes to one or more lines within a rule.
- Use primary commands to make changes that potentially affect the whole rule.
- Use the **EXPAND** command to display browsable information about rules, tables, fields, screens, reports, and routines.

After you edit your changes, syntax checking is performed when you attempt to save your rule using PF3 or the **END** or **SAVE** commands.

Syntax Checking Performed on a Rule

The syntax checker checks each rule for accuracy and displays appropriate error messages when corrections are necessary. A rule cannot be saved until it is syntactically correct. For detailed information on the syntax of rules, refer to Appendix A, Syntax of the Rules Language, on page 211.

A rule that is syntactically correct could contain other kinds of errors, such as misspelled names or incorrect logic, which is detected only on execution. Refer to Chapter 16, Processing in Standard Execution Mode, on page 175 for more information.

Available Line Commands and Associated PF Keys

I	PF4	Insert a line.
R	n/a	Replicate a line.
D	PF16	Delete a line.
S	n/a	Split a line.
M	n/a	Move a line
A	n/a	Move or copy a line to after the line with the A.
B	n/a	Move or copy a line to before the line with the B.
C	n/a	Copy a line.

When using line commands:

- The initial screen contains the rules definition, the conditions, the actions and the exception handlers. If you issue a line command in the first line command field of the action section, it applies to the action section.
- If the rule is longer than the screen that first appears, when you press PF8 the rules definition and a horizontal separator line between the definition and conditions of the rule appear, followed by the actions. After the screen is scrolled, the line command issued in the first line command field below the horizontal line is applied to the conditions section of the rule.

Available Primary Commands

The following commands are described in Using Available Primary Commands on page 165.

APPEND	n/a	Append a rule to another rule.
CANCEL	PF12	Leave the editor without saving changes.
CHANGE	n/a	Change a token.
CHANGE...ALL	n/a	Change all instances of a specified token.
CLOSE		Close all open windows. Refer to Expanding Token Information on page 172 for more information.

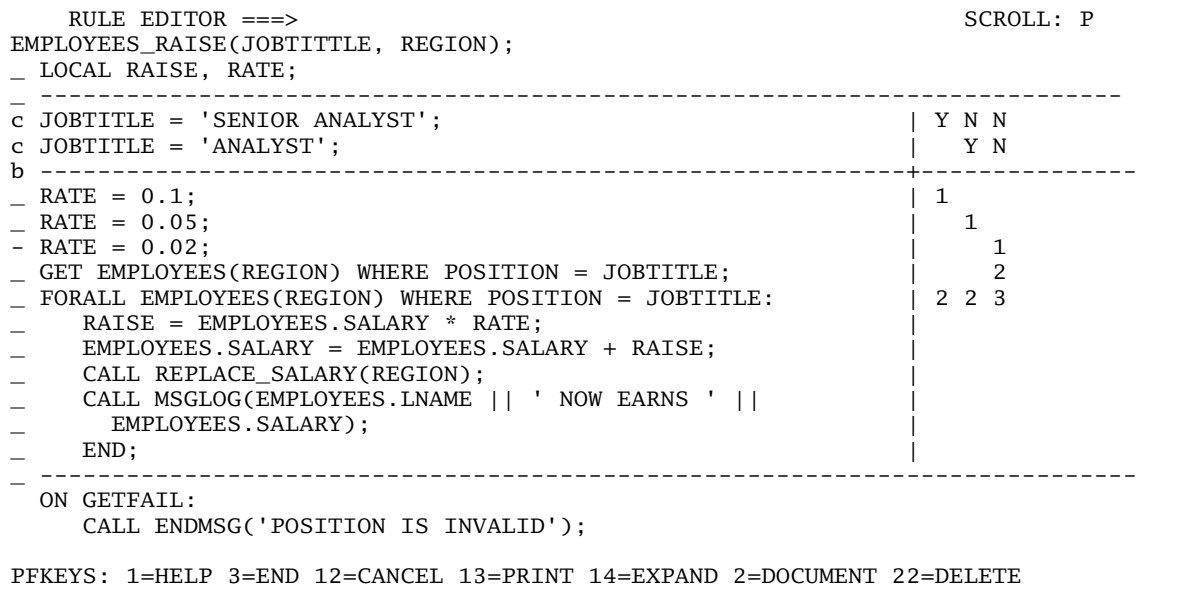
CONFIRM		Confirm a deletion.
COPY	n/a	Copy a rule.
DELETE	PF22	Delete a rule. You must confirm the deletion using the **CONFIRM** command or PF22.
DOCUMENT	PF2	Document a rule.
EDIT	n/a	Save the current rule and edit a different rule or create a new one.
END	PF3	Save the rule and exit the editor.
EXPAND	PF14	Provide expanded information for the token specified. Refer to Expanding Token Information on page 172 for more information.
FIND	n/a	Find a token.
HELP	PF1	Display online help.
LOWER	n/a	Allow the use of lowercase in a string.
PRINT	PF13	Print a hardcopy of a rule.
SAVE	n/a	Save the rule and remain in the editor.
UPPER	n/a	Allow the use of uppercase in a string.
XEDIT	n/a	Cancel the current editing session and initiate a new one.
n/a	PF5	When used in conjunction with the **CHANGE** or **FIND** commands, find the next occurrence of the token.
n/a	PF6	When used in conjunction with the **CHANGE** command, change the next occurrence of the token. When used in conjunction with a **FIND** command, change the current token to a null value.
n/a	PF9	Re-display the most recent primary command.

Using Available Line Commands

Copying Lines

The **C** line command copies a line. Type **C** in the line command field of the line to be copied and use <Enter>. The line is copied after the line where the cursor is located. You can also copy several lines at once by typing **C** in the line command fields of several lines. All the lines are copied to the destination, in the order in which they already appear. The action sequence numbers are supplied for you.

You can specify a destination with one of the destination indicators **A** (after) or **B** (before). Type **A** or **B** in the line command field of the line to be used as the reference point. Press Enter to copy the line to the specified destination. In the following screen, the two `JOBTITLE` lines are explicitly copied after the line `JOBTITLE = 'ANALYST'`.



```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------
c JOBTITLE = 'SENIOR ANALYST';                                 | Y N N
c JOBTITLE = 'ANALYST';                                        |   Y N
b ---------------------------------------------------------------+---------
_ RATE = 0.1;                                                  | 1
_ RATE = 0.05;                                                 |   1
- RATE = 0.02;                                                 |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;             |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:          | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                          |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;              |
_    CALL REPLACE_SALARY(REGION);                              |
_    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||           |
_      EMPLOYEES.SALARY);                                      |
_    END;                                                      |
_ ------------------------------------------------------------------------
  ON GETFAIL:
     CALL ENDMSG('POSITION IS INVALID');

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Deleting Lines

The **D** line command deletes lines. Type **D** in the line command field of the line and press Enter to delete the line. If the line is an action containing an action sequence number, the Rule Editor re-sequences the remaining actions.

Using PF16 has the same effect as the **D** line command; pressing PF16 while the cursor is positioned anywhere on the line deletes the line.

Inserting Lines

The **I** line command inserts an empty line. Type **I** in the line command field of a line and press Enter. This creates a new, empty line below the selected line and places the cursor in column one of the new line. You must type something in the empty line or it disappears when you press Enter.

Using PF4 has the same effect as the **I** line command. Pressing PF4 while the cursor is positioned anywhere on the line creates a blank line below the selected one.

If you are inserting lines in the action part of a rule and your rule has conditions, you must type in the action sequence numbers for the new lines or type in any character except a blank. If the rule does not have conditions, the action sequence numbers are supplied for you.

Moving Lines

The **M** line command moves a line. Type an **M** in the line command field of the line to be moved and move the cursor to the destination, then press Enter. The line is moved after the line where the cursor is located. You can also move several lines at once by typing **M** in the command fields of several lines. All the lines are moved to the destination, in the order in which they already appear. The Rule Editor re-sequences the action sequence numbers of all the lines.

You can also specify a destination with one of the destination indicators, **A** (after) or **B** (before). Type **A** or **B** in the line command field of the line to be used as the reference point. Press Enter to move the line to the destination indicated. In the following screen, the lines `RATE = 0.05` and `RATE = 0.02` are placed before `RATE = 0.1`.

```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------
_ JOBTITLE = 'SENIOR ANALYST';                                  | Y N N
_ JOBTITLE = 'ANALYST';                                         |   Y N
_ ------------------------------------------------------------+----------
b RATE = 0.1;                                                   | 1
m RATE = 0.05;                                                  |   1
m RATE = 0.02;                                                  |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;              |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:           | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                           |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;               |
_    CALL REPLACE_SALARY(REGION);                               |
_    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||            |
_      EMPLOYEES.SALARY);                                       |
_    END;                                                       |
_ ------------------------------------------------------------------------
  ON GETFAIL:
     CALL ENDMSG('POSITION IS INVALID');

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Replicating Lines

The **R** line command replicates a line. You can use this replica as a template for a similar line. Typing **R** in the line command field and pressing Enter replicates the selected line below itself. The cursor is placed in column one of the new line so that you can modify it. If you are replicating lines in the action part of a rule, the action sequence numbers are supplied for you.

Splitting Lines

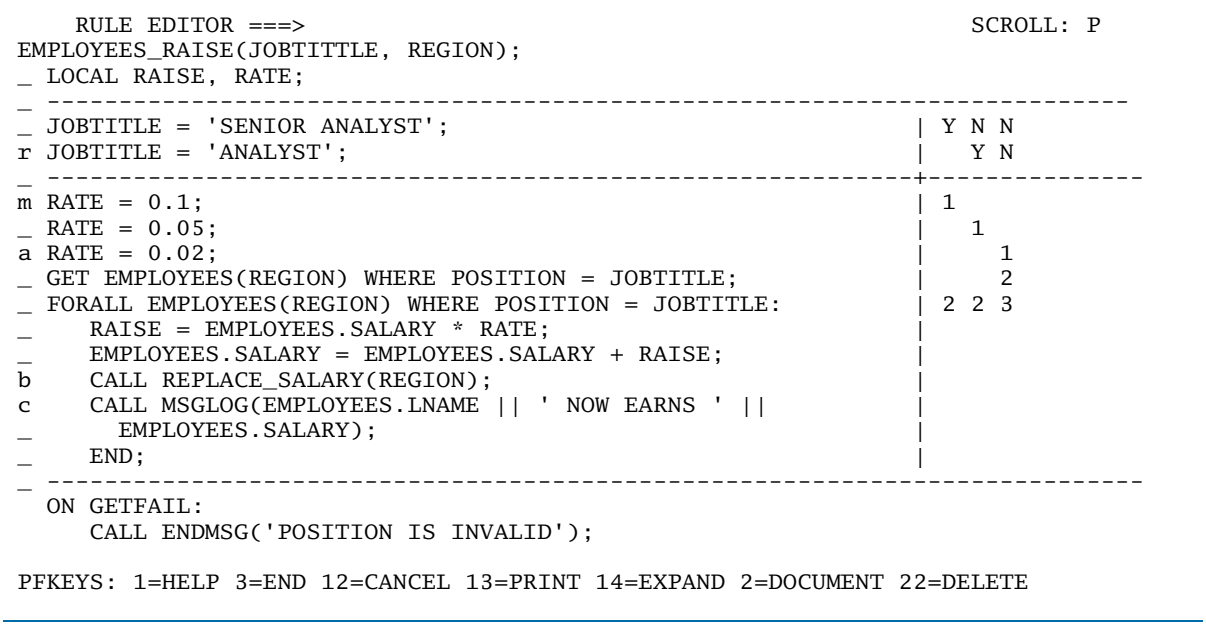
The **S** line command splits a line into two parts. Use this, for example, if a line of code is too long to fit on one line. To split a line, first type **S** in the line command field and then move the cursor to the place where you want the line to be split. When you press Enter, the Rule Editor creates a new line immediately below the current line and places everything to the right of the cursor position on the new line.

The split command is token-sensitive. When a line is split, the Rule Editor places the token where the cursor is at the start of the new line. Refer to Tokens on page 18 for the description of a token. If the cursor is between tokens, the split occurs there. If the cursor is not on the same line as the **S** command, the **S** is treated as an **I** (insert) command.

Combining Line Commands

You can enter several different kinds of line commands at once. These commands are processed in order from top to bottom. In the following screen, the line beginning with:

- `JOBTITLE ='ANALYST'` is replicated
- `RATE = 0.1` is moved after `RATE = 0.02`
- `CALL MSGLOG` is copied before `CALL REPLACE_SALARY`



```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------
_ JOBTITLE = 'SENIOR ANALYST';                                | Y N N
r JOBTITLE = 'ANALYST';                                       |   Y N
_ ------------------------------------------------------------+----------
m RATE = 0.1;                                                 | 1
_ RATE = 0.05;                                                |   1
a RATE = 0.02;                                                |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;            |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:         | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                         |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;             |
b    CALL REPLACE_SALARY(REGION);                             |
c    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||          |
_      EMPLOYEES.SALARY);                                     |
_    END;                                                     |
_ ------------------------------------------------------------------------
  ON GETFAIL:
     CALL ENDMSG('POSITION IS INVALID');

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Using Available Primary Commands

APPEND Command

The primary command **APPEND** appends a rule onto the rule currently being edited. The **APPEND** command appends the actions at the end of the action part and the exception handlers at the end of the exception handler part of the rule being edited. The rule is checked for syntax before it can be saved.

APPEND does not append local variables and conditions, and it does not add action sequence numbers for the appended action part. You must supply these as required.

Issue the **APPEND** command as shown:

```
RULE EDITOR ===> APPEND DEPARTMENTS<Enter>
```

The following screen shows the DEPARTMENTS rule appended to the EMPLOYEES_RAISE rule, after the line `END;`.

```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE(JOBTITTLE, REGION);
_ LOCAL RAISE, RATE;
_ ------------------------------------------------------------------------
_ JOBTITLE = 'SENIOR ANALYST';                                | Y N N
_ JOBTITLE = 'ANALYST';                                       |   Y N
_ ------------------------------------------------------------+----------
_ RATE = 0.1;                                                 | 1
_ RATE = 0.05;                                                |   1
_ RATE = 0.02;                                                |     1
_ GET EMPLOYEES(REGION) WHERE POSITION = JOBTITLE;            |     2
_ FORALL EMPLOYEES(REGION) WHERE POSITION = JOBTITLE:         | 2 2 3
_    RAISE = EMPLOYEES.SALARY * RATE;                         |
_    EMPLOYEES.SALARY = EMPLOYEES.SALARY + RAISE;             |
_    CALL REPLACE_SALARY(REGION);                             |
_    CALL MSGLOG(EMPLOYEES.LNAME || ' NOW EARNS ' ||          |
_      EMPLOYEES.SALARY);                                     |
_    END;                                                     |
_ FORALL DEPARTMENTS:                                         |
_    FORALL EMPLOYEES('EDUC') WHERE DEPTNO =                  |
_         DEPARTMENTS.DEPTNO:                                 |
_    CALL MSGLOG(' LAST NAME = '|| EMPLOYEE.LNAME ||          |
PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

CANCEL Command (PF12)

If you decide not to save changes to a rule, you can cancel the changes by typing the primary command **CANCEL** in the primary command field and pressing Enter, or by pressing PF12. When you issue the command, the Rule Editor cancels editing changes and returns you to your previous screen.

CHANGE Command

To change an instance of a particular token use the primary command **CHANGE**. Type in the **CHANGE** command as follows:

```
RULE EDITOR ===> CHANGE search_token replacement_token<Enter>
```

To change the first instance of the token REGION to the token LOCATION in the EMPLOYEES_RAISE rule, issue the **CHANGE** command as shown:

```
RULE EDITOR ===> CHANGE REGION LOCATION<Enter>
```

Usage Notes

- Concatenation of tokens is not supported.
- The search for the token begins immediately to the right of the cursor position (not at the beginning of the rule) and, if successful, replaces the first instance of the token and positions the cursor at the start of the replaced token. Use PF6 to find the next instance of the token and replace it.
- If you are unsure whether you want to change the next instance, you can first find the instance by pressing PF5. To replace it press PF6. To skip it and search for the next instance press PF5.
- If the search for the token is unsuccessful, a message appears on the message line informing you that the token is not found. The message means that the search failed to find an instance of the token between the current cursor position and the end of the rule. Pressing PF5 or PF6 resumes the search from the beginning of the rule.

CHANGE … ALL Command

The **CHANGE** command has an optional argument, **ALL**. This argument changes the scope of the **CHANGE** command from the next occurrence of the search token to all the occurrences of the search token in the rule. Specify the **ALL** argument as follows:

```
RULE EDITOR ===> CHANGE search_token replacement_token ALL<Enter>
```

If a **CHANGE ... ALL** command makes changes, the message line tells how many occurrences of the search token are replaced. If the search token does not occur in the rule, the message line informs you of this.

COPY Command

The **COPY** primary command causes the current rule to be replaced, except its name, with a copy of the rule specified. The **COPY** command provides a convenient way to replace one rule with another or to fill in an empty template.

Issue the **COPY** command as shown:

```
RULE EDITOR ===> COPY DEPARTMENTS<Enter>
```

The contents of the current EMPLOYEES_RAISE rule are replaced with a copy of the DEPARTMENTS rule, as shown in the following screen:

```
    RULE EDITOR ===>                                                SCROLL: P
EMPLOYEES_RAISE;
_
_ ------------------------------------------------------------------------------
_ ------------------------------------------------------------+---------------
_ FORALL DEPARTMENTS :                                        | 1
_    FORALL EMPLOYEES WHERE REGION = 'EDUC' & DEPTNO =        |
_         DEPARTMENTS.DEPTNO :                                |
_       CALL MSGLOG('LAST NAME = ' || EMPLOYEES.LNAME ||      |
_         '  DEPARTMENT =  ' || DEPARTMENTS.DEPTNO);          |
_       END;                                                  |
_    END;                                                     |
_ ------------------------------------------------------------------------------

PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

If you change the name of the rule that you are editing (without making other changes) and if there is no rule with the new name in your local library, the Rule Editor creates a copy of the original rule under the new name. If a rule with the new name already exists, you are prompted to overwrite it.

DELETE Command (PF22)

The primary command **DELETE** deletes a rule from your local library. To issue the command, type **DELETE** in the primary command field, or display the rule and press PF22.

The Rule Editor does not delete a rule without first prompting you to type the command **CONFIRM** or press PF22 to confirm the deletion. To cancel the deletion, press Enter or any other function key.

DOCUMENT Command (PF2)

The primary command **DOCUMENT** causes a screen to appear where you document the description and usage of your rule. To issue the command, type **DOCUMENT** in the primary command field, or display the rule and then press PF2.

See Also *TIBCO Object Service Broker Getting Started* about documenting rules and other TIBCO Object Service Broker objects.

EDIT Command

You can use the primary command **EDIT** to edit a different rule or to create a new rule from within an editing session. The rule being edited is checked for syntax errors, and when it can be saved the new editing session begins.

Editing a Different Rule

To edit a different rule from within an editing session, type the **EDIT** command and the name of the required rule in the primary command field, and press Enter. For example, if you are editing the EMPLOYEES_RAISE rule and you want to edit the DEPARTMENTS rule, type the **EDIT** command as shown:

```
RULE EDITOR ===> EDIT DEPARTMENTS<Enter>
```

If there are no syntax errors, the Rule Editor saves the EMPLOYEES_RAISE rule before it displays the DEPARTMENTS rule. If there are syntax errors, you must correct them before the Rule Editor can begin the editing session for the DEPARTMENTS rule.

Creating a New Rule

To create a new rule from within an editing session, type the **EDIT** command and the name of the new rule in the primary command field and press Enter.

END Command (PF3)

You can end the editing session and save the changes by typing **END** in the primary command field and pressing Enter, or by pressing PF3. If there are no syntax errors, when you issue the **END** command the Rule Editor saves the rule in the local library you are using, not in the library of origin, and returns you to the workbench. You must correct your syntax errors before the rule can be saved. Refer to Syntax Checking Performed on a Rule on page 158 for more information.

FIND Command

To find a particular token, use the primary command **FIND**. Type in the **FIND** command as follows:

```
RULE EDITOR ===> FIND search_token<Enter>
```

Usage Notes

- The search for the token begins immediately to the right of the cursor position (not at the beginning of the rule) and, if successful, positions the cursor at the start of the first occurrence of the token. Use PF5 to continue the search from the current cursor position. Use PF6 to change the current occurrence to a null value.
- If the search is unsuccessful, a message appears on the message line informing you that the search is unsuccessful. This means that the search failed to find an occurrence of the token between the current cursor position and the end of the rule. Use PF5 to resume the search from the beginning of the rule.

HELP Command (PF1)

You can display online help by typing **HELP** in the primary command field or by pressing PF1. The online help contains a description of the Rule Editor and a list of the Rule Editor commands and their syntax. To return to the rule, press PF3 or PF12.

LOWER Command

You can use the primary command **LOWER** to type string literals in mixed case (that is, text within single quotation marks). If you are editing an existing rule, you must issue the primary command **LOWER** and press Enter before the lowercase text is typed in. Existing lowercase text remains in lowercase when it is edited.

PRINT Command (PF13)

To print the current rule, press PF13 or type **PRINT** in the primary command field and press Enter. The hardcopy output includes some of the documentation for the rule. The documentation that is included appears between the rule definition and the conditions.

SAVE Command

Rules are not actually updated in your local library until you save your changes. You can save changes by typing **SAVE** in the primary command field and pressing Enter. The Rule Editor saves the rule in the local library you are using, not in the library of origin, and continues the editing session.

If the Rule Editor cannot save a rule, it gives a reason for the failure on the message line and positions the cursor where the first correction is required. You must correct the rule before you can save the changes. Refer to Syntax Checking Performed on a Rule on page 158 for more information.

UPPER Command

If you issue the primary command **LOWER** and you want to revert back to using uppercase for your string literals, type **UPPER** in the primary command field and press Enter.

XEDIT Command

You use the **XEDIT** command to edit an existing or new rule within an editing session. The difference between **EDIT** and **XEDIT** is that **XEDIT** cancels changes made to the rule currently being edited.

To cancel editing changes made to EMPLOYEES_RAISE and start editing the DEPARTMENTS rule, type the **XEDIT** command as shown:

```
RULE EDITOR ===> XEDIT DEPARTMENTS<Enter>
```

Redisplay the Most Recent Primary Command–(PF9)

The Rule Editor stores a command until it is replaced with another command. PF9 redisplays the most recent command, which you can edit if necessary. For example, suppose that the previous command was the following:

```
RULE EDITOR ===> CHANGE REGION LOCATION ALL<Enter>
```

You now want to change the token DEPT to the token DEPARTMENT#. Re-display the previous command by pressing PF9. Place the cursor where editing is required (on the token REGION) and alter the command by overtyping, as follows:

```
RULE EDITOR ===> CHANGE DEPT DEPARTMENT# ALL<Enter>
```

Expanding Token Information

While editing a rule, you can display information in browse format for the following types of tokens: rules, tables, fields, screens, reports, and routines. Selected information appears for each token type. You can expand a token named in the rule using a function key or primary command, or expand on any allowable token type not named within the rule using a primary command.

A routine is an TIBCO Object Service Broker shareable tool that is not written in the rules language.

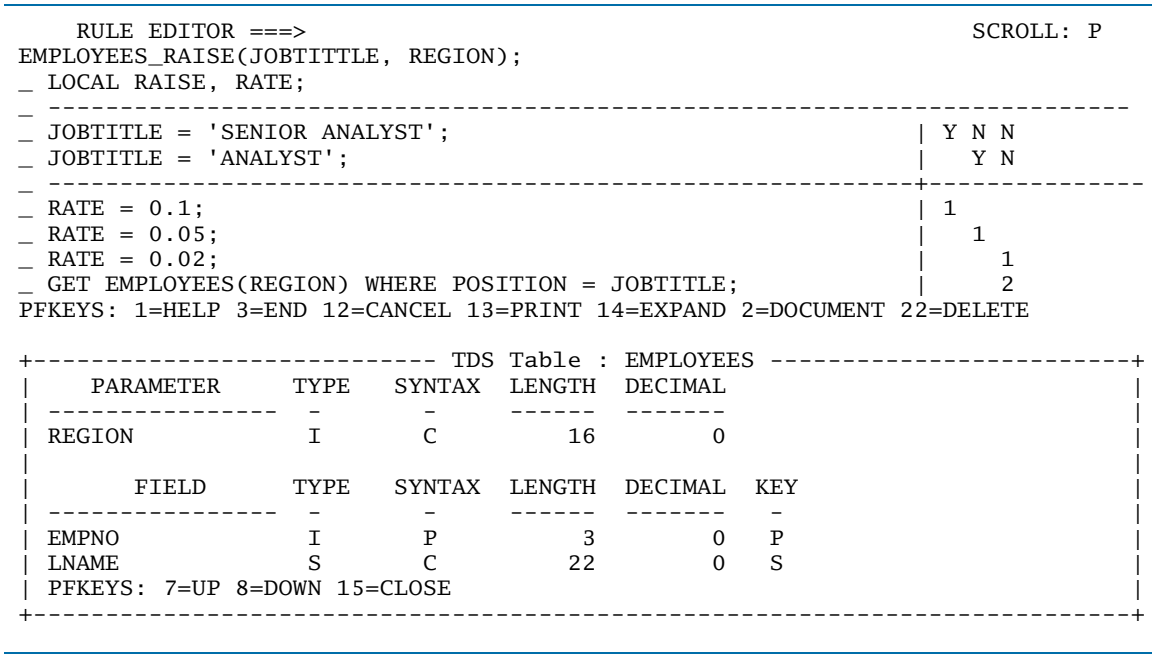
Types of Token Information Available

The following list outlines what kind of information appears, based on the type of token:

Rule	The name of the rule and its complete contents
Table	The name of the table, and the names and attributes of its parameters and fields
Field	The name of the field and its attributes If the field is a global field (a field defined in the centralized field dictionary table in your TIBCO Object Service Broker database), a summary of global usage appears. If the field is a referenced field, a listing of reference values appears.
Screen	The name of the screen, and the names of its screen tables and some of their attributes. Within the display, the screen tables can be expanded to the level of screen fields.
Report	The name of the report, and the names of its report tables and some of their attributes. Within the display, the report tables can be expanded to the level of report fields.
Routine	The name of the routine, the arguments of the routine, its keywords, and a summary

Expanding a Token at the Cursor Position

Positioning your cursor on a token name and pressing PF14 causes the screen to split horizontally into two portions. As shown in the following screen, the upper portion displays the rule being edited and the lower portion displays information about the selected token in a browsable format. To scroll in either portion, position your cursor in the desired portion and press PF7 to scroll up or PF8press to scroll down.



Displaying Rule, Routine, or Table Information

To display a rule, routine, or table, position your cursor on the desired token name and press PF14. Because the information appears in browse mode, you can also display the rule you are currently editing.

Displaying Screen Information

To display screen information, the cursor must be positioned on the screen name within a DISPLAY statement. After positioning the cursor, press PF14.

Displaying Report Information

To display report information, the cursor must be positioned on the report name within a call to the $SETRPTMEDIUM or $RPTPRINT tools or within a PRINT statement. After positioning the cursor, press PF14.

Expanding With the Primary Command EXPAND

The primary command **EXPAND** also invokes the expand facility. You specify **EXPAND** using the format:

EXPAND *token_name token_type*

token_ type is optional and can be one of the following: RULE, REPORT or RPT, SCREEN or SCR, TABLE or TBL. If no type is specified the search hierarchy is: reports, screens, tables, rules, and routines. The first match found is expanded within a display area.

The following shows an example of the command **EXPAND**:

```
RULE EDITOR ===> EXPAND EMP_EXPENSE RPT<Enter>
```

Nesting Expanded Displays

You can expand more than one token at a time. This is known as nesting. To nest an expanded display, position the cursor on the desired object within the display and press PF14.

Nesting expanded displays is permitted for rules, screens, and reports. Within screens you can also nest screen tables, and within reports you can also nest report tables. Nesting is not permitted from within a routine, table, or field.

Closing the Expanded Display

To close an expanded display and exit to the previous display, press PF15. If you press PF15 in the top-most display, it closes the expand facility. If you are in a lower nested level, press PF15 outside of the expanded display to exit the expand facility, or type in the command **CLOSE** in the primary command field and press Enter. Either method closes all open displays.

See Also *TIBCO Object Service Broker Managing Data* about tables.

TIBCO Object Service Broker Defining Reports about reports.

TIBCO Object Service Broker Defining Screens and Menus about screens.

TIBCO Object Service Broker Shareable Tools about the use of tools, including routines.

Chapter 16 Processing in Standard Execution Mode

This chapter describes how to do processing in standard execution mode.

Topics

Using Execute Rule

TIBCO Object Service Broker UI

This chapter describes how to perform various tasks in TIBCO Object Service Broker using the text-based workbench. You can also perform these tasks using the TIBCO Object Service Broker UI, which provides a graphical environment for TIBCO Object Service Broker development. For information about using the TIBCO Object Service Broker UI, refer to the TIBCO Object Service Broker UI online help.

Available Methods

You can invoke the Rule Executor, by using the Execute Rule option from one of the following places:

- The workbench. For example:

  ```
  EX execute rule ==> EMPLOYEES_RAISE<Enter>
  ```

- The primary command field. For example:

  ```
  COMMAND ==> EX EMPLOYEES_RAISE<Enter>
  ```

- The line command **x** within the Object Manager screen of the Rule Editor

You can also execute a rule asynchronously or in debug mode. For more information about these methods, refer to Chapter 17, Processing Asynchronously in Batch Mode, on page 191 and Chapter 18, Processing in Debug Mode, on page 199.

Supplying Argument Names

You can provide the arguments within parentheses after the rule name in the command itself. For example:

```
COMMAND ===> EX EMPLOYEES_RAISE('ANALYST','MIDWEST')<Enter>
```

or through the arguments screen presented if you do not supply argument values.

The mode of your 3270 terminal session determines the maximum number of arguments that you can enter through the arguments screen. For example, you can enter a maximum of twelve arguments in a Model 5 3270 terminal session.

Using the Arguments Screen

If the rule has arguments defined and you do not specify values when you invoke the Rule Executor, you are prompted to supply them. For example, because the EMPLOYEES_RAISE rule has two argument, JOBTITLE and REGION, the following screen prompts for two argument values.

```
---------------------------- RULE EXECUTION -------------------------

ENTER ARGUMENTS FOR RULE EMPLOYEES_RAISE

JOBTITLE         ===>

REGION           ===>
```

Library Search Order

If the specified rule exists in your local library, installation library, or system library, the rule is executed. The rules libraries are searched in the order: local library, installation library, system library. This is the default order; you or your system administrator can modify this search order through your user profile or session options.

If the rule is executed from a menu or object list, the search order for the rule can also be set by the tool used to create the menu, such as DEFINE_MENU and DEFINE_OBJLIST.

Results After Execution

After the rule has executed, one of the following types of messages appears across the bottom of the workbench:

- The ENDMSG, if there is one, for example:

  ```
  POSITION IS INVALID
  ```

- The message OK, if the rule executes successfully, for example:

  ```
  11:02:18 OK
  ```

  If a message is sent to the Message Log, press PF2 to access the message.

- A reason for execution failure, if the rule does not execute successfully, for example:

  ```
  ACCESS ERROR ON TABLE "EMPLOYEES"
  ```

  Press PF2 for more information about the error. Information about the failure is sent to one of the logs maintained by TIBCO Object Service Broker. Refer to Logging of Output on page 181.

See Also *TIBCO Object Service Broker Defining Screens and Menus* about defining menus.

TIBCO Object Service Broker Managing Security and *External Environments* manuals about modifying the search order for rules.

TIBCO Object Service Broker Shareable Tools about tools.

Accessing a Listing of Rules to Execute

Using the Object Manager Screen

If you invoke Execute Rule without specifying a rule name, the Object Manager screen appears. This screen contains a listing of all the rules in the local library you are using and you can select your rules from this library listing.

```
                    List of rules to execute     LIBRARY: DOCEXMPL
Command ==>                                                      Scroll P
           Select one or more rule or enter a primary command
         NAME         DATE      TIME   UNIT                        DESCRIPTION*
   ---------------- ---------- ---- -------- ---------------------------------
  _ ABC              2000-03-20 1154 USR      Sample rule
  _ ABEND_CODES      1999-11-02 1146 USR40    GENERATED TO PRINT REPORT ABEND_C
  _ ABS_1            2000-08-13 1626 DOCUMENT Sample rule for Shareable Tools C
  _ ADDOBJSEL        1997-06-09 1130 OBJ      Add a selected object to the list
  _ CHANGE_LOCATION  1997-07-09 0838 USR40    sample rule to change @session ta
  _ CHECK_DONE       1998-03-25 0850 ACC      Sample rule for Report Writer
  _ CHECK_PARMVALU   1997-06-20 113  TEST
  _ CHK_OVRWRITE     1999-06-16 1615 TEST     ensure no overwrite when SAVE
  _ COMMIT50         1997-12-14 1106 USR      COMMITs are applied periodically
  _ COPYCODES        1998-03-26 0830 DOCMSG   copy abend_codes to CODES table
  _ COPYCOMPONENTS   1997-11-11 1203 DOCMSG   copy COMPONENTS to UTIL table
  _ COPYUTILITIES    1997-12-06 0955 DOCMSG
  _ COUNT            1997-09-01 1453 DOCSAMP  Sample for Processing manual
  _ DEPARTMENTS      1997-06-23 0958 USR      Lists the employees in a departme
  _ EMPLOYEE_EXPENSE 1998-01-16 1405 USR      Display the employee_expense scre
  _ EMPLOYEES_RAISE  1998-02-16 0945 USR      Provides raises for selected empl
E-EDIT     G-Debug  S-Select

PFKEYS: 12=EXIT 13=PRINT 3=END 5=FIND NEXT 9=RECALL
```

Available Commands

E	Invoke the Rule Editor for the rule.
G	Invoke the Rule Debugger for the rule.
S	Execute the rule.

You can use the primary command **SELECT** to create a shorter list of rules, using selection criteria based on any of the fields in the display. To select a rule, type **S** in the line command field of the required rule. If the rule has arguments, you are prompted for them when you press Enter.

Logging of Output

Message Logs Available

TIBCO Object Service Broker maintains two message logs to log user and system output: a user log and a system log. Refer to User Log on page 183 and System Log on page 185 for information about these logs. To access the message logs, press PF2 from the workbench. Messages from the system log appear first; pressing PF2 again displays the user log. If there is no message in the system log, the user log appears after the first PF2 command.

Message Log Primary Commands

From the command line at the top of the message log screen, you can issue commands to:

- Print the log.
- Find a string.

PRINT Command

To print either the user or system log, type the following in the primary command field:

```
COMMAND ===> PRINT
```

while you view the log you want to print.

FIND Command

To find a string, use the **FIND** command, which you can abbreviate as **F**. Follow the **FIND** command with the string that you want to find. If the string contains blanks, enclose the string in single quotation marks. For example, if you want to find the word “message”, type in this command:

```
f message
```

If you want to find the phrase “error message log”, type in the command:

```
f 'error message log'
```

In these two examples, the first occurrence of the string is found regardless of what case it is in. “Message”, “MESSAGE”, and “message” all satisfy the search. If you want to limit the search, you can specify case sensitivity with the keyword CASE, which you can shorten to C. If you issue the command:

```
f Message c
```

the first occurrence of the word “Message” with a capital M is found.

PF Keys for the Message Logs

PF1	Display the Help screen.
PF2	Display the user log (if you are looking at the system log).
PF3	Exit the message log.
PF5	Repeat the search after a FIND command is issued.
PF7	Scroll up.
PF8	Scroll down.
PF10	Scroll left.
PF11	Scroll right.
Pf12	Exit the message log.
PF13	Print the log that you are viewing.

User Log

The user log contains any messages generated by a rule, using tools such as MSGLOG or $PRINTLINE. The messages can be generated by a parent transaction and its descendant (child) transactions. Refer to Nesting Transactions on page 108 for more information about parent and child relationships.

Sample User Log

The following screen shows an example message sent to the user log by the MSGLOG tool.

```
------------------------- INFORMATIONAL MESSAGE LOG -------------------------
COMMAND ===>                                                  SCROLL ===> P
HRODEK NOW EARNS 745.50
CANNON NOW EARNS 735.00
BOIVIN NOW EARNS 745.00









PFKEYS: 2=NEXT LOG 3=EXIT 5=REPEAT 12=EXIT 13=PRINT
```

What Output is Available?

Output from previous descendant transactions (at the same descendant level) is cleared when a new transaction is started. Only output from the most recent transaction, at a given descendant level, is available to display.

Output from the EXECUTE and TRANSFERCALL Statements

The statement that you use determines whether output is maintained:

- If you use the EXECUTE statement more than once in a rule (parent transaction) to invoke other rules (child transactions), the output from each of the parent transactions is maintained.

  This output appears sequentially in the first part of the message log. The messages of the last child transaction to be executed, at each descendant level, appear next.

- If you use the TRANSFERCALL statement, the messages from the sibling transactions are maintained.

See Also *TIBCO Object Service Broker Shareable Tools* about using the tools.

System Log

When a rule fails in its execution and the exception is not handled, error messages and debugging information for that transaction (and parent transactions) are supplied in the system log. Using the information from the system log, you can make corrections to your rule so that it can execute successfully. If you require more debugging capabilities, you can use the Rule Debugger. Refer to Chapter 18, Processing in Debug Mode, on page 199 for more information.

Sample Rule

The following is an example of a rule that could produce the system log information shown in Sample System Log on page 186.

```
     RULE EDITOR ===>                                                SCROLL: P
NEWMANAGER(YEAR, DEPT#);
_
_ ------------------------------------------------------------------------------
_ YEAR < 96;                                                    | Y N N
_ YEAR > 97;                                                    |   Y N
_ --------------------------------------------------------------+--------------
_ CALL ENDMSG('95 IS THE EARLIEST VALID YEAR');                  | 1
_ CALL ENDMSG('THIS AN INVALID YEAR');                          |   1
_ FORALL EMPLOYEES WHERE REGION = 'MIDWEST' & DEPTNO = DEPT#    |     1
_      :                                                        |
_    EMPLOYEES.MGR# = EMPLOYEES.EMPNO;                          |
_    REPLACE EMPLOYEES;                                         |
_    END;                                                       |
_ ------------------------------------------------------------------------------




PFKEYS: 1=HELP 3=END 12=CANCEL 13=PRINT 14=EXPAND 2=DOCUMENT 22=DELETE
```

Sample System Log

```
------------------------- INFORMATIONAL MESSAGE LOG -------------------------
 COMMAND ===>                                                  SCROLL ===> P
 Error detected in rule "NEWMANAGER" at action 3 FORALL statement 2
 Access error on TABLE "EMPLOYEES"
 REPLACE  EMPLOYEES
 Invalid number of parameters supplied for table "EMPLOYEES"
 Traceback of rules called at time of error
   NEWMANAGER(96,10)
 End of traceback
 ------------------------------------------------------------------------------
 No local variables
 Dump of active tables
 Table EMPLOYEES
   EMPNO = 44385
   LNAME = 'SOUZA'
   POSITION = 'SALES'
   MGR# = 44385
   DEPTNO = 10
   SALARY = 719.00
   ADDRESS = ''
   CITY = ''
   STATE_PROV = ''
 PFKEYS: 2=NEXT LOG 3=EXIT 5=REPEAT 12=EXIT 13=PRINT 4=PRS
```

What Output is Available?

The system log is cleared if the transaction ends and a new transaction begins (for example, as with the TRANSFERCALL statement). Only output from the most recent transaction is available to display. The following types of output appear:

- Location of the error
- The reason the rule stopped executing
- A traceback of the rules called in the transaction
- List of active local variables
- List of the buffers for active tables
- List of last sixteen rules called

Event Logging

The system log can also contain an event log for event rules. This information identifies the location, cause, and dump of active tables and local variables at the time of an error in a derivation, trigger, or validation rule for a table. The following is an example of an event log.

```
=========== Trigger for table "DEPARTMENTS" ==== level 1 ==========
    Trigger rule "INSERTDEPTNO" for table "DEPARTMENTS" failed at action 3
    Traceback of rules called at time of error
      INSERTDEPTNO
    End of traceback
    ------------------------------------------------------------
    No local variables
    Dump of active tables
    Table DEPARTMENTS
      DEPTNO = 30
      DEPTNAME = 'CUST SUPPORT'
     End of table dump




PFKEYS: 2=NEXT LOG 3=EXIT 5=REPEAT 12=EXIT 13=PRINT 4=PRS
```

Examples of System Log Information

The following sections contain samples of the different types of output available. They use examples from the sample rule shown in System Log on page 185.

Location of the Error

```
Error detected in rule “NEWMANAGER” at action 3 FORALL
statement 2
```

In the rule called NEWMANAGER, the error occurred at the second line in the FORALL loop, and the FORALL statement is the third statement (or action) in the rule. The action sequence numbers are not referenced.

If the error occurs in the exception handler part of the rule, this is explained. The statement numbers begin counting from 1 for each exception handler.

Reason the Rule Stopped Executing

```
Access error on TABLE "EMPLOYEES"
REPLACE EMPLOYEES
Invalid number of parms supplied for table "EMPLOYEES"
```

Traceback of the Rules

```
Traceback of rules called at time of error
NEWMANAGER(96,10)
End of traceback
```

If the rule had been called by another rule, both rule names would appear in this section, starting with the most recently called rule. The traceback lists only the rules called that lead to the point of error.

List of Active Local Variables

```
No local variables
```

If local variables had been assigned values, messages similar to the following could appear:

```
Dump of local variables
X = 10
```

If the same local variable is declared in several rules, the value of the variable in the rule closest to the failure is the value displayed in the dump.

List of the Buffers for Active Tables

```
Dump of active tables
Table EMPLOYEES
EMPNO = 44385
LNAME = 'SOUZA'
POSITION = 'SALES'
MGR# = '44385'
DEPTNO = 10
SALARY = 719.00
ADDRESS = ''
CITY = ''
STATE_PROV = ''
ZP_CODE = ''
HIREDATE = *NULL*
End of table dump
```

Up to the Last Sixteen Rules Invoked

```
Last 16 rules called
NEWMANAGER
End of call list
```

The rules invoked in this transaction are listed starting with the rule that failed and proceeding in order from the most recently invoked rule to the least recently invoked rule.

If the failed transaction has parent transactions, this information is repeated for each transaction level, starting with the first-level transaction. Information for the next transaction level is indicated by a message similar to:

```
Transaction at depth 2
```

Chapter 17 Processing Asynchronously in Batch Mode

This chapter describes how to do processing asynchronously in batch mode.

Topics

Functional Overview

Batch Processing Options Available

Using the SCHEDULE statement from within a rule, you can do one of the following to process rules asynchronously in batch:

- Submit JCL, a Windows batch program, or a Solaris script to process your rule immediately.
- Send a rule to a batch queue by using the TO clause, if you are using TIBCO Object Service Broker for z/OS.

The SCHEDULE statement causes an instance of the @SCHEDULEMODEL table to be scheduled for processing in batch. The instance that is scheduled is determined by your rule specifications.

JCL, Batch Programs, and Scripts Provided

Operating system specific default instances of @SCHEDULEMODEL are provided with TIBCO Object Service Broker and are customized for your site when TIBCO Object Service Broker is installed. For more information, refer to Editing @SCHEDULEMODEL on page 196.

Dynamic Creation of an Execution Environment

Through the use of variable substitution in the @SCHEDULEMODEL instances, you can dynamically create the environment where to run the scheduled rule. The session parameter variables are enclosed in delimiters such as braces ({ }), for example {LIBRARY} and {RULE}. You can specify values for the variables directly or as a reference to a TIBCO Object Service Broker table.

The Execution Environment parameters VARLDELIMITER and VARRDELIMITER determine the delimiters that are to be used.

Precedence of Processing Values

Values that you provide in your @SCHEDULEMODEL instance scheduled for processing take precedence over current interactive session attributes and any attributes you specify in the SCHEDULE statement.

See Also *TIBCO Object Service Broker Application Administion* for the setup required to allow users to reference values in a table for @SCHEDULEMODEL.

Installing and Operating manual for your operating environment about customizing instances of @SCHEDULEMODEL.

TIBCO Object Service Broker Parameters about session and Execution Environment parameters.

Scheduling for Direct Batch Processing

How to Schedule a Rule for Direct Batch Processing

To schedule a rule for direct batch processing, use the SCHEDULE statement in your rule without the TO clause. This causes an instance from the table @SCHEDULEMODEL to be submitted for processing when the rule is run. The instance submitted can contain any z/OS JCL, Windows batch program, or Solaris script. If the JCL, batch program, or script invokes TIBCO Object Service Broker, the Data Object Broker must be operational.

Instance Sent for Processing

The @SCHEDULEMODEL table is parameterized by the data parameters OPERATING_SYSTEM and MODELNAME. By default the system searches for the @SCHEDULEMODEL instance that matches your operating system environment and has the name of the scheduled rule as the value for the model name. If this instance does not exist, the *DEFAULT* model for your operating environment, which is one of the models shipped with TIBCO Object Service Broker, is sent for processing.

Example of Scheduling for Direct Processing

The following is an example of the SCHEDULE statement scheduling an instance of the @SCHEDULEMODEL table:

```
   RULE EDITOR ===>                                                  SCROLL: P
SCHED_RPT25;
 _
_------------------------------------------------------------------------------
 _ ------------------------------------------------------------+---------------
 _ SCHEDULE INCOME_TAX;                                       | 1
 _ ------------------------------------------------------------------------------
```

About the Example

When you execute the SCHED_RPT25 rule, a search is made for the table instance @SCHEDULEMODEL(*operating system*, INCOME_TAX). If it is found, TIBCO Object Service Broker submits it; otherwise, TIBCO Object Service Broker submits the instance @SCHEDULEMODEL(*operating system*, *DEFAULT*) for processing.

Scheduling a Batch Queue in TIBCO Object Service Broker for z/OS

How to Schedule to a Batch Queue

In the TIBCO Object Service Broker for z/OS, you can use the TO clause in the SCHEDULE statement to schedule and send a rule to a queue that you specify. The definition of the queue determines when the batch job is submitted. You can specify rules settings and output settings for a queued rule by using the TIBCO Object Service Broker tools $BATCHOPT or BATCH before the SCHEDULE statement.

Example of Scheduling to a Queue

The following is an example of using the TO clause in the SCHEDULE statement:

```
   RULE EDITOR ===>                                                    SCROLL: P
SCHED_RPT01(MONTH);
 _
 _ ------------------------------------------------------------------------------
 _ ------------------------------------------------------------+-------------
 _ SCHEDULE TO 'MONTH_END' MONTHLY_REPORT(MONTH);               | 1
_ ------------------------------------------------------------------------------
```

In this example, when you execute the SCHED_RPT01 rule, TIBCO Object Service Broker sends the MONTHLY_REPORT rule with the argument MONTH to the MONTH_END queue.

Features Available with the BATCH Tool

If you schedule a rule to a queue, these features are available to you through the BATCH tool:

- You can view a list of available queues and their definitions.
- When your rule is scheduled to a queue, you can check the status of your batch job.
- You can remove a job from the queue.
- You can create JCL for your batch job (and specify the JCL before the SCHEDULE statement with the BATCH tool).

See Also *TIBCO Object Service Broker for z/OS Installing and Operating* about defining queues, and *TIBCO Object Service Broker Shareable Tools* about the tools.

Editing @SCHEDULEMODEL

Provided Default Instances

You can copy and modify the default instances of the @SCHEDULEMODEL table that are provided with TIBCO Object Service Broker. The following generic default instances are provided for general usage:

- @SCHEDULEMODEL(MVS,*DEFAULT*)
- @SCHEDULEMODEL(NT,*DEFAULT*)
- @SCHEDULEMODEL(UNIX,*DEFAULT*)

Other instances are provided for specific tasks such as printing and for promotions.

About the Default JCL

The default JCL, @SCHEDULEMODEL(MVS,*DEFAULT*), executes the program S6BBATCH. This program runs a new TIBCO Object Service Broker Execution Environment where the rule can be processed asynchronously.

Example of Customized JCL

The following is an extract of customized batch JCL:

```
_     60 //HURON EXEC PGM=S6BBATCH,REGION=4096K,
_     80 //PARM=('TDS={TDS},RULE=INCOME_TAX')
_     91 //HRNIN    DD *
_     92  {TEST},
_     93  SEA={SEARCH},
_     94  C={CHARSET},
_     95  L=REPORT
_     96 /*
```

About the Example

The customized JCL specifies the REPORT local library and the INCOME_TAX rule. This ensures that when the INCOME_TAX rule is scheduled for batch processing the local library is always REPORT and the rule that is executed is always INCOME_TAX.

About the Default Windows Batch Program and Solaris Script

The default batch program, @SCHEDULEMODEL(NT,*DEFAULT*) and the default script, @SCHEDULEMODEL(UNIX,*DEFAULT*), execute the utility osBatch. This utility starts a new Execution Environment where the rule can be processed asynchronously.

As well, the following @SCHEDULEMODEL table instances for “NT” and for “UNIX” should not be modified: *DEFAULT*, APPLY_CHANGE, APPLY_PROM, TREE, XRF. These instances should contain the following occurrences:

Windows

Number	Card
10	OSBATCH DOB={DOB} "R={RULE}({PARM})" \
20	U={USERID} P={PASSWORD} \
30	EENAME={EENAME} \
40	L={LIBRARY} I={INSTLIB} \
50	SEA={SEARCH} AC=T \
60	TEST={TEST} BROWSE={BROWSE} \
70	SESSIONLOGCLEAR={SESSIONLOGCLEAR} \
80	MSGLOGMAX={MSGLOGMAX} \
90	TRANMAXNUM={TRANMAXNUM} \
100	CHARSET={CHARSET} \
110	"PRINTDEST={DEST}" \
120	DSBIFTYPE={DSBIFTYPE} \
130	DSIXFTYPE={DSIXFTYPE} \
140	DSFIELDSEP={DSFIELDSEP} \
150	"DSDIR={DSDIR}"

Solaris

Number	Card
10	osBatch DOB={DOB} "R={RULE}({PARM})" \
20	U={USERID} P={PASSWORD} \
30	EENAME={EENAME} \
40	L={LIBRARY} I={INSTLIB} \
50	SEA={SEARCH} AC=T \
60	TEST={TEST} BROWSE={BROWSE} \
70	SESSIONLOGCLEAR={SESSIONLOGCLEAR} \
80	MSGLOGMAX={MSGLOGMAX} \
90	TRANMAXNUM={TRANMAXNUM} \
100	CHARSET={CHARSET} \
110	"PRINTDEST={DEST}" \
120	DSBIFTYPE={DSBIFTYPE} \
130	DSIXFTYPE={DSIXFTYPE} \
140	DSFIELDSEP={DSFIELDSEP} \
150	"DSDIR={DSDIR}" &

Editing Instances of @SCHEDULEMODEL

You use the Table Editor to create new instances or modify existing instances of the @SCHEDULEMODEL table.

When editing an instance of the @SCHEDULEMODEL table:

- If possible, use a Model 5 3270 terminal (or its equivalent, if you are emulating a 3270 terminal) to edit the table. If you do not use a Mod 5 terminal, you must edit the table with the Single Occurrence Editor.
- Data entered into the table is case sensitive.
- Data in the Card field of the @SCHEDULEMODEL table is limited to 72 characters. A command in the table can be continued over several lines by including a continuation character (\) at the end of the line. TIBCO Object Service Broker removes this character to pass a single line to the operating system.
- Windows commands are limited to 2046 characters.
- Solaris systems work more efficiently if you put an ampersand (&) at the end of the last line.

See Also

TIBCO Object Service Broker for z/OS Installing and Operating for details about the instances of @SCHEDULEMODEL supplied with TIBCO Object Service Broker.

TIBCO Object Service Broker for z/OS External Environments about the parameters for S6BBATCH. Also refer to this manual about the PARM statement and the HRNIN DD statement, both of which are used in this sample JCL.

TIBCO Object Service Broker for Open Systems Utilities about the parameters for osBatch.

TIBCO Object Service Broker Managing Data about using the Table Editor.

Chapter 18 Processing in Debug Mode

This chapter describes how to do processing in debug mode.

Topics

Using the Rule Debugger

Processing Rules in Debug Mode

Using the Rule Debugger, you can process your rules in debug mode. In debug mode, you can view and modify the execution state of a rules program. You can choose particular events, called break events, that are of interest to you and have the Debugger suspend the execution of the rules program when these events occur. When execution is suspended, you can issue commands to examine the state of execution or change the values of:

- Fields
- Parameters
- Local variables

and then instruct the Debugger to resume the execution of the rule.

How to Invoke and Navigate the Rule Debugger

Invoke the Debugger from one of the following places:

- The workbench:

  ```
  DB debug rule ==> rulename<Enter>
  ```

  If you do not specify a rule, the Object Manager screen for the Debugger appears. Refer to Chapter 16, Processing in Standard Execution Mode, on page 175 for more information about using the Object Manager.

- The EX execute rule workbench option:

  ```
  EX execute rule ==> Debug(rulename)<Enter>
  ```

- The primary command field:

  ```
  COMMAND ==> DB rulename<Enter>
  ```

- Using the line command **G** within the Object Manager screen of the Rule Editor or the Rule Executor.

Main Screen of the Rule Debugger

Any of those methods display the following screen:

```
COMMAND ===>


 -------------------------------Debug Information-----------------------------
 Starting TRANSFERCALL to rule EMPLOYEES_RAISE









 ---------------------------- Breakevent Control--------------------------------
 Break Event      Object name




 PFKEYS: 1=HELP 3=SAVE BREAKS  6=DBGCOM 12=EXIT 14=EXPAND 15=GO 22=REMOVE BREAK
```

Layout of the Rule Debugger Screen

The Rule Debugger screen is divided into three sections:

- The primary command line
- The Debug Information section, which displays the execution state of the rule in browse mode as the rule is being debugged
- The Breakevent Control section, which displays where the Debugger is to suspend the execution of your rules program

PF Keys and Commands for the Rule Debugger Screen

The following PF keys and primary commands are available from within the Debugger screen:

PF Key	Command	Description
n/a	**AT**	Stop the execution of the rule at the event specified. Refer to How to Specify Break Events on page 204 for more information.
n/a	**RUN**	Ignore the break event for the current transaction.
n/a	**STOPAPPLICATION**	Exit the application and return to the starting point for the debugging session (for example, workbench, Object Manager for the Rule Editor).
n/a	**STOPDEBUGGER**	Stop the debugging process and let the application finish processing.
PF2	n/a	Display the message logs for additional information.
PF3	n/a	Save the current break events that are defined and exit one transaction level of the application that is being debugged.
PF4	n/a	Save the current break events and remain in the Debugger.
PF5	**GO**	Continue executing.
PF6	**DBGCOM**	Go to the Debugger Command Editor screen. The cursor must be positioned on a break event. Refer to How to Examine and Modify the Execution State on page 207 for more information.
PF9	n/a	Re-display the last command typed in the primary command field.
PF12	**EXIT**	Exit the Debugger. Changes made since you last pressed PF4 persist until you exit your TIBCO Object Service Broker session.

PF Key	Command	Description
PF13	n/a	Print the Debugger screen.
PF14	**EXPAND**	Expand the rule that is currently being executed by the Debugger. The action statement that is about to be performed is highlighted.
PF15	**GO**	Continue executing.
PF17	**DUMP CALLSTACK**	Display the execution stack.
PF18	**DUMP LASTCALLED**	Display a list of the last 16 rules called.
PF19	**DUMP LOCALS**	Display a list of all active local variables and their values.
PF20	**DUMP TABLES**	Display a list of all active tables and the field values being currently accessed.
PF22	**OFF**	Delete the break event where the cursor is positioned.

How to Specify Break Events

Methods Available to Specify Break Events

There are two ways that you can specify break events to suspend the execution of your rule:

- In the primary command field, issue the command **AT** in combination with the event of your choice. If you use the **AT** command, specify a break event using the format:

  AT *event qualifier*

- In the Breakevent Control section, type in one or more events. On each line specify a break event and a qualifier (that is, the name of the object where you want the event to stop processing).

In either method, the break events that you specify are maintained and appear in the Breakevent Control section, in the order in which you entered them. After specifying the break events, use the **GO** command or press PF5 or PF15 to execute the rule.

Break events are not processed for event rules. This includes trigger rules, validation rules, and rules used to derive values from within tables.

Valid Break Event Values

The following describes the valid events and any further qualifiers that you can use with the **AT** command or specify in the Breakevent Control section of the screen.

Event	Description	Type of Qualifier
ACCESS	Any table access.	* *table name*
CALL	CALL statement or reference to a function.	* *rule name*

Event	Description	Type of Qualifier
COMMIT	COMMIT statement. Do not use this break event if an external database is open for update.	
DELETE	DELETE statement.	* *table name*
DISPLAY	DISPLAY statement.	* *screen name*
DISPXCALL	DISPLAY & TRANSFERCALL statement.	* *screen name*
EXECUTE	EXECUTE statement.	* *rule name*
FORALL	FORALL statement.	* *table name*
GET	GET statement.	* *table name*
INSERT	INSERT statement.	* *table name*
ON	Handling of an exception.	* *exception name* *exception name table name*
REFFIELD	Reference to a field in a table.	* *table name.** *table name.field name*
REFLOCAL	Reference to a local variable.	* *variable name*

Event	Description	Type of Qualifier
REPLACE	REPLACE on a table occurrence.	* *table name*
RETURN	Return from a rule.	* *rule name*
ROLLBACK	ROLLBACK statement. Do not use this break event if an external database is open for update.	
SCHEDULE	SCHEDULE statement.	* i*nstance name* of the @SCHEDULEMODEL table
SETFIELD	Assign a value to a field of a table.	* *table name.** *table name.field name*
SETLOCAL	Assign a value to a local variable.	* *variable name*
SIGNAL	Raise an exception.	* *exception name* *exception name table name*
TRANSACTIONEND	End of transaction.	
TRANSFERCALL	TRANSFERCALL to a rule.	* *rule name*
UNRECERR	Unrecoverable error.	
*	At all events.	

How to Examine and Modify the Execution State

Suspension of the Execution State

The Rule Debugger suspends the execution of the rule just prior to the event taking place and displays the name of the event in the Debug Information section. When the Debugger has suspended the execution of your rule, you can examine the execution state of the present break event or modify the execution state of the next break event to be encountered. To continue processing the rule, use the **GO** command or press PF5 or PF15.

Commands Available to Examine the Execution State

To examine or modify the execution state use the primary commands listed in the following table. Press Enter to execute the command.

Command	Description	Syntax
LIST	Display a variable and its current value.	**LIST** *local variable name* **LIST** *table.field* **LIST** *table.**
SET	Assign a new value to a variable.	**SET** *local=value* **SET** *table.field=value* **SET** *table.*=value*
DUMP CALLSTACK	Display the execution stack.	
DUMP LASTCALLED	Display a list of the last 16 rules called.	
DUMP LOCALS	Display a list of all active local variables and their values.	
DUMP TABLES	Display a list of all active tables and the field values being currently accessed.	

Associating a Series of Commands With a Break Event

Using the commands described in Commands Available to Examine the Execution State on page 207, you can also create a series of commands to be associated with a specific break event. When created, this series of commands is saved. It is then executed when the specific break event with which it is associated is encountered during execution, in the same manner as if you had entered each of the commands interactively.

How to Create a Series of Commands

To create a series of commands for a specific break event, complete the following steps:

1. Position your cursor on the required break event in the Breakevent Control section of the Debugger screen and press PF6.
2. Use the text editor TED to type in the required commands, one line at a time, on the displayed screen.

Editing Commands

You can add to or delete any of the existing commands, using the following line commands:

I	Insert a line after the line your cursor is on.
D	Delete the line the command is on.
M	Move the line either before (**B**) or after (**A**) the line indicated.
C	Copy a line after the line the command is on. Or copy the line either before (**B**) or after (**A**) the line indicated.
B	Move or copy the line before the line the command is on.
A	Move or copy the line after the line the command is on.

Debugger Command Editor Screen

The Debugger Command Editor screen, similar to the following one, appears when you press PF6 from the main Debugger screen:

```
                    Debugger Command Editor
                         TEXTTEMP
Command:
 _ ------------------------------------------------------------------------
 _ DUMP callstack                                                          1
 _ DUMP lastcalled                                                         2




PFKEYS: 1=HELP 3=SAVE 4=VERIFY 12=CANCEL 13=PRINT 22=DELETE COMMANDS
```

Available PF Keys

PF1	Display the Help screen.
PF3	Save the list of commands and exit to the Debug screen.
PF4	Validate the commands listed.
PF12	Cancel changes and exit to the Debug screen.
PF13	Print the list of commands.
PF22	Delete the list of commands. You are prompted to confirm the deletion.

See Also *TIBCO Object Service Broker Shareable Tools* about using TED.

Appendix A Syntax of the Rules Language

This appendix describes the syntax of the rule language.

Topics

BNF Notation

Conventions Used

The following sections describe the syntax of the rules language using BNF notation. This notation makes use of the following conventions:

- Lowercase words enclosed in angle brackets (< >) denote syntactic categories. For example:

  ```
  <start character>
  ```

- In a list of alternatives, each alternative starts on a new line except in the case of the start characters, digits, and hexadecimal characters.
- A repeated item is enclosed in braces ({ }). The item can appear zero or more times. For example:

  ```
  {<digit>}
  ```

- Optional categories are enclosed in square brackets ([]). For example:

  ```
  [<exponent>]
  ```

- `::=` means “is defined to be”.

Identifiers

Rules language identifiers cannot exceed sixteen characters in length and are comprised of the following characters:

```
<identifier> ::=
   <start character> {<follow character>}
<start character> ::=
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z @ # $
<follow character> ::=
   <start character>
   <digit>
   _
<digit> ::=
   0 1 2 3 4 5 6 7 8 9
```

Numeric Literals

The rules language supports the following forms of numeric literals:

```
 <numeric literal> ::=
   <digits> [ . <digits> ] [ <exponent> ]
   [<digits> ] . <digits> [ <exponent> ]
<digits> ::=
   digit { <digit> }
```

```
<digit> ::=
   0 1 2 3 4 5 6 7 8 9
<exponent> ::=
   E [ <sign> ] <digits>
   e [ <sign> ] <digits>
<sign> ::=
   +
   -
```

String Literals

A string literal is zero or more characters enclosed in single quotes:

```
<string literal> ::=
   <plain character string>
   <hexadecimal character string>
   <Unicode character string>
   <hexadecimal byte string>

<plain character string> ::=
   '{<character>}'

<hexadecimal character string> ::=
   x'{<hexadecimal digit><hexadecimal digit>}'
   X'{<hexadecimal digit><hexadecimal digit>}'

<Unicode character string> ::=
   u'{<Unicode character>}'
   U'{<Unicode character>}'

<hexadecimal byte string> ::=
   r'{<hexadecimal digit><hexadecimal digit>}'
   R'{<hexadecimal digit><hexadecimal digit>}'

<hexadecimal character> ::=
   0 1 2 3 4 5 6 7 8 9 a A b B c C d D e E f F

<Unicode character> ::=
   <character other than />
  /<hexadecimal digit><hexadecimal digit><hexadecimal digit><hexadecimal digit>
   //
```

where:

- <character> can be <letters>, <digits>, <special characters>, and the space character (described in Character Set on page 19), as well as any other characters that you can input from your terminal.
- <character other than /> denotes the same characters as <character>, except for the exclusion of the forward slash (escape) character.
- The double forward slash (//) denotes the forward slash character (/).
- A forward slash followed by four hexadecimal digits denotes the Unicode character with the corresponding UTF-16 representation.

- <plain character string> denotes a typeless string literal of syntax V containing the designated characters.
- <hexadecimal character string> denotes a typeless string literal of syntax V containing characters designated by their two-digit hexadecimal values in the relevant EBCDIC code page. When a rule containing <hexadecimal character string> is stored and subsequently retrieved for editing, the literal in question appears as <plain character string> if all the characters designated by the original literal are printable.
- <Unicode character string> denotes a typeless string literal of syntax UN containing the characters designated by their printable EBCDIC value, an escaped UTF-16 hexadecimal value, or the escaped escape character.
- <hexadecimal byte string> denotes a typeless literal of syntax RD, containing the byte values designated by the pairs of hexadecimal digits.

Syntax of Rules

Supported Syntax

The rules language supports the following syntax within rules:

```
<rule> ::=
    <rule declaration> <condition list> <action list> <exception list>
<rule declaration> ::=
   <rule header> [<local name declaration>]
<rule header> ::=
   <rule name> [<rule header argument list>] ;
<rule header argument list> ::=
   (<rule argument name> {,<rule argument name>})
<local name declaration> ::=
   LOCAL <local name> {,<local name>} ;
<condition list> ::=
   {<condition>;}
<condition> ::=
   [<not>] <logical value>
   <expression> <relational operator in condition> <expression>
<logical value> ::=
   <field of a table>
   <rule argument name>
   <function call>
<action list> ::=
   <action> {<action>}
<exception list> ::=
   {<on exception>}
<on exception> ::=   ON <exception designation> : {<action>}
<action> ::=
   <statement> ;
<statement> ::=
   <assignment>
   <rule invocation>
   <function return>
   <table access statement>
   <synchronous processing>
   <output processing>
   <signal exception>
   <asynchronous call>
   <iterative processing>
<assignment> ::=
   <assignment target> = <expression>
   <assignment-by-name>
<assignment target> ::=
   <field of a table>
   <local name>
<assignment-by-name> ::=
   <table reference> .* = <table reference> .*
   <table reference> .* = NULL
<rule invocation> ::=
   <invocation> <invocation specification> [<invocation arguments>]
```

```
<invocation> ::=
   CALL
   EXECUTE [IN <browse specification>]
   TRANSFERCALL [IN <browse specification>]
<invocation specification> ::=
   <rule name>
   <rule argument name>
   <table name> . <field name>
<invocation arguments> ::=
   <argument list>
   WHERE <where argument list>
<where argument list> ::=
   <where argument item> {<and> <where argument item>}
<where argument item> ::=
   <identifier> = <expression>
<function return> ::=
   RETURN ( <expression>
<table access statement> ::=
   <get statement>
   <insert statement>
   <replace statement>
   <delete statement>
   <forall statement>
<get statement> ::=
   GET <occurrence specification> [<table order>] [WITH MINLOCK]
<occurrence specification> ::=
   <table specification> [WHERE <where predicate>]
<table specification> ::=
   <table name> [<argument list>]
   <rule argument name> [<argument list>]
   <table name> . <field name> [<argument list>]
<where predicate> ::=
   <where not expression> {<logical operator> <where not expression>}
<where not expression> ::=
   [<not>] <where expression>
<where expression> ::=
   <where relation>
   (<where predicate>)
<where relation> ::=
   <field reference> <relational operator> <where expression>
<where expression> ::=
   [<unary operator>] <where expression term> {<add operator>
<where expression term>}
<where expression term> ::=
   <where expression factor> {<multiplication operator>
   <where expression factor>}
<where expression factor> ::=
   <where expression primary> [<exponent operator> <where
   expression primary>]
<where expression primary> ::=
   (<where expression>)
   <where field of a table>
   <rule argument name>
   <local name>
   <function call>
   <literal>
<where field of a table> ::=
```

```
   <where table reference> . <field reference>
<where parameter list> ::=
   <where parameter item> {<and> <where parameter item>}
<where parameter item> ::=
   <table parameter name> = <expression>
<where table reference> ::=
   *
   <table name>
   (<rule argument name>)
   (<table name> . <field name>)
```

The <where table reference> construction allows an asterisk (*) to be specified as the table name.

```
<insert statement> ::=
   INSERT <table specification> [WHERE <where parameter list>]
<replace statement> ::=
   REPLACE <table specification> [WHERE <where parameter list>]
<delete statement> ::=
   DELETE <table specification> [WHERE <where primary key>]
<where primary key> ::=
   <primary key> = <expression>
<primary key> ::=
   <field name>
<forall statement> ::=
   FORALL <occurrence specification> [ <table order>] [<until
   specification> ]: <for action list> END
<until specification> ::=
   UNTIL <exceptions>
<exceptions> ::=
   <exception designation> {<or> <exception designation>}
<exception designation> ::=
   <exception name> [<table name>]
<exception name> ::=
   <identifier>
<for action list> ::=
   <for action> ; }
<for action> ::=
   <assignment>
   <rule invocation>
   <table access statement>
   <output processing>
   <asynchronous call>
   <iterative processing>   COMMIT
<table order> ::=
   <table order item> {AND <table order item>}
<table order item> ::=
   ORDERED [<ordering>] <field reference>
<ordering> ::=
   ASCENDING
   DESCENDING
<synchronous processing> ::=
   COMMIT
   ROLLBACK
<output processing> ::=
```

```
   <PRINT> <report reference>
   DISPLAY <screen reference> [<and option>]
<report reference> ::=
   <report name>
   <table name> . <field name>
   <rule argument name>
<report name> ::=
   <identifier>
<screen reference> ::=
   <screen name>
   <table name> . <field name>
   <rule argument name>
<screen name> ::=
   <identifier>
<and option> ::=
   <and> TRANSFERCALL [IN <browse specification>] <invocation
   specification> [<invocation arguments>]
<signal exception> ::=
   SIGNAL <exception name>
<asynchronous call> ::=
   SCHEDULE [IN <browse specification>] [<queue
   specification>] <rule name> [<call arguments>]
<browse specification> ::=
   BROWSE
   UPDATE
<queue specification> ::=
   TO <expression>
<iterative processing> ::=
<until statement> ::=
   UNTIL <exceptions> [DISPLAY<screen reference>]:{<action>} END
<field reference> ::=
   <field name>
   (<rule argument name>)
   (<table name> . <field name>)
<table parameter name> ::=
   <identifier>
<function call> ::=
   <function name> [<argument list>]
<argument list> ::=
   (<expression> {,<expression>})
<expression> ::=
   [<unary op>] <expression term> {<add operator>
   <expression term>}
<expression term> ::=
   <expression factor> {<multiplication operator> <expression
   factor>}
<expression factor> ::=
   <expression primary> [<exponent operator> <expression
   primary>]
<expression primary> ::=
   <expression>
   <field of a table>
   <rule argument name>
   <local name>
   <function call>
   <literal>
<field of a table> ::=
```

```
   <table reference> . <field reference>
<table reference> ::=
   <table name>
   (<rule argument name>)
   (<table name> . <field name>)
<rule name> ::=
   <identifier>
<function name> ::=
   <identifier><rule argument name> ::=
   <identifier>
<table name> ::=
   <identifier>
<field name> ::=
   <identifier>
<local name> ::=
   <identifier><unary operator> ::=
   -
   +
<add operator> ::=
   +
   -
   ||
<multiplication operator> ::=
   *
   /
<exponent operator> ::=
   **
<logical operator> ::=
   <and>
   <or>
<and> ::=
   AND
   &
<or> ::=
   OR
   |
<not> ::=
   NOT    ¬
<relational operator in section> ::=
   <relational operator in condition>
   LIKE
<relational operator in condition> ::=
   =
   ¬=
   >
   >=
   <
   <=
<literal> ::=
   <string literal>
   <numeric literal>
   NULL
```

The keyword NULL can appear only by itself and not as an operand in an expression. For example, it is syntactically incorrect to write NULL + 1.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z