Oracle® Fusion Middleware

Language Reference Guide for Oracle Business Rules

11*g* Release 1 (11.1.1.7)

E10227-10

November 2013

The Oracle Business Rules Language Reference Guide contains a detailed and complete reference to the Oracle Business Rules RL Language syntax, semantics, and built-in functions. You can create and edit the Oracle Rules language directly but the language is normally generated by high-level tools.

ORACLE

Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules, 11*g* Release 1 (11.1.1.7)

E10227-10

Primary Author: Oracle Corporation

Contents

Preface

This Preface contains these topics:

Audience

Oracle Fusion Middleware Language Reference Guide for Oracle Business Rules is intended for application developers and Oracle Application Server administrators who perform the following tasks:

- Develop rule enabled applications
- Debug rule enabled applications
- Deploy and Administer rule enabled applications.
- Develop rulesets for those who prefer a technical language environment instead of the Oracle Business Rules Rule Author graphical environment for rule authoring.
- Need to use Oracle Business Rules RL Language advanced features that are not available in the Oracle Business Rules Rule Author environment.

To use this document, you need to be familiar with the Java programming language.

Documentation Accessibility

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

Access to Oracle Support

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

Related Documents

For more information, see the following documents in the Oracle Business Process Management and SOA Suite 11*g* Release 1 (11.1.1.7) documentation sets:

- Oracle Fusion Middleware User's Guide for Oracle Business Rules
- Oracle Fusion Middleware User's Guide for Oracle Business Process Management
- Oracle Fusion Middleware Developer's Guide for Oracle SOA Suite

Conventions

This section describes the conventions used in the text and code examples of this documentation set. It describes:

- Conventions in Text
- RL Language Backus-Naur Form Grammar Rules

Conventions in Text

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

RL Language Backus-Naur Form Grammar Rules

Each RL Language command in the guide is shown in a format description that consists of a variant of Backus-Naur Form (BNF) that includes the symbols and conventions in the following table.

Symbol or Convention	Meaning
[ ]	Brackets enclose optional items.
{ }	Braces enclose items only one of which is required.
|	A vertical bar separates alternatives within brackets or braces.
*	A star indicates that an element can be repeated.
delimiters	Delimiters other than brackets, braces, vertical bars, stars, and ellipses must be entered as shown.
boldface	Words appearing in boldface are keywords. They must be typed as shown. (Keywords are case-sensitive in some, but not all, operating systems.) Words that are not in boldface are placeholders for which you must substitute a name or value
underline	When on the left side of a production (`::=`) indicates a definition for a non-terminal symbol.

Symbol or Convention	Meaning
underline	When found on the right side of a production, `::=` , a link, which is a non-terminal symbol, links to the definition for the non-terminal symbol.
italic text	Semantic information about non-terminals, such as the required data type for an expression or a descriptive tag used in following discussion, is in italics.

What's New in This Guide for Release 11.1.1.7

The following table lists the content that has been added or updated.

For a list of known issues (release notes), see the "Known Issues for Oracle SOA Products and Oracle AIA Foundation Pack" at http://www.oracle.com/technetwork/middleware/docs/soa-aiafp-knownissuesindex-364630.html.

Content	Changes Made
Chapter 2, Built-in Functions	Added these built-in functions: ■ isErrorInRuleConditionSuppressed ■ setErrorInRuleConditionSuppressed ■ getFactByType
Chapter 2, Built-in Functions	Added two new signatures for the Step function.
Chapter 4, table 4.1	Added configuration parameters that can be set in a Map passed to the RuleSession constructor.

1 Rules Programming Concepts

This chapter introduces Oracle Business Rules RL Language (RL Language) concepts.

This chapter includes the following sections:

1.1 Starting the Oracle Business Rules RL Language Command-Line

The Oracle Business Rules environment is implemented in a JVM or in a J2EE container by the classes supplied with `rl.jar`. Start the RL Language command-line interface using the following command:

```
java -jar $ORACLE_HOME/soa/modules/oracle.rules_11.1.1/rl.jar -p "RL> "
```

Where ORACLE_HOME is where SOA modules are installed (for example, c:/Oracle/Middleware). The `-p` option specifies the prompt.

The RL Language command-line interface provides access to an Oracle Business Rules *RuleSession*. The RuleSession is the API that allows Java programmers to access the RL Language in a Java application (the command-line interface uses a RuleSession internally).

You can run the program in Example 1–1 using the command-line interface by entering the text shown at the `RL>` prompt.

Example 1–1 Using the Command-Line Interface

```
RL> println(1 + 2);
3
RL> final int low = -10;
RL> final int high = 10;
RL> println(low + high * high);
90
RL> exit;
```

See Also:

- Chapter 3, "Using the Command-line Interface" for more details and for a list of command-line options
- Chapter 4, "Using a RuleSession" for details on Oracle Business Rules RuleSession API

1.2 Introducing Rules and Rulesets

An RL Language *ruleset* provides a namespace, similar to a Java package, for RL classes, functions, and rules. In addition, you can use rulesets to partially order rule firing. A ruleset may contain executable actions, may include or contain other rulesets, and may import Java classes and packages.

An RL Language *rule* consists of *rule conditions*, also called fact-set-conditions, and an action-block or list of actions. Rules follow an if-then structure with rule conditions followed by rule actions.

Example 1–2 shows a program that prints, "Hello World." This example demonstrates a program that contains a single top-level action in the default ruleset (named `main`). Example 1–2 contains only an action, and does not define a rule, so the action executes immediately at the command-line.

Example 1–2 Hello World Programming Example

```
RL> println("Hello World");
Hello World
RL>
```

See Also: Understanding and Controlling Rule Firing on page 1-5 for details on rule firing

1.2.1 Rule Conditions

A rule condition is a component of a rule that is composed of conditional expressions that refer to facts.

In the following example the conditional expression refers to a fact (`Driver` instance `d1`), followed by a test that the fact's data member, `age`, is less than 16.

```
if (fact Driver d1 && d1.age < 16)
```

Example 1–3 shows the complete rule, written in RL Language (the rule includes a rule condition and a rule action).

The Oracle Rules Engine activates a rule whenever there is a combination of facts that makes the rule's conditional expression true. In some respects, a rule condition is like a query over the available facts in the Oracle Rules Engine, and for every row that returns from the query, the rule activates.

Note: Rule activation is different from rule firing. For more information, see Section 1.4, "Understanding and Controlling Rule Firing".

Example 1–3 Defining a Driver Age Rule

```
RL> rule driverAge{
     if (fact Driver d1 && d1.age < 16)
```

```
    {
       println("Invalid Driver");
    }
}
```

1.2.2 Rule Actions

A rule action is activated if all of the rule conditions are satisfied. There are several kinds of actions that a rule's action-block might perform. For example, an action in the rule's action-block can add new facts by calling the assert function or remove facts by calling the retract function. An action can also execute a Java method or perform an RL Language function (Example 1–3 uses the `println` function). Using actions, you can call functions that perform a desired task associated with a pattern match.

1.3 Introducing Facts and RL Language Classes

This section describes Oracle Business Rules facts and includes the following sections:

- What Are Facts?
- Adding Facts to Working Memory with Assert
- Using RL Language Classes as Facts
- Using Java Classes as Facts

1.3.1 What Are Facts?

Oracle Business Rules facts are asserted objects. For Java objects, a fact is a shallow copy of the object, meaning that each property is cloned, if possible, and if not, then the fact is a copy of the Java object reference.

In RL Language, a Java object is an instance of a Java class and an RL Object is an instance of an RL Language class. You can use Java classes in the classpath or you can define and use RL Language classes in a ruleset. You can also declare additional properties that are associated with the existing properties or methods of a Java class using a fact class declaration. You can hide properties of a Java class that are not needed in facts using a fact class declaration.

An RL Language class is similar to a Java Bean without methods. An RL class contains set of named properties. Each property has a type that is either an RL class, a Java object, or a primitive type.

Using Oracle Business Rules, you typically use Java classes, including JAXB generated classes that support the use of XML, to create rules that examine the business objects in a rule enabled application, or to return results to the application. You typically use RL classes to create intermediate facts that can trigger other rules in the Oracle Rules Engine.

1.3.2 Adding Facts to Working Memory with Assert

Oracle Business Rules uses *working memory* to contain facts (facts do not exist outside of working memory). A RuleSession contains the working memory.

A fact in RL Language is an asserted instance of a class. Example 1–4 shows the assert function that adds an instance of the RL class `enterRoom` as a fact to working memory. A class that is the basis for asserted facts may be defined in Java or in RL Language.

In Example 1–4 the `sayHello` rule matches facts of type `enterRoom`, and for each such fact, prints a message. The action `new,` shown in the `assert` function, creates an instance of the `enterRoom` class.

In Example 1–4 the `run` function fires the `sayHello` rule.

> **Note:** The RL Language `new` keyword extends the Java `new` functionality with the capability to specify initial values for properties.

Example 1–4 Matching a Fact Defined by an RL Language Class

```
RL> class enterRoom { String who; }
RL> assert(new enterRoom(who: "Bob"));
RL> rule sayHello {
  if ( fact enterRoom ) {
    println("Hello " + enterRoom.who);
  }
}
RL> run();
Hello Bob
RL>
```

See Also: "Understanding and Controlling Rule Firing" on page 1-5

1.3.3 Using RL Language Classes as Facts

You can use RL Language classes in a rules program to supplement a Java application's object model, without having to change the application code for the Java application that supplies Java Objects.

Example 1–5 shows the `goldCust` rule uses a Java class containing customer data, `cust`; the rule's action asserts an instance of the `GoldCustomer` RL class, representing a customer that spends more than 500 dollars in a three month period. The Java `Customer` class includes a method `SpentInLastMonths` that is supplied an integer representing a number of months of customer data to add.

Example 1–5 goldCust Rule

```
rule goldCust {
  if (fact Customer cust && cust.SpentInLastMonths(3) > 500 ){
              assert (new GoldCustomer(cust: cust));
  }
}
```

Example 1–6 shows the `goldDiscount` rule uses the RL fact `GoldCustomer` to infer that if a customer spent $500 within the past 3 months, then the customer is eligible for a 10% discount.

Example 1–6 goldDiscount Rule

```
rule goldDiscount {
   if (fact Order ord & fact GoldCustomer(cust: ord.customer) )
       {
            ord.discount = 0.1;
            assert(ord);
       }
}
```

Example 1–7 shows the declaration for the `GoldCustomer` RL class (this assumes that you also have the `Customer` class available in the classpath).

Example 1–7 Declaring an RL Language Class

```
class GoldCustomer {
 Customer cust;
}
```

See Also: "Adding Facts to Working Memory with Assert" on page 1-3

1.3.4 Using Java Classes as Facts

You can use asserted Java objects as facts in an RL Language program. You are not required to explicitly define or declare the Java classes. However, you must include the Java classes in the classpath when you run the program. This lets you use the Java classes in rules, and allows a rules program to access and use the public attributes, public methods, and bean properties defined in the Java class (bean properties are preferable for some applications because the Oracle Rules Engine can detect that a Java object supports `PropertyChangeListener`; in this case it uses that mechanism to be notified when the object changes).

In addition, Fact class declarations can fine tune the properties available to use in an RL program, and may be required for certain multiple inheritance situations.

When you work with Java classes, using the `import` statement lets you omit the package name (see Example 1–8).

Example 1–8 Sample Java Fact with Import

```
ruleset main
{
  import example.Person;
  import java.util.*;
  rule hasNickNames
  {
    if (fact Person p && ! p.nicknames.isEmpty() )
    {
       // accessing properties as fields:
       println(p.firstName + " " + p.lastName + " has nicknames:");
       Iterator i = p.nicknames.iterator();
       while (i.hasNext())
       {
          println(i.next());
       }
    }
}
```

See Also:

- "Fact Class Declarations" on page 2-21
- "Import Statement" on page 2-25

1.4 Understanding and Controlling Rule Firing

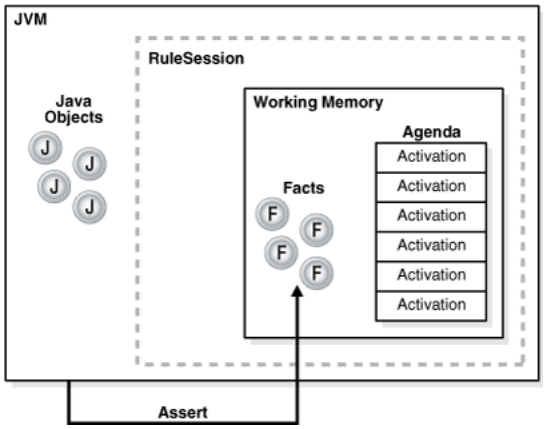
This section covers the following topics:

1.4.1 Rule Activation and the Agenda

The Oracle Rules Engine matches facts against the rule conditions (fact-set-conditions) of all rules as the state of working memory changes. The Oracle Rules Engine only checks for matches when the state of working memory changes, typically when a fact is asserted or retracted. A group of facts that makes a given rule condition true is called a *fact set row*. A *fact set* is a collection of all the fact set rows for a given rule. Thus a fact set consists of the facts that match the rule conditions for a rule. For each fact set row in a fact set, an *activation*, consisting of a *fact set row* and a reference to the rule is added to the *agenda* (the agenda contains the complete list of activations).

Figure 1–1 shows a RuleSession with an agenda containing activations in working memory.

Figure 1–1 RuleSession with Working Memory and the Agenda Containing Activations



The run, runUntilHalt, and step functions execute the activations on the agenda, that is, these commands *fire* the rules (use the step command to fire a specified number of activations).

Rules fire when the Oracle Rules Engine removes activations, by popping the activations off the agenda and performing the rule's actions.

The Oracle Rules Engine may remove activations without firing a rule if the rule conditions are no longer satisfied. For example, if the facts change or the rule is cleared then activations may be removed without firing. Further, the Oracle Rules Engine removes activations from the agenda when the facts referenced in a fact set row are modified or the facts are retracted, such that they no longer match a rule condition (and this can also happen in cases where new facts are asserted, when the ! operator applies).

Note the following concerning rule activations:

1. Activations are created, and thus rules fire only when facts are asserted, modified, or retracted (otherwise, the rules would fire continuously).
2. If a rule asserts a fact that is mentioned in the rule condition, and the rule condition is still true, then a new activation is added back to the agenda and the

rule fires again (in this case the rule would fire continuously). This behavior is often a bug.

3. The actions associated with a rule firing can change the set of activations on the agenda, by modifying facts, asserting facts, or retracting facts, and this can change the next rule to fire.
4. Rules fire sequentially, not in parallel.

See Also: Ordering Rule Firing on page 1-8

1.4.2 Watching Facts, Rules, and Rule Activations

You can use the functions `watchActivations`, `watchFacts`, `watchRules`, and `showFacts` to help write and debug RL Language programs.

This section covers the following topics:

- Watching and Showing Facts in Working Memory
- Watching Activations and Rule Firing

1.4.2.1 Watching and Showing Facts in Working Memory

Example 1–9 shows the `watchFacts` function that prints information about facts entering and leaving working memory.

As shown in Example 1–9, the `watchFacts` function prints `==>` when a fact is asserted. Each fact is assigned a short identifier, beginning with `f-`, so that the fact may be referenced. For example, activations include a reference to the facts that are passed to the rule actions.

In Example 1–9, notice that the program uses the default ruleset `main`. This ruleset contains the `enterRoom` class.

Example 1–9 Using watchFacts with enterRoom Facts

```
RL> watchFacts();
RL>  class enterRoom {String who;}
RL> assert(new enterRoom(who: "Rahul"));
 ==> f-1 main.enterRoom(who : "Rahul")
RL> assert(new enterRoom(who: "Kathy"));
 ==> f-2 main.enterRoom(who : "Kathy")
RL> assert(new enterRoom(who: "Tom"));
 ==> f-3 main.enterRoom(who : "Tom")
RL>
```

You can use `showFacts` to show the current facts in working memory. Example 1–10 shows that the Oracle Rules Engine asserts the initial-fact, `f-0` (the Oracle Rules Engine uses this fact internally).

Example 1–10 Show Facts in Working Memory

```
RL> showFacts();
f-0   initial-fact()
f-1   main.enterRoom(who : "Rahul")
f-2   main.enterRoom(who : "Kathy")
f-3   main.enterRoom(who : "Tom")
For a total of 4 facts.
```

Use retract to remove facts from working memory, as shown in Example 1–11. When `watchFacts` is enabled, the Oracle Rules Engine prints <== when a fact is retracted.

Example 1–11 Retracting Facts from Working Memory

```
RL> watchFacts();
RL> retract(object(2));
 <== f-2 main.enterRoom(who : "Kathy")
RL> showFacts();
f-0   initial-fact()
f-1   main.enterRoom(who : "Rahul")
f-3   main.enterRoom(who : "Tom")
For a total of 3 facts.
```

1.4.2.2 Watching Activations and Rule Firing

The `watchActivations` function monitors the Oracle Rules Engine and prints information about rule activations entering and leaving the agenda. The `watchRules` function prints information about rules firing.

Example 1–12 shows how `run` causes the activations to fire. Notice that Rahul is greeted last even though he entered the room first (this is due to the firing order).

> **Note:** Activations may be removed from the agenda before they are fired if their associated facts no longer make the condition true.

Example 1–12 Using WatchActivations and WatchRules

```
RL> clear;
RL> class enterRoom {String who;}
RL> assert(new enterRoom(who: "Rahul"));
RL> assert(new enterRoom(who: "Kathy"));
RL> assert(new enterRoom(who: "Tom"));
RL> watchActivations();
RL> rule sayHello {
if (fact enterRoom) {
      println("Hello " + enterRoom.who);
   }
}
==> Activation: main.sayHello :  f-1
==> Activation: main.sayHello :  f-2
==> Activation: main.sayHello :  f-3
RL> watchRules();
RL> run();
Fire 1 main.sayHello f-3
Hello Tom
Fire 2 main.sayHello f-2
Hello Kathy
Fire 3 main.sayHello f-1
Hello Rahul
RL>
```

1.4.3 Ordering Rule Firing

To understand the ordering algorithm for firing rule activations on the agenda, we introduce the *ruleset stack*. Each `RuleSession` includes one ruleset stack. The `RuleSession`'s ruleset stack contains the top of the stack, called the *focus* ruleset, and any non focus rulesets that are also on the ruleset stack. You place additional rulesets on the ruleset stack using either the pushRuleset or setRulesetStack built-in functions. You can manage the rulesets on the ruleset stack with the clearRulesetStack,

popRuleset, and setRulesetStack functions. In this case, the focus of the ruleset stack is the current top ruleset in the ruleset stack (see Example 1–13).

Example 1–13 Ruleset Stack - Picture

```
RuleSet Stack

          Focus Ruleset  -->     Top_Ruleset
                                 Next_down_Ruleset
                                 Lower_Ruleset
                                 Bottom_Ruleset
```

When activations are on the agenda, the Oracle Rules Engine fires rules when run, runUntilHalt, or step executes. The Oracle Rules Engine sequentially selects a rule activation from all of the activations on the agenda, using the following ordering algorithm:

1. The Oracle Rules Engine selects all the rule activations for the focus ruleset, that is the ruleset at the top of the ruleset stack (see the pushRuleset and setRulesetStack built-in functions).
2. Within the set of activations associated with the focus ruleset, rule priority specifies the firing order, with the higher priority rule activations selected to be fired ahead of lower priority rule activations (the default priority level is 0).
3. Within the set of rule activations of the same priority, within the focus ruleset, the most recently added rule activation is the next rule to fire. However, note that in some cases multiple activations may be added to the agenda at the same time, the ordering for such activations is not defined.
4. When all of the rule activations in the current focus fire, the Oracle Rules Engine pops the ruleset stack, and the process returns to Step 1, with the current focus.

If a set of rules named R1 must all fire before any rule in a second set of rules named R2, then you have two choices:

- Use a single ruleset and set the priority of the rules in R1 higher than the priority of rules in R2.
- Use two rulesets R1 and R2, and push R2 and then R1 on the ruleset stack.

Generally, using two rulesets with the ruleset stack is more flexible than using a single ruleset and setting the priority to control when rules fire. For example if some rule R in R1 must trigger a rule in R2 before all rules in R1 fire, a return in R pops the ruleset stack and allows rules in R2 to fire.

If execution must alternate between two sets of rules, for example, rules to produce facts and rules to consume facts, it is easier to alternate flow with different rulesets than by using different priorities.

Example 1–14 shows that the priority of the `keepGaryOut` rule is set to high, this is higher than the priority of the `sayHello` rule (the default priority is 0). If the activations of both rules are on the agenda, the higher priority rule fires first. Notice that just before calling `run`, `sayHello` has two activations on the agenda. Because `keepGaryOut` fires first, it retracts the `enterRoom(who: "Gary")` fact, which removes the corresponding `sayHello` activation, resulting in only one `sayHello` firing.

The rule shown in Example 1–14 illustrates two additional RL Language features.

1. The `fact` operator, also known as a fact set pattern, uses the optional `var` keyword to define a variable, in this case the variable `g`, that is bound to the matching facts.
2. You can remove facts in working memory using the `retract` function.

Example 1–14 Using Rule Priority with keepGaryOut Rule

```
RL> final int low = -10;
RL> final int high = 10;
RL> rule keepGaryOut {
  priority = high;
  if (fact enterRoom(who: "Gary") var g) {
    retract(g);
  }
}
RL> assert(new enterRoom(who: "Gary"));
==> f-4 main.enterRoom(who: "Gary")
==> Activation: main.sayHello : f-4
==> Activation: main.keepGaryOut : f-4
RL> assert(new enterRoom(who: "Mary"));
==> f-5 main.enterRoom(who: "Mary")
==> Activation: main.sayHello : f-5
RL> run();
Fire 1 main.keepGaryOut f-4
<== f-4 main.enterRoom(who: "Gary")
<== Activation: main.sayHello : f-4
Fire 2 main.sayHello f-5
Hello Mary
RL>
```

Example 1–15 shows the `sayHello` rule that includes a condition that matches the asserted `enterRoom` fact; this match adds an activation to the agenda. Example 1–15 demonstrates the following RL Language programming features.

1. The Oracle Rules Engine matches facts against the rule conditions (fact-set-conditions) of all rules as the state of working memory changes. Thus, it does not matter whether facts are asserted before the rule is defined, or after.
2. The run function processes any activations on the agenda. No activations on the agenda are processed before calling run.

Example 1–15 enterRoom Class with sayHello Rule

```
RL> class enterRoom { String who; }
RL> rule sayHello {
  if ( fact enterRoom ) {
    println("Hello " + enterRoom.who);
  }
}
RL> assert(new enterRoom(who: "Bob"));
RL> run();
Hello Bob
RL>
```

Notes for ordering rule firing.

1. When you use the `return` action, this changes the behavior for firing rules. A `return` action in a rule pops the ruleset stack, so that execution continues with the activations on the agenda that are from the ruleset that is currently at the top of the ruleset stack.

If rule execution was initiated with either the run or step functions, and a return action pops the last ruleset from the ruleset stack, then control returns to the caller of the run or step function.

If rule execution was initiated with the runUntilHalt function, then a return action does not pop the last ruleset from the ruleset stack. The last ruleset is popped with runUntilHalt when there are not any activations left. The Oracle Rules Engine then waits for more activations to appear. When they do, it places the last ruleset on the ruleset stack before resuming ruleset firing.

2. Rule priority is only applicable within rules in a given ruleset. Thus, the priority of rules in different rulesets are not comparable.

1.5 Using Effective Dates

By default, the value of the effective date is managed implicitly by the rules engine. In this case, when a run family of built-in functions is invoked, the effective date is updated to the current system date. This is done before any rules fire so that the new effective date is applied before rules begin to fire. In the case of `runUntilHalt`, this update occurs each time there is a transition from 0 rules on the agenda to > 0 rules on the agenda.

In Oracle Business Rules RL Language, the effective start and end dates and the active property are only applied to rules (and do not apply for rulesets). The effective start and end date properties of a rule can be specified in the rule.

For example,

```
rule myrule2 {
   active = true;
   effectiveDateForm = Rule.EDFORM_DATETIME:
   effectiveStartDate = JavaDate.fromDateTimeString("2008-11-01");
   effectiveEndDate = JavaDate.fromDateTimeString("2008-11-16");

   if (fact Foo)
   {
    .
    .
   }

}

If you use the RuleSession Java API, you can access the effective start and end
date.
```

Setting a property from RL Language requires a long expression or several statements.

For example, given a ruleset:

```
ruleset MyRules {
  rule myRule { if fact foo { }}
}
```

To set the active property, use the following:

```
Rule r = getRuleSession().getRuleset("MyRules").getRule("myRule");

r.setActive(false);
```

1.6 Integrating RL Language Programs with Java Programs

This section describes integrating RL Language programs with Java programs. This section covers the following topics:

- Using Java Beans Asserted as Facts
- Using RuleSession Objects in Java Applications

> **See Also:** "Working with Rules SDK Decision Point API" in the *Oracle Business Rules User's Guide*

1.6.1 Using Java Beans Asserted as Facts

Example 1–16 shows the Java source for a simple bean. Use the `javac` command to compile the bean, `example.Person` shown in Example 1–16 into a directory tree.

The following shows how an RL Language command-line can be started that can access this Java bean:

```
java -classpath $ORACLE_HOME/soa/modules/oracle.rules_11.1.1/rl.jar;BeanPath
oracle.rules.rl.session.CommandLine -p "RL> "
```

Where *BeanPath* is the classpath component to any supplied Java Bean classes.

Example 1–16 Java Source for Person Bean Class

```
package example;
import java.util.*;
public class Person
{
  private String firstName;
  private String lastName;
  private Set nicknames = new HashSet();

  public Person(String first, String last, String[] nick) {
    firstName = first; lastName = last;
    for (int i = 0; i < nick.length; ++i)
      nicknames.add(nick[i]);
  }
  public Person() {}
  public String getFirstName() {return firstName;}
  public void setFirstName(String first) {firstName = first;}
  public String getLastName() {return lastName;}
  public void setLastName(String last) {lastName = last;}
  public Set getNicknames() {return nicknames;}
}
```

Example 1–17 shows how the RL Language command-line can execute an RL Language program that uses `example.Person`. The `import` statement, as in Java, allows a reference to the `Person` class using "`Person`" instead of "`example.Person`". Rules reference the `Person` bean class and its properties and methods. In order to create a `Person` fact you must `assert` a Java `Person` bean.

Example 1–17 uses the `new` operator to create an array of `Person` objects, named `people`. The `people` array is declared final so that `reset` does not create more `people`. The `numPeople` variable is not declared `final` so that `reset` re-invokes the `assertPeople` function and re-asserts the `Person` facts using the existing `Person` objects.

Example 1–17 Ruleset Using Person Bean Class

```
ruleset main
{
  import example.Person;
  import java.util.*;
  rule hasNickNames
  {
    if (fact Person(nicknames: var nns) p && !nns.isEmpty())
    {
             // accessing properties as fields:
       println(p.firstName + " " + p.lastName + " has nicknames:");
       Iterator i = nns.iterator();
       while (i.hasNext())
       {
          println(i.next());
       }
    }
}
rule noNickNames
{
   if fact Person(nicknames: var nns) p && nns.isEmpty()
   {
          // accessing properties with getters:
   println(p.getFirstName() + " " + p.getLastName() + " does not have nicknames");
   }
}
  final Person[] people = new Person[] {
new Person("Robert", "Smith", new String[] { "Bob", "Rob" }), // using constructor
new Person(firstName: "Joe", lastName: "Schmoe") // using attribute value pairs
};

function assertPeople(Person[] people) returns int
{
   for (int i = 0; i < people.length; ++i) {
      assert(people[i]);
}
  return people.length;
  }
  int numPeople = assertPeople(people);
  run();
}
```

Note the following when working with Java beans as facts:

1. The `fact` operator can include a pattern that matches or retrieves the bean properties. The properties are defined by getter and setter methods in the bean class.
2. The `new` operator can include a pattern that sets property values after invoking the default no-argument constructor, or can pass arguments to a user-defined constructor.
3. Outside of the `fact` and `new` operators, the bean properties may be referenced or updated using getter and setter methods, or using the property name as if it were a field.
4. If a bean has both a property and a field with the same name, then the field cannot be referenced in RL Language.

If Example 1–18 executes using the same `RuleSession` following the execution of Example 1–17, the output is identical to the Example 1–17 results (both `person` facts are reasserted).

> **Note:** The RL Language command-line interpreter internally creates a `RuleSession` when it starts (and when you use the `clear` command).

Example 1–18 Using Reset with a RuleSession

```
reset();
run();
```

1.6.2 Using RuleSession Objects in Java Applications

Java programs can use the `RuleSession` interface to execute rulesets, invoke RL Language functions passing Java objects as arguments, and redirect RL Language `watch` and `println` output. Example 1–19 and Example 1–20 each contain a Java program fragment that uses a `RuleSession` that prints "hello world". Like many Java program fragments, these examples are also legal RL Language programs.

The RL Language environment provides multiple rule sessions. Each rule session can be used by multiple threads, but rules are fired by a single thread at a time.

Each rule RuleSession has its own copy of facts and rules. To create a fact from a Java Object, use a call such as:

```
rs.callFunctionWithArgument("assert", Object;);
```

To create a rule, a function, or an RL Language class, define a string containing a ruleset, and use the `executeRuleset` method.

Example 1–19 Using a RuleSession Object with callFunctionWithArgument

```
import oracle.rules.rl.*;
try {
  RuleSession rs = new RuleSession();
  rs.callFunctionWithArgument("println", "hello world");
} catch (RLException rle) {
   System.out.println(rle);
}
```

Example 1–20 Using a RuleSession with ExecuteRuleset

```
import oracle.rules.rl.*;
try {
   RuleSession rs = new RuleSession();
   String rset =
     "ruleset main {" +
     " function myPrintln(String s) {" +
     " println(s);" +
     " }" +
     "}";
   rs.executeRuleset(rset);
   rs.callFunctionWithArgument("myPrintln", "hello world");
} catch (RLException rle) {
  System.out.println(rle);
```

```
}
```

1.7 Using Decision Tracing

Using Oracle Business Rules, a decision trace is a trace of rule engine execution that includes information on the state of the rule engine, including the state of facts when rule fire. The Oracle Business Rules rule engine constructs and returns a decision trace using JAXB generated Java classes generated from the decision trace XML schema.

1.7.1 Introduction to Rule Engine Level Decision Tracing

To provide a business analyst friendly presentation of a decision trace requires that the associated rule dictionary is available. Using the rule dictionary associated with a trace allows for a more flexible and efficient approach, as the trace output does not need to include all of the related dictionary content information in the trace.

The XML schema is in the file `decisiontrace.xsd` and it is part of the Jar file rl.jar as: `oracle/rules/rl/trace/decisiontrace.xsd`. The packages of interest are `oracle.rules.rl.trace`, `oracle.rules.rl.extensions.trace`, and `oracle.rules.sdk2.decisiontrace`. The Java classes packages generated from the decisiontrace XML schema are in the package `oracle.rules.rl.trace` and are included in the Javadoc. For more information, see *Oracle Business Rules Java API Reference*.

1.7.2 Using Rule Engine Level Decision Tracing

A decision trace is a set of XML elements showing rule engine events that occur during rule evaluation. The types of events that are added to a decision trace depend on the trace level selected, and can include:

- Fact operations (assert, retract, modify)
- Rules fired
- Rule activations added or removed from the agenda
- Ruleset stack changes
- Rule compilation
- Reset (which is needed for maintaining state for decision trace analysis)

Each trace contains information about a particular event. For example, a fact operation event entry consists of:

- The operation type (assert, modify, retract)
- The ID of the fact in working memory
- Fact type name (fact classed in RL)
- Runtime object type name
- The fact object data, including the property name and value for zero or more fact properties
- Name of rule, RL name, if the operation was the result of a rule action
- Millisecond timestamp

In a fact operation event trace, the fact object content reflects the structure of the object as a Java Bean. If the bean properties are references to other beans the related bean

content is included in the trace. The value of a bean property can be one of the following alternatives.

- A string representation of the property. This is the case for primitive types and classes in the java.* and javax.* packages.
- A nested bean object with its property values.
- A fact ID. This occurs when the property value is an object which has itself been asserted as a fact. The data for the fact at the time of the trace can be retrieved from the RuleEngineState using the fact ID when analyzing the trace.
- A collection of values accessed as a `List` in the trace.
- An array of values accessed as a `List` in the trace.

At runtime, to determine which alternative is included in the trace you can test for null; only the correct alternative has a non-null value.

Table 1–1 shows the RL functions that provide control over decision tracing in the rule engine and provide access to a decision trace.

Table 1–1 RL Decision Trace Functions

Function	Description
`getDecisionTrace`	Returns the current trace and starts a new trace.
`getDecisionTraceLevel`	Gets the current decision trace level.
`getDecisionTraceLimit`	Returns the current limit on the number of events in a trace.
`setDecisionTraceLevel`	Sets the decision trace level to the specified level.
`setDecisionTraceLimit`	Sets the limit on the number of events in a trace. Default limit value is 10000.

The decision trace level may be set by invoking the setDecisionTraceLevel function. You can also configure the initial trace level in a RuleSession or in a RuleSessionPool by including the `RuleSession.CFG_DECISION_TRACE_LEVEL` initialization parameter and specifying a level in the configuration Map passed to the RuleSession or RuleSessionPool constructor. This sets the decision trace level at the time a RuleSession is created.

You can invoke the setDecisionTraceLevel function on a RuleSession or a RuleSessionPool object after initialization. When you invoke `reset()`, this function returns the decision trace level to the configured value (if the level was changed during rule execution). Thus, the `reset()` function resets the decision trace limit to the value set during initialization of a RuleSession or a RuleSessionPool. In these cases, reset() restores the values established using the initialization parameters.

> **Note:** These `reset()` semantics for a RuleSession are only valid for a RuleSession initialized with either or both of the `CFG_DECISION_TRACE_LIMIT` and the `CFG_DECISION_TRACE_LEVEL` initialization parameters (or that is obtained from a RuleSessionPool when the pool is created with either or both of the `CFG_DECISION_TRACE_LIMIT` and the `CFG_DECISION_TRACE_LEVEL` initialization parameters.

The size of a trace is limited by limiting the number of entries in a decision trace. This necessary to avoid infinite rule fire loops, due to a possible bug in the rules, from creating a trace that consumes all available heap in the JVM. Set the trace limit with the

setDecisionTraceLimit function. The limit may also be configured in a RuleSession (or RuleSessionPool) by including the `RuleSession.CFG_DECISION_TRACE_LIMIT` initialization parameter with the desired limit in the configuration Map passed to the RuleSession or RuleSessionPool constructor.

For rules applications that use runUntilHalt, it is the responsibility of the application to invoke getDecisionTrace before the trace limit is hit.

The decision trace provides structure to the trace data so that it can be manipulated programmatically. However, the trace by itself can be cumbersome to analyze. A trace analysis class (`oracle.rules.rl.extensions.trace.TraceAnalysis`) analyzes a decision trace and facilitates exploration of the trace. Use this class to construct the state of working memory, the agenda, and the ruleset stack from the trace.

The `TraceAnalysis` API supports the following:

- Obtain a list of fact types that appear in the trace.
- Obtain a list of names of the rules that fired in the trace.
- Obtain a list of the last fact traces for each instance of a specific fact type.
- Obtain the last fact trace for a specific fact identified by its fact ID.
- Obtain all of the fact traces for a fact identified by its fact ID.
- For a fact trace, if the fact trace was created by a rule action, get the rule trace that rule firing in which the action executed.
- For a rule trace, get the list of fact traces for each fact that matched and resulted in the rule firing.
- Get the next or previous trace. Exploration of the trace is typically not an iteration over the trace. For example, obtaining a rule trace from a fact trace is essentially jumping to that rule trace. The traces near the rule trace can be explored directly.
- Obtain a list of rule traces for a rule identified by its name.
- Obtain the rule engine state for a trace entry. The rule engine state reflects the state of the rule engine after the activity that generated the trace. This API enables inspecting the rule engine state at the time of each trace. This API is most useful with development level tracing. With production level tracing, only the facts in working memory can be tracked and they will not include any fact contents.

Example 1–21 shows a code sample that uses the decision trace analysis API.

Example 1–21 Decision Trace Analysis API Usage

```
DecisionTrace trace;
...
TraceAnalysis ta = new TraceAnalysis(trace);
// Get all of the last fact traces for a fact type.
List<FactTrace> factTraces = ta.getLastFactTraces("com.example.MyFactType");
// From a particular fact trace, how it was arrived at may be explored, first by
// obtaining the rule that asserted or modified the fact.
// From the FactRecord, the rule that resulted in the record can be obtained.
FactTrace factTrace = factTraces.get(0);  // assumes there is one
RuleTrace ruleTrace = ta.whichRule(factTrace);
  // The ruleTrace will be null if the fact operation was not part of a rule
action.
System.out.print("Fact " + factTrace.getFactId() + ", a " +
factTrace.getFactType() + " " + factRecord.getFactOp());
if (ruleTrace != null)
```

```
            System.out.println(" by rule " + ruleTrace.getRuleName());
        else
            System.out.println("");
        // The analysis can continue by obtaining the list of FactRecords that matched the
        rule and
        // proceeding to analyze the trace back in time.
        List<FactTrace> matchingFacts = ta.getRuleMatchedFacts(ruleTrace);
```

1.7.3 Decision Trace Samples for Production and Development Level Tracing

Example 1–22 shows a sample production level trace document.

Example 1–22 Sample Production Level Decision Trace

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<decision-trace xmlns="http://xmlns.oracle.com/rules/decisiontrace">
    <trace-entries xsi:type="rule-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975549890</timestamp>
        <rule-name>OrderDiscount.goldCustomer</rule-name>
        <token-time>0</token-time>
        <sequence-number>1</sequence-number>
    </trace-entries>
    <trace-entries xsi:type="rule-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975549893</timestamp>
        <rule-name>OrderDiscount.goldCustomerDiscount</rule-name>
        <token-time>0</token-time>
        <sequence-number>2</sequence-number>
    </trace-entries>
    <trace-entries xsi:type="rule-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975549894</timestamp>
        <rule-name>OrderDiscount.applyDiscount</rule-name>
        <token-time>0</token-time>
        <sequence-number>3</sequence-number>
    </trace-entries>
</decision-trace>
```

Example 1–23 Sample Development Level DecisionTrace

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<decision-trace xmlns="http://xmlns.oracle.com/rules/decisiontrace">
    <trace-entries xsi:type="fact-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491008</timestamp>
        <fact-id>1</fact-id>
        <operation>assert</operation>
        <fact-type>com.example.Customer</fact-type>
        <object-type>com.example.Customer</object-type>
        <fact-object>
            <properties>
                <name>YTDOrderAmount</name>
                <value>
                    <string>2000.0</string>
                </value>
            </properties>
            <properties>
                <name>level</name>
                <value>
                    <string>null</string>
```

```
                </value>
            </properties>
            <properties>
                <name>name</name>
                <value>
                    <string>OneLtd</string>
                </value>
            </properties>
            <properties>
                <name>pastDue</name>
                <value>
                    <string>false</string>
                </value>
            </properties>
        </fact-object>
    </trace-entries>
    <trace-entries xsi:type="activation-trace"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491024</timestamp>
        <rule-name>OrderDiscount.goldCustomer</rule-name>
        <token-time>2</token-time>
        <fact-ids>1</fact-ids>
        <operation>add</operation>
    </trace-entries>
    <trace-entries xsi:type="fact-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491025</timestamp>
        <fact-id>2</fact-id>
        <operation>assert</operation>
        <fact-type>com.example.Order</fact-type>
        <object-type>com.example.Order</object-type>
        <fact-object>
            <properties>
                <name>customerName</name>
                <value>
                    <string>OneLtd</string>
                </value>
            </properties>
            <properties>
                <name>discount</name>
                <value>
                    <string>0.0</string>
                </value>
            </properties>
            <properties>
                <name>grossAmount</name>
                <value>
                    <string>400.0</string>
                </value>
            </properties>
            <properties>
                <name>netAmount</name>
                <value>
                    <string>0.0</string>
                </value>
            </properties>
            <properties>
                <name>number</name>
                <value>
                    <string>1001</string>
                </value>
```

```
            </properties>
        </fact-object>
    </trace-entries>
    <trace-entries xsi:type="activation-trace"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491035</timestamp>
        <rule-name>OrderDiscount.goldCustomerDiscount</rule-name>
        <token-time>5</token-time>
        <fact-ids>2</fact-ids>
        <fact-ids>1</fact-ids>
        <operation>add</operation>
    </trace-entries>
    <trace-entries xsi:type="rule-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491036</timestamp>
        <rule-name>OrderDiscount.goldCustomerDiscount</rule-name>
        <token-time>5</token-time>
        <fact-ids>2</fact-ids>
        <fact-ids>1</fact-ids>
        <sequence-number>2</sequence-number>
    </trace-entries>
...
    <trace-entries xsi:type="rule-trace" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491036</timestamp>
        <rule-name>OrderDiscount.applyDiscount</rule-name>
        <token-time>7</token-time>
        <fact-ids>2</fact-ids>
        <sequence-number>3</sequence-number>
    </trace-entries>
...
    <trace-entries xsi:type="ruleset-stack-trace"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
        <timestamp>1248975491037</timestamp>
        <operation>pop</operation>
        <ruleset-name>OrderDiscount</ruleset-name>
    </trace-entries>
</decision-trace>
```

1.8 Building a Coin Counter Rules Program

This section shows a sample that uses RL Language to solve a puzzle:

How many ways can 50 coins add up to $1.50?

The rules program that solves this puzzle illustrates an important point for rule-based programming; *knowledge representation*, that is, the fact classes that you select, can be the key design issue. It is often worthwhile to write procedural code to shape your data into a convenient format for the rules to match and process.

To use this example, first copy the RL Language program shown in Example 1–25 to a file named `coins.rl`. You can include this from the RL Language command-line using the `include` command. Before you include the coins program, use the `clear;` command to erase everything in the current rule session, as follows:

```
RL> clear;
RL> include file:coins.rl;
RL>
```

Example 1–24 shows the debugging functions that show the count coins sample facts, activations, and rules for the coin counter. All facts are asserted, and activations for all solutions are placed on the agenda. Notice that the facts are matched to the rule condition as they are generated by `populate_facts`, and that `find_solution` prints the matches.

Example 1–24 Using Debugging Functions with Coins Example

```
RL> watchFacts();
RL> watchActivations();
RL> watchRules();
RL> reset();
RL> showActivations();
RL> run();
The rule is fired for each activation, printing out the solutions
RL>
```

In Example 1–25, the keyword `final` in front of a global variable definition such as `coinCount` and `totalAmount` marks that variable as a constant, as in Java. You can reference constants in rule conditions, but you cannot reference variables in rule conditions.

In RL Language, you must initialize all variables. The initialization expression for a `final` variable is evaluated once when the variable is defined. The initialization expression for a non-final variable is evaluated when the variable is defined, and again each time the `reset` function is called. Because the `reset` function retracts all facts from working memory, it is good practice to `assert` initial facts in a global variable initialization expression, so that the facts are re-asserted when `reset` is called.

Example 1–25 illustrates how to use global variable initialization expressions. The `initialized` global variable is initialized with the `populate_facts` function. This function is re-executed whenever `reset` is called. The `populate_facts` function has a `while` loop nested within a `for` loop. The `for` loop iterates over an array of coin denomination Strings. For each denomination, the `while` loop asserts a fact that expresses a count and a total that does not exceed the total amount of $1.50. For example, for half dollars:

```
coin(denomination "half-dollar", count:0, amount:0)
coin(denomination "half-dollar", count:1, amount:50)
coin(denomination "half-dollar", count:2, amount:100)
coin(denomination "half-dollar", count:3, amount:150)
```

With such facts in working memory, the rule `find_solution` matches against each denomination with a condition that requires that the counts sum to `coinCount` and the amounts sum to `totalAmt`. The `run` function fires the `find_solutions` activations.

Example 1–25 Count Coins Program Source

```
final int coinCount = 50;
final int totalAmt = 150;
final String[] denominations = new String[]
{"half-dollar" , "quarter", "dime", "nickel", "penny" };
class coin {
   String denomination;
   int count;
   int amount;
}
function populate_facts() returns boolean
{
```

```
   for (int i = 0; i < denominations.length; ++i) {
      String denom = denominations[i];
      int count = 0;
      int total = 0;
      int amount = 0;
      if (denom == "half-dollar" ) { amount = 50; }
      else if (denom == "quarter" ) { amount = 25; }
      else if (denom == "dime" ) { amount = 10; }
      else if (denom == "nickel" ) { amount = 5; }
      else { amount = 1; }

      while (total <= totalAmt && count <= coinCount)
      {
       assert(new coin(denomination: denom,
          count : count,
          amount : total));
          total += amount;
          count ++;
     }
   }
return true;
}
boolean initialized = populate_facts();
rule find_solution
{
   if(fact coin(denomination: "penny") p
   && fact coin(denomination: "nickel") n
   && fact coin(denomination: "dime") d
   && fact coin(denomination: "quarter") q
   && fact coin(denomination: "half-dollar") h
   && p.count + n.count + d.count + q.count + h.count == coinCount
   && p.amount + n.amount + d.amount + q.amount + h.amount == totalAmt)
   {
      println("Solution:"
             + " pennies=" + p.count
             + " nickels=" + n.count
             + " dimes=" + d.count
             + " quarters=" + q.count
             + " half-dollars=" + h.count
      );
   }
}
run();
```

2

Rule Language Reference

This chapter contains a detailed and complete reference to the Oracle Business Rules RL Language (RL Language) syntax, semantics, and built-in functions.

Grammar rules define the RL Language. Each grammar rule defines a non-terminal symbol on the left of the ::= symbol in terms of one or more non-terminal and terminal symbols on the right of the ::= symbol.

Reserved Words

aggregate, **boolean**, break, **byte**, **catch**, **char**, **class**, **constant**, continue, **double**, **else**, **exists**, **extends**, **fact**, **factpath**, **false**, **final**, **finally**, **float**, **for**, **function**, **hide**, **if**, **import**, **include**, **instanceof**, **int**, **long**, **modify**, **new**, **null**, **property**, **public**, query, **return**, **returns**, **rule**, **rulegroup**, **ruleset**, **short**, **supports**, **synchronized**, **throw**, **true**, **try**, **while**, **var**

> **Note:** Reserved words in bold apply to the current release. Reserved words that are not shown in **bold** typeface are planned for a future RL Language release, and include the words: break, continue, and query.

This chapter includes the following sections:

Ruleset

A ruleset groups a set of definitions. A ruleset is a collection of rules and other definitions that are all intended to be evaluated at the same time. A ruleset may also contain executable actions, may include or contain other rulesets, and may import Java classes and packages.

Format

ruleset ::= named-ruleset | unnamed-ruleset

named-ruleset ::= **ruleset** ruleset-name **{** unnamed-ruleset **}**

unnamed-ruleset ::= (import | include | named-ruleset | definition | action | fact-class | rulegroup)*

ruleset-name ::= identifier

Usage Notes

A named-ruleset creates or adds definitions to the specified ruleset named ruleset-name.

An unnamed-ruleset adds definitions to the default ruleset named `main`.

Rulesets may be nested, that is they may contain or include other rulesets. Nesting does not affect ruleset naming, but it does affect ruleset visibility in a way similar to Java import's affect on package visibility.

You can execute a ruleset using the RL Language command-line, or using the Java RuleSession API.

A named-ruleset ruleset-name must be unique within a `RuleSession`.

Examples

Example 2–1 contains two definitions, `enterRoom` and `sayHello`, and two actions (assert and run).

The rule shown in Example 2–1 will not fire until:

1. An `enterRoom` fact is asserted.
2. The run function executes, which pushes the rule's containing ruleset, `hello` onto the ruleset stack.

Example 2–1 Using a Named Ruleset

```
ruleset hello {
  class enterRoom { String who; }
  rule sayHello {
    if (fact enterRoom) {
      println("Hello " + enterRoom.who);
    }
  }
  assert(new enterRoom(who: "Bob"));
  run("hello");
}
```

In Example 2–2, if ruleset R2 is nested in ruleset R1, the name R2 must be unique within the rule session. R2 is not named relative to R1. For example, the class C2 defined in R2 is globally named R2.C2, not R1.R2.C2. If R2 is nested in R1, a public

class C1 defined in R1 may be referenced in R2 using either the full name R1.C1 or the short name C1 (assuming R2 does not also define C1).

Example 2–2 Using a Nested Ruleset

```
ruleset R1 {
  public class C1 {
    public String s;
  }
  C1 apple = new C1(s: "apple");
  ruleset R2 {
    public class C2 {
      public String s;
    }
    C1 c1 = apple;        // finds C1 and apple in containing ruleset R1
    c1.s = "delicious";
    C2 c2 = new C2(s: "pear");
  }
  R2.C2 pear = R2.c2; // finds R2.C2 and R2.c2 because they are fully qualified
  println(apple.s + " " + pear.s); // prints "delicious pear"

  pear = c2;  // UndefinedException: c2 not in R1 or a containing ruleset
}
```

Types

RL Language is a strongly typed language. Each variable and value has a specified type.

Format

type ::= simple-type [**[ ]**]

simple-type ::= primitive | object-type

primitive ::= **boolean** | numeric

numeric ::= **int** | **double** | **float** | **long** | **short** | **byte** | **char**

object-type ::= class-definition-name | Java-class-name

class-definition-name ::= qname

Java-class-name ::= qname

Type Conversion

There are several ways that a value can be converted from one type to another:

1. Conversion from any type to `String` using the `String` concatenation operator `+` .
2. Implicitly from context. For example, by adding an `int` to a `double` first converts the `int` to a `double` and then adds the 2 `doubles`.
3. Casting between 2 numeric types.
4. Casting between 2 classes related by inheritance.
5. Invoking a function or method that performs the conversion. For example, `toString`.

Table 2–1 summarizes the implicit conversions for various types. Rows indicate how the type in the From column may be implicitly converted, as shown in the list of types shown in the To column.

Table 2–1 Implicit Type Conversions

From	To
`int`	`double`, `float`, `long`
`float`	`double`
`long`	`double`, `float`
`short`	`int`, `double`, `float`, `long`
`byte`	`int`, `double`, `float`, `long`, `short`
`char`	`int`, `double`, `float`, `long`
`String`	`Object`
`Object`	`Object` (if the From Object is a subclass of the To Object)
`fact set`	`boolean`
`array`	`Object`

Note: An Object is an instance of a Java or RL Language class or array. Type conversion is possible only if the classes are related by inheritance (implements or extends).

Table 2–2 summarizes the allowed cast conversions for various types where a cast can be used to convert a primitive with more bits to a primitive with fewer bits, without throwing an exception.

The type conversions shown in Table 2–2 require an explicit cast operator. For example,

```
int i = 1;
short s = (short)i;
```

Note: Type conversions such as those shown in Table 2–2 that involve numeric types may lose high order bits, and such conversions involving Objects may throw a `RLClassCastExeption`.

Table 2–2 Explicit Type Conversions

From	To
`double`	`float`, `long`, `int`, `short`, `byte`, `char`
`float`	`long`, `int`, `short`, `byte`, `char`
`long`	`int`, `short`, `byte`, `char`
`short`	`byte`, `char`
`byte`	`char`
`char`	`byte`

When you use a cast to convert a primitive with more bits, to a primitive with fewer bits, the RL Language discards extra, high order, bits without throwing an exception.

For example,

```
short s = -134;
byte b = (byte)s;
println("s = " + s + ", b = " + b);
prints: s = -134, b = 122
```

Primitive Types

A primitive type may be any of the following

- An `int`, which is a 32 bit integer. Literal values are scanned by `java.lang.Integer.parseInt`
- A `long`. Literal values are scanned by `java.lang.Long.parseLong`
- A `short`. Literal values are scanned by `java.lang.Short.parseShort`
- A `byte`. Literal values are scanned by `java.lang.Byte.parseByte`
- A `char`.
- A `double`. Literal values are scanned by `java.lang.Double.parseDouble`

- A `float`. Literal values are scanned by `java.lang.Float.parseFloat`
- A `boolean` `true` or `false`

Object Types

An object type may be:

- A java `Object`, identified by the qualified name, qname, of its class. For example, `java.lang.String`.
- An RL Language `Object`, identified by the qualified name, qname of its class. For example, `ruleset1.Class1`.

String Types

RL Language uses Java strings, where:

- Strings are instances of the class `java.lang.String`.
- A string literal is delimited by double quotes ("string").

  Use \" to include the double quote character in a string.
- Strings may be concatenated using the + operator as follows:
  - If any operand of a + operator is a String then the remaining operands are converted to String and the operands are concatenated.
  - An Object is converted to a String using its `toString` method.
  - An instance of an RL Language class is converted to a String using a built-in conversion.

Array Types

Square brackets `[]` denote arrays. An array in RL Language has the same syntax and semantics as a Java 1-dimensional array.

Note: RL Language does not support multi-dimensional arrays.

Identifiers

RL Language supports both the Java and the XML variant of identifiers and namespace packages. To use the XML variant, the identifier must be enclosed in back quotes.

Format

identifier ::= java-identifier | xml-identifier

java-identifier ::= *valid-Java-identifier*

xml-identifier ::= `*valid-xml-identifier* or *URI*`

Where:

valid-Java-identifier is: a legal Java identifier, for example, JLd_0.

valid-xml-identifier is: a legal XML identifier, for example x-1.

URI is: a legal Uniform Resource Identifier, for example, http://www.oracle.com/rules

Usage Notes

An xml-identifier can contain characters that are illegal Java identifier characters, for example, ':' and '-'. The JAXB specification defines a standard mapping of XML identifiers to Java identifiers, and includes preserving the Java conventions of capitalization. The JAXB specification also defines a standard mapping from the schema target namespace URI to a Java package name, and a mapping from anonymous types to Java static nested classes.

Examples

RL Language supports both the Java and the XML variant of identifiers and namespaces or packages. Enclose an identifier in back quotes to use the XML variant, as shown in Example 2–3.

You can use the back quote notation anywhere an identifier or package name is legal in RL Language. To use the XML variant of identifiers in `String` arguments to assertXPath, back quotes are not needed.

Example 2–3 Sample Mapping for XML Identifiers Using Back Quotes

```
`http://www.example.com/po.xsd` -> com.mycompany.po
`my-attribute` -> myAttribute
`Items/item` -> Items$ItemType
```

Literals

Table 2–3 summarizes the RL Language literals. The literals are the same as Java literals.

Table 2–3 RL Language Literals

A literal such as	Can be assigned to variables of these types
An integer in range 0..127 or a `char` with UCS2 encoding in range 0...127	`byte`, `char`, `short`, `int`, `long`, `float`, `double`
An integer in range 0..65535 or a `char`	`char`, `int`, `long`, `float`, `double`
An integer in range -128..127	`byte`, `short`, `int`, `long`, `float`, `double`
An integer in range -32768..32767	`short`, `int`, `long`, `float`, `double`
An integer	`int`, `long`, `float`, `double`
An integer with `L` suffix	`long`, `float`, `double`
A floating point constant	`double`
A floating point constant with `F` suffix	`float`, `double`
A String enclosed in ""	`String`, `Object`

Definitions

When a definition within a ruleset is executed, it is checked for correctness and then saved for use later in the rule session.

Format

definition ::= variable | rule | rl-class-definition | function

name ::= identifier

qname ::= [ruleset-or-packagename**.**]name

ruleset-or-packagename ::= qname

Usage Notes

Every definition has a unique name within its containing ruleset, and thus a unique qualified name, qname, within the rule session.

Variables defined at the ruleset level are global. Global variables are visible to all expressions contained in the ruleset using the name of the variable and visible to expressions in other rulesets using the variable qname. Functions and public classes may also be referenced from other rulesets using the respective qname.

Java classes and their methods and properties also have qnames.

Example

The qname of the class definition in Example 2–4 is `hello.enterRoom`.

Example 2–4 Class Definition Within a Named Ruleset

```
ruleset hello {
  class enterRoom { String who; }
  rule sayHello {
    if (fact enterRoom) {
      println("Hello " + enterRoom.who);
    }
  }
  assert(new enterRoom(who: "Bob"));
  run("hello");
}
```

Variable Definitions

Variables are declared as in Java, but initialization is always required.

Format

variable ::= [**final**] (numeric name = numeric-expression

| **boolean** name = boolean-expression

| type **[ ]** name = array-expression | **null**

| object-type name = object-expression | **null**)

);

Usage Notes

The type of the array initialized with the array-expression must be the same as the type specified for the array elements.

A variable can have a primitive type, a Java class name, or an RL Language class name, and may be an array of elements of the same type.

The type of the object-expression must be the same as the object-type of the variable being declared. A class instance or array may be initialized to null.

Variables may be local or global in scope. The initialization expression is required. Local variables may not be final.

Global Variables

Variables immediately enclosed in a ruleset, that is, in a definition, are global to a rule session in scope. The initialization expression for a final global variable is executed when the global variable is defined.

The initialization expression for a non-final global variable is executed both:

- When the global variable is defined.
- Each time the `reset` function is called.

Global variables declared as final may not be modified after they are initialized.

Global variables referenced in a rule condition (fact-set-condition) must be final.

Examples

Example 2–5 shows that the reset function performs initialization for the non-final global variable `i`. Thus, this example prints 0, not 1.

Example 2–5 Non-Final Global Variable Initialization After Reset Function

```
RL> int i = 0;
RL> i++;
RL> reset();
RL> println(i);
```

Be careful when initializing global variables with functions that have side effects. If you do not want the side effects repeated when calling reset, you should declare the

variable `final`. For example, Example 2–6 prints "once" twice and Example 2–7 prints "once" once.

Example 2–6 Initializing a Global Variable with Side Effects with Reset

```
RL> clear;
RL> function once() returns int
{
  println("once");
  return 1;
}
RL> int i = once();
once
RL> reset();
once
RL>
```

Example 2–7 Initializing a Final Global Variable to Avoid Side Effects with Reset

```
RL> clear;
RL> function once() returns int
{
  println("once");
  return 1;
}
RL> final int i = once();
once
RL> reset();
RL>
```

Rule Definitions

The Oracle Rules Engine matches facts against the fact-set-conditions of all rules in the rule session to build the agenda of rules to execute. A fact set row is a combination of facts that makes the conditions of a rule true. An activation is a fact set row paired with a reference to the action-block of the rule. The agenda is the list of all activations in the rules session. The Oracle Rules Engine matches facts and rules when the state of working memory changes, typically when a fact is asserted or retracted.

The run, runUntilHalt, and step functions execute activations. Activations are removed from the agenda after they are executed, or if the facts referenced in their fact set row are modified or retracted such that they no longer match the rule's condition.

Activations are executed in order of the ruleset stack. You can manage the ruleset stack with the getRulesetStack, clearRulesetStack, pushRuleset, and popRuleset functions.

In order for a rule to fire, three things must occur:

1. An activation of that rule must be on the agenda.
2. The containing ruleset must be at the top of the ruleset stack.
3. You must invoke run, runUntilHalt, or step.

The fact set produced in a fact-set-condition is available to the rule actions. For each row in the fact set, the action-block is activated as follows:

- The rule's action-block is scheduled for execution at the specified rule priority.
- References from the action-block to the matched facts are bound to the current row.
- If a matched fact is retracted before the action-block is executed, the dependent activations are destroyed (removed from the agenda).

Format

rule ::= **rule** rule-name { property* fact-set-condition action-block }

rule-name ::= name

property ::= priority | autofocus | logical | active

priority ::= **priority =** numeric-expression

autofocus ::= **autofocus =** boolean-literal

logical ::= **logical =** (boolean-literal | *positive-integer-literal*)

active ::= **active =** boolean-literal

effectiveDateForm ::= **effectiveDateForm =** an int restricted to one of values defined in oracle.rules.rl.Rule: EDFORM_DATE, EDFORM_DATETIME, or EDFORM_TIME

effectiveStartDate ::= **effectiveStartDate =** expression of type `java.util.Calendar`

effectiveEndDate ::= **effectiveEndDate =** expression of type `java.util.Calendar`

Where:

positive-integer-literal is: an integer literal that is > 0

Usage Notes

The priority property specifies the priority for a rule. Within a set of activations of rules from the same ruleset, activations are executed in priority order (see "Ordering Rule Firing" on page 1-8). When rules have different priorities, the rules with a higher priority are activated before those with a lower priority. The default priority is 0. Within a set of activations of rules of the same priority, the most recently added activations are executed first, but this behavior can be changed (see the getStrategy and setStrategy functions).

A rule with the autofocus property equal to `true` automatically pushes its containing ruleset onto the ruleset stack whenever it is activated.

A rule with the logical property makes all facts asserted by the rule's action block dependent on some or all facts matched by the rule's condition. An integer value of *n* for the logical property makes the dependency on the first *n* top-level `&&`ed fact set expressions in the rule's condition. A boolean value of `true` for the logical property makes the dependency on the fact set expression of the condition. Anytime a fact referenced in a row of the fact set changes such that the rule's logical conditions no longer apply, the facts asserted by the activation associated with that fact set row are automatically retracted. A rule with the logical property enabled makes all facts that are asserted by an action block in the rule dependent on facts matched in the rule condition. Anytime a fact referenced in the rule condition changes, such that the rule's conditions no longer apply, the facts asserted by the rule condition are automatically retracted.

The active property defaults to `true`.

effectiveStartDate date defaults to null.

effectiveEndDate date default to null.

effectiveDateForm defaults to `Rule.EDFORM_DATETIME`

Examples

Example 2–8 shows a rule with the inference, Socrates is mortal, which depends on the fact, Socrates is a man.

Example 2–8 Defining and Using Rule allMenAreMortal

```
RL> clear;
RL> class Man    {String name;}
RL> class Mortal {String name;}
RL> Mortal lastMortal = null;
RL> rule allMenAreMortal {
  logical = true;
  if (fact Man)
  {
    assert(lastMortal = new Mortal(name: Man.name));
  }
}
RL> watchAll();
RL> Man socrates = new Man(name: "Socrates");
RL> assert(socrates);
 ==> f-1 main.Man (name : "Socrates")
==> Activation: main.allMenAreMortal :  f-1
RL> run();
Fire 1 main.allMenAreMortal f-1
 ==> f-2 main.Mortal (name : "Socrates")
 <== Focus main, Ruleset stack: {}
```

```
RL> retract(socrates);
 <== f-1 main.Man (name : "Socrates")
 <== f-2 main.Mortal (name : "Socrates")
RL> showFacts();
f-0   initial-fact()
```

Example 2–9 shows that it is possible for the same fact to be asserted by multiple rules, or to be asserted by a top-level ruleset action or function. Such a fact will not be automatically retracted unless all asserters have logical clauses that call for automatic retraction. A fact that is asserted by a top level action or function will never be automatically retracted.

Note that the fact that Socrates is mortal is not retracted, because it was asserted by a top level action that is not dependent on the fact that Socrates is a man.

Example 2–9 Asserting Facts Unconditionally

```
RL> assert(socrates);
 ==> f-3 main.Man(name : "Socrates")
==> Activation: main.allMenAreMortal :  f-3
RL> run();
Fire 1 main.allMenAreMortal f-3
 ==> f-4 main.Mortal(name : "Socrates")
 <== Focus main, Ruleset stack: {}
RL> assert(lastMortal);
 <=> f-4 main.Mortal(name : "Socrates")
RL> retract(socrates);
 <== f-3 main.Man(name: "Socrates")
RL> showFacts();
f-0   initial-fact()
f-2   main.Mortal(name: "Socrates")
```

Class Definitions

All referenced classes must be defined with an RL Language class definition or must be on the Java classpath (Java classes must be imported).

Both RL Language classes and Java classes can support xpath using the supports keyword, with a supplied xpath.

Format

rl-class-definition ::= [**public**] [**final**] **class** name [extends] [supports] { type-property* }

type-property ::= [**public**] type name [= expression] ;

extends ::= **extends** qname extended-class-name

extended-class-name ::= qname

Usage Notes

The type of the optional initialization expression must be the same as the type of the property or implicitly convertible to that type.

A public class is visible from all rulesets. A non-public class is visible only in the containing ruleset.

A final class cannot be extended.

The extended class must be a defined RL Language class not an imported Java class.

Each property may have an optional initializer. The initializer is evaluated when the class is instantiated by new. If an initial value is also passed to new, the value passed to new overwrites the value computed by the initializer in the class definition.

A public property is visible from all rulesets. A non-public property is visible only within its containing ruleset.

Examples

In RL Language, the type of an property may be the name of the containing class definition (see Example 2–10). RL Language, unlike Java, does not support forward references to class definitions (see Example 2–11).

Example 2–10 Class Definition with Type of Property with Name of Containing Class

```
class C0 {
  C0 next;
}
```

Example 2–11 Class Definitions with Forward References are Not Allowed

```
class C1 {
  C2 c2;  // causes an UndefinedException
}
class C2 {
  C1 c1;
}
```

xpath Support

> **Note:** xpath support has been deprecated. For more information, see assertTree.

Both RL Language classes and Java classes support xpath.

An XML identifier does not need to be surrounded by back quotes within an xpath.

The built-in assertXPath function supports a simple xpath-like syntax to assert a tree of objects as facts. The nodes in the tree are objects from classes in the same package or ruleset that support xpath. The links between parent and child nodes are instances of the `XLink` class. All of the properties in a class that supports xpath may be used in the xpath expression.

Format

supports ::= **supports xpath**

xpath ::= first-step next-step*

first-step ::= (**.** | **/*** | [**//**] (identifier | *****)) predicate*

predicate ::= **[** identifier xrelop literal **]**

next-step ::= (**/** | **//**) (identifier | *) predicate*

xrelop ::= **eq** | **lt** | **gt** | **le** | **ge** | **ne** | **==** | **<** | **>** | **<=** | **>=** | **!=**

literal ::= integer-literal | decimal-literal | double-literal | string-literal | **true** | **false** | dateTime-literal | date-literal | time-literal

integer-literal ::= [**-**] d+

d ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**

decimal-literal ::= [**-**] (**.** d+ | d+ **.** d*)

double-literal ::= [**-**] (**.** d+ | d+ [**.** d*]) (**e** | **E**) [**+** | **-**] d+

string-literal ::= " char* " | ' char* '

dateTime-literal ::= local-date **T** time-literal

date-literal ::= local-date [time-zone]

time-zone ::= **Z** | (**+** | **-**) d d **:** d d

local-date ::= d d d d **-** d d **-** d d

time-literal ::= d d **:** d d **:** d d [**.** d+] [time-zone]

Usage Notes

RL Language xpath support was designed to work with classes that conform to the Java XML Binding (JAXB) 1.0 standard. All JAXB elements from the given root to the elements selected by the xpath, inclusive, are asserted. Additional XLink facts are asserted to aid in writing rules about the parent-child relationships among the asserted elements.

If a JAXB element is retracted or re-asserted, using assert, then all of its children, and XLinks, are retracted. Instead of re-asserting, use assertXPath again.

Note that RL Language Xpath is not a proper subset of W3C Xpath 2.0. Note the following differences:

- The `lt` and `<`, `gt` and `>`, are synonymous in RL Language but different in W3C.
- Date literals must use `xs:date()` and other constructors in W3C.
- Constructors are not supported in RL Language, and literals, other than string literals, must not be quoted in RL Language.

Examples

Table 2–4 shows the xpath selection options for use with the built-in assertXPath function. In the descriptions, select means that the element is asserted as a fact, and the selected property of the XLink whose element property refers to the asserted element is `true`. The ancestors of a selected element, up to and including the root element, are always asserted, but not necessarily selected.

Table 2–4 xpath Selection Strings

xpath Select String	Description of Selection
//*	Select all elements including the root
.//*	Select all but the root
.	Select only the root
//*foo*	Select all objects that are the value of a property named *foo*.
.[*x*==1]/*y*	Select children or attributes of root named *y* only if the root has a child element or attribute named *x* and equal to 1

Example 2–12 instantiates an RL Language class called Person to build a family tree, as follows:

```
First Generation   Second Generation     Third Generation
Ida
                   Mary
                                          Fred
                                          John
                   Rachel
                                          Sally
                                          Evan
```

Example 2–12 uses the assertXPath function twice, with two xpaths:

```
//kids[male==true]
//kids[male==false]
```

Example 2–12 defines two rules:

- `sibling`: prints all pairs of siblings.
- `brotherSister`: prints all pairs of brothers and all pairs of sisters.

Example 2–13 shows the output from running Example 2–12.

Example 2–12 Sample Family Tree Rule Using supports xpath

```
import java.util.*;
ruleset xp {
  public class Person supports xpath {
    public String name;
```

```
    public boolean male;
    public List kids;
  }
        // Build the Family Tree
  Person p = new Person(name: "Fred", male: true);
  List k = new ArrayList();
  k.add(p);
  p = new Person(name: "John", male: true);
  k.add(p);
  p = new Person(name: "Mary", male: false, kids: k);
  Person gramma = new Person(name: "Ida", male: false, kids: new ArrayList());
  gramma.kids.add(p);
  p = new Person(name: "Sally", male: false);
  k = new ArrayList();
  k.add(p);
  p = new Person(name: "Evan", male: true);
  k.add(p);
  p = new Person(name: "Rachel", male: false, kids: k);
  gramma.kids.add(p);
  // test for siblings.
  // Note the test id(p1) < id(p2) halves the Cartesian product p1 X p2.
  rule sibling {
    if (fact Person p1 && fact Person p2 && id(p1) < id(p2) &&
        exists(fact XLink(element: p1) x &&
               fact XLink(element: p2, parent: x.parent))) {
      println(p1.name + " is sibling of " + p2.name);
    }
  }
  // test for brothers and sisters, given the following 2 assertXPath() calls
  rule brotherSister {
    if (fact Person p1 && fact Person p2 && id(p1) < id(p2) &&
        exists(fact XLink(element: p1, selected: true) x &&
               fact XLink(element: p2, selected: true,
                          parent: x.parent) y &&
               x.samePath(y))) {
      println(p1.name + " and " + p2.name + " are " +
              (p1.male ? "brothers" : "sisters"));
    }
  }
  assertXPath("xp", gramma, "//kids[male==true]");
  assertXPath("xp", gramma, "//kids[male==false]");
  run("xp");
}
```

Example 2–13 Output from Run of Family Tree Example

```
Mary and Rachel are sisters
Evan is sibling of Sally
Fred and John are brothers
Fred is sibling of John
Mary is sibling of Rachel
```

Example 2–14 shows that when you retract an element that was asserted with `assertXPath`, all its descendents are retracted as well.

The result is:

```
f-0   initial-fact()
```

For a total of 1 fact.

Example 2–14 Retract the Family Tree

```
retract(xp.gramma);
showFacts();
```

Example 2–15 prints all pairs of ancestors. First, the family tree is asserted. Example 2–16 shows the output of a run of the code from Example 2–15.

Example 2–15 Print Ancestor Pairs with Class Ancestor

```
assertXPath("xp", xp.gramma, "//*");
class Ancestor { Object element; Object ancestor; }
rule parents {
  if (fact XLink x) {
    assert(new Ancestor(element: x.element, ancestor: x.parent));
  }
}
rule ancestors {
  if (fact XLink x && fact Ancestor(ancestor: x.element) a) {
    assert(new Ancestor(element: a.element, ancestor: x.parent));
  }
}
rule printAncestor {
  if (fact xp.Person p && fact xp.Person a &&
      fact Ancestor(element: p, ancestor: a) {
    println(a.name + " is an ancestor of " p.name);
  }
}
run();
```

Example 2–16 Output from Run of Ancestor Example

```
Mary is an ancestor of John
Ida is an ancestor of John
Mary is an ancestor of Fred
Ida is an ancestor of Fred
Ida is an ancestor of Mary
Rachel is an ancestor of Evan
Ida is an ancestor of Evan
Rachel is an ancestor of Sally
Ida is an ancestor of Sally
Ida is an ancestor of Rachel
```

Function Definitions

A function is similar to a Java static method.

Format

function ::= **function** name parameters [**returns** type] action-block

parameters ::= **(** [type identifier (, type identifier)*] **)**

Usage Notes

The action-block may invoke the function being defined. However, the action-block may not contain a forward reference to a function that has not already been defined (see Example 2–17 and Example 2–18).

Functions may be overloaded. For example, the built-in println function is overloaded.

Examples

Example 2–17 Valid Function Definition Containing Recursive Reference

```
function factorial(long x) returns long {
  if (x <= 1) { return 1; }
  else { return x * factorial(x - 1); }
}
```

Example 2–18 Invalid Function Definition Containing Reference to Undefined Function

```
function f1() {
  f2();  // causes an UndefinedException
}
function f2() {
}
```

Fact Class Declarations

Any Java class can be used as an RL Language fact in a fact context.

A fact context is one of:

- The class of a fact-class declaration.
- The class of a fact-set-pattern.
- The declared class of an argument to the assert function.
- The declared class of an argument to the retract function.
- The declared class of an element argument to the assertXPath function.

If a class or interface `B` implements or extends class or interface `A`, and both `A` and `B` appear in fact contexts, then `A` must appear before `B`. Failure to follow this rule will result in a `FactClassException`.

Fact class definitions are not required when using RL Language classes.

For xpath support, use the supports xpath clause of the RL Language class definition.

Format

fact-class ::= **fact class** class-name [supports] (fact-class-body | **;**)

class-name ::= qname

fact-class-body ::= **{** [hidden-properties | properties]**}**

hidden-properties ::= **hide property * ;** | (**hide property** ((name **,**)* name | *) **;**)+

properties ::= **property * ;** | (**property** ((name **,**)* name | *) **;**)+

Usage Notes

The fact-class-body is optional in a fact-class declaration. The default fact-class-body is:

```
{ property *; }
```

Either the `property` or `hide property` keywords can be used in a body, but not both.

If `hide property` is used with a list of property names, then those property names are hidden and not available for use in RL Language.

If `hide property` is used with the wildcard "*", then no properties other than those exposed by a superclass or superinterface are available for use in RL Language.

If `property` is used with a list of property names, then those properties are exposed and available for use in RL Language. If property is used with the wildcard `*`, then all properties other than those hidden by a superclass or superinterface are available for use in RL Language.

A `HiddenPropertyException` will be thrown if a superclass exposes a property that its subclass hides or if a subclass exposes a property that its superclass hides.

Examples

Suppose a Java class `Vehicle` has subclasses `Car` and `Truck`. The rule shown in Example 2–19, `matchVehicle`, generates a `TypeCheckException` wrapping a `FactClassException` because the subclasses are referenced before the superclass. Wrapping is used instead of subclassing for both `FactClassException` and `MultipleInheritanceException` because in some fact contexts, these exceptions are not thrown until runtime and then are wrapped by a `RLRuntimeException`.

Example 2–19 matchVehicle Rule with Subclasses Referenced Before the Superclass

```
assert(new Car());   // fact context for Car
assert(new Truck()); // fact context for Truck
rule matchVehicle {
  if (fact Vehicle v) { // fact context for Vehicle - too late!
    if (v instanceof Car) {
      println("car");
     } else {
      println("truck");
     }
   }
 } // generates a TypeCheckException wrapping a FactClassException
```

In Example 2–20, the `matchVehicle` rule is the first reference to the superclass, so no exception is thrown.

Example 2–20 matchVehicle Rule with References to Superclass First

```
clear;
rule matchVehicle {
  if (fact Vehicle v) {
    if (v instanceof Car) {
      println("car");
     } else {
      println("truck");
     }
   }
 }
assert(new Car());
assert(new Truck());
run();  // prints "truck" then "car"
```

In Example 2–21, a fact class declaration is the first reference to the superclass, so no exception is thrown.

Example 2–21 matchVehicle Rule with Fact Class Declaration with Reference to Superclass First

```
clear;
fact class Vehicle;
assert(new Car());
assert(new Truck());
rule matchVehicle {
  if (fact Vehicle v) {
    if (v instanceof Car) {
      println("car");
     } else {
      println("truck");
     }
   }
```

```
 }
run();  // prints "truck" then "car"
```

Facts do not support multiple inheritance. Consider the Java classes and interfaces shown in Example 2–22.

Example 2–22 Java Classes and Sample Multiple Inheritance

```
package example;
public class Car {}
public interface Sporty {}
public class SportsCar extends Car implements Sporty {}
```

Example 2–23 entered at the command-line results in a `TypeCheckException` that wraps a `MultipleInheritanceException`. Use the `getCause` method on the `TypeCheckException` to retrieve the wrapped `MultipleInheritanceException` exception.

Example 2–23 MultipleInheritance Exception for Facts

```
import example.*;
fact class Sporty;
fact class Car;
fact class SportsCar; // throws TypeCheckException wrapping a
MultipleInheritanceException
```

Example 2–24 illustrates an exception that occurs at runtime when the Oracle Rules Engine attempts to assert the `rx8` object and discovers its true type is `SportsCar`, not `Object`. To avoid the `MultipleInheritanceException`, you must choose whether to use `Sporty` or `Car` in a fact class context. You cannot use both.

Example 2–24 RLRuntimeException wraps MultipleInheritanceException

```
import example.*;
fact class Sporty;
fact class Car;
Object rx8 = new SportsCar();
assert(rx8);  // throws RLRuntimeException wrapping a MultipleInheritanceException
```

Example 2–25 FactClassException Possible Cause

```
oracle.rules.rl.FactClassException: fact class for 'X' should be declared earlier
in rule session
```

Note the fact context rule is:

If X is a subclass or subinterface, of Y, then Y must appear in a fact context before X. A fact context is a fact-class declaration, a rule fact pattern, or the argument of assert, assertXPath, or retract.

In some cases you need to consider the fact context. For example, with an XML schema such as the following:

```
<schema>
  <element name=A type=T/>
  <complexType name=T>
    <sequence>
      <element name=B type=T/>
```

```
    </sequence>
  </complexType>
</schema>
```

JAXB generates:

```
interface T {
  List getB();  // List has TImpl objects
}
interface A extends T;
class AImpl implements A extends TImpl;
class TImpl implements T;
```

In an example with the following order of appearance in fact contexts:

1. `fact class T`
2. `assertXPath AImpl`
3. `assert TImpl` (performed internally by the `assertXPath` implementation)

The, `AImpl` precedes `TImpl` in the ordering, yet `AImpl` extends `TImpl`, which would give the exception. The fix for this fact context is to explicitly issue `fact class TImpl;` anywhere before Step 2.

Import Statement

An import statement makes it possible to omit the package name qualification when referencing Java classes.

Format

import ::= **import** (Java-class-name | Java-package-name**.***) **;**

Java-package-name ::= qname

Usage Notes

Import commands can be placed inside a ruleset, implying that the scope of the import is the ruleset where the import is located, but the import actually applies globally. For example, in the following code if the imports were scoped to the rulesets, then the `PrintWriter` reference in `r2` would not compile.

```
class X { }

ruleset rl {
  import java.io.*;
  rule A {
  if ( fact X ) {
    @ PrintWriter pw = null;
  }
}
}

ruleset r2 {
  rule B {
    if ( fact X ) {
      @ PrintWriter pw = null;
    }
  }
}
```

Include Statement

Include the ruleset at the location specified by the URL.

Format

include ::= **include** *URL* **;**

Where:

URL is: A legal Uniform Resource Locator.

Usage Notes

The `file:` and `http:` schemes are supported.

Example

```
include file:example.rl;
```

Using Expressions

Expressions in RL Language use familiar Java syntax (with minor variations as noted). For example,

```
(a + 1) * (b - 2)
```

Use expressions in a condition or in an action, with some restrictions. Expressions are strongly typed.

Format

expression ::= boolean-expression
| numeric-expression
| string-expression
| array-expression
| fact-set-expression
| object-expression
| comparable-expression

Boolean Expressions

Boolean expressions, as in Java, may be either `true` or `false`.

Format

boolean-expression ::= boolean-assignment

| boolean-expression ? boolean-expression : boolean-expression

| boolean-expression || boolean-expression

| boolean-expression && boolean-expression

| numeric-expression equal-op numeric-expression

| object-expression equal-op object-expression

| boolean-expression equal-op boolean-expression

| object-expression **instanceof** type-name

| numeric-expression relop numeric-expression

| string-expression relop string-expression

| **!** boolean-expression

| boolean-primary-expression

boolean-assignment ::= boolean-target-expression = boolean-expression

equal-op ::= **==** | **!=**

relop ::= **<** | **>** | **<=** | **>=**

type-name ::= qname

Usage Notes

For strings, < is Unicode UCS2 code point order.

For objects, `!=` does not test for inequality of object references, but rather is the negation of the equals methods.

Thus, the statement:

```
if (object1 != object2){}
```

Is equivalent to the statement:

```
if (! (object1.equals(object2)){}
```

RL Language, unlike Java, does not support testing for equality of object references.

Example

Example 2–26 shows use of a boolean expression in RL Language.

Example 2–26 RL Boolean Expression

```
if (
  (true ? "a" < "b" : false)
  && (1 == 0 || 1.0 > 0)
  && "x" instanceof Object )
{
  println("all true");
};
```

Numeric Expressions

Numeric expressions, as in Java, implicitly convert integer operands to floating point if other operands are floating point. Table 2–1 shows other implicit conversions.

Format

numeric-expression ::= numeric-assignment

| boolean-expression **?** numeric-expression **:** numeric-expression

| numeric-expression(**+** | **-**) numeric-expression

| numeric-expression (***** | **/** | **%**) numeric-expression

| numeric-expression ** numeric-expression

| (numeric-cast) numeric-expression

| (**+** | **-**) numeric-expression

| (**++** | **--**) numeric-primary-expression

| numeric-primary-expression [**++** | **--**]

numeric-assignment ::= numeric-target-expression (**=** | **+=** | **-=** | ***=** | **/=** | **%=**) numeric-expression

numeric-cast ::= numeric

Usage Notes

Table 2–5 shows the precedence order, from highest to lowest, for a numeric-expression.

Table 2–5 Expression Operator Precedence

Symbols	Category	Description
++ --	Post-increment or Post-decrement	numeric-primary-expression [++ | --]
++ --	Pre-increment or Pre-decrement	(++ | --) numeric-primary-expression
- +	Unary minus or Unary plus	(+ | -) numeric-expression
(type)	Type cast	(numeric *cast*) numeric-expression
**	Exponentiation	numeric-expression ** numeric-expression
*, /, %	Multiply or Divide or Remainder	numeric-expression (* | / | %) numeric-expression
+, -	Addition or Subtraction	numeric-expression(+ | -) numeric-expression
	Conditional	boolean-expression ? numeric-expression : numeric-expression
=	Assignment Operators	numeric-target-expression (= | += | -= | *= | /= | %=) numeric-expression

String Expressions

As in Java, any expression can be converted to a string using the concatenation + operator. In RL Language, unlike Java, when an array is converted to a string, the array contents are converted to a string, with array elements separated by commas and surrounded with curly braces. When an instance of an RL Language class is converted to a string, the class name appears followed by property value pairs separated by commas and surrounded with parentheses. This RL Language feature is useful for logging, tracing, and debugging.

When + operator is applied to an operand that is a `String`, then all operands are converted to `Strings` and the operands are concatenated.

Format

string-expression ::= string-assignment
| boolean-expression ? string-expression : string-expression
| string-expression + expression
| expression + string-expression
| string-primary-expression
|

string-assignment ::= string-target-expression (= | +=) string-expression

Example

Example 2–27 shows use of a string expression in RL Language. The example prints "1 2.0 true {1,2}"

Example 2–27 RL String Expression

```
int i = 1;
double f = 2.0;
boolean b = true;
int[] v = new int[]{i, 2};
println(i + " " + f + " " + b + " " + v);
```

Array Expressions

RL Language arrays behave just like Java arrays, but are limited to one dimension. The base type of an array is the type of the members of the array. All members must be of the same type. An array element may contain an array but only if the containing array is of type `Object[]`.

> **Note:** RL Language does not directly support multi-dimensional arrays.

Format

array-expression ::= array-assignment

| boolean-expression **?** array-expression : array-expression

| (array-cast) (array-expression | object-expression)

| array-primary-expression

array-assignment ::= array-target-expression = array-expression

array-cast ::= type

Usage Notes

The type of an array-cast must be an array type.

Fact Set Expressions

A fact-set-expression matches, filters, and returns facts from working memory. A fact-set-expression is legal only in a rule fact-set-condition. The `if` keyword indicates a fact-set-condition; however, a fact-set-condition is different from an if action. A rule's fact-set-condition iterates through all the rows in a fact set that match the fact-set-condition. The if action tests a boolean expression.

Format

fact-set-condition ::= **if** fact-set-expression

fact-set-expression ::= fact-set-expression **||** fact-set-expression
| fact-set-expression **&&** fact-set-expression
| fact-set-expression **&&** boolean-expression
| **!** fact-set-expression
| **exists** fact-set-expression
| fact-set-pattern
| (fact-set-expression)
| aggregate

fact-set-pattern ::= **fact** [(property-pattern (, property-pattern)*)]
[**var**] local-object-variable

local-object-variable ::= identifier

property-pattern ::= property-name : field-pattern

field-pattern ::= **var** local-property-variable | constraint

local-property-variable ::= identifier

simple-expression ::= string literal
| object-target-expression
| numeric literal
| numeric-target-expression
| boolean-literal
| boolean-target-expression

constraint ::= simple-expression

property-name ::= name

aggregate ::= **aggregate** fact-set-expression : aggregate-spec (, aggregate-spec)*

aggregate-spec ::= aggregate-function [**var**] identifier

aggregate-function ::=

average (numeric-expression **)**

| **sum (** numeric-expression **)**

| **minimum (** comparable-expression **)**

| **maximum** (comparable-expression)

| **count ()**

| **collection (** object-expression **)**

| user-defined **(** expression type Tin **)**

user-defined ::= qname

Usage Notes

A fact-set-expression can limit the facts it returns using either a simple-expression as a constraint in a fact-set-pattern or using a supported operator with the fact-set-expression.

A fact-set-expression may not contain any of the following:

- assert
- modify
- `new`
- References to non-final global variables.

Operator precedence is as in Java. Use parentheses to force desired precedence. For example,

```
fact person var p && (p.age < 21 || p.age > 65)
```

Without the parentheses, the `p` in `p.age` is undefined (see Table 2–5 for more details on operator precedence).

A local-object-variable or local-property-variable is in scope for all expressions following the pattern that are following the pattern and connected with the `&&` operator. If the pattern is not contained in an `exists`, `||`, or `!` expression, the variable is also in scope in the rule's action-block. The `&&`'ed expressions may filter the returned facts, so that only the facts surviving the filter are returned.

Fact Set Pattern - Fetch From Working Memory

The most primitive fact-set-expression is the fact-set-pattern that returns some or all facts of the given class that have been asserted into working memory. A fact-set-pattern searches working memory for facts of the given class and with the optional constraint on the property values. The returned fact set contains a row for each matching fact. A local row variable can be defined to refer to each row, or local field variables can be defined to refer to fields within a row. If no local row variable is supplied, the name part of the class qname can be used to refer to each row (see Example 2–31).

Join Operator

The `&&` operator defines the cross product or join of two fact-set-expression operands. The left-hand-side of a fact-set-expression `&&` operator must be a fact set. The right-hand-side of a *join* operator is another fact-set-expression. The result of applying the `&&` operator to two fact sets is the joined fact set.

Filter Operator

The `&&` operator defines a filter operator that rejects facts in its left-hand-side fact-set-expression that do not match the right-hand-side boolean-expression. The

left-hand-side of filter must be a fact-set-expression. The right-hand-side of a *filter* is a boolean-expression.

A filter right-hand-side may include references to variables defined, using the `var` keyword, in the left-hand-side.

Union Operator

The `||` operator defines the union of two fact-set-expression operands. When the `||` operator is applied to fact-set-expressions, the following is true:

- The expression's `vars` cannot be referenced outside the containing expression.
- The `||` returns the concatenation of its input fact sets, but the contents of the produced fact set are not accessible. Thus, `||` is typically used in a `!` or `exists` expression. Rather than a top-level `||` in a condition, it is usually better to use two or more rules with top-level `&&` operators so that `vars` defined in the condition can be referenced in the action-block.

Note: In the following construction:

```
if (fact X || fact W) {}
```

If both an `X` and a `W` are asserted, this rule fires twice, one time for each fact.

Empty Operator

The `!` operator tests if the fact-set-expression is empty. When the `!` is applied to the fact-set-expression, the following is true:

- The expression's `vars` cannot be referenced outside the containing `!` expression.
- The `!` operator returns a single row if the fact-set-expression is empty, else the `!` operator returns an empty fact set.

Exists (Not Empty) Operator

The `exists` operator tests if the fact-set-expression is not empty.

When the `exists` operator is applied to the fact-set-expression, the following is true:

- The expression's `vars` cannot be referenced outside the containing `exists` expression.
- The `exists` returns a single row if the expression is not empty, else `exists` returns an empty fact set.

Var Keyword

Note that when you use `var`, the fact is only visible using the `var` defined variable (and not using the original name). Thus, the following example works, assuming `action.kind` is defined:

```
if (fact action) {
   println(action.kind);
}
```

However, for the following example, after `var a` is defined, the `action.kind` reference produces a syntax error because you need to use `a.kind` after the `var a` definition.

```
if (fact action var a) {
```

```
  println(action.kind);
}
```

Aggregate

Aggregates support the following functions:

Table 2–6 Aggregate Functions

Function	Description
`average()`	Provides the average for matching facts. The result is `double`.
`sum()`	Provides the sum for the matching facts. The result is `double` or `long`.
`count()`	The result is `long`.
`minimum()`	Provides the minimum for the matching facts.
`maximum()`	Provides the maximum for the matching facts.
`collection()`	The result is `java.util.List` of Facts.
user defined	For a user-defined function the result is type Tout.

RL Language supports the aggregate pattern that applies one or more aggregate functions to a `factRowCollection`, and binds the aggregates to pattern variables. The usual SQL set of built-in aggregates is supported, and user-defined aggregates are supported when a user-supplied Java class is supplied.

If an aggregate function uses primitive wrapper types, for example Long, Double, then these will be unboxed such that the bind variable for the result has the appropriate raw primitive type.

If the fact expression in an aggregate is empty, then the rule will not fire. This ensures that if there are no matching facts for the expression, the aggregate function does not return a number that is meaningless in this context. For example, the "sum" of a property of a zero-size set is not meaningful.

For example, print the names of employees who are paid better than average:

```
if fact Emp emp && aggregate fact Emp(salary: var sal) : average(sal) var avgSal
&& emp.salary > avgSal {
   println(emp.name);
}
```

Print the names of employees, who are paid better than the average of employees who make over $100,000:

```
if fact Emp emp && aggregate fact Emp(salary: var sal) && Emp.salary > 100000 :
average(sal) var avgSal
&& emp.salary > avgSal {
   println(emp.name);
}
```

User-defined aggregates are supported by providing a public class named `user-defined` with a public 0-arg constructor that implements:

```
public interface IncrementalAggregate<Tin, Tout> extends Serializable
{
        public void initialize();
        public void add(Tin value);
        public void remove(Tin value);
        public Tout getResult();
```

```
        public boolean isValid();
}
```

Implementations must support the following invocation sequence:

```
new (initialize (add|remove)+ isValid getResult)*
```

`isValid` should return `true` when the result of the user defined aggregate is valid and `false` otherwise.

Examples

Example 2–28 shows the action is placed on the agenda for all `Counter` facts with a value of 1.

Example 2–28 Fact Set Expression for Counter.value

```
class Counter  { int id; int value; }
rule ex1a {
  if (fact Counter c && c.value == 1)
  { println("counter id " + c.id + " is 1"); }
}
```

Example 2–29 shows an equivalent way to express the rule from Example 2–28, using a constraint.

Example 2–29 Using a Fact Set Constraint

```
rule ex1b {
  if (fact Counter(value: 1) c)
  { println("counter id " + c.id + " is 1"); }
}
assert(new Counter(id: 1, value: 99));
run();  // prints twice, once for each rule
```

Example 2–30 shows an illegal use of a fact set, because `c` is used before it is defined.

Example 2–30 Illegal Use of Fact Set

```
rule ex2 {
  if (c.value == 1 && fact Counter c)
  { println("counter id " + c.id + " is 1"); }
}
```

Example 2–31 shows an action is placed on the agenda for all `AttFacts` with the property `a2==0` and without a matching, equal first elements, Counter.

Example 2–31 Using a Fact Set with && Operator for Counter Fact

```
class AttFact {int a1; int a2;}
rule ex3 {
  if (fact AttFact(a2: 0) && ! fact Counter(id: AttFact.a1))
  { println(AttFact.a1); }
}
assert(new AttFact()); // will match because a1=a2=0
assert(new AttFact(a1: 1, a2: 0));  // will not match
run();  // rule fires once
```

Example 2–32 shows the condition, `if (fact Ca a && fact Cb(v: a.v) b)` is interpreted as follows:

- The `fact Ca a` returns a fact set containing `a(v: 1), a(v: 2), a(v: 3)`
- The `&&` operator returns a fact set containing the two rows `{a(v: 1),b(v: 1)}, {a(v: 2),b(v: 2)}`

Example 2–32 Using a Fact Set with && Operator

```
class Ca {int v;}
assert(new Ca(v: 1));
assert(new Ca(v: 2));
assert(new Ca(v: 3));
class Cb {int v;}
assert(new Cb(v: 0));
assert(new Cb(v: 1));
assert(new Cb(v: 2));
rule r {
  if (fact Ca a && fact Cb(v: a.v) b) {
    println("row: " + a + " " + b);
  }
}
run();  // prints 2 rows
```

Comparable Expression

Comparable expressions allow objects that implement java.lang.Comparable to be compared using the ==, !=, <, <=, >, and >= operators. This allows dates to be easily compared. Also, `BigDecimal`, often used to represent money, can be compared in such expressions.

Format

comparable-expression ::=

qname *variable of type* implementing java.lang.Comparable

| member of type implementing java.lang.Comparable

Object Expressions

The only expression operators for objects are assignment and cast.

Format

object-expression ::= object-assignment | (ob-cast) object-expression |
boolean-expression ? object-expression : object-expression

object-assignment ::= object-target-expression = object-primary-expression

ob-cast ::= object-type

Primary Expressions

Primary expressions include assignment targets such as variables, properties, array elements, class members and other tightly binding expression syntax such as literals, method and function calls, and object and fact construction. The syntax is very similar to Java except where noted.

Format

primary-expression ::= array-primary-expression

| string-primary-expression

| numeric-primary-expression

| boolean-primary-expression

| object-primary-expression

array-primary-expression ::=

array-constructor

| function-call *returning array*

| method-call* *returning 1-dim Java array*

| **(** array-expression **)**

| array-target-expression

array-constructor ::= **new** (

simple-type **[** numeric-expression *integer* **]**

| numeric **[ ] {** numeric-expression (, numeric-expression)* **}** *numeric expression must be implicitly convertible to base*

| **boolean [ ] {** boolean-expression (, boolean-expression)* **}**

| object-type **[ ] {** object-expression (, object-expression)* **}**

)

array-target-expression ::=

qname *variable of type array*

| member *of type array*

| array-primary-expression *base type is Object* **[** numeric-expression *int* **]**

string-primary-expression ::=

string literal (see "Literals" on page 2-8)

| object-primary-expression *object is java.lang.String*

string-target-expression ::= object-target-expression *object is java.lang.String*

numeric-primary-expression ::=

numeric literal

| function-call *returning numeric*

| method-call *returning numeric*

| array-primary-expression . **length**

| **(** numeric-expression **)**

| numeric-target-expression

numeric-target-expression ::=

qname *variable of type numeric*

| member *of type numeric*

| array-primary-expression *base type is numeric* **[** numeric-expression **]**

boolean-primary-expression ::=

boolean-literal

| function-call *returning boolean*

| method-call *returning boolean*

| **(** boolean-expression **)**

| boolean-target-expression

boolean-literal ::= **true** | **false**

boolean-target-expression ::=

qname *variable of type boolean*

| member *of type boolean*

| array-primary-expression *base type is boolean* **[** numeric-expression *int* **]**

object-primary-expression ::=

new class-definition-name **(** [expression (, expression)*] *argument list* **)**

| **new** class-definition-name **(** [property-pattern (**,** property-pattern)*] *property-value pairs* **)**

| function-call *returning Java object*

| method-call *returning Java object*

| object-target-expression

object-target-expression ::=

qname *variable of type object*

| member *of type Java object*

| array-primary-expression *base type is object* [numeric-expression *int*]

function-call ::= qname *function name* **(** [expression (, expression)*] *argument list* **)**

method-call ::= object-primary-expression **.** identifier *method name*
([expression (**,** expression)*] *argument list* **)**

member ::= object-primary-expression **.** identifier *member name*

Examples

Example 2–33 shows the RL Language literal syntax (which is the same as Java).

Example 2–33 Use of Literals

```
String s = "This is a string."
int i = 23;
double f = 3.14;
boolean b = false;
```

Methods and functions can be overloaded. However, unlike Java, RL Language uses a first fit algorithm to match an actual argument list to the overloaded functions.

Example 2–34 shows an example of example of overloading

Example 2–34 Overloading

```
function f(int i);
function f(Object o);
function f(String s);  // can never be called

f(1);        // calls first f
f("a"); // calls second f, because "a" is an Object
```

new

RL Language classes do not have user-defined constructors. The default constructor initializes properties to their default values. The RL Language `new` operator permits specifying some property values (this works for Java bean properties, too).

A Java bean property may have a getter but no setter. Such a property may not be modified.

Example

Example 2–35 Initialization Using the New Operator

```
class C { int i = 1; int j = 2; }
C c = new C();
println(c); // c.i == 1 and c.j == 2
c = new C(i: 3);
```

```
println(c); // c.i == 3 and c.j == 2
c = new C(i: 0, j: 0);
println(c); // c.i == c.j == 0
```

Actions and Action Blocks

RL Language, unlike Java, requires action blocks and does not allow a single semicolon terminated action.

Format

action ::= action-block | if | while | for | try | synchronized | return | throw

| assign | incr-decr-expression | primary-action

action-block ::= **{** (variable | action)* **}**

Usage Notes

An action block is any number of local variable declarations and actions. The variables are visible to subsequent variable initialization expressions and actions within the same action block.

In RL Language, unlike in Java, all local variables must be initialized when they are declared. Local variables may not be final.

To exit, you can invoke the System.exit(*int*) method from within an action.

Example

Example 2–36 Action Block Sample

```
RL> {
  int i = 2;
  while (i-- > 0) { println("bye"); }
}
bye
bye
RL>
```

If Else Action Block

Using the if else action, if the *test* is `true`, execute the first action block, and if the *test* is `false`, execute the optional else part, which may be another if action or an action block.

RL Language, unlike Java, requires action blocks and does not allow a single semicolon terminated action.

Format

if ::= **if** if-test action-block [**else** if | action-block]

if-test ::= boolean-expression

Examples

Example 2–37 shows an RL Language if else action block. Example 2–38 shows that an action block is required.

Example 2–37 Sample If Else Action

```
String s = "b";

if (s=="a") { println("no"); } else
if (s=="b") { println("yes");}
else        { println("no"); }
```

Example 2–38 Illegal If Action Without an Action Block

```
if (s=="a") println("no");
```

While Action Block

While the test is `true`, execute the action block. A return, throw, or halt may exit the action block.

Format

while ::= **while** while-test action-block

while-test ::= boolean-expression

Usage Notes

RL Language, unlike Java, requires action blocks and does not allow single semicolon terminated action.

Examples

Example 2–39 prints "bye" twice.

Example 2–39 Sample While Action

```
int i = 2;
while (i-- > 0) {
  println("bye");
}
```

Example 2–40 Illegal While Action Without an Action Block

```
while (i-- > 0) println("no");
```

For Action Block

RL Language, like Java, has a for loop. Using the for action block, the for-init portion executes, then while the boolean-expression is `true`, first the specified *action block* is executed then the for-update executes. A return, throw, or halt may exit the action block.

Format

for ::= **for** **(** for-init ; boolean-expression ; for-update **)** action-block

for-init ::= variable | for-update

for-update ::= incr-decr-expression | assign | primary-expression

Usage Notes

RL Language does not allow a comma separated list of expressions in the `for init` or for update clauses (Java does allow this).

Example

Example 2–41 shows RL Language code that converts an `int[]` to a `double[]`.

Example 2–41 For Action

```
int[]    is = new int[]{1,2,3};
double[] fs = is; // error!
double[] fs = new double[3];
for (int i = 0; i < is.length; ++i) {
  fs[i] = is[i];
}
println(fs);
```

Try Catch Finally Action Block

Execute the first action block. Catch exceptions thrown during executions that match the `Throwable` class in a catch clause. For the first match, execute the associated catch action block. Bind the `Throwable` class instance to the given identifier and make it available to the catch action block. Whether an exception is thrown in the try action block, execute the finally action block, if given.

Uncaught exceptions are printed as error messages when using the RL Language command-line and are thrown as `RLExceptions` when using a RuleSession's `executeRuleset` or `callFunction` methods. The `try`, `catch`, and `finally` in RL Language is like Java both in syntax and in semantics. There must be at least one `catch` or `finally` clause.

Format

try ::= **try** action-block

(**catch (**class-implementing-throwable identifier **)** action-block)*

[**finally** action-block]

class-implementing-throwable ::= qname

Usage Notes

In order to fully understand how to catch exceptions in RL Language, one must understand how the stack frames are nested during rule execution. Rules do not call other rules the way that functions or methods may call functions or methods. Therefore, you cannot use a catch block in one rule's action block to catch exceptions in another rule's action block. Exceptions thrown during rule firing must either be handled by the firing rule's action block, or must be handled by a caller to the run, runUntilHalt, or step functions that caused the rule to fire.

Examples

Example 2–42 shows the try catch and finally actions. The output from running this example is:

```
exception in invoked Java method
this is really bad!
but at least it's over!
```

Example 2–42 Try Catch and Finally Action Blocks

```
try {
  throw new Exception("this is really bad!");
} catch (Exception e) {
  println(e.getMessage());
  println(e.getCause().getMessage());
} finally {
  println("but at least it's over!");
}
```

Note that RL Language treats the explicitly thrown `Exception ("this is really bad!")` as an exception from an invoked Java method, and wraps the `Exception` in a `JavaException`. The explicitly thrown `Exception` is available as the cause of the `JavaException`.

Synchronized Action Block

As in Java, the synchronized action is useful for synchronizing the actions of multiple threads. The synchronized action block lets you acquire the specified object's lock, then execute the action-block, then release the lock.

Format

synchronized ::= **synchronized** object-primary-expression action-block

Example

Example 2–43 changes the name of a `Person` object, adding old names to the nicknames, and synchronizes so that a concurrent reader of the Java object who is also synchronizing will see a consistent view of the `Person` (See Example 2–12 details on the `Person` bean).

Example 2–43 Synchronized Action

```
import example.Person;  // this Java bean is defined in example J1
function changeName(Person p, String first, String last) {
  synchronized(p) {
    java.util.Set s = p.getNicknames();
    s.add(p.getFirstName());
    s.add(p.getLastName());
    p.setFirstName(first);
    p.setLastName(last);
  }
  assert(p);
}
Person person = new Person("Elmer", "Fudd", new String[]{"Wabbit Wuver"});
println(person.nicknames.toArray());
changeName(person, "Bugs", "Bunny");
println(person.nicknames.toArray());
```

Modify Action

Modify updates the named properties using the associated expressions. It also updates the associated shadow fact, if any, and causes rules whose conditions reference the updated properties and evaluate to `true` to be activated. Rules whose conditions do not reference the updated properties are not activated.

The object argument to modify must be an object that has already been asserted, then the values of that object are updated and network is updated with the slot-specific semantics. The result is the object and the network are consistent.

Format

modify ::= **modify (**object-expression , property-update (, property-update)* **)**

property-update ::= property-name **:** expression.

Usage Notes

It is common for a fact to have properties that are set or modified by rules. For example, a customer in an application might have a status of "", "silver", or "gold". The status may be set by rules that examine other properties of customer and related facts (such as past orders). It is also common for these *computed properties* to be used in further rule conditions. For example, give gold customers a 10% discount. A rule that modifies a fact and reasserts it must be careful to add an extra condition so that it does not reactive itself over and over. For example, if the following rule fires once, it will fire over and over:

```
if fact Customer c && c.pastYearSpend > 1000 {
   c.status = "gold";
   assert(c);
}
```

You can fix this looping using the following rule definition:

```
if fact Customer c && c.pastYearSpend > 1000 && c.status != "gold" {
   c.status = "gold";
   assert(c);
}
```

Example 2–44 prevents the loop but does not activate rules that are looking for gold customers

Example 2–44 Example Showing Bad Rules Programming Practice to be Avoided

```
if fact Customer c && c.pastYearSpend > 1000 {
   c.status = "gold";
}
```

Example 2–44 demonstrates bad rules programming practice because it changes the value of the customer object but not the value of the, shadow, customer fact. The modify action lets you modify the object and fact together. Modify also activates rules that test the modified properties but does not activate rules that test non-modified properties.

```
if fact Customer c && c.pastYearSpend > 1000 {
   modify(c, status: "gold");
}
```

This rule does not loop because the tested properties and modified properties are disjoint. This rule can be used in an inference to fire subsequent rules that test for status=="gold".

A second rule that illustrates infinite looping is the rule described as follows:

Give Employees earning at least $50,000 a 5% raise.

```
if Employee emp && emp.sal > 50000 {
   modify(emp, sal: sal * 1.05);
}
```

Even using modify, this rule will self-trigger because it is testing the same property (`sal`) that it is modifying, and the test is `true` after modification. To avoid looping in this case, you could also add a `raise` property test, as follows:

```
if Employee emp && emp.sal > 50000 && !emp.raise {
  modify(emp, sal: emp.sal * 1.05, raise: true);
}
```

Alternatively, to avoid looping in this case you could also add a fact to handle the raise. For example:

```
public class RaiseGiven
{
   Employee emp;  // or possibly just an employee ID
}

if Employee emp && emp.sal > 500000 && !RaiseGiven(emp: emp) {
   modify(emp, sal: sal * 1.05);
   assert(new RaiseGiven(emp: emp));
}
```

Return Action

The return action returns from the action block of a function or a rule.

A return action in a rule pops the ruleset stack, so that execution continues with the activations on the agenda that are from the ruleset that is currently at the top of the ruleset stack.

If rule execution was initiated with either the run or step functions, and a return action pops the last ruleset from the ruleset stack, then control returns to the caller of the `run` or step function.

If rule execution was initiated with the runUntilHalt function, then a return action will not pop the last ruleset from the ruleset stack. The last ruleset is popped with runUntilHalt when there are not any activations left. The Oracle Rules Engine then waits for more activations to appear. When they do, it places the last ruleset on the ruleset stack before resuming ruleset firing.

Format

return ::= **return** [return-value] **;**

return-value ::= expression

If the function has a `returns` clause, then the return-value must be specified and it must be of the type specified by the `returns` clause.

Usage Notes

A return action in a rule or a `function` without a `returns` clause must not specify a return-value.

Throw Action

Throw an exception, which must be a Java object that implements `java.lang.Throwable`. A thrown exception may be caught by a `catch` in a try action block.

Format

throw ::= **throw** throwable **;**

throwable ::= object-primary-expression

Assign Action

An assignment in RL Language, as in Java, is an expression that can appear as an action.

Format

assign ::= assignment-expression ;

assignment-expression ::= boolean-assignment
| numeric-assignment
| string-assignment
| object-assignment
| array-assignment

Example

Example 2–45 shows the use of the RL Language assignment expression. This prints "6 5".

Example 2–45 Assignment Expression

```
clear;
int i = 1;
int j = 2;
i += j += 3;
println(i + " " + j);
```

Increment or Decrement Expressions

Increment and decrement in RL Language, as in Java, are expressions that can appear as actions.

Format

incr-decr ::= incr-decr-expression ;

incr-decr-expression ::= (++ | --) numeric-target-expression | numeric-target-expression (++ | --)

Examples

Example 2–46 shows the use of the RL Language decrement action. This example prints "0".

Example 2–46 Decrement Action

```
clear;
int i = 1;
--i;
println(i);
```

Primary Actions

A primary action is a primary expression such as a function call, assert, or Java method call executed for its side-effects. For example, the `println` function is often used as a primary action.

Format

primary-action ::= primary-expression ;

Rulegroup

A rulegroup provides support for decision table overrides. This supports the following decision table features: the ability for one rule to override one or more other rules

Format

rulegroup ::= **rulegroup** rulegroup-name { rulegroup-property* (rule | rulegroup)* }

rulegroup-name ::= identifier

rulegroup-property ::= mutex

mutex ::= **mutex =** boolean-literal ;

Usage Notes

A rulegroup construct is a top level construct in a ruleset. A rulegroup has an optional boolean property:

- mutex: The mutex property enables mutual exclusion between rules. The common case is that the rules reside in the same rulegroup that has set mutex to true. In more complex rulegroup hierarchies, two rules mutually exclude each other if their closest common ancestor rulegroup has mutex set to true. When a rules fires, existing activations for rules which it mutually excludes are removed from the agenda preventing them from firing. Theses activations must be directly related as identified by the set of facts that resulted in the activations being placed on the agenda. This leads to the requirement that the conditions of all rules that mutually exclude each other have the same fact clauses specified in the same order. This occurs naturally for decision tables.

Example

Example 2–47 demonstrates the use of rulegroups with the mutex property. Note that r2 and r3 reference the same set of fact types in the same order in the rule condition. This is required within a mutex group. The tests in the patterns can be different. The restriction on the shape of the rule condition in a mutex group extends to its descendent groups. Thus, rules r4 and r5 also must reference the same set of fact types in the same order. Assume that one instance of A and one instance of B have been asserted and that r1, r2, r3, r4 and r5 have activations on the agenda. r3 will fire since it has a higher priority. The activation of r2 will be removed without firing since r2 is in a group with mutex=true. Since group2 is a member of group1, the activations of r4 and r5 will also be removed without firing. Rule r1 will fire. Now, assume that r2, r4, and r5 have activations on the agenda and assume that r4 fires first. The activation for r2 will be remove without firing since any rule in group2 firing mutually excludes both r2 and r3. r5 will fire. It is not mutually excluded since group2 does not have mutex=true.

Example 2–47 Ruleset with Rulegroup with Specified Fact Order of Reference

```
ruleset set1
{
 rule r1 { ... }
 rulegroup group1
 {
   mutex = true;
   rule r2 { if fact A && fact B ... { ... }}
   rule r3 { priority = 2; if fact A && fact B ... { ... }}
```

```
  rulegroup group2
  {
    rule r4 { if fact A && fact B ... { ... }}
    rule r5 { if fact A && fact B ... { ... }}
  }
 }
}
```

Built-in Functions

This section covers the following RL Language built-in functions:

assert, assertTree, assertXPath, clearRule, clearRulesetStack, clearWatchRules, clearWatchActivations, clearWatchFacts, clearWatchFocus, clearWatchCompilations, clearWatchAll, contains, getCurrentDate, getDecisionTrace, getDecisionTraceLevel, getDecisionTraceLimit, getEffectiveDate, getFactByType, getFactsByType, getRulesetStack, getRuleSession, getStrategy, halt, id, isErrorInRuleConditionSuppressed, object, println, popRuleset, pushRuleset, retract, reset, run, runUntilHalt, setCurrentDate, setDecisionTraceLevel, setDecisionTraceLimit, setErrorInRuleConditionSuppressed, setEffectiveDate, setRulesetStack, setStrategy, showActivations, showFacts, step, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

assert

Adds a fact to working memory or updates a fact already in working memory based on the properties of the supplied object *obj*. If the supplied object *obj* is a Java instance, then properties are Java bean properties defined by an associated `BeanInfo` class or by the existence of getter and setter methods. If *obj* is an RL Language class instance, then the properties are the fields of the class.

Format

function assert(Object *obj*);

Usage Notes

The fact in working memory is a shadow of the supplied object *obj*, and this shadow contains a copy, clone, or reference to each property *prop*. If *prop* is a primitive type, then *prop* is copied to the shadow. If *prop* implements the Java `Cloneable` interface, then a clone, shallow copy, of *prop* is shadowed. Otherwise, only the reference to *prop* is shadowed. The more a shadow can copy its object's properties, the better a rule with references to several facts can be optimized.

Note that because `==` and `!=` when applied to an `Object` in RL Language always invokes the `Object` equals method, whether a shadow contains copies, clones, or references is transparent to the RL Language program.

Assert may affect the agenda. Rules whose conditions now return a fact set because of a new fact place activations on the agenda. Activations that test for non-existence of facts, using `!`, may be removed from the agenda. Updates to facts may affect the agenda. Activations whose rule conditions no longer match the changed facts are removed from the agenda. Rules whose conditions return a fact set because of the changed facts have activations placed on the agenda.

Assert should be used to update the fact in working memory if any part of the *obj*'s state has been updated that could possibly have an effect on a rule condition, unless the *obj* is a Java bean that supports registering property change listeners, and all that is changed is the value of a bean property.

Examples

Example 2–48 prints, "Pavi has highest salary 65000.0" and Example 2–49 prints, "dept 10 has no employees!".

Example 2–48 Using Assert Function in the highestSalary Rule

```
class Emp { String ename; double salary; }
               rule highestSalary {
                    if (fact Emp hi && !(fact Emp e && e.salary > hi.salary))
                       {
                          println(hi.ename + " has highest salary " + hi.salary);
                       }
               }
               Emp e1 = new Emp(ename: "Pavi", salary: 55000.00);
               assert(e1); // put in working memory
               Emp e2 = new Emp(ename: "Fred", salary: 60000.00);
               assert(e2);        // put in working memory
               e1.salary += 10000.00;    // Pavi is now the highest paid
               assert(e1);       // MUST re-assert before allowing rules to fire
               run();
```

Example 2–49 Using Assert Function in the emptyDept Rule

```
import java.util.*;
class Dept { int deptno; List emps = new ArrayList(); }
             rule emptyDept {
                  if (fact Dept d && d.emps.isEmpty()) {
                         println("dept " + d.deptno + " has no employees!");
                        }
                }
                Dept d = new Dept(deptno: 10);
                d.emps.add(e1);
                assert(d);         // put in working memory with 1 employee
                d.emps.remove(0);
                assert(d);         // MUST re-assert before allowing rules to fire
                run();
```

See Also

assertTree, id, object, retract

assertTree

The assertTree built-in function asserts object in an object tree as facts.

The assertTree built-in function supports JAXB 2.0.

Format

assertTree(Object *root*)

assertTree(String *spec*, Object *root*)

Usage Notes

There are two assertTree() signatures:

- assertTree(Object *root*) is necessary with SDK2. This format traverses the object graph and asserts objects as directed by internal metadata generated by SDK2.
- assertTree(String *spec*, Object *root*) is necessary when SDK2 is not involved. Similar to assertXPath, **spec** is a package specification enhanced to allow the specification of multiple packages separated by a colon. This format asserts objects and traverses the bean properties of an object if the object is in one of the specified packages. Typically, the package specification will be the same one used in establishing the JAXBContext.

The assertTree function does the following:

- asserts objects starting with the root and every fact (object with a visible declared fact type) referenced from the root, recursively.
- assert JAXBElement instances it encounters, extract the value class object it contains and continue.

See Also

assert

assertXPath

The assertXPath function is deprecated. Use assertTree instead.

Add a tree of facts to working memory using the specified *element* as the root and an XML xpath-like expression to define the objects in the tree. The *pkg* is the Java package or RL Language ruleset that contains the classes of objects in the tree. All objects in the tree must be in the same package or ruleset.

In addition to asserting "element" and selected descendants, XLink facts are asserted that link parent and child objects. The classes of all objects in the tree must use the **supports xpath** (supports) clause of the RL class (rl-class-definition) or fact-class declaration.

Format

function assertXPath(String *pkg*, Object *element*, String xpath);

See Also

assert, id, object, retract

clearRule

Clears the named rule from the rule session. Removes all of the rule's activations from the agenda.

Format

function clearRule(String *name*);

See Also

getRuleSession

clearRulesetStack

Empties the ruleset stack.

Format

function clearRulesetStack();

See Also

getRulesetStack, getStrategy, popRuleset, pushRuleset, run, setStrategy

clearWatchRules, clearWatchActivations, clearWatchFacts, clearWatchFocus, clearWatchCompilations, clearWatchAll

The clearWatch functions stop printing debug information.

Format

```
function clearWatchRules();
function clearWatchActivations();
function clearWatchFacts();
function clearWatchFocus();
function clearWatchCompilations();
function clearWatchAll();
```

See Also

watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

contains

The `contains()` function is similar to the `contains()` method on Java `Collection` but with includes the ability to handle the presence of JAXBElement in the collection.

Format

```
contains(Collection c, Object o) returns boolean
```

Usage

When `contains()` encounters a JAXBElement in the collection `c`, it obtains the value class from the JAXBElement and compares it to the Object `o`.

See Also

assertTree

getCurrentDate

The getCurrentDate function returns the date associated with the CurrentDate fact.

Format

getCurrentDate() returns Calendar

Usage Notes

The effective date and the current date are two orthogonal items. The effective date is used to determine which rules are in affect according to the start and end effective date properties. The CurrentDate fact allows reasoning on a rules engine managed fact representing the "current" date.

Setting the current date does not affect the effective date semantics.

If you want to create rules to reason on the date explicitly in the rules, then use the CurrentDate fact. If you want to assign start and or end effective dates to rules and have the rules in effect determined from a date in the data, then use `setEffectiveDate()`.

See Also

setCurrentDate, getEffectiveDate, setEffectiveDate

getDecisionTrace

Returns the current trace and starts a new trace.

Format

getDecisionTrace() returns DecisionTrace

Usage Notes

See Also

getDecisionTraceLevel, getDecisionTraceLevel, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

getDecisionTraceLevel

Gets the current decision trace level.

Format

getDecisionTraceLevel() returns int

Usage Notes

Supported decision trace levels include the following levels:

- Off: decision tracing is disabled. This is defined as `RuleSession.DECISION_TRACE_OFF`.
- Production: rules fired are shown in trace. This is defined as `RuleSession.DECISION_TRACE_PRODUCTION`.
- Development: full decision tracing similar in detail to `watchAll()`. This is defined as `RuleSession.DECISION_TRACE_DEVELOPMENT`.

See Also

getDecisionTrace, getDecisionTraceLimit, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

getDecisionTraceLimit

Returns the current limit on the number of events in a trace.

Format

getDecisionTraceLimit() returns int

Usage Notes

See Also

getDecisionTrace, getDecisionTraceLevel

getEffectiveDate

The getEffectiveDate function returns the current value of the effective date.

Format

getEffectiveDate() returns Calendar

Usage Notes

The effective date and the current date are two orthogonal items. The effective date is used to determine which rules are in affect according to the start and end effective date properties. The CurrentDate fact allows reasoning on a engine managed fact representing the "current" date. Both may be set explicitly.

Setting the current date does not affect the effective date semantics. Setting the effective date does not affect the CurrentDate fact.

If you want to create rules to reason on the date explicitly in the rules, then use the CurrentDate (`setCurrentDate()`). If you want to assign start and or end effective dates to rules and have the rules in effect determined from a date in the data, then use `setEffectiveDate()`.

See Also

setEffectiveDate, getCurrentDate, setCurrentDate

getFactByType

This function returns an instance of the fact type identified by className if exactly one instance exists in working memory. If no existence exists, null is returned. If more than one instance of the fact type exists in working memory an exception is thrown.

Format

function getFactByType(String *className*) returns Object

See Also

showFacts

getFactsByType

Returns a list of all facts in working memory that are instances of a specified class.

Format

function getFactsByType(String *className*) returns List

See Also

showFacts

getRulesetStack

Returns the ruleset stack as an array of ruleset names.

Format

```
function getRulesetStack() returns String[];
```

Usage Notes

Returns: the ruleset stack as an array of ruleset names.

Entry 0, the top of the stack, is the focus ruleset. The focus ruleset is the ruleset whose activations are fired first by a subsequent run, runUntilHalt, or step function execution.

See Also

clearRulesetStack, getStrategy, popRuleset, pushRuleset, setRulesetStack, setStrategy

getRuleSession

Returns a Java `RuleSession` object. An RL Language program could use this `RuleSession` to dynamically define new classes, rules, functions, or variables.

Format

function getRuleSession() returns RuleSession;

Example

```
rule learn {
  if (fact f1 && …)
  {
    RuleSession rs = getRuleSession();
    rs.executeRuleset("rule newRule { if fact f1 && fact f2 && … { … } }");
  }
}
```

See Also

clearRule

getStrategy

Returns the current strategy. Table 2–7 shows the possible strategy values.

Format

```
function getStrategy() returns String;
```

See Also

clearRulesetStack, getRulesetStack, popRuleset, pushRuleset, setStrategy

halt

The halt function halts execution of the currently firing rule, and returns control to the run, runUntilHalt, or step function that caused the halted rule to run. The agenda is left intact, so that a subsequent run, runUntilHalt, or step can be executed to resume rule firings.

The halt function has no effect if it is invoked outside the context of a run, runUntilHalt, or step function.

Format

```
function halt();
```

See Also

reset, run, runUntilHalt, step

id

Return the fact id associated with the object *obj*. If *obj* is not associated with a fact, returns -1.

Format

function id(Object *obj*) returns int;

See Also

assert, object, retract

isErrorInRuleConditionSuppressed

Error suppression happens at each test in the rule condition. Each test is evaluated separately and if an error happens during the evaluation and error suppression is enabled, the error is suppressed. When an error occurs during the evaluation of a test, it means that the result of the test is unknown. Internally, the rule engine uses three-valued logic to represent this state (true/false/unknown, see http://en.wikipedia.org/wiki/Three-valued_logic). When a true/false value is required, a value of unknown is mapped to false.

Errors in a test can either be in the RL domain (detected by the engine) or in the Java domain (thrown by a Java method invoked in the test). In the RL domain, `RLNullPointerException`, an attempt to de-reference a null object reference in an RL expression, is the most common error. `RLArithmeticException` (integer divide by 0) and RLIllegalArgumentException are infrequent. `RLArrayIndexOutOfBoundsException, RLClassCastException`, and `RLCloneNotSupportedException` are rare.

Java domain exceptions are those thrown by a Java method invoked from the rule condition. Those exceptions are wrapped in an RL exception, `JavaException` with the exception thrown by the method available via getCause(). When an exception is thrown by a Java method the engine can not make any further distinction. For example, it is possible that a null value passed to the method caused the method to throw an exception but the rule engine has no way to know whether or not the null was the cause.

Error suppression can be controlled by invoking a built-in function. A common place to invoke it is in the initial actions of a decision function.

Format

isErrorInRuleConditionSuppressed() returns boolean

Usage Notes

The RL signature is: isErrorInRuleConditionSuppressed() returns boolean.

See Also

setErrorInRuleConditionSuppressed

object

Return the object associated with the given fact id. If there is no such fact id, returns null.

Format

function object(int *factId*) returns Object;

See Also

assert, id, retract

println

Print the given value to the RuleSession output writer.

Format

function println(char *c*);

function println(char[] *ca*);

function println(int *i*);

function println(long *l*);

function println(float *f*);

function println(double *d*);

function println(boolean *b*);

function println(Object *obj*);

popRuleset

If the stack is empty, `popRuleset` throws `RLRuntimeException`. If the stack is not empty, `popRuleset` pops the focus off the stack and returns it.

All entries are shifted down one position, and the new focus is the new top of stack, entry 0.

Entry 0, the top of the stack, is the focus ruleset. The focus ruleset is the ruleset whose activations are fired first by a subsequent run, runUntilHalt, or step function execution.

Format

function popRuleset() returns String;

Example 2–50 Using popRuleset and Throwing RLRuntimeException

```
clearRulesetStack();
popRuleset();        // RLRuntimeException
```

See Also

clearRulesetStack, getRulesetStack, getStrategy, pushRuleset, setStrategy

pushRuleset

Push the given ruleset onto the stack and make it the focus. It is an error to push a ruleset that is already the focus (`RLIllegalArgumentException` is thrown for this error).

Entry 0, the top of the stack, is the focus ruleset. The focus ruleset is the ruleset whose activations are fired first by a subsequent run, runUntilHalt, or step function execution.

Format

function pushRuleset(String *focus*);

Examples

Example 2–51 shows the RL Language using the `pushRuleset` function.
Example 2–52 shows the RL Language using the `popRuleset` function.

Example 2–51 Using pushRuleset - Throws RLIllegalArgumentException

```
clearRulesetStack();
pushRuleset("main");        // focus is "main"
pushRuleset("main");        // RLIllegalArgumentException
```

Example 2–52 Using popRuleset - Throws RLRuntimeException

```
clearRulesetStack();
popRuleset();        // RLRuntimeException
```

See Also

clearRulesetStack, getRulesetStack, getStrategy, popRuleset, setStrategy

retract

Remove the fact associated with the object *obj* from working memory.

Format

function retract(Object *obj*);

Usage Notes

Retract may affect the agenda. Activations that depend on the retracted fact are removed from the agenda.

Note, rules that have conditions that test for non-existence of facts (using `!`) may place new activations on the agenda.

See Also

assert, id, object

reset

Clears all facts from working memory, clears all activations from the agenda, and reevaluates non-final global variable initialization expressions.

Format

```
function reset();
```

See Also

halt, run, runUntilHalt, step

run

Fire rule activations on the agenda until:

- A rule action calls halt directly or indirectly. For example, when halt is called by a function called by a rule action.
- The agenda is empty.
- The ruleset stack is empty.

Format

function run() returns int;

function run(String *rulesetName*) returns int;

Usage Notes

If the argument, *rulesetName* is supplied, the named ruleset is pushed on the top of the ruleset stack before firing any rules.

If a null *rulesetName* is supplied, the ruleset stack is not modified before firing rules.

If no *rulesetName* is supplied and the default `main` ruleset is not on the ruleset stack, then the `main` ruleset is placed at the bottom of the ruleset stack before firing any rules.

Returns: `int`, the number of rules fired.

See Also

halt, reset, runUntilHalt, step

runUntilHalt

This function fires rule activations until halt is called. Unlike run and step, runUntilHalt does not return when the agenda is empty. Also, runUntilHalt does not pop the bottommost ruleset name from the ruleset stack. Instead, it waits for the agenda to contain activations.

Format

```
function runUntilHalt() returns int;
```

Usage Notes

The only way for activations to be added to the agenda while the main RuleSession thread is busy executing runUntilHalt is for a second thread to either:

1. Modify Java bean facts with `PropertyChangeListeners`.
2. Execute assert or `retract` functions.

Rules must be designed carefully when using runUntilHalt. For example, a rule that attempts to find a fact with the minimum value of a property will fire when the first instance of the fact is asserted, and then every time another instance is asserted with a lower valued property.

See Also

halt, reset, run, step

setCurrentDate

The `setCurrentDate` function sets the date for reasoning on an engine managed fact representing the "current" date (with the `CurrentDate` fact).

Format

setCurrentDate(Calendar *newDate*)

Usage Notes

The `RLIllegalArgumentException` exception is thrown if the *newDate* argument is null.

If you need to reason on the date explicitly in the rules, then use the `CurrentDate` fact. If you want to assign start and end effective dates to rules and have the rules in effect determined from a date in the data, then they should use setEffectiveDate. The setEffectiveDate function does not affect the `CurrentDate` fact.

By default the value of the current date is managed implicitly by the rules engine. The value of the `CurrentDate` fact is updated to the current system date and (re)asserted internally when a run family of built-in functions is invoked. This is done before any rules fire so that the new current date is evaluated in rule conditions. In the case of runUntilHalt, this update occurs each time there is a transition from 0 rules on the agenda to > 0 rules on the agenda.

After the user invokes the `setCurrentDate` function, it becomes the responsibility of the user to update the current date as required. The rules engine no longer manages it implicitly. This remains in effect until the reset function is invoked. After the current date is set explicitly with `setCurrentDate`, any invocation of `setCurrentDate` function that would result in time going backward, set to an earlier point in time, is an error and an `RLIllegalArgumentException` is thrown. After the reset function is invoked, the current date may be set to any value.

See Also

getCurrentDate, getEffectiveDate, setEffectiveDate

setDecisionTraceLevel

Sets the decision trace level to the specified level.

Format

setDecisionTraceLevel(int level)

Usage Notes

Supported decision trace levels include the following levels:

- Off: decision tracing is disabled. This is defined as `RuleSession.DECISION_TRACE_OFF`.
- Production: rules fired are shown in trace. This is defined as `RuleSession.DECISION_TRACE_PRODUCTION`.
- Development: full decision tracing similar in detail to `watchAll()`. This is defined as `RuleSession.DECISION_TRACE_DEVELOPMENT`.

See Also

getDecisionTrace, getDecisionTraceLevel, getDecisionTraceLevel, setDecisionTraceLimit, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

setDecisionTraceLimit

Sets the limit on the number of events in a trace.

Format

setDecisionTraceLimit(int count)

Usage Notes

The default value is 10000.

See Also

getDecisionTrace, getDecisionTraceLevel, getDecisionTraceLevel, setDecisionTraceLevel

setEffectiveDate

The setEffectiveDate function updates the effective date in the rules engine.

By default, the value of the effective date is managed implicitly by the rules engine. In this case, when a run family of built-in functions is invoked the effective date is updated to the current system date. This is done before any rules fire so that the new effective date is applied before rules begin to fire. In the case of `runUntilHalt`, this update occurs each time there is a transition from 0 rule activations on the agenda to > 0 rule activations on the agenda.

Format

setEffectiveDate(Calendar *newDate*)

Usage Notes

Invoking setEffectiveDate is the only way that you can alter the effective date. After the reset function is invoked, the effective date may be set to any value.

The `RLIllegalArgumentException` exception is thrown if the *newDate* argument is null.

After you invoke the setEffectiveDate function, it becomes the responsibility of the application to update the effective date as required. The rules engine no longer manages it implicitly.

This remains in effect until the reset function is invoked.

This is useful for debugging, performing rule evaluation at a "point in time", or other use cases that require application control of the effective date.

See Also

getEffectiveDate, getCurrentDate, setCurrentDate

setErrorInRuleConditionSuppressed

Error suppression happens at each test in the rule condition. Each test is evaluated separately and if an error happens during the evaluation and error suppression is enabled, the error is suppressed. When an error occurs during the evaluation of a test, it means that the result of the test is unknown. Internally, the rule engine uses three-valued logic to represent this state (true/false/unknown, see http://en.wikipedia.org/wiki/Three-valued_logic). When a true/false value is required, a value of unknown is mapped to false.

Errors in a test can either be in the RL domain (detected by the engine) or in the Java domain (thrown by a Java method invoked in the test). In the RL domain, `RLNullPointerException`, an attempt to de-reference a null object reference in an RL expression, is the most common error. `RLArithmeticException` (integer divide by 0) and RLIllegalArgumentException are infrequent. `RLArrayIndexOutOfBoundsException, RLClassCastException`, and `RLCloneNotSupportedException` are rare.

Java domain exceptions are those thrown by a Java method invoked from the rule condition. Those exceptions are wrapped in an RL exception, `JavaException` with the exception thrown by the method available via getCause(). When an exception is thrown by a Java method the engine can not make any further distinction. For example, it is possible that a null value passed to the method caused the method to throw an exception but the rule engine has no way to know whether or not the null was the cause.

Error suppression can be controlled by invoking a built-in function. A common place to invoke it is in the initial actions of a decision function.

Format

setErrorInRuleConditionSuppressed(boolean bv) returns boolean

Usage Notes

The RL signature is: setErrorInRuleConditionSuppressed(boolean bv) returns boolean.

Passing a value of true will enable error suppression. The return value is the previous setting.

The current setting can be queried with the isErrorInRuleConditionSuppressed built-in function.

See Also

isErrorInRuleConditionSuppressed

setRulesetStack

Sets the ruleset stack to the given array of ruleset names.

Entry 0, the top of the stack, is the focus ruleset, which is the ruleset whose activations will be fired first by a subsequent run, runUntilHalt, or step function execution.

Format

```
function setRulesetStack(String[] rulesetStack
```

See Also

clearRulesetStack, getRulesetStack, getStrategy, popRuleset, pushRuleset, setStrategy

setStrategy

Strategy specifies the order in which activations from the same ruleset and with the same priority are executed. Table 2–7 shows the valid strategy values.

Table 2–7 Strategy Values for setStrategy and getStrategy Functions

Strategy	Description
queue	Activations are fired in order from oldest to newest.
stack	Activations are fired in order from newest to oldest.

Format

function setStrategy(String *strategy*);

See Also

clearRulesetStack, getRulesetStack, getStrategy, popRuleset, pushRuleset

showActivations

The show functions print rule session state to the output Writer. State that can be shown is: Activations all activations on the agenda

Format

```
function showActivations();
```

See Also

clearWatchRules, clearWatchActivations, clearWatchFacts, clearWatchFocus, clearWatchCompilations, clearWatchAll, showFacts, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

showFacts

The show functions print rule session state to the output Writer. State that can be shown is: all facts in working memory.

Format

```
function showFacts();
```

See Also

clearWatchRules, clearWatchActivations, clearWatchFacts, clearWatchFocus, clearWatchCompilations, clearWatchAll, showActivations, watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

step

Fire rule activations on the agenda until:

- The specified number of rule activations, *numRulesToFire* have been fired.
- A rule action calls halt directly or indirectly. For example, by a function called by a rule action.
- The agenda is empty.
- The ruleset stack is empty.

Format

function step(int numRulesToFire) returns int;

function step(int numRulesToFire, String rulesetName) returns int;

function step(int numRulesToFire, boolean errorIfLimitHit) returns int;

function step(int numRulesToFire, String rulesetName, boolean errorIfLimitHit) returns int;

Usage Notes

If no ruleset name is supplied and the `main` ruleset is not on the ruleset stack, then the `main` ruleset is placed at the bottom of the ruleset stack before firing any rules.

If a ruleset named, *rulesetName*, is supplied, the specified ruleset is pushed on the top of the ruleset stack before firing any rules. If a null ruleset name is supplied, the ruleset stack is not modified before firing rules.

Returns the integer number of rules fired.

The last two signatures allow the caller to specify whether hitting the firing limit (numRulesToFire) should be treated as an error condition or not. The default is false, it is not an error condition to hit the limit.

Invoking the step function with a value of true for the errorIfLimitHit argument can be used to catch infinite rule firing loops that can occur due to a bug in the rules as written.

See Also

halt, reset, run, runUntilHalt

watchRules, watchActivations, watchFacts, watchFocus, watchCompilations

The watch functions turn on printing of information about important rule session events. The information is printed to the output Writer whenever the events occur. Use a clearWatch function to turn off printing.

Table 2–8 describes the available debugging information.

Table 2–8 Watch Functions Event Descriptions

Debug Watch	Rule Session Event Description
watch	Rule session event description
Rules	Information about rule firings (execution of activations)
Activations	Addition or removal of activations from the agenda
Facts	Assertion, retraction, or modification of facts in working memory
Focus	Pushing or popping of the ruleset stack. The top of the ruleset stack is called the *focus ruleset*, and all activations on the agenda from the focus ruleset will be fired before the focus is popped and the next ruleset on the stack becomes the focus.
Compilations	When a rule's conditions are added to the rete network, information about how the condition parts are shared with existing rules is printed. "=" indicates sharing. The order that rules are defined can affect sharing and thus can affect performance.
All	Includes information shown with watch Rules, watch Activations, watch Facts, watch Compilations and watch Focus.

Format

```
function watchRules();

function watchActivations();

function watchFacts();

function watchFocus();

function watchCompilations();

function watchAll();
```

See Also

clearWatchRules, clearWatchActivations, clearWatchFacts, clearWatchFocus, clearWatchCompilations, clearWatchAll, showActivations, showFacts

3

Using the Command-line Interface

This chapter describes the RL command-line that reads rulesets from `System.in` and writes output from the functions `println`, `watch`, and `show` to `System.out`.

This chapter includes the following sections:

3.1 Starting and Using the Command-Line Interface

The following invocation provides a simple command-line interface, with the prompt, `RL>`. Example without Java Beans:

```
java -jar SOA_ORACLE_HOME/soa/modules/oracle.rules_11.1.1/rl.jar -p "RL> "
```

Where *SOA_ORACLE_HOME* is where SOA modules are installed (for example, `c:/Oracle/Middleware`). The `-p` option specifies the prompt.

The following shows how an RL Language command-line can be started that can access this Java bean:

```
java -classpath SOA_ORACLE_HOME/soa/modules/oracle.rules_11.1.1/rl.jar;BeanPath
 oracle.rules.rl.session.CommandLine -p "RL> "
```

Where *BeanPath* is the classpath component to any supplied Java Bean classes.

To exit the command-line interface, use the special action `exit;` at the command prompt. The `exit;` action cannot be in an included ruleset. Alternatively, to exit you can invoke the `System.exit(`*int*`)` method in any action.

The RL command-line interface accumulates input line by line, and interprets the input when the input stream includes either:

- A complete named ruleset
- One or more complete `import`, `include`, `ruleset`, `definition`, *action* commands within an unnamed ruleset.

Note: The `if`,`else` and `try`, `catch`, and `finally` actions require lookahead to determine where they end. In order to execute an `if` without an `else` clause, or a `try` without a `finally` clause at the RL command-line, you should add a semicolon terminator.

This is not necessary if you execute RL using `include`, or using the RuleSession API.

Example 3–1 Sample RL Command-Line Input Processing

```
RL> int i = 1;
RL> if (i > 0) {println("i positive");}
// nothing happens - waiting for possible "else"
;
i positive
RL>
```

Input must be complete at the end of a line. For example, if an action ends in the middle of a line, then that action is not interpreted until some following action is complete at the end of a line.

Example 3–2 Sample Command-Line Input Processing - Waiting for End of Line

```
RL> println("delayed"
); println("hello"
); println("world");
delayed
hello
world
RL>
```

Notes for using command-line input processing:

1. The command-line segments its input into blocks and then feeds each block to the interpreter. If you never type a closing brace or semicolon, no error is raised because the command line waits for input before it does a full parse of the block
2. The command-line interpreter, when used interactively or with the –`i` option, collapses the input, for line numbering purposes, into "small" rulesets ending at a newline. Errors are reported with numbers within the ruleset.

   For example, if the input consists of the following:

   ```
   int i = 0; i = 1; // this is a ruleset
   i = "i";  // this is another ruleset
   ```

   For this example, command-line reports an error as follows:

   ```
   Oracle Business Rules RL: type check error
   ConversionException: cannot convert from type 'java.lang.String' to type 'int'
   at line 1 column 5 in main
   ```

   To avoid this behavior, you can explicitly enclose the input in a ruleset. For example,

   ```
   ruleset main {
        int i = 0; i = 1;
        i = "i";
   }
   ```

Now, the error is on line 3 or, you can include the input file using an include.

3.2 RL Command-Line Options

Table 3–1 RL Command-Line Options

Flag	Description
–i	Read rulesets from the file named by the next argument, instead of from the default, `System.in`. For example, `-i myInput.rl` Note: the command-line segments its input into blocks and then feeds each block to the interpreter. If the file `myInput.rl` does not include a closing brace or semicolon at the end, then, no error is raised because the command line waits for additional input before it does a full parse of the block. Thus, there are cases where an incomplete input file supplied using the `–i` option could run and execute the valid part of the code from the file `myInput.rl`, and exit, while still waiting for command line input.

–c	Executes the next argument as the first RL command, then start reading input. This option is useful to include a file of settings and functions for debugging.
	For example,
	`-c "include file:debugSettings.rl;"`
	If you do not want to read from the input after executing the command, include "exit;" after the command.
	For example,
	`-c "include file:script.rl; exit;"`

–p	Sets the next argument as the prompt string. For example, `-p "RL> "`
-o	Specifies where to write output from `println`, `watch`, and `show` to the file named by the next argument, instead of to `System.out`. For example: `-o debug.log`
–v	Print version information.

3.3 RL Command-Line Built-in Commands

This section lists commands that are implemented by the RL command-line interface (these commands are not part of RL). Thus, these commands cannot appear in blocks or be included rulesets.

3.3.1 Clear Command

Discard the current RuleSession object and allocate a new one. The effect is that all rules, variables, classes, and functions are discarded.

Instead of using `clear;` to restart a command-line you can also type `exit;` and then reissue the Java command to start another command-line.

3.3.2 Exit Command

Exit the command-line interface. The command-line interface also exits when end-of-file is reached on its input.

4

Using a RuleSession

This chapter describes how to use a RulesSession object.

This chapter includes the following sections:

4.1 RuleSession Constructor Properties

This section shows you the steps for creating a rule enabled application and describes using a `RuleSession` object. The package oracle.rules.rl contains the `RuleSession` object.

The `RuleSession` no argument constructor returns a `RuleSession` with the default locale and logging options set with default configuration parameters.

Table 4–1 describes the configuration parameters that can be set in a Map passed to the RuleSession constructor.

Table 4–1 Configuration Parameters for a RuleSession Constructor

Parameter Key	Value
CFG_LOGGING	Boolean. True enables logging. False disables logging. The default is true.
CFG_LOCALE	The Locale instance for the desired locale. The default is the JVM default locale.

Table 4–1 (Cont.) Configuration Parameters for a RuleSession Constructor

Parameter Key	Value
CFG_WATCH	The desired setting for the watch raw activity trace facility. The setting is restored when the session is reset. WATCH_RULES: watch rules that fire. WATCH_ACTIVATIONS: watch rule activations and deactivations. WATCH_FACTS: watch fact operations assert, modify, retract. WATCH_FOCUS: watch ruleset stack changes. WATCH_COMPILATION: watch rule definintion. WATCH_ALL: watch all of the above. The default is that no trace settings are enabled.
CFG_DECISION_TRACE_LEVEL	Sets the decision trace level which controls the rule engine activity traced. This level is restored when the session is reset. DECISION_TRACE_OFF: disables all decision tracing. DECISION_TRACE_PRODUCTION: traces rules that fire. DECISION_TRACE_DEVELOPMENT: detailed decision tracing. Equivalent to WATCH_ALL plus tracing of reset. The default is DECISION_TRACE_OFF.
CFG_DECISION_TRACE_LIMIT	An integer that sets the limit on the number of trace entries that will be kept internally until the trace is retrieved. The default decision trace limit is 10000.

4.2 RuleSession Methods

The `outputWriter` property determines where `println`, `watch`, and `show` output goes.

The rulesetName property sets the ruleset when RL statements are executed without an explicit named ruleset. The default rulesetName is `main`.

The executeRuleset methods parse and execute the given ruleset text (given as a String or a java.io.Reader).

The callFunction method invokes the named RL function (which must either be a built-in RL function or must have been previously defined with no parameters using one of the executeRuleset methods) and returns its result. Functions with a single argument can be invoked with the callFunctionWithArgument method. Functions taking any number of arguments can be called using the `callFunctionWithArgumentList` or `callFunctionWithArgumentArray` methods. The argument List or array must contain a Java Object for each RL function parameter.

4.3 RL to Java Type Conversion

Table 4–2 describes how Java Object types are be converted to RL types for passing arguments to RL functions, and conversely how RL types are converted to Java types for passing the RL function return value to Java.

Table 4–2 RL to Java Object Conversion

Java Class	RL Type
java.lang.Integer	int
java.lang.Character	char
java.lang.Byte	byte
java.lang.Short	short
java.lang.Long	long
java.lang.Double	double
java.lang.Float	float
java.lang.Boolean	boolean
Object	Object
int[]	int[]
char[]	char[]
byte[]	byte[]
short[]	short[]
long[]	long[]
double[]	double[]
float[]	float[]
boolean[]	boolean[]
Object[]	Object[]

4.4 Error Handling

RuleSession method invocations that throw a `ParseException` or `TypeCheckException` do not affect the state of the RuleSession. A Java application, for example, an interactive command-line, can catch these exceptions and continue using the RuleSession.

RuleSession method invocations that throw a `RLRuntimeException` may have affected the state of the RuleSession and the RuleSession may not be in a usable state for the application to proceed. Robust applications should attempt to catch and recover from `RLRuntimeExceptions` in RL at a point near where the exception is thrown.

Other exceptions likely indicate a serious problem that the application cannot handle.

4.5 RL Class Reflection

You can use an RL class like a Java class in an RL program. The `new`, `instanceof`, and `cast` operators work on both kinds of class. However, when an instance of an RL class is passed to a Java program, it is actually an instance of `oracle.rules.rl.RLObject`. A Java program can use the following classes: `RLClass`, `RLProperty`, and `RLArray` to examine the `RLObject` in a manner similar to using the `java.lang.Class`, `java.lang.reflect.Field`, and `java.lang.Array` classes to reflect a `java.lang.Object`. The package `oracle.rules.rl` contains `RLCLass`, `RLProperty`, and `RLArray`.

4.6 XML Navigation

XLink objects are created and asserted as facts by the assertTree function. An RL rule can use XLinks to reason about the hierarchy of elements asserted by assertTree.

4.7 Obtaining Results from a Rule Enabled Program

When you create a a rule enabled program with Oracle Business Rules, a common question is, "How do I get the results of the evaluation?"

This section one approaches to extracting or exposing results of rule evaluation from the rule engine.

This section covers the following:

- Overview of Results Examples
- Using External Resources to Obtain Results

> **See Also:** "Working with Rules SDK Decision Point API" in the *Oracle Business Rules User's Guide*

4.7.1 Overview of Results Examples

The examples in this section show a highway incident notification system. These examples show the different approaches to access the results of rule engine evaluation. The examples use two Java classes: `traffic.TrafficIncident` and `traffic.IncidentSubscription`.

> **Note:** The `traffic.*` sample classes are not included in the Oracle Business Rules distribution.

The `TrafficIncident` class represents information about an incident affecting traffic and contains the following properties:

- Which highway
- Which direction
- Type of incident
- Time incident occurred
- Estimated delay in minutes

The `IncidentSubscription` class describes a subscription to notifications for incidents on a particular highway and contains the following properties:

- Subscriber - the name of the subscriber
- The highway
- The direction

In the example using these classes, when an incident occurs that affects traffic on a highway, a `TrafficIncident` object is asserted and rule evaluation determines to whom notifications are sent.

In the examples, the `sess` object is a `RuleSession` and a number of incident subscriptions are asserted. As a simplification, it is assumed that the `TrafficIncident` objects are short lived. They are effectively an event that gets asserted and only those subscribers registered at that time are notified.

The classes in these examples are all Java classes. However, it is possible to manipulate instances of RL classes in Java using the RL class reflection.

> **See Also:** For documentation see the Javadoc for the RLClass, RLObject, RLProperty and RLArray classes in the `oracle.rules.rl` package. Thus, RL objects, or instances of RL classes, can be used to hold rule engine results as well as Java objects.

4.7.2 Using External Resources to Obtain Results

This approach is similar to asserting a container for results, except that instead of a container, the object is a means to affecting resources external to the rules engine. For example, this could involve queuing up or scheduling work to be done, updating a database, sending a message. Any Java method accessible in the action may be invoked to effect the results. As with the container use case, the objects used in this example to access the external resources are not re-asserted since their content is not being reasoned on.

Example 4–1 shows the `IncidentDispatcher` object that is asserted and then used to dispatch the notification.

Example 4–1 Obtaining Results Using External Resources

```
rule incidentAlert
{
    if (fact TrafficIncident ti &&
        fact IncidentSubscription s &&
             s.highway == ti.highway &&
             s.direction == ti.direction &&
        fact IncidentDispatcher dispatcher)
    {
        dispatcher.dispatch(s.subscriber, ti);
    }
}
```

Example 4–2 shows Java code that asserts an `IncidentDispatcher` and a `TrafficIncident`, and then invokes the rule engine. This could also be accomplished using an object that is being reasoned on, but this would require a test in the rule condition to avoid an infinite loop of rule firing.

Example 4–2 Sample Showing Results with External Resources

```
sess.callFunctionWithArgument("assert", new IncidentDispatcher());

// An accident has happened
TrafficIncident ti = new TrafficIncident();
ti.setHighway("I5");
ti.setDirection("south");
ti.setIncident("accident");
ti.setWhen(new GregorianCalendar(2005, 1, 25, 5, 4));
ti.setDelay(45);

sess.callFunctionWithArgument("assert", ti);
sess.callFunction("run");
```

4.8 Debugging an RL Stacktrace

The runtime provides detailed debugging information in an RL stacktrace. When possible, if there is an error, the runtime provides extra context that helps identify the location of a problem. This extra context is useful when working with Rules SDK and Rules Designer.

The stacktrace includes the extra context showing the information for rule conditions, rule actions, functions, variables, and RL class definitions. The XPath style format consists of an RL construct and, if named, followed by the name enclosed in parentheses. If a number, `n`, appears in brackets after a construct it indicates the `n`th item following the previous construct. In combination with Rules SDK, RL generation should significantly assist in identifying a location for an error in Rules Designer.

For example, consider the ruleset shown in Example 4–3. When this ruleset executes, it gives the following report:

```
RLNullPointerException: object cannot be null
    at line 12 column 13 in stackTraceContext /Rule(porsche)/Pattern(car)/Test[1]
    at line 17 column 5 in stackTraceContext
```

Example 4–3 Test Ruleset

```
ruleset stackTraceContext
{
    class Car
    {
        String make;
        String model;
    }

    rule porsche
    {
        if (fact Car car &&
            car.make.startsWith("Porsche"))
        {
            println(car.make + " " + car.model);
        }
    }

    assert(new Car());
}
```

ruleset stackTraceContext

{

class Car

{

String make;

String model;

}

rule porsche

{

if (fact Car car &&

```
      car.make.startsWith("Porsche"))
    {
      println(car.make + " " + car.model);
    }
}
assert(new Car());
```

4.9 Using RuleSession Pooling

A typical application that uses rules evaluates the same rules multiple times, with different facts corresponding to separate requests. Initializing a RuleSession typically takes a few seconds depending on the number of rules involved. In contrast, the time to execute the rules is typically much less. Therefore, better performance can be achieved by initializing a RuleSession one time and reusing it for each new request. Using RuleSession pooling, you can create a pool of RuleSession instances that supports improved performance and scalability of applications that use rules.

4.9.1 How to Create a RuleSession Pool

In order for performance to scale up with increasing load, more than one RuleSession is required. A pool of RuleSession instances supports improved performance and scalability of applications that use rules. A pool is instantiated with a list of the RL code that is used to initialize each RuleSession created by the pool. The RL code is executed in the order in which it appears in the list. The number of RuleSession instances to create initially may be specified. In general, this should be a small value and usually the default should be sufficient.

Typically, the RL code is generated from a RuleDictionary created with the Rules SDK. Example 4–4 demonstrates creating and using a RuleSessionPool with RL code from a RuleDictionary.

Example 4–4 Creating a RuleSession Pool

```
RuleDictionary rd;
// Code to load rule dictionary not shown
List rlList = new ArrayList();
rlList.add(rd.dataModelRL());
List rulesetAliases = rd.getRuleSetAliases(true);
for (String alias : rulesetAliases)
{
    rlList.add(rd.ruleSetRL(alias));
}

RuleSessionPool pool = new RuleSessionPool(rlList);
```

If the rules in use by an application are updated, the application may need to load the new rules so that subsequent rule executions use the new rules. This is supported by the pool by invoking the refreshPool method passing it a list of the new RL. After the pool has been refreshed, RuleSessions returned by getPoolableRuleSession will have been initialized with the new RL code. When RuleSessions that were obtained before the refresh are returned using returnPoolableRuleSession, they are not placed back in the pool. The refreshed pool will only contain RuleSessions initialized with the new RL code.

4.9.2 How to Use a RuleSession Pool

To execute rules using a RuleSession, you obtain a RuleSession from the pool and then return it after execution is complete. A poolable RuleSession is acquired by invoking the getPoolableRuleSession method. The pool creates new RuleSessions as required. An invocation of getPoolableRuleSession will not block waiting for a free RuleSession.

When rule execution has been completed, the poolable RuleSession is returned to the pool by invoking the returnPoolableRuleSession method. When a RuleSession is returned to the pool it is reset by the pool by invoking the built-in RL function, reset(). This removes all facts from working memory to prepare the RuleSession for the next execution. Every RuleSession that is retrieved from the pool should be returned to the pool. If an error has occurred during rule execution that results in the RuleSession being unfit for further use, the pool detects this and discards it.

Besides clearing working memory, the reset() function re-executes the initializers of all non-final global variables. The initializer of a non-final global variable can be used to perform other initialization at reset if this is required.

Example 4–5 demonstrates using a RuleSession from the pool.

Example 4–5 Using a Rule Session Pool

```
PoolableObject po = pool.getPoolableRuleSession();
RuleSession engine = po.getPooledObject();
// use the RuleSession to execute rules as required here
pool.returnPoolableRuleSession(po);
```

A soft upper bound on the size of the pool can be specified. This allows the pool to respond to temporary increases in demand by growing the pool while allowing the pool to shrink down to this soft upper bound when demand subsides.

Using the RuleSession pooling implementation, you create RuleSession instances when the getPoolableRuleSession method is invoked and the pool is empty. If the load is heavy enough, this will result in an instance count that is greater than the soft limit.

As the load subsides, the number of RuleSesion instances in the pool will automatically be decreased to the soft limit.

4.10 Using RuleSession Options

The RL runtime with a RuleSession supports the following options:

- `RuleSession.CFG_LOGGING`.
- RuleSession.CFG_DECISION_TRACE_LEVEL
- `RuleSession.CFG_DECISION_TRACE_LIMIT`

4.10.1 Using the CFG_LOGGING System Property

RL Language runtime looks for `CFG_LOGGING` as a system property as well as a Boolean in the config Map passed to the RuleSession constructor. A value in the Map overrides the system property value.

4.10.2 Using the CFG_DECISION_TRACE_LEVEL Option

You can configure the trace level in a RuleSession or in a RuleSessionPool by including the `RuleSession.CFG_DECISION_TRACE_LEVEL` initialization parameter and specifying a level in the configuration Map passed to the RuleSession or RuleSessionPool constructor. This sets the decision trace level at the time a RuleSession is created; invoking `reset()` guarantees that the level after the `reset()` is returned to the configured value, in case it had been changed during rule execution. For more information, see Section 1.7.2, "Using Rule Engine Level Decision Tracing".

4.10.3 Using the CFG_DECISION_TRACE_LIMIT Option

The size of a trace is limited by limiting the number of entries in a decision trace. This necessary to avoid infinite rule fire loops, due to a possible bug in the rules, from creating a trace that consumes all available heap in the JVM. Set the trace limit with the setDecisionTraceLimit function. The limit may also be configured in a RuleSession (or RuleSessionPool) by including the `RuleSession.CFG_DECISION_TRACE_LIMIT` initialization parameter with the desired limit in the configuration Map passed to the RuleSession or RuleSessionPool constructor. For more information, see Section 1.7.2, "Using Rule Engine Level Decision Tracing".

A

Summary of Java and RL Differences

This appendix describes the differences between the RL Language and Java languages.

A.1 RL Differences from Java

- RL does not include interfaces or methods.
- RL global variables are similar to Java static class variables, but there is one instance for each rule session.
- RL does not have a `static` keyword.
- RL has rulesets instead of packages. Rulesets group definitions and actions.
- Instances of RL and Java classes can be asserted as facts in working memory.
- RL facts are not garbage collected; they must be explicitly retracted.
- RL is interpreted. There is no compilation or class loading. Classes and functions must be defined before they are used.
- RL classes may not contain constructors or methods, only data fields. The data fields behave like Java bean properties.
- Java bean properties can be accessed as fields in RL.
- The `new` operator can explicitly assign values to named properties, regardless of whether a constructor is defined. The `fact` operator can match values to named properties and retrieve, using the `var` keyword, values from named properties. A property is either a Java bean property, for Java objects, or a field, for RL objects.
- RL arrays are limited to one dimension.
- The `if` and `while` actions must be in a block, enclosed in curly braces (`{}`).
- RL does not include a `switch` action, `continue` statement, `break` statement, or labeled statements for breaking out of nested loops.
- An RL for loop cannot contain multiple comma separated initialization or update expressions.
- RL does not support bitwise `&` and | operators.
- RL supports function overloading and Java method overloading using best fit.
- RL variables must be initialized when they are defined.

- For RL and Java objects, == always invokes the object `equals` method. RL does not allow testing for object reference equality. For objects, `!=` does not test for inequality of object references, but rather is the negation of the equals methods.

  Thus, the statement:

  ```
  if (object1 != object2){}
  ```

  Is equivalent to the statement:

  ```
  if (! (object1.equals(object2)){}
  ```

- Forward references to classes or functions is not allowed.
- In Java the Java Bean introspector will include write only properties. RL does not include such properties as Beans, since they cannot be reasoned on in a rule. Thus, in order for Java fact type bean properties to be properly accessed in RL they must have both a getter and setter. Properties which have a setter but not a getter, that is write-only properties, are not allowed in the RL `new` syntax.

  For example, if a bean `Foo` only has the method `setProp1(int i)`, then you cannot use the following in RL, `Foo f = new Foo(prop1: 0)`

Index

F

G

H

I

J

L

T

U

V

W

X