Red Hat Decision Manager 7.6

Designing a decision service using DRL rules

Red Hat Decision Manager 7.6 Designing a decision service using DRL rules

Red Hat Customer Content Services
brms-docs@redhat.com

Legal Notice

Abstract

This document describes how to design a decision service using DRL rules in Red Hat Decision Manager 7.6.

Table of Contents

PREFACE

As a business rules developer, you can define business rules using the DRL (Drools Rule Language) designer in Business Central. DRL rules are defined directly in free-form **.drl** text files instead of in a guided or tabular format like other types of rule assets in Business Central. These DRL files form the core of the decision service for your project.

> **NOTE**
>
> You can also design your decision service using Decision Model and Notation (DMN) models instead of rule-based or table-based assets. For information about DMN support in Red Hat Decision Manager 7.6, see the following resources:
>
> - *Getting started with decision services* (step-by-step tutorial with a DMN decision service example)
>
> - *Designing a decision service using DMN models* (overview of DMN support and capabilities in Red Hat Decision Manager)

Prerequisites

- The space and project for the DRL rules have been created in Business Central. Each asset is associated with a project assigned to a space. For details, see *Getting started with decision services*.

CHAPTER 1. DECISION-AUTHORING ASSETS IN RED HAT DECISION MANAGER

Red Hat Decision Manager supports several assets that you can use to define business decisions for your decision service. Each decision-authoring asset has different advantages, and you might prefer to use one or a combination of multiple assets depending on your goals and needs.

The following table highlights the main decision-authoring assets supported in Red Hat Decision Manager projects to help you decide or confirm the best method for defining decisions in your decision service.

Table 1.1. Decision-authoring assets supported in Red Hat Decision Manager

Asset	Highlights	Authoring tools	Documentation
Decision Model and Notation (DMN) models	- Are decision models based on a notation standard defined by the Object Management Group (OMG) - Use graphical decision requirements diagrams (DRDs) with one or more decision requirements graphs (DRGs) to trace business decision flows - Use an XML schema that allows the DMN models to be shared between DMN-compliant platforms - Support Friendly Enough Expression Language (FEEL) to define decision logic in DMN decision tables and other DMN boxed expressions - Are optimal for creating comprehensive, illustrative, and stable decision flows	Business Central or other DMN-compliant editor	*Designing a decision service using DMN models*

Asset	Highlights	Authoring tools	Documentation
Guided decision tables	- Are tables of rules that you create in a UI-based table designer in Business Central - Are a wizard-led alternative to spreadsheet decision tables - Provide fields and options for acceptable input - Support template keys and values for creating rule templates - Support hit policies, real-time validation, and other additional features not supported in other assets - Are optimal for creating rules in a controlled tabular format to minimize compilation errors	Business Central	*Designing a decision service using guided decision tables*
Spreadsheet decision tables	- Are XLS or XLSX spreadsheet decision tables that you can upload into Business Central - Support template keys and values for creating rule templates - Are optimal for creating rules in decision tables already managed outside of Business Central - Have strict syntax requirements for rules to be compiled properly when uploaded	Spreadsheet editor	*Designing a decision service using spreadsheet decision tables*
Guided rules	- Are individual rules that you create in a UI-based rule designer in Business Central - Provide fields and options for acceptable input - Are optimal for creating single rules in a controlled format to minimize compilation errors	Business Central	*Designing a decision service using guided rules*

Asset	Highlights	Authoring tools	Documentation
Guided rule templates	- Are reusable rule structures that you create in a UI-based template designer in Business Central - Provide fields and options for acceptable input - Support template keys and values for creating rule templates (fundamental to the purpose of this asset) - Are optimal for creating many rules with the same rule structure but with different defined field values	Business Central	*Designing a decision service using guided rule templates*
DRL rules	- Are individual rules that you define directly in **.drl** text files - Provide the most flexibility for defining rules and other technicalities of rule behavior - Can be created in certain standalone environments and integrated with Red Hat Decision Manager - Are optimal for creating rules that require advanced DRL options - Have strict syntax requirements for rules to be compiled properly	Business Central or integrated development environment (IDE)	*Designing a decision service using DRL rules*

Asset	Highlights	Authoring tools	Documentation
Predictive Model Markup Language (PMML) models	- Are predictive data-analytic models based on a notation standard defined by the Data Mining Group (DMG) - Use an XML schema that allows the PMML models to be shared between PMML-compliant platforms - Support Regression, Scorecard, Tree, Mining, and other model types - Can be included with a standalone Red Hat Decision Manager project or imported into a project in Business Central - Are optimal for incorporating predictive data into decision services in Red Hat Decision Manager	PMML or XML editor	*Designing a decision service using PMML models*

CHAPTER 2. DRL (DROOLS RULE LANGUAGE) RULES

DRL (Drools Rule Language) rules are business rules that you define directly in **.drl** text files. These DRL files are the source in which all other rule assets in Business Central are ultimately rendered. You can create and manage DRL files within the Business Central interface, or create them externally as part of a Maven or Java project using Red Hat CodeReady Studio or another integrated development environment (IDE). A DRL file can contain one or more rules that define at a minimum the rule conditions (**when**) and actions (**then**). The DRL designer in Business Central provides syntax highlighting for Java, DRL, and XML.

DRL files consist of the following components:

Components in a DRL file

```
package

import

function  // Optional

query  // Optional

declare   // Optional

global   // Optional

rule "rule name"
   // Attributes
   when
      // Conditions
   then
      // Actions
end

rule "rule2 name"

...
```

The following example DRL rule determines the age limit in a loan application decision service:

Example rule for loan application age limit

```
rule "Underage"
  salience 15
  agenda-group "applicationGroup"
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```

A DRL file can contain single or multiple rules, queries, and functions, and can define resource declarations such as imports, globals, and attributes that are assigned and used by your rules and

queries. The DRL package must be listed at the top of a DRL file and the rules are typically listed last. All other DRL components can follow any order.

Each rule must have a unique name within the rule package. If you use the same rule name more than once in any DRL file in the package, the rules fail to compile. Always enclose rule names with double quotation marks (**rule "rule name"**) to prevent possible compilation errors, especially if you use spaces in rule names.

All data objects related to a DRL rule must be in the same project package as the DRL file in Business Central. Assets in the same package are imported by default. Existing assets in other packages can be imported with the DRL rule.

2.1. PACKAGES IN DRL

A package is a folder of related assets in Red Hat Decision Manager, such as data objects, DRL files, decision tables, and other asset types. A package also serves as a unique namespace for each group of rules. A single rule base can contain multiple packages. You typically store all the rules for a package in the same file as the package declaration so that the package is self-contained. However, you can import objects from other packages that you want to use in the rules.

The following example is a package name and namespace for a DRL file in a mortgage application decision service:

Example package definition in a DRL file

```
package org.mortgages;
```

2.2. IMPORT STATEMENTS IN DRL

Similar to import statements in Java, imports in DRL files identify the fully qualified paths and type names for any objects that you want to use in the rules. You specify the package and data object in the format **packageName.objectName**, with multiple imports on separate lines. The decision engine automatically imports classes from the Java package with the same name as the DRL package and from the package **java.lang**.

The following example is an import statement for a loan application object in a mortgage application decision service:

Example import statement in a DRL file

```
import org.mortgages.LoanApplication;
```

2.3. FUNCTIONS IN DRL

Functions in DRL files put semantic code in your rule source file instead of in Java classes. Functions are especially useful if an action (**then**) part of a rule is used repeatedly and only the parameters differ for each rule. Above the rules in the DRL file, you can declare the function or import a static method from a helper class as a function, and then use the function by name in an action (**then**) part of the rule.

The following examples illustrate a function that is either declared or imported in a DRL file:

Example function declaration with a rule (option 1)

```
function String hello(String applicantName) {
    return "Hello " + applicantName + "!";
}

rule "Using a function"
  when
    // Empty
  then
    System.out.println( hello( "James" ) );
end
```

Example function import with a rule (option 2)

```
import function my.package.applicant.hello;

rule "Using a function"
  when
    // Empty
  then
    System.out.println( hello( "James" ) );
end
```

2.4. QUERIES IN DRL

Queries in DRL files search the working memory of the decision engine for facts related to the rules in the DRL file. You add the query definitions in DRL files and then obtain the matching results in your application code. Queries search for a set of defined conditions and do not require **when** or **then** specifications. Query names are global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, you construct a **QueryResults** definition using **ksession.getQueryResults("name")**, where **"name"** is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. You define the query and query results parameters above the rules in the DRL file.

The following example is a query definition in a DRL file for underage applicants in a mortgage application decision service, with the accompanying application code:

Example query definition in a DRL file

```
query "people under the age of 21"
    $person : Person( age < 21 )
end
```

Example application code to obtain query results

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age  of 21" );
```

You can also iterate over the returned **QueryResults** using a standard **for** loop. Each element is a **QueryResultsRow** that you can use to access each of the columns in the tuple.

Example application code to obtain and iterate over query results

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
```

```
   System.out.println( "we have " + results.size() + " people under the age of 21" );

   System.out.println( "These people are under the age of 21:" );

   for ( QueryResultsRow row : results ) {
      Person person = ( Person ) row.get( "person" );
      System.out.println( person.getName() + "\n" );
   }
```

2.5. TYPE DECLARATIONS AND METADATA IN DRL

Declarations in DRL files define new fact types or metadata for fact types to be used by rules in the DRL file:

- **New fact types:** The default fact type in the **java.lang** package of Red Hat Decision Manager is **Object**, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the decision engine, without creating models in a lower-level language like Java. You can also declare a new type when a domain model is already built and you want to complement this model with additional entities that are used mainly during the reasoning process.

- **Metadata for fact types:** You can associate metadata in the format **@key(value)** with new or existing facts. Metadata can be any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. The metadata can be queried at run time by the decision engine and used in the reasoning process.

2.5.1. Type declarations without metadata in DRL

A declaration of a new fact does not require any metadata, but must include a list of attributes or fields. If a type declaration does not include identifying attributes, the decision engine searches for an existing fact class in the classpath and raises an error if the class is missing.

The following example is a declaration of a new fact type **Person** with no metadata in a DRL file:

Example declaration of a new fact type with a rule

```
declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end

rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then   // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName( "Mark" );
    insert( mark );
end
```

In this example, the new fact type **Person** has the three attributes **name**, **dateOfBirth**, and **address**. Each attribute has a type that can be any valid Java type, including another class that you create or a fact type that you previously declared. The **dateOfBirth** attribute has the type **java.util.Date**, from the Java API, and the **address** attribute has the previously defined fact type **Address**.

To avoid writing the fully qualified name of a class every time you declare it, you can define the full class name as part of the **import** clause:

Example type declaration with the fully qualified class name in the import

```
import java.util.Date

declare Person
    name : String
    dateOfBirth : Date
    address : Address
end
```

When you declare a new fact type, the decision engine generates at compile time a Java class representing the fact type. The generated Java class is a one-to-one JavaBeans mapping of the type definition.

For example, the following Java class is generated from the example **Person** type declaration:

Generated Java class for the Person fact type declaration

```java
public class Person implements Serializable {
    private String name;
    private java.util.Date dateOfBirth;
    private Address address;

    // Empty constructor
    public Person() {...}

    // Constructor with all fields
    public Person( String name, Date dateOfBirth, Address address ) {...}

    // If keys are defined, constructor with keys
    public Person( ...keys... ) {...}

    // Getters and setters
    // `equals` and `hashCode`
    // `toString`
}
```

You can then use the generated class in your rules like any other fact, as illustrated in the previous rule example with the **Person** type declaration:

Example rule that uses the declared Person fact type

```
rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then   // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName( "Mark" );
    insert( mark );
end
```

2.5.2. Enumerative type declarations in DRL

DRL supports the declaration of enumerative types in the format **declare enum <factType>**, followed by a comma-separated list of values ending with a semicolon. You can then use the enumerative list in the rules in the DRL file.

For example, the following enumerative type declaration defines days of the week for an employee scheduling rule:

Example enumerative type declaration with a scheduling rule

```
declare enum DaysOfWeek

SUN("Sunday"),MON("Monday"),TUE("Tuesday"),WED("Wednesday"),THU("Thursday"),FRI("Friday"),SAT("Saturday");

   fullName : String
end

rule "Using a declared Enum"
when
   $emp : Employee( dayOff == DaysOfWeek.MONDAY )
then
   ...
end
```

2.5.3. Extended type declarations in DRL

DRL supports type declaration inheritance in the format **declare <factType1> extends <factType2>**. To extend a type declared in Java by a subtype declared in DRL, you repeat the parent type in a declaration statement without any fields.

For example, the following type declarations extend a **Student** type from a top-level **Person** type, and a **LongTermStudent** type from the **Student** subtype:

Example extended type declarations

```
import org.people.Person

declare Person end

declare Student extends Person
   school : String
end

declare LongTermStudent extends Student
   years : int
   course : String
end
```

2.5.4. Type declarations with metadata in DRL

You can associate metadata in the format **@key(value)** (the value is optional) with fact types or fact attributes. Metadata can be any kind of data that is not represented by the fact attributes and is

consistent among all instances of that fact type. The metadata can be queried at run time by the decision engine and used in the reasoning process. Any metadata that you declare before the attributes of a fact type are assigned to the fact type, while metadata that you declare after an attribute are assigned to that particular attribute.

In the following example, the two metadata attributes **@author** and **@dateOfCreation** are declared for the **Person** fact type, and the two metadata items **@key** and **@maxLength** are declared for the **name** attribute. The **@key** metadata attribute has no required value, so the parentheses and the value are omitted.

Example metadata declaration for fact types and attributes

```
import java.util.Date

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )

    name : String @key @maxLength( 30 )
    dateOfBirth : Date
    address : Address
end
```

For declarations of metadata attributes for existing types, you can identify the fully qualified class name as part of the **import** clause for all declarations or as part of the individual **declare** clause:

Example metadata declaration for an imported type

```
import org.drools.examples.Person

declare Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

Example metadata declaration for a declared type

```
declare org.drools.examples.Person
    @author( Bob )
    @dateOfCreation( 01-Feb-2009 )
end
```

2.5.5. Metadata tags for fact type and attribute declarations in DRL

Although you can define custom metadata attributes in DRL declarations, the decision engine also supports the following predefined metadata tags for declarations of fact types or fact type attributes.

> **NOTE**
>
> The examples in this section that refer to the **VoiceCall** class assume that the sample application domain model includes the following class details:
>
> **VoiceCall fact class in an example Telecom domain model**
>
> ```
> public class VoiceCall {
>   private String  originNumber;
>   private String  destinationNumber;
>   private Date    callDateTime;
>   private long    callDuration;  // in milliseconds
>
>   // Constructors, getters, and setters
> }
> ```

@role

This tag determines whether a given fact type is handled as a regular fact or an event in the decision engine during complex event processing.
Default parameter: **fact**

Supported parameters: **fact**, **event**

```
@role( fact | event )
```

Example: Declare VoiceCall as event type

```
declare VoiceCall
  @role( event )
end
```

@timestamp

This tag is automatically assigned to every event in the decision engine. By default, the time is provided by the session clock and assigned to the event when it is inserted into the working memory of the decision engine. You can specify a custom time stamp attribute instead of the default time stamp added by the session clock.
Default parameter: The time added by the decision engine session clock

Supported parameters: Session clock time or custom time stamp attribute

```
@timestamp( <attributeName> )
```

Example: Declare VoiceCall timestamp attribute

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
end
```

@duration

This tag determines the duration time for events in the decision engine. Events can be interval-

based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the decision engine until their duration time has lapsed. Point-in-time events have no duration and are essentially interval-based events with a duration of zero. By default, every event in the decision engine has a duration of zero. You can specify a custom duration attribute instead of the default.

Default parameter: Null (zero)

Supported parameters: Custom duration attribute

```
@duration( <attributeName> )
```

Example: Declare VoiceCall duration attribute

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
end
```

@**expires**

This tag determines the time duration before an event expires in the working memory of the decision engine. By default, an event expires when the event can no longer match and activate any of the current rules. You can define an amount of time after which an event should expire. This tag definition also overrides the implicit expiration offset calculated from temporal constraints and sliding windows in the KIE base. This tag is available only when the decision engine is running in stream mode.

Default parameter: Null (event expires after event can no longer match and activate rules)

Supported parameters: Custom **timeOffset** attribute in the format **[#d][#h][#m][#s][[ms]]**

```
@expires( <timeOffset> )
```

Example: Declare expiration offset for VoiceCall events

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
  @expires( 1h35m )
end
```

@**typesafe**

This tab determines whether a given fact type is compiled with or without type safety. By default, all type declarations are compiled with type safety enabled. You can override this behavior to type-unsafe evaluation, where all constraints are generated as MVEL constraints and executed dynamically. This is useful when dealing with collections that do not have any generics or mixed type collections.

Default parameter: **true**

Supported parameters: **true**, **false**

```
@typesafe( <boolean> )
```

Example: Declare VoiceCall for type-unsafe evaluation

```
declare VoiceCall
  @role( fact )
  @typesafe( false )
end
```

@serialVersionUID

This tag defines an identifying **serialVersionUID** value for a serializable class in a fact declaration. If a serializable class does not explicitly declare a **serialVersionUID**, the serialization run time calculates a default **serialVersionUID** value for that class based on various aspects of the class, as described in the Java Object Serialization Specification . However, for optimal deserialization results and for greater compatibility with serialized KIE sessions, set the **serialVersionUID** as needed in the relevant class or in your DRL declarations.
Default parameter: Null

Supported parameters: Custom **serialVersionUID** integer

```
@serialVersionUID( <integer> )
```

Example: Declare serialVersionUID for a VoiceCall class

```
declare VoiceCall
  @serialVersionUID( 42 )
end
```

@key

This tag enables a fact type attribute to be used as a key identifier for the fact type. The generated class can then implement the **equals()** and **hashCode()** methods to determine if two instances of the type are equal to each other. The decision engine can also generate a constructor using all the key attributes as parameters.
Default parameter: None

Supported parameters: None

```
<attributeDefinition> @key
```

Example: Declare Person type attributes as keys

```
declare Person
    firstName : String @key
    lastName : String @key
    age : int
end
```

For this example, the decision engine checks the **firstName** and **lastName** attributes to determine if two instances of **Person** are equal to each other, but it does not check the   **age** attribute. The decision engine also implicitly generates three constructors: one without parameters, one with the **@key** fields, and one with all fields:

Example constructors from the key declarations

```
Person() // Empty constructor

Person( String firstName, String lastName )

Person( String firstName, String lastName, int age )
```

You can then create instances of the type based on the key constructors, as shown in the following example:

Example instance using the key constructor

```
Person person = new Person( "John", "Doe" );
```

@position

This tag determines the position of a declared fact type attribute or field in a positional argument, overriding the default declared order of attributes. You can use this tag to modify positional constraints in patterns while maintaining a consistent format in your type declarations and positional arguments. You can use this tag only for fields in classes on the classpath. If some fields in a single class use this tag and some do not, the attributes without this tag are positioned last, in the declared order. Inheritance of classes is supported, but not interfaces of methods.
Default parameter: None

Supported parameters: Any integer

```
<attributeDefinition> @position ( <integer> )
```

Example: Declare a fact type and override declared order

```
declare Person
    firstName : String @position( 1 )
    lastName : String @position( 0 )
    age : int @position( 2 )
    occupation: String
end
```

In this example, the attributes are prioritized in positional arguments in the following order:

1. **lastName**

2. **firstName**

3. **age**

4. **occupation**

In positional arguments, you do not need to specify the field name because the position maps to a known named field. For example, the argument **Person(lastName == "Doe")** is the same as **Person("Doe";)**, where the **lastName** field has the highest position annotation in the DRL declaration. The semicolon **;** indicates that everything before it is a positional argument. You can mix positional and named arguments on a pattern by using the semicolon to separate them. Any variables in a positional argument that have not yet been bound are bound to the field that maps to that position.

The following example patterns illustrate different ways of constructing positional and named

arguments. The patterns have two constraints and a binding, and the semicolon differentiates the positional section from the named argument section. Variables and literals and expressions using only literals are supported in positional arguments, but not variables alone.

Example patterns with positional and named arguments

```
Person( "Doe", "John", $a; )

Person( "Doe", "John"; $a : age )

Person( "Doe"; firstName == "John", $a : age )

Person( lastName == "Doe"; firstName == "John", $a : age )
```

Positional arguments can be classified as *input arguments* or *output arguments*. Input arguments contain a previously declared binding and constrain against that binding using unification. Output arguments generate the declaration and bind it to the field represented by the positional argument when the binding does not yet exist.

In extended type declarations, use caution when defining **@position** annotations because the attribute positions are inherited in subtypes. This inheritance can result in a mixed attribute order that can be confusing in some cases. Two fields can have the same **@position** value and consecutive values do not need to be declared. If a position is repeated, the conflict is solved using inheritance, where position values in the parent type have precedence, and then using the declaration order from the first to last declaration.

For example, the following extended type declarations result in mixed positional priorities:

Example extended fact type with mixed position annotations

```
declare Person
   firstName : String @position( 1 )
   lastName : String @position( 0 )
   age : int @position( 2 )
   occupation: String
end

declare Student extends Person
   degree : String @position( 1 )
   school : String @position( 0 )
   graduationDate : Date
end
```

In this example, the attributes are prioritized in positional arguments in the following order:

1. **lastName** (position 0 in the parent type)

2. **school** (position 0 in the subtype)

3. **firstName** (position 1 in the parent type)

4. **degree** (position 1 in the subtype)

5. **age** (position 2 in the parent type)

6. **occupation** (first field with no position annotation)

7. **graduationDate** (second field with no position annotation)

2.5.6. Property-change settings and listeners for fact types

By default, the decision engine does not re-evaluate all fact patterns for fact types each time a rule is triggered, but instead reacts only to modified properties that are constrained or bound inside a given pattern. For example, if a rule calls **modify()** as part of the rule actions but the action does not generate new data in the KIE base, the decision engine does not automatically re-evaluate all fact patterns because no data was modified. This property reactivity behavior prevents unwanted recursions in the KIE base and results in more efficient rule evaluation. This behavior also means that you do not always need to use the **no-loop** rule attribute to avoid infinite recursion.

You can modify or disable this property reactivity behavior with the following **KnowledgeBuilderConfiguration** options, and then use a property-change setting in your Java class or DRL files to fine-tune property reactivity as needed:

- **ALWAYS**: (Default) All types are property reactive, but you can disable property reactivity for a specific type by using the **@classReactive** property-change setting.

- **ALLOWED**: No types are property reactive, but you can enable property reactivity for a specific type by using the **@propertyReactive** property-change setting.

- **DISABLED**: No types are property reactive. All property-change listeners are ignored.

Example property reactivity setting in KnowledgeBuilderConfiguration

```
KnowledgeBuilderConfiguration config =
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();
config.setOption(PropertySpecificOption.ALLOWED);
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
```

Alternatively, you can update the **drools.propertySpecific** system property in the **standalone.xml** file of your Red Hat Decision Manager distribution:

Example property reactivity setting in system properties

```
<system-properties>
  ...
  <property name="drools.propertySpecific" value="ALLOWED"/>
  ...
</system-properties>
```

The decision engine supports the following property-change settings and listeners for fact classes or declared DRL fact types:

@classReactive

If property reactivity is set to **ALWAYS** in the decision engine (all types are property reactive), this tag disables the default property reactivity behavior for a specific Java class or a declared DRL fact type. You can use this tag if you want the decision engine to re-evaluate all fact patterns for the specified fact type each time the rule is triggered, instead of reacting only to modified properties that are constrained or bound inside a given pattern.

Example: Disable default property reactivity in a DRL type declaration

```
declare Person
  @classReactive
    firstName : String
    lastName : String
end
```

Example: Disable default property reactivity in a Java class

```
@classReactive
public static class Person {
    private String firstName;
    private String lastName;
}
```

@propertyReactive

If property reactivity is set to **ALLOWED** in the decision engine (no types are property reactive unless specified), this tag enables property reactivity for a specific Java class or a declared DRL fact type. You can use this tag if you want the decision engine to react only to modified properties that are constrained or bound inside a given pattern for the specified fact type, instead of re-evaluating all fact patterns for the fact each time the rule is triggered.

Example: Enable property reactivity in a DRL type declaration (when reactivity is disabled globally)

```
declare Person
  @propertyReactive
    firstName : String
    lastName : String
end
```

Example: Enable property reactivity in a Java class (when reactivity is disabled globally)

```
@propertyReactive
public static class Person {
    private String firstName;
    private String lastName;
}
```

@watch

This tag enables property reactivity for additional properties that you specify in-line in fact patterns in DRL rules. This tag is supported only if property reactivity is set to **ALWAYS** in the decision engine, or if property reactivity is set to **ALLOWED** and the relevant fact type uses the **@propertyReactive** tag. You can use this tag in DRL rules to add or exclude specific properties in fact property reactivity logic.
Default parameter: None

Supported parameters: Property name, ***** (all), **!** (not), **!*** (no properties)

```
<factPattern> @watch ( <property> )
```

Example: Enable or disable property reactivity in fact patterns

```
// Listens for changes in both `firstName` (inferred) and `lastName`:
Person(firstName == $expectedFirstName) @watch( lastName )

// Listens for changes in all properties of the `Person` fact:
Person(firstName == $expectedFirstName) @watch( * )

// Listens for changes in `lastName` and explicitly excludes changes in `firstName`:
Person(firstName == $expectedFirstName) @watch( lastName, !firstName )

// Listens for changes in all properties of the `Person` fact except `age`:
Person(firstName == $expectedFirstName) @watch( *, !age )

// Excludes changes in all properties of the `Person` fact (equivalent to using `@classReactivity`
tag):
Person(firstName == $expectedFirstName) @watch( !* )
```

The decision engine generates a compilation error if you use the **@watch** tag for properties in a fact type that uses the **@classReactive** tag (disables property reactivity) or when property reactivity is set to **ALLOWED** in the decision engine and the relevant fact type does not use the **@propertyReactive** tag. Compilation errors also arise if you duplicate properties in listener annotations, such as **@watch(firstName, ! firstName)**.

@propertyChangeSupport

For facts that implement support for property changes as defined in the JavaBeans Specification, this tag enables the decision engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
    @propertyChangeSupport
end
```

2.5.7. Access to DRL declared types in application code

Declared types in DRL are typically used within the DRL files while Java models are typically used when the model is shared between rules and applications. Because declared types are generated at KIE base compile time, an application cannot access them until application run time. In some cases, an application needs to access and handle facts directly from the declared types, especially when the application wraps the decision engine and provides higher-level, domain-specific user interfaces for rules management.

To handle declared types directly from the application code, you can use the **org.drools.definition.type.FactType** API in Red Hat Decision Manager. Through this API, you can instantiate, read, and write fields in the declared fact types.

The following example code modifies a **Person** fact type directly from an application:

Example application code to handle a declared fact type through the FactType API

```
import java.util.Date;

import org.kie.api.definition.type.FactType;
import org.kie.api.KieBase;
import org.kie.api.runtime.KieSession;
```

```
...

// Get a reference to a KIE base with the declared type:
KieBase kbase = ...

// Get the declared fact type:
FactType personType = kbase.getFactType("org.drools.examples", "Person");

// Create instances:
Object bob = personType.newInstance();

// Set attribute values:
personType.set(bob, "name", "Bob" );
personType.set(bob, "dateOfBirth", new Date());
personType.set(bob, "address", new Address("King's Road","London","404"));

// Insert the fact into a KIE session:
KieSession ksession = ...
ksession.insert(bob);
ksession.fireAllRules();

// Read attributes:
String name = (String) personType.get(bob, "name");
Date date = (Date) personType.get(bob, "dateOfBirth");
```

The API also includes other helpful methods, such as setting all the attributes at once, reading values from a **Map** collection, or reading all attributes at once into a **Map** collection.

Although the API behavior is similar to Java reflection, the API does not use reflection and relies on more performant accessors that are implemented with generated bytecode.

2.6. GLOBAL VARIABLES IN DRL

Global variables in DRL files typically provide data or services for the rules, such as application services used in rule consequences, and return data from rules, such as logs or values added in rule consequences. You set the global value in the working memory of the decision engine through a KIE session configuration or REST operation, declare the global variable above the rules in the DRL file, and then use it in an action (**then**) part of the rule. For multiple global variables, use separate lines in the DRL file.

The following example illustrates a global variable list configuration for the decision engine and the corresponding global variable definition in the DRL file:

Example global list configuration for the decision engine

```
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

Example global variable definition with a rule

```
global java.util.List myGlobalList;

rule "Using a global"
```

```
when
   // Empty
then
   myGlobalList.add( "My global list" );
end
```

> **WARNING**
>
> Do not use global variables to establish conditions in rules unless a global variable has a constant immutable value. Global variables are not inserted into the working memory of the decision engine, so the decision engine cannot track value changes of variables.
>
> Do not use global variables to share data between rules. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory of the decision engine.

A use case for a global variable might be an instance of an email service. In your integration code that is calling the decision engine, you obtain your **emailService** object and then set it in the working memory of the decision engine. In the DRL file, you declare that you have a global of type **emailService** and give it the name **"email"**, and then in your rule consequences, you can use actions such as **email.sendSMS(number, message)**.

If you declare global variables with the same identifier in multiple packages, then you must set all the packages with the same type so that they all reference the same global value.

2.7. RULE ATTRIBUTES IN DRL

Rule attributes are additional specifications that you can add to business rules to modify rule behavior. In DRL files, you typically define rule attributes above the rule conditions and actions, with multiple attributes on separate lines, in the following format:

```
rule "rule_name"
   // Attribute
   // Attribute
   when
       // Conditions
   then
       // Actions
end
```

The following table lists the names and supported values of the attributes that you can assign to rules:

Table 2.1. Rule attributes

Attribute	Value

Attribute	Value
salience	An integer defining the priority of the rule. Rules with a higher salience value are given higher priority when ordered in the activation queue. Example: **salience 10**
enabled	A Boolean value. When the option is selected, the rule is enabled. When the option is not selected, the rule is disabled. Example: **enabled true**
date-effective	A string containing a date and time definition. The rule can be activated only if the current date and time is after a **date-effective** attribute. Example: **date-effective "4-Sep-2018"**
date-expires	A string containing a date and time definition. The rule cannot be activated if the current date and time is after the **date-expires** attribute. Example: **date-expires "4-Oct-2018"**
no-loop	A Boolean value. When the option is selected, the rule cannot be reactivated (looped) if a consequence of the rule re-triggers a previously met condition. When the condition is not selected, the rule can be looped in these circumstances. Example: **no-loop true**
agenda-group	A string identifying an agenda group to which you want to assign the rule. Agenda groups allow you to partition the agenda to provide more execution control over groups of rules. Only rules in an agenda group that has acquired a focus are able to be activated. Example: **agenda-group "GroupName"**
activation-group	A string identifying an activation (or XOR) group to which you want to assign the rule. In activation groups, only one rule can be activated. The first rule to fire will cancel all pending activations of all rules in the activation group. Example: **activation-group "GroupName"**
duration	A long integer value defining the duration of time in milliseconds after which the rule can be activated, if the rule conditions are still met. Example: **duration 10000**
timer	A string identifying either **int** (interval) or **cron** timer definitions for scheduling the rule. Example: **timer (cron:* 0/15 * * * ?)** (every 15 minutes)

Attribute	Value
calendar	A Quartz calendar definition for scheduling the rule. Example: **calendars "* * 0-7,18-23 ? * *"** (exclude non-business hours)
auto-focus	A Boolean value, applicable only to rules within agenda groups. When the option is selected, the next time the rule is activated, a focus is automatically given to the agenda group to which the rule is assigned. Example: **auto-focus true**
lock-on-active	A Boolean value, applicable only to rules within rule flow groups or agenda groups. When the option is selected, the next time the ruleflow group for the rule becomes active or the agenda group for the rule receives a focus, the rule cannot be activated again until the ruleflow group is no longer active or the agenda group loses the focus. This is a stronger version of the **no-loop** attribute, because the activation of a matching rule is discarded regardless of the origin of the update (not only by the rule itself). This attribute is ideal for calculation rules where you have a number of rules that modify a fact and you do not want any rule re-matching and firing again. Example: **lock-on-active true**
ruleflow-group	A string identifying a rule flow group. In rule flow groups, rules can fire only when the group is activated by the associated rule flow. Example: **ruleflow-group "GroupName"**
dialect	A string identifying either **JAVA** or **MVEL** as the language to be used for code expressions in the rule. By default, the rule uses the dialect specified at the package level. Any dialect specified here overrides the package dialect setting for the rule. Example: **dialect "JAVA"** **NOTE** When you use Red Hat Decision Manager without the executable model, the **dialect "JAVA"** rule consequences support only Java 5 syntax. For more information about executable models, see *Packaging and deploying a Red Hat Decision Manager project*.

2.7.1. Timer and calendar rule attributes in DRL

Timers and calendars are DRL rule attributes that enable you to apply scheduling and timing constraints to your DRL rules. These attributes require additional configurations depending on the use case.

The **timer** attribute in DRL rules is a string identifying either **int** (interval) or **cron** timer definitions for scheduling a rule and supports the following formats:

Timer attribute formats

```
timer ( int: <initial delay> <repeat interval> )

timer ( cron: <cron expression> )
```

Example interval timer attributes

```
// Run after a 30-second delay
timer ( int: 30s )

// Run every 5 minutes after a 30-second delay each time
timer ( int: 30s 5m )
```

Example cron timer attribute

```
// Run every 15 minutes
timer ( cron:* 0/15 * * * ? )
```

Interval timers follow the semantics of **java.util.Timer** objects, with an initial delay and an optional repeat interval. Cron timers follow standard Unix cron expressions.

The following example DRL rule uses a cron timer to send an SMS text message every 15 minutes:

Example DRL rule with a cron timer

```
rule "Send SMS message every 15 minutes"
  timer ( cron:* 0/15 * * * ? )
  when
    $a : Alarm( on == true )
  then
    channels[ "sms" ].insert( new Sms( $a.mobileNumber, "The alarm is still on." );
end
```

Generally, a rule that is controlled by a timer becomes active when the rule is triggered and the rule consequence is executed repeatedly, according to the timer settings. The execution stops when the rule condition no longer matches incoming facts. However, the way the decision engine handles rules with timers depends on whether the decision engine is in *active mode* or in *passive mode*.

By default, the decision engine runs in *passive mode* and evaluates rules, according to the defined timer settings, when a user or an application explicitly calls **fireAllRules()**. Conversely, if a user or application calls **fireUntilHalt()**, the decision engine starts in *active mode* and evaluates rules continually until the user or application explicitly calls **halt()**.

When the decision engine is in active mode, rule consequences are executed even after control returns from a call to **fireUntilHalt()** and the decision engine remains *reactive* to any changes made to the working memory. For example, removing a fact that was involved in triggering the timer rule execution causes the repeated execution to terminate, and inserting a fact so that some rule matches causes that rule to be executed. However, the decision engine is not continually *active*, but is active only after a rule is executed. Therefore, the decision engine does not react to asynchronous fact insertions until the next execution of a timer–controlled rule. Disposing a KIE session terminates all timer activity.

When the decision engine is in passive mode, rule consequences of timed rules are evaluated only when **fireAllRules()** is invoked again. However, you can change the default timer-execution behavior in passive mode by configuring the KIE session with a **TimedRuleExecutionOption** option, as shown in the following example:

KIE session configuration to automatically execute timed rules in passive mode

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption( TimedRuleExecutionOption.YES );
KSession ksession = kbase.newKieSession(ksconf, null);
```

You can additionally set a **FILTERED** specification on the **TimedRuleExecutionOption** option that enables you to define a callback to filter those rules, as shown in the following example:

KIE session configuration to filter which timed rules are automatically executed

```
KieSessionConfiguration ksconf = KieServices.Factory.get().newKieSessionConfiguration();
conf.setOption( new TimedRuleExecutionOption.FILTERED(new TimedRuleExecutionFilter() {
    public boolean accept(Rule[] rules) {
        return rules[0].getName().equals("MyRule");
    }
}) );
```

For interval timers, you can also use an expression timer with **expr** instead of **int** to define both the delay and interval as an expression instead of a fixed value.

The following example DRL file declares a fact type with a delay and period that are then used in the subsequent rule with an expression timer:

Example rule with an expression timer

```
declare Bean
  delay   : String = "30s"
  period  : long = 60000
end

rule "Expression timer"
  timer ( expr: $d, $p )
  when
    Bean( $d : delay, $p : period )
  then
    // Actions
end
```

The expressions, such as **$d** and **$p** in this example, can use any variable defined in the pattern-matching part of the rule. The variable can be any **String** value that can be parsed into a time duration or any numeric value that is internally converted in a **long** value for a duration in milliseconds.

Both interval and expression timers can use the following optional parameters:

- **start** and **end**: A **Date** or a **String** representing a **Date** or a **long** value. The value can also be a **Number** that is transformed into a Java **Date** in the format **new Date(((Number) n).longValue())**.

- **repeat-limit**: An integer that defines the maximum number of repetitions allowed by the timer. If both the **end** and the **repeat-limit** parameters are set, the timer stops when the first of the two is reached.

Example timer attribute with optional start, end, and repeat-limit parameters

```
timer (int: 30s 1h; start=3-JAN-2020, end=4-JAN-2020, repeat-limit=50)
```

In this example, the rule is scheduled for every hour, after a delay of 30 seconds each hour, beginning on 3 January 2020 and ending either on 4 January 2020 or when the cycle repeats 50 times.

If the system is paused (for example, the session is serialized and then later deserialized), the rule is scheduled only one time to recover from missing activations regardless of how many activations were missed during the pause, and then the rule is subsequently scheduled again to continue in sync with the timer setting.

The **calendar** attribute in DRL rules is a Quartz calendar definition for scheduling a rule and supports the following format:

Calendar attribute format

```
calendars "<definition or registered name>"
```

Example calendar attributes

```
// Exclude non-business hours
calendars "* * 0-7,18-23 ? * *"

// Weekdays only, as registered in the KIE session
calendars "weekday"
```

You can adapt a Quartz calendar based on the Quartz calendar API and then register the calendar in the KIE session, as shown in the following example:

Adapting a Quartz Calendar

```
Calendar weekDayCal = QuartzHelper.quartzCalendarAdapter(org.quartz.Calendar quartzCal)
```

Registering the calendar in the KIE session

```
ksession.getCalendars().set( "weekday", weekDayCal );
```

You can use calendars with standard rules and with rules that use timers. The calendar attribute can contain one or more comma-separated calendar names written as **String** literals.

The following example rules use both calendars and timers to schedule the rules:

Example rules with calendars and timers

```
rule "Weekdays are high priority"
  calendars "weekday"
  timer ( int:0 1h )
  when
```

```
    Alarm()
  then
    send( "priority high - we have an alarm" );
  end

  rule "Weekends are low priority"
    calendars "weekend"
    timer ( int:0 4h )
    when
      Alarm()
    then
      send( "priority low - we have an alarm" );
  end
```

2.8. RULE CONDITIONS IN DRL (WHEN)

The **when** part of a DRL rule (also known as the *Left Hand Side (LHS)* of the rule) contains the conditions that must be met to execute an action. Conditions consist of a series of stated *patterns* and *constraints*, with optional *bindings* and supported rule condition elements (keywords), based on the available data objects in the package. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition of an **"Underage"** rule would be **Applicant(age < 21)**.

> **NOTE**
>
> DRL uses **when** instead of **if** because **if** is typically part of a procedural execution flow during which a condition is checked at a specific point in time. In contrast, **when** indicates that the condition evaluation is not limited to a specific evaluation sequence or point in time, but instead occurs continually at any time. Whenever the condition is met, the actions are executed.

If the **when** section is empty, then the conditions are considered to be true and the actions in the **then** section are executed the first time a **fireAllRules()** call is made in the decision engine. This is useful if you want to use rules to set up the decision engine state.

The following example rule uses empty conditions to insert a fact every time the rule is executed:

Example rule without conditions

```
rule "Always insert applicant"
  when
    // Empty
  then   // Actions to be executed once
    insert( new Applicant() );
end

// The rule is internally rewritten in the following way:

rule "Always insert applicant"
  when
    eval( true )
  then
    insert( new Applicant() );
end
```

If rule conditions use multiple patterns with no defined keyword conjunctions (such as **and**, **or**, or **not**), the default conjunction is **and**:

Example rule without keyword conjunctions

```
rule "Underage"
  when
    application : LoanApplication()
    Applicant( age < 21 )
  then
    // Actions
end

// The rule is internally rewritten in the following way:

rule "Underage"
  when
    application : LoanApplication()
    and Applicant( age < 21 )
  then
    // Actions
end
```

2.8.1. Patterns and constraints

A *pattern* in a DRL rule condition is the segment to be matched by the decision engine. A pattern can potentially match each fact that is inserted into the working memory of the decision engine. A pattern can also contain *constraints* to further define the facts to be matched.

In the simplest form, with no constraints, a pattern matches a fact of the given type. In the following example, the type is **Person**, so the pattern will match against all **Person** objects in the working memory of the decision engine:

Example pattern for a single fact type

```
Person()
```

The type does not need to be the actual class of some fact object. Patterns can refer to superclasses or even interfaces, potentially matching facts from many different classes. For example, the following pattern matches all objects in the working memory of the decision engine:

Example pattern for all objects

```
Object() // Matches all objects in the working memory
```

The parentheses of a pattern enclose the constraints, such as the following constraint on the person's age:

Example pattern with a constraint

```
Person( age == 50 )
```

A *constraint* is an expression that returns **true** or **false**. Pattern constraints in DRL are essentially Java expressions with some enhancements, such as property access, and some differences, such as **equals()** and **!equals()** semantics for **==** and **!=** (instead of the usual **same** and **not same** semantics).

Any JavaBeans property can be accessed directly from pattern constraints. A bean property is exposed internally using a standard JavaBeans getter that takes no arguments and returns something. For example, the **age** property is written as **age** in DRL instead of the getter **getAge()**:

DRL constraint syntax with JavaBeans properties

```
Person( age == 50 )

// This is the same as the following getter format:

Person( getAge() == 50 )
```

Red Hat Decision Manager uses the standard JDK **Introspector** class to achieve this mapping, so it follows the standard JavaBeans specification. For optimal decision engine performance, use the property access format, such as **age**, instead of using getters explicitly, such as **getAge()**.

> ⚠️ **WARNING**
>
> Do not use property accessors to change the state of the object in a way that might affect the rules because the decision engine caches the results of the match between invocations for higher efficiency.
>
> For example, do not use property accessors in the following ways:
>
> ```
> public int getAge() {
>     age++; // Do not do this.
>     return age;
> }
> ```
>
> ```
> public int getAge() {
>     Date now = DateUtil.now(); // Do not do this.
>     return DateUtil.differenceInYears(now, birthday);
> }
> ```
>
> Instead of following the second example, insert a fact that wraps the current date in the working memory and update that fact between **fireAllRules()** as needed.

However, if the getter of a property cannot be found, the compiler uses the property name as a fallback method name, without arguments:

Fallback method if object is not found

```
Person( age == 50 )

// If `Person.getAge()` does not exist, the compiler uses the following syntax:
```

```
Person( age() == 50 )
```

You can also nest access properties in patterns, as shown in the following example. Nested properties are indexed by the decision engine.

Example pattern with nested property access

```
Person( address.houseNumber == 50 )

// This is the same as the following format:

Person( getAddress().getHouseNumber() == 50 )
```

> **WARNING**
>
> In stateful KIE sessions, use nested accessors carefully because the working memory of the decision engine is not aware of any of the nested values and does not detect when they change. Either consider the nested values immutable while any of their parent references are inserted into the working memory, or, if you want to modify a nested value, mark all of the outer facts as updated. In the previous example, when the **houseNumber** property changes, any **Person** with that **Address** must be marked as updated.

You can use any Java expression that returns a **boolean** value as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access:

Example pattern with a constraint using property access and Java expression

```
Person( age == 50 )
```

You can change the evaluation priority by using parentheses, as in any logical or mathematical expression:

Example evaluation order of constraints

```
Person( age > 100 && ( age % 10 == 0 ) )
```

You can also reuse Java methods in constraints, as shown in the following example:

Example constraints with reused Java methods

```
Person( Math.round( weight / ( height * height ) ) < 25.0 )
```

> **WARNING**
>
> Do not use constraints to change the state of the object in a way that might affect the rules because the decision engine caches the results of the match between invocations for higher efficiency. Any method that is executed on a fact in the rule conditions must be a read-only method. Also, the state of a fact should not change between rule invocations unless those facts are marked as updated in the working memory on every change.
>
> For example, do not use a pattern constraint in the following ways:
>
> ```
> Person( incrementAndGetAge() == 10 ) // Do not do this.
> ```
>
> ```
> Person( System.currentTimeMillis() % 1000 == 0 ) // Do not do this.
> ```

Standard Java operator precedence applies to constraint operators in DRL, and DRL operators follow standard Java semantics except for the **==** and **!=** operators.

The **==** operator uses null-safe **equals()** semantics instead of the usual **same** semantics. For example, the pattern **Person(firstName == "John")** is similar to **java.util.Objects.equals(person.getFirstName(), "John")**, and because **"John"** is not null, the pattern is also similar to **"John".equals(person.getFirstName())**.

The **!=** operator uses null-safe **!equals()** semantics instead of the usual **not same** semantics. For example, the pattern **Person(firstName != "John")** is similar to **!java.util.Objects.equals(person.getFirstName(), "John")**.

If the field and the value of a constraint are of different types, the decision engine uses type coercion to resolve the conflict and reduce compilation errors. For instance, if **"ten"** is provided as a string in a numeric evaluator, a compilation error occurs, whereas **"10"** is coerced to a numeric 10. In coercion, the field type always takes precedence over the value type:

Example constraint with a value that is coerced

```
Person( age == "10" ) // "10" is coerced to 10
```

For groups of constraints, you can use a delimiting comma **,** to use implicit **and** connective semantics:

Example patterns with multiple constraints

```
// Person is at least 50 years old and weighs at least 80 kilograms:
Person( age > 50, weight > 80 )

// Person is at least 50 years old, weighs at least 80 kilograms, and is taller than 2 meters:
Person( age > 50, weight > 80, height > 2 )
```
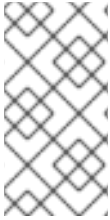
NOTE

Although the **&&** and **,** operators have the same semantics, they are resolved with different priorities. The **&&** operator precedes the **||** operator, and both the **&&** and **||** operators together precede the **,** operator. Use the comma operator at the top-level constraint for optimal decision engine performance and human readability.

You cannot embed a comma operator in a composite constraint expression, such as in parentheses:

Example of misused comma in composite constraint expression

```
// Do not use the following format:
Person( ( age > 50, weight > 80 ) || height > 2 )

// Use the following format instead:
Person( ( age > 50 && weight > 80 ) || height > 2 )
```

2.8.2. Bound variables in patterns and constraints

You can bind variables to patterns and constraints to refer to matched objects in other portions of a rule. Bound variables can help you define rules more efficiently or more consistently with how you annotate facts in your data model. To differentiate more easily between variables and fields in a rule, use the standard format **$variable** for variables, especially in complex rules. This convention is helpful but not required in DRL.

For example, the following DRL rule uses the variable **$p** for a pattern with the **Person** fact:

Pattern with a bound variable

```
rule "simple rule"
  when
    $p : Person()
  then
    System.out.println( "Person " + $p );
end
```

Similarly, you can also bind variables to properties in pattern constraints, as shown in the following example:

```
// Two persons of the same age:
Person( $firstAge : age ) // Binding
Person( age == $firstAge ) // Constraint expression
```

NOTE

Ensure that you separate constraint bindings and constraint expressions for clearer and more efficient rule definitions. Although mixed bindings and expressions are supported, they can complicate patterns and affect evaluation efficiency.

```
// Do not use the following format:
Person( $age : age * 2 < 100 )

// Use the following format instead:
Person( age * 2 < 100, $age : age )
```

The decision engine does not support bindings to the same declaration, but does support *unification* of arguments across several properties. While positional arguments are always processed with unification, the unification symbol **:=** exists for named arguments.

The following example patterns unify the **age** property across two **Person** facts:

Example pattern with unification

```
Person( $age := age )
Person( $age := age )
```

Unification declares a binding for the first occurrence and constrains to the same value of the bound field for sequence occurrences.

2.8.3. Nested constraints and inline casts

In some cases, you might need to access multiple properties of a nested object, as shown in the following example:

Example pattern to access multiple properties

```
Person( name == "mark", address.city == "london", address.country == "uk" )
```

You can group these property accessors to nested objects with the syntax **.(<constraints>)** for more readable rules, as shown in the following example:

Example pattern with grouped constraints

```
Person( name == "mark", address.( city == "london", country == "uk") )
```

NOTE

The period prefix **.** differentiates the nested object constraints from a method call.

When you work with nested objects in patterns, you can use the syntax **<type>#<subtype>** to cast to a subtype and make the getters from the parent type available to the subtype. You can use either the object name or fully qualified class name, and you can cast to one or multiple subtypes, as shown in the following examples:

Example patterns with inline casting to a subtype

```
// Inline casting with subtype name:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000 )
```

These example patterns cast **Address** to **LongAddress**, and additionally to **DetailedCountry** in the last example, making the parent getters available to the subtypes in each case.

You can use the **instanceof** operator to infer the results of the specified type in subsequent uses of that field with the pattern, as shown in the following example:

```
Person( name == "mark", address instanceof LongAddress, address.country == "uk" )
```

If an inline cast is not possible (for example, if **instanceof** returns **false**), the evaluation is considered **false**.

2.8.4. Date literal in constraints

By default, the decision engine supports the date format **dd-mmm-yyyy**. You can customize the date format, including a time format mask if needed, by providing an alternative format mask with the system property **drools.dateformat="dd-mmm-yyyy hh:mm"**. You can also customize the date format by changing the language locale with the **drools.defaultlanguage** and **drools.defaultcountry** system properties (for example, the locale of Thailand is set as **drools.defaultlanguage=th** and **drools.defaultcountry=TH**).

Example pattern with a date literal restriction

```
Person( bornBefore < "27-Oct-2009" )
```

2.8.5. Supported operators in DRL pattern constraints

DRL supports standard Java semantics for operators in pattern constraints, with some exceptions and with some additional operators that are unique in DRL. The following list summarizes the operators that are handled differently in DRL constraints than in standard Java semantics or that are unique in DRL constraints.

.(), #

Use the **.()** operator to group property accessors to nested objects, and use the **#** operator to cast to a subtype in nested objects. Casting to a subtype makes the getters from the parent type available to the subtype. You can use either the object name or fully qualified class name, and you can cast to one or multiple subtypes.

Example patterns with nested objects

```
// Ungrouped property accessors:
Person( name == "mark", address.city == "london", address.country == "uk" )

// Grouped property accessors:
Person( name == "mark", address.( city == "london", country == "uk") )
```
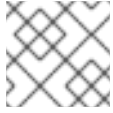
NOTE

The period prefix **.** differentiates the nested object constraints from a method call.

Example patterns with inline casting to a subtype

```
// Inline casting with subtype name:
Person( name == "mark", address#LongAddress.country == "uk" )

// Inline casting with fully qualified class name:
Person( name == "mark", address#org.domain.LongAddress.country == "uk" )

// Multiple inline casts:
Person( name == "mark", address#LongAddress.country#DetailedCountry.population > 10000000
)
```

!.

Use this operator to dereference a property in a null-safe way. The value to the left of the **!.** operator must be not null (interpreted as **!= null**) in order to give a positive result for pattern matching.

Example constraint with null-safe dereferencing

```
Person( $streetName : address!.street )

// This is internally rewritten in the following way:

Person( address != null, $streetName : address.street )
```

[]

Use this operator to access a **List** value by index or a **Map** value by key.

Example constraints with **List** and **Map** access

```
// The following format is the same as `childList(0).getAge() == 18`:
Person(childList[0].age == 18)

// The following format is the same as `credentialMap.get("jdoe").isValid()`:
Person(credentialMap["jdoe"].valid)
```

<, <=, >, >=

Use these operators on properties with natural ordering. For example, for **Date** fields, the **<** operator means *before*, and for **String** fields, the operator means *alphabetically before*. These properties apply only to comparable properties.

Example constraints with **before operator**

```
Person( birthDate < $otherBirthDate )

Person( firstName < $otherFirstName )
```

==, !=

Use these operators as **equals()** and **!equals()** methods in constraints, instead of the usual **same** and **not same** semantics.

Example constraint with null-safe equality

```
Person( firstName == "John" )

// This is similar to the following formats:

java.util.Objects.equals(person.getFirstName(), "John")
"John".equals(person.getFirstName())
```

Example constraint with null-safe not equality

```
Person( firstName != "John" )

// This is similar to the following format:

!java.util.Objects.equals(person.getFirstName(), "John")
```

&&, ||

Use these operators to create an abbreviated combined relation condition that adds more than one restriction on a field. You can group constraints with parentheses **()** to create a recursive syntax pattern.

Example constraints with abbreviated combined relation

```
// Simple abbreviated combined relation condition using a single `&&`:
Person(age > 30 && < 40)

// Complex abbreviated combined relation using groupings:
Person(age ((> 30 && < 40) || (> 20 && < 25)))

// Mixing abbreviated combined relation with constraint connectives:
Person(age > 30 && < 40 || location == "london")
```

matches, not matches

Use these operators to indicate that a field matches or does not match a specified Java regular expression. Typically, the regular expression is a **String** literal, but variables that resolve to a valid regular expression are also supported. These operators apply only to **String** properties. If you use **matches** against a **null** value, the resulting evaluation is always **false**. If you use **not matches** against a **null** value, the resulting evaluation is always **true**. As in Java, regular expressions that you write as **String** literals must use a double backslash \\ to escape.

Example constraint to match or not match a regular expression

```
Person( country matches "(USA)?\\S*UK" )

Person( country not matches "(USA)?\\S*UK" )
```

contains, not contains

Use these operators to verify whether a field that is an **Array** or a **Collection** contains or does not contain a specified value. These operators apply to **Array** or **Collection** properties, but you can also use these operators in place of **String.contains()** and **!String.contains()** constraints checks.

Example constraints with contains and not contains for a Collection

```
// Collection with a specified field:
FamilyTree( countries contains "UK" )

FamilyTree( countries not contains "UK" )


// Collection with a variable:
FamilyTree( countries contains $var )

FamilyTree( countries not contains $var )
```

Example constraints with contains and not contains for a String literal

```
// Sting literal with a specified field:
Person( fullName contains "Jr" )

Person( fullName not contains "Jr" )


// String literal with a variable:
Person( fullName contains $var )

Person( fullName not contains $var )
```

> **NOTE**
>
> For backward compatibility, the **excludes** operator is a supported synonym for **not contains**.

memberOf, not memberOf

Use these operators to verify whether a field is a member of or is not a member of an **Array** or a **Collection** that is defined as a variable. The **Array** or **Collection** must be a variable.

Example constraints with memberOf and not memberOf with a Collection

```
FamilyTree( person memberOf $europeanDescendants )

FamilyTree( person not memberOf $europeanDescendants )
```

soundslike

Use this operator to verify whether a word has almost the same sound, using English pronunciation, as the given value (similar to the **matches** operator). This operator uses the Soundex algorithm.

Example constraint with soundslike

```
// Match firstName "Jon" or "John":
Person( firstName soundslike "John" )
```

str

Use this operator to verify whether a field that is a **String** starts with or ends with a specified value. You can also use this operator to verify the length of the **String**.

Example constraints with str

```
// Verify what the String starts with:
Message( routingValue str[startsWith] "R1" )

// Verify what the String ends with:
Message( routingValue str[endsWith] "R2" )

// Verify the length of the String:
Message( routingValue str[length] 17 )
```

in, notin

Use these operators to specify more than one possible value to match in a constraint (compound value restriction). This functionality of compound value restriction is supported only in the **in** and **not in** operators. The second operand of these operators must be a comma-separated list of values enclosed in parentheses. You can provide values as variables, literals, return values, or qualified identifiers. These operators are internally rewritten as a list of multiple restrictions using the operators **==** or **!=**.

Example constraints with in and notin

```
Person( $color : favoriteColor )
Color( type in ( "red", "blue", $color ) )

Person( $color : favoriteColor )
Color( type notin ( "red", "blue", $color ) )
```

2.8.6. Operator precedence in DRL pattern constraints

DRL supports standard Java operator precedence for applicable constraint operators, with some exceptions and with some additional operators that are unique in DRL. The following table lists DRL operator precedence where applicable, from highest to lowest precedence:

Table 2.2. Operator precedence in DRL pattern constraints

Operator type	Operators	Notes
Nested or null-safe property access	.(), !.	Not standard Java semantics
List or **Map** access	[]	Not standard Java semantics
Constraint binding	:	Not standard Java semantics

Operator type	Operators	Notes
Multiplicative	***, /%**	
Additive	**+, -**	
Shift	**>>, >>>, <<**	
Relational	**<, <=, >, >=, instanceof**	
Equality	**== !=**	Uses **equals()** and **!equals()** semantics, not standard Java **same** and **not same** semantics
Non-short-circuiting **AND**	**&**	
Non-short-circuiting exclusive **OR**	**^**	
Non-short-circuiting inclusive **OR**	**|**	
Logical **AND**	**&&**	
Logical **OR**	**||**	
Ternary	**? :**	
Comma-separated **AND**	**,**	Not standard Java semantics

2.8.7. Supported rule condition elements in DRL (keywords)

DRL supports the following rule condition elements (keywords) that you can use with the patterns that you define in DRL rule conditions:

and

Use this to group conditional components into a logical conjunction. Infix and prefix **and** are supported. You can group patterns explicitly with parentheses **()**. By default, all listed patterns are combined with **and** when no conjunction is specified.

Example patterns with and

```
//Infix `and`:
Color( colorType : type ) and Person( favoriteColor == colorType )

//Infix `and` with grouping:
(Color( colorType : type ) and (Person( favoriteColor == colorType ) or Person( favoriteColor ==
colorType ))
```

```
// Prefix `and`:
(and Color( colorType : type ) Person( favoriteColor == colorType ))

// Default implicit `and`:
Color( colorType : type )
Person( favoriteColor == colorType )
```

> **NOTE**
>
> Do not use a leading declaration binding with the **and** keyword (as you can with **or**, for example). A declaration can only reference a single fact at a time, and if you use a declaration binding with **and**, then when **and** is satisfied, it matches both facts and results in an error.
>
> **Example misuse of and**
>
> ```
> // Causes compile error:
> $person : (Person( name == "Romeo" ) and Person( name == "Juliet"))
> ```

or

Use this to group conditional components into a logical disjunction. Infix and prefix **or** are supported. You can group patterns explicitly with parentheses **()**. You can also use pattern binding with **or**, but each pattern must be bound separately.

Example patterns with or

```
//Infix `or`:
Color( colorType : type ) or Person( favoriteColor == colorType )

//Infix `or` with grouping:
(Color( colorType : type ) or (Person( favoriteColor == colorType ) and Person( favoriteColor ==
colorType ))

// Prefix `or`:
(or Color( colorType : type ) Person( favoriteColor == colorType ))
```

Example patterns with or and pattern binding

```
pensioner : (Person( sex == "f", age > 60 ) or Person( sex == "m", age > 65 ))

(or pensioner : Person( sex == "f", age > 60 )
    pensioner : Person( sex == "m", age > 65 ))
```

The behavior of the **or** condition element is different from the connective **||** operator for constraints and restrictions in field constraints. The decision engine does not directly interpret the **or** element but uses logical transformations to rewrite a rule with **or** as a number of sub-rules. This process ultimately results in a rule that has a single **or** as the root node and one sub-rule for each of its condition elements. Each sub-rule is activated and executed like any normal rule, with no special behavior or interaction between the sub-rules.

Therefore, consider the **or** condition element a shortcut for generating two or more similar rules that, in turn, can create multiple activations when two or more terms of the disjunction are true.

exists

Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches. If you use this element with multiple patterns, enclose the patterns with parentheses **()**.

Example patterns with exists

```
exists Person( firstName == "John")

exists (Person( firstName == "John", age == 42 ))

exists (Person( firstName == "John" ) and
     Person( lastName == "Doe" ))
```

not

Use this to specify facts and constraints that must not exist. If you use this element with multiple patterns, enclose the patterns with parentheses **()**.

Example patterns with not

```
not Person( firstName == "John")

not (Person( firstName == "John", age == 42 ))

not (Person( firstName == "John" ) and
    Person( lastName == "Doe" ))
```

forall

Use this to verify whether all facts that match the first pattern match all the remaining patterns. When a **forall** construct is satisfied, the rule evaluates to **true**. This element is a scope delimiter, so it can use any previously bound variable, but no variable bound inside of it is available for use outside of it.

Example rule with forall

```
rule "All full-time employees have red ID badges"
  when
    forall( $emp : Employee( type == "fulltime" )
            Employee( this == $emp, badgeColor = "red" ) )
  then
    // True, all full-time employees have red ID badges.
end
```

In this example, the rule selects all **Employee** objects whose type is **"fulltime"**. For each fact that matches this pattern, the rule evaluates the patterns that follow (badge color) and if they match, the rule evaluates to **true**.

To state that all facts of a given type in the working memory of the decision engine must match a set of constraints, you can use **forall** with a single pattern for simplicity.

Example rule with forall and a single pattern

```
rule "All full-time employees have red ID badges"
  when
    forall( Employee( badgeColor = "red" ) )
  then
    // True, all full-time employees have red ID badges.
  end
```

You can use **forall** constructs with multiple patterns or nest them with other condition elements, such as inside a **not** element construct.

Example rule with **forall** and multiple patterns

```
rule "All employees have health and dental care programs"
  when
    forall( $emp : Employee()
        HealthCare( employee == $emp )
        DentalCare( employee == $emp )
      )
  then
    // True, all employees have health and dental care.
  end
```

Example rule with **forall** and **not**

```
rule "Not all employees have health and dental care"
  when
    not ( forall( $emp : Employee()
            HealthCare( employee == $emp )
            DentalCare( employee == $emp ) )
      )
  then
    // True, not all employees have health and dental care.
  end
```
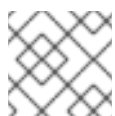
> **NOTE**
>
> The format **forall(p1 p2 p3 ...)** is equivalent to **not(p1 and not(and p2 p3 ...))**.

from

Use this to specify a data source for a pattern. This enables the decision engine to reason over data that is not in the working memory. The data source can be a sub-field on a bound variable or the result of a method call. The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, the **from** element enables you to easily use object property navigation, execute method calls, and access maps and collection elements.

Example rule with **from** and pattern binding

```
rule "Validate zipcode"
  when
    Person( $personAddress : address )
    Address( zipcode == "23920W" ) from $personAddress
```

```
  then
    // Zip code is okay.
  end
```

Example rule with from and a graph notation

```
rule "Validate zipcode"
  when
    $p : Person()
    $a : Address( zipcode == "23920W" ) from $p.address
  then
    // Zip code is okay.
end
```

Example rule with from to iterate over all objects

```
rule "Apply 10% discount to all items over US$ 100 in an order"
  when
    $order : Order()
    $item  : OrderItem( value > 100 ) from $order.items
  then
    // Apply discount to `$item`.
end
```
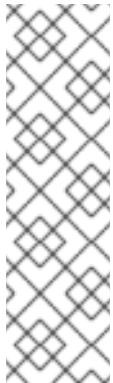
> **NOTE**
>
> For large collections of objects, instead of adding an object with a large graph that the
> decision engine must iterate over frequently, add the collection directly to the KIE
> session and then join the collection in the condition, as shown in the following
> example:
>
> ```
>   when
>     $order : Order()
>     OrderItem( value > 100, order == $order )
> ```

Example rule with from and lock-on-active rule attribute

```
rule "Assign people in North Carolina (NC) to sales region 1"
  ruleflow-group "test"
  lock-on-active true
  when
    $p : Person()
    $a : Address( state == "NC" ) from $p.address
  then
    modify ($p) {} // Assign the person to sales region 1.
end

rule "Apply a discount to people in the city of Raleigh"
  ruleflow-group "test"
  lock-on-active true
  when
    $p : Person()
    $a : Address( city == "Raleigh" ) from $p.address
```

```
    then
        modify ($p) {} // Apply discount to the person.
    end
```

> **IMPORTANT**
>
> Using **from** with **lock-on-active** rule attribute can result in rules not being executed. You can address this issue in one of the following ways:
>
> - Avoid using the **from** element when you can insert all facts into the working memory of the decision engine or use nested object references in your constraint expressions.
>
> - Place the variable used in the **modify()** block as the last sentence in your rule condition.
>
> - Avoid using the **lock-on-active** rule attribute when you can explicitly manage how rules within the same ruleflow group place activations on one another.

The pattern that contains a **from** clause cannot be followed by another pattern starting with a parenthesis. The reason for this restriction is that the DRL parser reads the **from** expression as **"from $l (String() or Number())"** and it cannot differentiate this expression from a function call. The simplest workaround to this is to wrap the **from** clause in parentheses, as shown in the following example:

Example rules with from used incorrectly and correctly

```
// Do not use `from` in this way:
rule R
  when
    $l : List()
    String() from $l
    (String() or Number())
  then
    // Actions
end

// Use `from` in this way instead:
rule R
  when
    $l : List()
    (String() from $l)
    (String() or Number())
  then
    // Actions
end
```

entry-point

Use this to define an entry point, or *event stream*, corresponding to a data source for the pattern. This element is typically used with the **from** condition element. You can declare an entry point for events so that the decision engine uses data from only that entry point to evaluate the rules. You can declare an entry point either implicitly by referencing it in DRL rules or explicitly in your Java application.

Example rule with from entry point

Example rule with from entry-point

```
rule "Authorize withdrawal"
  when
    WithdrawRequest( $ai : accountId, $am : amount ) from entry-point "ATM Stream"
    CheckingAccount( accountId == $ai, balance > $am )
  then
    // Authorize withdrawal.
end
```

Example Java application code with EntryPoint object and inserted facts

```
import org.kie.api.runtime.KieSession;
import org.kie.api.runtime.rule.EntryPoint;

// Create your KIE base and KIE session as usual:
KieSession session = ...

// Create a reference to the entry point:
EntryPoint atmStream = session.getEntryPoint("ATM Stream");

// Start inserting your facts into the entry point:
atmStream.insert(aWithdrawRequest);
```

collect

Use this to define a collection of objects that the rule can use as part of the condition. The rule obtains the collection either from a specified source or from the working memory of the decision engine. The result pattern of the **collect** element can be any concrete class that implements the **java.util.Collection** interface and provides a default no-arg public constructor. You can use Java collections like **List**, **LinkedList**, and **HashSet**, or your own class. If variables are bound before the **collect** element in a condition, you can use the variables to constrain both your source and result patterns. However, any binding made inside the **collect** element is not available for use outside of it.

Example rule with collect

```
import java.util.List

rule "Raise priority when system has more than three pending alarms"
  when
    $system : System()
    $alarms : List( size >= 3 )
          from collect( Alarm( system == $system, status == 'pending' ) )
  then
    // Raise priority because `$system` has three or more `$alarms` pending.
end
```

In this example, the rule assesses all pending alarms in the working memory of the decision engine for each given system and groups them in a **List**. If three or more alarms are found for a given system, the rule is executed.

You can also use the **collect** element with nested **from** elements, as shown in the following example:

Example rule with collect and nested from

```
import java.util.LinkedList;

rule "Send a message to all parents"
  when
    $town : Town( name == 'Paris' )
    $mothers : LinkedList()
          from collect( Person( children > 0 )
                  from $town.getPeople()
                      )
  then
    // Send a message to all parents.
end
```

accumulate

Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to **true**). This element is a more flexible and powerful form of the **collect** condition element. You can use predefined functions in your **accumulate** conditions or implement custom functions as needed. You can also use the abbreviation **acc** for **accumulate** in rule conditions.

Use the following format to define **accumulate** conditions in rules:

Preferred format for accumulate

```
accumulate( <source pattern>; <functions> [;<constraints>] )
```
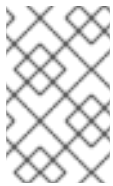
> **NOTE**
>
> Although the decision engine supports alternate formats for the **accumulate** element for backward compatibility, this format is preferred for optimal performance in rules and applications.

The decision engine supports the following predefined **accumulate** functions. These functions accept any expression as input.

- **average**

- **min**

- **max**

- **count**

- **sum**

- **collectList**

- **collectSet**

In the following example rule, **min**, **max**, and **average** are **accumulate** functions that calculate the minimum, maximum, and average temperature values over all the readings for each sensor:

Example rule with accumulate to calculate temperature values

```
rule "Raise alarm"
  when
    $s : Sensor()
    accumulate( Reading( sensor == $s, $temp : temperature );
            $min : min( $temp ),
            $max : max( $temp ),
            $avg : average( $temp );
            $min < 20, $avg > 70 )
  then
    // Raise the alarm.
end
```

The following example rule uses the **average** function with **accumulate** to calculate the average profit for all items in an order:

Example rule with accumulate to calculate average profit

```
rule "Average profit"
  when
    $order : Order()
    accumulate( OrderItem( order == $order, $cost : cost, $price : price );
            $avgProfit : average( 1 - $cost / $price ) )
  then
    // Average profit for `$order` is `$avgProfit`.
end
```

To use custom, domain-specific functions in **accumulate** conditions, create a Java class that implements the **org.kie.api.runtime.rule.AccumulateFunction** interface. For example, the following Java class defines a custom implementation of an **AverageData** function:

Example Java class with custom implementation of average function

```
// An implementation of an accumulator capable of calculating average values

public class AverageAccumulateFunction implements
org.kie.api.runtime.rule.AccumulateFunction<AverageAccumulateFunction.AverageData> {

  public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {

  }

  public void writeExternal(ObjectOutput out) throws IOException {

  }

  public static class AverageData implements Externalizable {
    public int    count = 0;
    public double total = 0;

    public AverageData() {}

    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
      count  = in.readInt();
      total  = in.readDouble();
    }
```

```java
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeInt(count);
        out.writeDouble(total);
    }

}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#createContext()
 */
public AverageData createContext() {
    return new AverageData();
}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#init(java.io.Serializable)
 */
public void init(AverageData context) {
    context.count = 0;
    context.total = 0;
}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#accumulate(java.io.Serializable,
java.lang.Object)
 */
public void accumulate(AverageData context,
                       Object value) {
    context.count++;
    context.total += ((Number) value).doubleValue();
}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#reverse(java.io.Serializable,
java.lang.Object)
 */
public void reverse(AverageData context, Object value) {
    context.count--;
    context.total -= ((Number) value).doubleValue();
}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#getResult(java.io.Serializable)
 */
public Object getResult(AverageData context) {
    return new Double( context.count == 0 ? 0 : context.total / context.count );
}

/* (non-Javadoc)
 * @see org.kie.api.runtime.rule.AccumulateFunction#supportsReverse()
 */
public boolean supportsReverse() {
    return true;
}
```

```
    /* (non-Javadoc)
     * @see org.kie.api.runtime.rule.AccumulateFunction#getResultType()
     */
    public Class< ? > getResultType() {
        return Number.class;
    }

}
```

To use the custom function in a DRL rule, import the function using the **import accumulate** statement:

Format to import a custom function

```
import accumulate <class_name> <function_name>
```

Example rule with the imported average function

```
import accumulate AverageAccumulateFunction.AverageData average

rule "Average profit"
  when
    $order : Order()
    accumulate( OrderItem( order == $order, $cost : cost, $price : price );
            $avgProfit : average( 1 - $cost / $price ) )
  then
    // Average profit for `$order` is `$avgProfit`.
end
```

2.8.8. OOPath syntax with graphs of objects in DRL rule conditions

OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

When the field of a fact is a collection, you can use the **from** condition element (keyword) to bind and reason over all the items in that collection one by one. If you need to browse a graph of objects in the rule condition constraints, the extensive use of the **from** condition element results in a verbose and repetitive syntax, as shown in the following example:

Example rule that browses a graph of objects with from

```
rule "Find all grades for Big Data exam"
  when
    $student: Student( $plan: plan )
    $exam: Exam( course == "Big Data" ) from $plan.exams
    $grade: Grade() from $exam.grades
  then
    // Actions
end
```

In this example, the domain model contains a **Student** object with a **Plan** of study. The **Plan** can have

zero or more **Exam** instances and an **Exam** can have zero or more **Grade** instances. Only the root object of the graph, the **Student** in this case, needs to be in the working memory of the decision engine for this rule setup to function.

As a more efficient alternative to using extensive **from** statements, you can use the abbreviated OOPath syntax, as shown in the following example:

Example rule that browses a graph of objects with OOPath syntax

```
rule "Find all grades for Big Data exam"
  when
    Student( $grade: /plan/exams[course == "Big Data"]/grades )
  then
    // Actions
end
```

Formally, the core grammar of an OOPath expression is defined in extended Backus-Naur form (EBNF) notation in the following way:

EBNF notation for OOPath expressions

```
OOPExpr = [ID ( ":" | ":=" )] ( "/" | "?/" ) OOPSegment { ( "/" | "?/" | "." ) OOPSegment } ;
OOPSegment = ID ["#" ID] ["[" ( Number | Constraints ) "]"]
```

In practice, an OOPath expression has the following features and capabilities:

- Starts with a forward slash / or with a question mark and forward slash **?**/ if it is a non-reactive OOPath expression (described later in this section).

- Can dereference a single property of an object with the period **.** operator.

- Can dereference multiple properties of an object with the forward slash / operator. If a collection is returned, the expression iterates over the values in the collection.

- Can filter out traversed objects that do not satisfy one or more constraints. The constraints are written as predicate expressions between square brackets, as shown in the following example:

  Constraints as a predicate expression

  ```
  Student( $grade: /plan/exams[ course == "Big Data" ]/grades )
  ```

- Can downcast a traversed object to a subclass of the class declared in the generic collection. Subsequent constraints can also safely access the properties declared only in that subclass, as shown in the following example. Objects that are not instances of the class specified in this inline cast are automatically filtered out.

  Constraints with downcast objects

  ```
  Student( $grade: /plan/exams#AdvancedExam[ course == "Big Data", level > 3 ]/grades )
  ```

- Can backreference an object of the graph that was traversed before the currently iterated graph. For example, the following OOPath expression matches only the grades that are above the average for the passed exam:

Constraints with backreferenced object

> Student($grade: /plan/exams/grades[result > ../averageResult])

- Can recursively be another OOPath expression, as shown in the following example:

Recursive constraint expression

> Student($exam: /plan/exams[/grades[result > 20]])

- Can access objects by their index between square brackets **[]**, as shown in the following example. To adhere to Java convention, OOPath indexes are 0-based, while XPath indexes are 1-based.

Constraints with access to objects by index

> Student($grade: /plan/exams[0]/grades)

OOPath expressions can be reactive or non-reactive. The decision engine does not react to updates involving a deeply nested object that is traversed during the evaluation of an OOPath expression.

To make these objects reactive to changes, modify the objects to extend the class **org.drools.core.phreak.ReactiveObject**. After you modify an object to extend the **ReactiveObject** class, the domain object invokes the inherited method **notifyModification** to notify the decision engine when one of the fields has been updated, as shown in the following example:

Example object method to notify the decision engine that an exam has been moved to a different course

```
public void setCourse(String course) {
    this.course = course;
    notifyModification(this);
    }
```

With the following corresponding OOPath expression, when an exam is moved to a different course, the rule is re-executed and the list of grades matching the rule is recomputed:

Example OOPath expression from "Big Data" rule

> Student($grade: /plan/exams[course == "Big Data"]/grades)

You can also use the **?/** separator instead of the **/** separator to disable reactivity in only one sub-portion of an OOPath expression, as shown in the following example:

Example OOPath expression that is partially non-reactive

> Student($grade: /plan/exams[course == "Big Data"]?/grades)

With this example, the decision engine reacts to a change made to an exam or if an exam is added to the plan, but not if a new grade is added to an existing exam.

If an OOPath portion is non-reactive, all remaining portions of the OOPath expression also become non-reactive. For example, the following OOPath expression is completely non-reactive:

Example OOPath expression that is completely non-reactive

```
Student( $grade: ?/plan/exams[ course == "Big Data" ]/grades )
```

For this reason, you cannot use the **?/** separator more than once in the same OOPath expression. For example, the following expression causes a compilation error:

Example OOPath expression with duplicate non-reactivity markers

```
Student( $grade: /plan?/exams[ course == "Big Data" ]?/grades )
```

Another alternative for enabling OOPath expression reactivity is to use the dedicated implementations for **List** and **Set** interfaces in Red Hat Decision Manager. These implementations are the **ReactiveList** and **ReactiveSet** classes. A **ReactiveCollection** class is also available. The implementations also provide reactive support for performing mutable operations through the **Iterator** and **ListIterator** classes.

The following example class uses these classes to configure OOPath expression reactivity:

Example Java class to configure OOPath expression reactivity

```
public class School extends AbstractReactiveObject {
    private String name;
    private final List<Child> children = new ReactiveList<Child>();  1

    public void setName(String name) {
        this.name = name;
        notifyModification();  2
    }

    public void addChild(Child child) {
        children.add(child);  3
        // No need to call `notifyModification()` here
    }
}
```

1 Uses the **ReactiveList** instance for reactive support over the standard Java **List** instance.

2 Uses the required **notifyModification()** method for when a field is changed in reactive support.

3 The **children** field is a **ReactiveList** instance, so the **notifyModification()** method call is not required. The notification is handled automatically, like all other mutating operations performed over the **children** field.

2.9. RULE ACTIONS IN DRL (THEN)

The **then** part of the rule (also known as the *Right Hand Side (RHS)* of the rule) contains the actions to be performed when the conditional part of the rule has been met. Actions consist of one or more *methods* that execute consequences based on the rule conditions and on available data objects in the package. For example, if a bank requires loan applicants to have over 21 years of age (with a rule condition **Applicant(age < 21)**) and a loan applicant is under 21 years old, the **then** action of an **"Underage"** rule would be **setApproved(false)**, declining the loan because the applicant is under age.

The main purpose of rule actions is to to insert, delete, or modify data in the working memory of the decision engine. Effective rule actions are small, declarative, and readable. If you need to use imperative or conditional code in rule actions, then divide the rule into multiple smaller and more declarative rules.

Example rule for loan application age limit

```
rule "Underage"
  when
    application : LoanApplication()
    Applicant( age < 21 )
  then
    application.setApproved( false );
    application.setExplanation( "Underage" );
end
```

2.9.1. Supported rule action methods in DRL

DRL supports the following rule action methods that you can use in DRL rule actions. You can use these methods to modify the working memory of the decision engine without having to first reference a working memory instance. These methods act as shortcuts to the methods provided by the **KnowledgeHelper** class in your Red Hat Decision Manager distribution.

For all rule action methods, download the **Red Hat Decision Manager 7.6.0 Source Distribution**ZIP file from the Red Hat Customer Portal and navigate to ~/**rhdm-7.6.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/spi/KnowledgeHelper.java**.

set

Use this to set the value of a field.

```
set<field> ( <value> )
```

Example rule action to set the values of a loan application approval

```
$application.setApproved ( false );
$application.setExplanation( "has been bankrupt" );
```

modify

Use this to specify fields to be modified for a fact and to notify the decision engine of the change. This method provides a structured approach to fact updates. It combines the **update** operation with setter calls to change object fields.

```
modify ( <fact-expression> ) {
    <expression>,
    <expression>,
    ...
}
```

Example rule action to modify a loan application amount and approval

```
modify( LoanApplication ) {
     setAmount( 100 ),
     setApproved ( true )
}
```

■

update

Use this to specify fields and the entire related fact to be updated and to notify the decision engine of the change. After a fact has changed, you must call **update** before changing another fact that might be affected by the updated values. To avoid this added step, use the **modify** method instead.

```
update ( <object, <handle> )  // Informs the decision engine that an object has changed

update ( <object> )  // Causes `KieSession` to search for a fact handle of the object
```

Example rule action to update a loan application amount and approval

```
LoanApplication.setAmount( 100 );
update( LoanApplication );
```
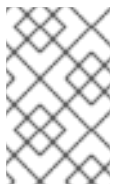
> **NOTE**
>
> If you provide property-change listeners, you do not need to call this method when an object changes. For more information about property-change listeners, see *Decision engine in Red Hat Decision Manager*.

insert

Use this to insert a **new** fact into the working memory of the decision engine and to define resulting fields and values as needed for the fact.

```
insert( new <object> );
```

Example rule action to insert a new loan applicant object

```
insert( new Applicant() );
```

insertLogical

Use this to insert a **new** fact logically into the decision engine. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts must be retracted explicitly. After logical insertions, the facts that were inserted are automatically retracted when the conditions in the rules that inserted the facts are no longer true.

```
insertLogical( new <object> );
```

Example rule action to logically insert a new loan applicant object

```
insertLogical( new Applicant() );
```

delete

Use this to remove an object from the decision engine. The keyword **retract** is also supported in DRL and executes the same action, but **delete** is typically preferred in DRL code for consistency with the keyword **insert**.

```
delete( <object> );
```

Example rule action to delete a loan applicant object

```
delete( Applicant );
```

2.9.2. Other rule action methods from drools and kcontext variables

In addition to the standard rule action methods, the decision engine supports methods in conjunction with the predefined **drools** and **kcontext** variables that you can also use in rule actions.

You can use the **drools** variable to call methods from the **KnowledgeHelper** class in your Red Hat Decision Manager distribution, which is also the class that the standard rule action methods are based on. For all **drools** rule action options, download the **Red Hat Decision Manager 7.6.0 Source Distribution** ZIP file from the Red Hat Customer Portal and navigate to ~/**rhdm-7.6.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/spi/KnowledgeHelper.java**.

The following examples are common methods that you can use with the **drools** variable:

- **drools.halt()**: Terminates rule execution if a user or application has previously called **fireUntilHalt()**. When a user or application calls **fireUntilHalt()**, the decision engine starts in *active mode* and evaluates rules continually until the user or application explicitly calls **halt()**. Otherwise, by default, the decision engine runs in *passive mode* and evaluates rules only when a user or an application explicitly calls **fireAllRules()**.

- **drools.getWorkingMemory()**: Returns the **WorkingMemory** object.

- **drools.setFocus("<agenda_group>")**: Sets the focus to a specified agenda group to which the rule belongs.

- **drools.getRule().getName()**: Returns the name of the rule.

- **drools.getTuple()**, **drools.getActivation()**: Returns the **Tuple** that matches the currently executing rule and then delivers the corresponding **Activation**. These calls are useful for logging and debugging purposes.

You can use the **kcontext** variable with the **getKieRuntime()** method to call other methods from the **KieContext** class and, by extension, the **RuleContext** class in your Red Hat Decision Manager distribution. The full Knowledge Runtime API is exposed through the **kcontext** variable and provides extensive rule action methods. For all **kcontext** rule action options, download the **Red Hat Decision Manager 7.6.0 Source Distribution** ZIP file from the Red Hat Customer Portal and navigate to ~/**rhdm-7.6.0-sources/src/kie-api-parent-$VERSION/kie-api/src/main/java/org/kie/api/runtime/rule/RuleContext.java**.

The following examples are common methods that you can use with the **kcontext.getKieRuntime()** variable-method combination:

- **kcontext.getKieRuntime().halt()**: Terminates rule execution if a user or application has previously called **fireUntilHalt()**. This method is equivalent to the **drools.halt()** method. When a user or application calls **fireUntilHalt()**, the decision engine starts in *active mode* and evaluates rules continually until the user or application explicitly calls **halt()**. Otherwise, by default, the decision engine runs in *passive mode* and evaluates rules only when a user or an application explicitly calls **fireAllRules()**.

- **kcontext.getKieRuntime().getAgenda()**: Returns a reference to the KIE session **Agenda**, and in turn provides access to rule activation groups, rule agenda groups, and ruleflow groups.

Example call to access agenda group "CleanUp" and set the focus

```
kcontext.getKieRuntime().getAgenda().getAgendaGroup( "CleanUp" ).setFocus();
```

This example is equivalent to **drools.setFocus("CleanUp")**.

- **kcontext.getKieRuntime().getQueryResults(<string> query)**: Runs a query and returns the results. This method is equivalent to **drools.getKieRuntime().getQueryResults()**.

- **kcontext.getKieRuntime().getKieBase()**: Returns the **KieBase** object. The KIE base is the source of all the knowledge in your rule system and the originator of the current KIE session.

- **kcontext.getKieRuntime().setGlobal()**, **~.getGlobal()**, **~.getGlobals()**: Sets or retrieves global variables.

- **kcontext.getKieRuntime().getEnvironment()**: Returns the runtime **Environment**, similar to your operating system environment.

2.9.3. Advanced rule actions with conditional and named consequences

In general, effective rule actions are small, declarative, and readable. However, in some cases, the limitation of having a single consequence for each rule can be challenging and lead to verbose and repetitive rule syntax, as shown in the following example rules:

Example rules with verbose and repetitive syntax

```
rule "Give 10% discount to customers older than 60"
  when
    $customer : Customer( age > 60 )
  then
    modify($customer) { setDiscount( 0.1 ) };
end

rule "Give free parking to customers older than 60"
  when
    $customer : Customer( age > 60 )
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
end
```

A partial solution to the repetition is to make the second rule extend the first rule, as shown in the following modified example:

Partially enhanced example rules with an extended condition

```
rule "Give 10% discount to customers older than 60"
  when
    $customer : Customer( age > 60 )
  then
    modify($customer) { setDiscount( 0.1 ) };
end

rule "Give free parking to customers older than 60"
    extends "Give 10% discount to customers older than 60"
```

```
  when
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
end
```

As a more efficient alternative, you can consolidate the two rules into a single rule with modified conditions and labelled corresponding rule actions, as shown in the following consolidated example:

Consolidated example rule with conditional and named consequences

```
rule "Give 10% discount and free parking to customers older than 60"
  when
    $customer : Customer( age > 60 )
    do[giveDiscount]
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
  then[giveDiscount]
    modify($customer) { setDiscount( 0.1 ) };
end
```

This example rule uses two actions: the usual default action and another action named **giveDiscount**. The **giveDiscount** action is activated in the condition with the keyword **do** when a customer older than 60 years old is found in the KIE base, regardless of whether or not the customer owns a car.

You can configure the activation of a named consequence with an additional condition, such as the **if** statement in the following example. The condition in the **if** statement is always evaluated on the pattern that immediately precedes it.

Consolidated example rule with an additional condition

```
rule "Give free parking to customers older than 60 and 10% discount to golden ones among them"
  when
    $customer : Customer( age > 60 )
    if ( type == "Golden" ) do[giveDiscount]
    $car : Car( owner == $customer )
  then
    modify($car) { setFreeParking( true ) };
  then[giveDiscount]
    modify($customer) { setDiscount( 0.1 ) };
end
```

You can also evaluate different rule conditions using a nested **if** and **else if** construct, as shown in the following more complex example:

Consolidated example rule with more complex conditions

```
rule "Give free parking and 10% discount to over 60 Golden customer and 5% to Silver ones"
  when
    $customer : Customer( age > 60 )
    if ( type == "Golden" ) do[giveDiscount10]
    else if ( type == "Silver" ) break[giveDiscount5]
    $car : Car( owner == $customer )
  then
```

```
    modify($car) { setFreeParking( true ) };
  then[giveDiscount10]
    modify($customer) { setDiscount( 0.1 ) };
  then[giveDiscount5]
    modify($customer) { setDiscount( 0.05 ) };
end
```

This example rule gives a 10% discount and free parking to Golden customers over 60, but only a 5% discount without free parking to Silver customers. The rule activates the consequence named **giveDiscount5** with the keyword **break** instead of **do**. The keyword **do** schedules a consequence in the decision engine agenda, enabling the remaining part of the rule conditions to continue being evaluated, while **break** blocks any further condition evaluation. If a named consequence does not correspond to any condition with **do** but is activated with **break**, the rule fails to compile because the conditional part of the rule is never reached.

2.10. COMMENTS IN DRL FILES

DRL supports single-line comments prefixed with a double forward slash // and multi-line comments enclosed with a forward slash and asterisk /* ... */. You can use DRL comments to annotate rules or any related components in DRL files. DRL comments are ignored by the decision engine when the DRL file is processed.

Example rule with comments

```
rule "Underage"
  // This is a single-line comment.
  when
    $application : LoanApplication()  // This is an in-line comment.
    Applicant( age < 21 )
  then
    /* This is a multi-line comment
    in the rule actions. */
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```
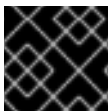
> **IMPORTANT**
>
> The hash symbol **#** is not supported for DRL comments.

2.11. ERROR MESSAGES FOR DRL TROUBLESHOOTING

Red Hat Decision Manager provides standardized messages for DRL errors to help you troubleshoot and resolve problems in your DRL files. The error messages use the following format:

Figure 2.1. Error message format for DRL file problems
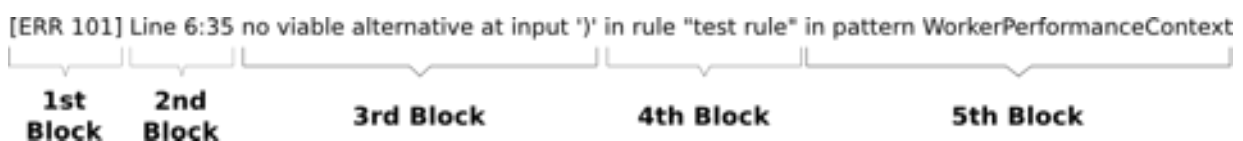
- **1st Block:** Error code

- **2nd Block:** Line and column in the DRL source where the error occurred

- **3rd Block:** Description of the problem

- **4th Block:** Component in the DRL source (rule, function, query) where the error occurred

- **5th Block:** Pattern in the DRL source where the error occurred (if applicable)

Red Hat Decision Manager supports the following standardized error messages:

101: no viable alternative

Indicates that the parser reached a decision point but could not identify an alternative.

Example rule with incorrect spelling

```
1: rule "simple rule"
2:   when
3:     exists Person()
4:     exits Student()  // Must be `exists`
5:   then
6: end
```

Error message

```
[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule "simple rule"
```

Example rule without a rule name

```
1: package org.drools.examples;
2: rule    // Must be `rule "rule name"` (or `rule rule_name` if no spacing)
3:   when
4:     Object()
5:   then
6:     System.out.println("A RHS");
7: end
```

Error message

```
[ERR 101] Line 3:2 no viable alternative at input 'when'
```

In this example, the parser encountered the keyword **when** but expected the rule name, so it flags **when** as the incorrect expected token.

Example rule with incorrect syntax

```
1: rule "simple rule"
2:   when
3:     Student( name == "Andy )  // Must be `"Andy"`
4:   then
5: end
```

Error message

```
[ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule "simple rule" in pattern Student
```

> **NOTE**
>
> A line and column value of **0:-1** means the parser reached the end of the source file (**<eof>**) but encountered incomplete constructs, usually due to missing quotation marks **"..."**, apostrophes **'...'**, or parentheses **(...)**.
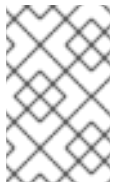
102: mismatched input

Indicates that the parser expected a particular symbol that is missing at the current input position.

Example rule with an incomplete rule statement

```
1: rule simple_rule
2:   when
3:     $p : Person(
       // Must be a complete rule statement
```

Error message

```
[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule "simple rule" in pattern Person
```

> **NOTE**
>
> A line and column value of **0:-1** means the parser reached the end of the source file (**<eof>**) but encountered incomplete constructs, usually due to missing quotation marks **"..."**, apostrophes **'...'**, or parentheses **(...)**.

Example rule with incorrect syntax

```
1: package org.drools.examples;
2:
3: rule "Wrong syntax"
4:   when
5:     not( Car( ( type == "tesla", price == 10000 ) || ( type == "kia", price == 1000 ) ) from $carList )
       // Must use `&&` operators instead of commas `,`
6:   then
7:     System.out.println("OK");
8: end
```

Error messages

```
[ERR 102] Line 5:36 mismatched input ',' expecting ')' in rule "Wrong syntax" in pattern Car
[ERR 101] Line 5:57 no viable alternative at input 'type' in rule "Wrong syntax"
[ERR 102] Line 5:106 mismatched input ')' expecting 'then' in rule "Wrong syntax"
```

In this example, the syntactic problem results in multiple error messages related to each other. The single solution of replacing the commas **,** with **&&** operators resolves all errors. If you encounter multiple errors, resolve one at a time in case errors are consequences of previous errors.

103: failed predicate

Indicates that a validating semantic predicate evaluated to **false**. These semantic predicates are typically used to identify component keywords in DRL files, such as **declare**, **rule**, **exists**, **not**, and others.

Example rule with an invalid keyword

```
 1: package nesting;
 2:
 3: import org.drools.compiler.Person
 4: import org.drools.compiler.Address
 5:
 6: Some text  // Must be a valid DRL keyword
 7:
 8: rule "test something"
 9:   when
10:     $p: Person( name=="Michael" )
11:   then
12:     $p.name = "other";
13:     System.out.println(p.name);
14: end
```

Error message

```
[ERR 103] Line 6:0 rule 'rule_key' failed predicate:
{(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule
```

The **Some text** line is invalid because it does not begin with or is not a part of a DRL keyword construct, so the parser fails to validate the rest of the DRL file.
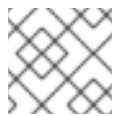
> **NOTE**
>
> This error is similar to **102: mismatched input**, but usually involves DRL keywords.

104: trailing semi-colon not allowed

Indicates that an **eval()** clause in a rule condition uses a semicolon **;** but must not use one.

Example rule with eval() and trailing semicolon

```
1: rule "simple rule"
2:   when
3:     eval( abc(); )  // Must not use semicolon `;`
4:   then
5: end
```

Error message

```
[ERR 104] Line 3:4 trailing semi-colon not allowed in rule "simple rule"
```

105: did not match anything

Indicates that the parser reached a sub-rule in the grammar that must match an alternative at least once, but the sub-rule did not match anything. The parser has entered a branch with no way out.

Example rule with invalid text in an empty condition

```
1: rule "empty condition"
2:   when
3:     None  // Must remove `None` if condition is empty
4:   then
5:      insert( new Person() );
6: end
```

Error message

```
[ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'WHEN' in rule "empty
condition"
```

In this example, the condition is intended to be empty but the word **None** is used. This error is resolved by removing **None**, which is not a valid DRL keyword, data type, or pattern construct.

> **NOTE**
>
> If you encounter other DRL error messages that you cannot resolve, contact your Red Hat Technical Account Manager.

2.12. RULE UNITS IN DRL RULE SETS

Rule units are groups of data sources, global variables, and DRL rules that function together for a specific purpose. You can use rule units to partition a rule set into smaller units, bind different data sources to those units, and then execute the individual unit. Rule units are an enhanced alternative to rule-grouping DRL attributes such as rule agenda groups or activation groups for execution control.

Rule units are helpful when you want to coordinate rule execution so that the complete execution of one rule unit triggers the start of another rule unit and so on. For example, assume that you have a set of rules for data enrichment, another set of rules that processes that data, and another set of rules that extract the output from the processed data. If you add these rule sets into three distinct rule units, you can coordinate those rule units so that complete execution of the first unit triggers the start of the second unit and the complete execution of the second unit triggers the start of third unit.

To define a rule unit, implement the **RuleUnit** interface as shown in the following example:

Example rule unit class

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    private int adultAge;
    private DataSource<Person> persons;

    public AdultUnit( ) { }

    public AdultUnit( DataSource<Person> persons, int age ) {
        this.persons = persons;
        this.age = age;
    }
```

```
    // A data source of `Persons` in this rule unit:
    public DataSource<Person> getPersons() {
        return persons;
    }

    // A global variable in this rule unit:
    public int getAdultAge() {
        return adultAge;
    }

    // Life-cycle methods:
    @Override
    public void onStart() {
        System.out.println("AdultUnit started.");
    }

    @Override
    public void onEnd() {
        System.out.println("AdultUnit ended.");
    }
}
```

In this example, **persons** is a source of facts of type **Person**. A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. The **adultAge** global variable is accessible from all the rules belonging to this rule unit. The last two methods are part of the rule unit life cycle and are invoked by the decision engine.

The decision engine supports the following optional life-cycle methods for rule units:

Table 2.3. Rule unit life-cycle methods

Method	Invoked when
onStart()	Rule unit execution starts
onEnd()	Rule unit execution ends
onSuspend()	Rule unit execution is suspended (used only with **runUntilHalt()**)
onResume()	Rule unit execution is resumed (used only with **runUntilHalt()**)
onYield(RuleUnit other)	The consequence of a rule in the rule unit triggers the execution of a different rule unit

You can add one or more rules to a rule unit. By default, all the rules in a DRL file are automatically associated with a rule unit that follows the naming convention of the DRL file name. If the DRL file is in the same package and has the same name as a class that implements the **RuleUnit** interface, then all of the rules in that DRL file implicitly belong to that rule unit. For example, all the rules in the **AdultUnit.drl** file in the **org.mypackage.myunit** package are automatically part of the rule unit **org.mypackage.myunit.AdultUnit**.

To override this naming convention and explicitly declare the rule unit that the rules in a DRL file belong to, use the **unit** keyword in the DRL file. The **unit** declaration must immediately follow the package declaration and contain the name of the class in that package that the rules in the DRL file are part of.

Example rule unit declaration in a DRL file

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : Person(age >= adultAge) from persons
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```

> **WARNING**
>
> Do not mix rules with and without a rule unit in the same KIE base. Mixing two rule paradigms in a KIE base results in a compilation error.

You can also rewrite the same pattern in a more convenient way using OOPath notation, as shown in the following example:

Example rule unit declaration in a DRL file that uses OOPath notation

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : /persons[age >= adultAge]
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```
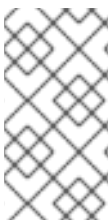
> **NOTE**
>
> OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

In this example, any matching facts in the rule conditions are retrieved from the **persons** data source defined in the **DataSource** definition in the rule unit class. The rule condition and action use the **adultAge** variable in the same way that a global variable is defined at the DRL file level.

To execute one or more rule units defined in a KIE base, create a new **RuleUnitExecutor** class bound to the KIE base, create the rule unit from the relevant data source, and run the rule unit executer:

Example rule unit execution

```
// Create a `RuleUnitExecutor` class and bind it to the KIE base:
KieBase kbase = kieContainer.getKieBase();
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

// Create the `AdultUnit` rule unit using the `persons` data source and run the executor:
RuleUnit adultUnit = new AdultUnit(persons, 18);
executor.run( adultUnit );
```

Rules are executed by the **RuleUnitExecutor** class. The **RuleUnitExecutor** class creates KIE sessions and adds the required **DataSource** objects to those sessions, and then executes the rules based on the **RuleUnit** that is passed as a parameter to the **run()** method.

The example execution code produces the following output when the relevant **Person** facts are inserted in the **persons** data source:

Example rule unit execution output

```
org.mypackage.myunit.AdultUnit started.
Jane is adult and greater than 18
John is adult and greater than 18
org.mypackage.myunit.AdultUnit ended.
```

Instead of explicitly creating the rule unit instance, you can register the rule unit variables in the executor and pass to the executor the rule unit class that you want to run, and then the executor creates an instance of the rule unit. You can then set the **DataSource** definition and other variables as needed before running the rule unit.

Alternate rule unit execution option with registered variables

```
executor.bindVariable( "persons", persons );
     .bindVariable( "adultAge", 18 );
executor.run( AdultUnit.class );
```

The name that you pass to the **RuleUnitExecutor.bindVariable()** method is used at run time to bind the variable to the field of the rule unit class with the same name. In the previous example, the **RuleUnitExecutor** inserts into the new rule unit the data source bound to the **"persons"** name and inserts the value **18** bound to the String **"adultAge"** into the fields with the corresponding names inside the **AdultUnit** class.

To override this default variable-binding behavior, use the **@UnitVar** annotation to explicitly define a logical binding name for each field of the rule unit class. For example, the field bindings in the following class are redefined with alternative names:

Example code to modify variable binding names with **@UnitVar**

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
   @UnitVar("minAge")
   private int adultAge = 18;
```

```
    @UnitVar("data")
    private DataSource<Person> persons;
}
```

You can then bind the variables to the executor using those alternative names and run the rule unit:

Example rule unit execution with modified variable names

```
executor.bindVariable( "data", persons );
      .bindVariable( "minAge", 18 );
executor.run( AdultUnit.class );
```

You can execute a rule unit in *passive mode* by using the **run()** method (equivalent to invoking **fireAllRules()** on a KIE session) or in *active mode* using the **runUntilHalt()** method (equivalent to invoking **fireUntilHalt()** on a KIE session). By default, the decision engine runs in *passive mode* and evaluates rule units only when a user or an application explicitly calls **run()** (or **fireAllRules()** for standard rules). If a user or application calls **runUntilHalt()** for rule units (or **fireUntilHalt()** for standard rules), the decision engine starts in *active mode* and evaluates rule units continually until the user or application explicitly calls **halt()**.

If you use the **runUntilHalt()** method, invoke the method on a separate execution thread to avoid blocking the main thread:

Example rule unit execution with runUntilHalt() on a separate thread

```
new Thread( () -> executor.runUntilHalt( adultUnit ) ).start();
```

2.12.1. Data sources for rule units

A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. A rule unit can have zero or more data sources and each **DataSource** definition declared inside a rule unit can correspond to a different entry point into the rule unit executor. Multiple rule units can share a single data source, but each rule unit must use different entry points through which the same objects are inserted.

You can create a **DataSource** definition with a fixed set of data in a rule unit class, as shown in the following example:

Example data source definition

```
DataSource<Person> persons = DataSource.create( new Person( "John", 42 ),
                              new Person( "Jane", 44 ),
                              new Person( "Sally", 4 ) );
```

Because a data source represents the entry point of the rule unit, you can insert, update, or delete facts in a rule unit:

Example code to insert, modify, and delete a fact in a rule unit

```
// Insert a fact:
Person john = new Person( "John", 42 );
FactHandle johnFh = persons.insert( john );

// Modify the fact and optionally specify modified properties (for property reactivity):
```

```
john.setAge( 43 );
persons.update( johnFh, john, "age" );

// Delete the fact:
persons.delete( johnFh );
```

2.12.2. Rule unit execution control

Rule units are helpful when you want to coordinate rule execution so that the execution of one rule unit triggers the start of another rule unit and so on.

To facilitate rule unit execution control, the decision engine supports the following rule unit methods that you can use in DRL rule actions to coordinate the execution of rule units:

- **drools.run()**: Triggers the execution of a specified rule unit class. This method imperatively interrupts the execution of the rule unit and activates the other specified rule unit.

- **drools.guard()**: Prevents (guards) a specified rule unit class from being executed until the associated rule condition is met. This method declaratively schedules the execution of the other specified rule unit. When the decision engine produces at least one match for the condition in the guarding rule, the guarded rule unit is considered active. A rule unit can contain multiple guarding rules.

As an example of the **drools.run()** method, consider the following DRL rules that each belong to a specified rule unit. The **NotAdult** rule uses the **drools.run(AdultUnit.class)** method to trigger the execution of the **AdultUnit** rule unit:

Example DRL rules with controlled execution using drools.run()

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    Person(age >= 18, $name : name) from persons
  then
    System.out.println($name + " is adult");
end
```

```
package org.mypackage.myunit
unit NotAdultUnit

rule NotAdult
  when
    $p : Person(age < 18, $name : name) from persons
  then
    System.out.println($name + " is NOT adult");
    modify($p) { setAge(18); }
    drools.run( AdultUnit.class );
end
```

The example also uses a **RuleUnitExecutor** class created from the KIE base that was built from these rules and a **DataSource** definition of **persons** bound to it:

Example rule executor and data source definitions

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Person> persons = executor.newDataSource( "persons",
                              new Person( "John", 42 ),
                              new Person( "Jane", 44 ),
                              new Person( "Sally", 4 ) );
```

In this case, the example creates the **DataSource** definition directly from the **RuleUnitExecutor** class and binds it to the **"persons"** variable in a single statement.

The example execution code produces the following output when the relevant **Person** facts are inserted in the **persons** data source:

Example rule unit execution output

```
Sally is NOT adult
John is adult
Jane is adult
Sally is adult
```

The **NotAdult** rule detects a match when evaluating the person **"Sally"**, who is under 18 years old. The rule then modifies her age to **18** and uses the **drools.run(AdultUnit.class)** method to trigger the execution of the **AdultUnit** rule unit. The **AdultUnit** rule unit contains a rule that can now be executed for all of the 3 **persons** in the **DataSource** definition.

As an example of the **drools.guard()** method, consider the following **BoxOffice** class and **BoxOfficeUnit** rule unit class:

Example BoxOffice class

```
public class BoxOffice {
    private boolean open;

    public BoxOffice( boolean open ) {
        this.open = open;
    }

    public boolean isOpen() {
        return open;
    }

    public void setOpen( boolean open ) {
        this.open = open;
    }
}
```

Example BoxOfficeUnit rule unit class

```
public class BoxOfficeUnit implements RuleUnit {
    private DataSource<BoxOffice> boxOffices;

    public DataSource<BoxOffice> getBoxOffices() {
        return boxOffices;
    }
}
```

The example also uses the following **TicketIssuerUnit** rule unit class to keep selling box office tickets for the event as long as at least one box office is open. This rule unit uses **DataSource** definitions of **persons** and **tickets**:

Example **TicketIssuerUnit** rule unit class

```
public class TicketIssuerUnit implements RuleUnit {
    private DataSource<Person> persons;
    private DataSource<AdultTicket> tickets;

    private List<String> results;

    public TicketIssuerUnit() { }

    public TicketIssuerUnit( DataSource<Person> persons, DataSource<AdultTicket> tickets ) {
        this.persons = persons;
        this.tickets = tickets;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public DataSource<AdultTicket> getTickets() {
        return tickets;
    }

    public List<String> getResults() {
        return results;
    }
}
```

The **BoxOfficeUnit** rule unit contains a **BoxOfficeIsOpen** DRL rule that uses the **drools.guard(TicketIssuerUnit.class)** method to guard the execution of the **TicketIssuerUnit** rule unit that distributes the event tickets, as shown in the following DRL rule examples:

Example DRL rules with controlled execution using **drools.guard()**

```
package org.mypackage.myunit;
unit TicketIssuerUnit;

rule IssueAdultTicket when
    $p: /persons[ age >= 18 ]
then
    tickets.insert(new AdultTicket($p));
end
rule RegisterAdultTicket when
    $t: /tickets
then
    results.add( $t.getPerson().getName() );
end
```

```
package org.mypackage.myunit;
unit BoxOfficeUnit;
```

```
rule BoxOfficeIsOpen
  when
    $box: /boxOffices[ open ]
  then
    drools.guard( TicketIssuerUnit.class );
end
```

In this example, so long as at least one box office is **open**, the guarded **TicketIssuerUnit** rule unit is active and distributes event tickets. When no more box offices are in **open** state, the guarded **TicketIssuerUnit** rule unit is prevented from being executed.

The following example class illustrates a more complete box office scenario:

Example class for the box office scenario

```
DataSource<Person> persons = executor.newDataSource( "persons" );
DataSource<BoxOffice> boxOffices = executor.newDataSource( "boxOffices" );
DataSource<AdultTicket> tickets = executor.newDataSource( "tickets" );

List<String> list = new ArrayList<>();
executor.bindVariable( "results", list );

// Two box offices are open:
BoxOffice office1 = new BoxOffice(true);
FactHandle officeFH1 = boxOffices.insert( office1 );
BoxOffice office2 = new BoxOffice(true);
FactHandle officeFH2 = boxOffices.insert( office2 );

persons.insert(new Person("John", 40));

// Execute `BoxOfficeIsOpen` rule, run `TicketIssuerUnit` rule unit, and execute `RegisterAdultTicket`
// rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "John", list.get(0) );
list.clear();

persons.insert(new Person("Matteo", 30));

// Execute `RegisterAdultTicket` rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "Matteo", list.get(0) );
list.clear();

// One box office is closed, the other is open:
office1.setOpen(false);
boxOffices.update(officeFH1, office1);
persons.insert(new Person("Mark", 35));
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "Mark", list.get(0) );
```

```
    list.clear();

    // All box offices are closed:
    office2.setOpen(false);
    boxOffices.update(officeFH2, office2); // Guarding rule is no longer true.
    persons.insert(new Person("Edson", 35));
    executor.run(BoxOfficeUnit.class); // No execution

    assertEquals( 0, list.size() );
```

2.12.3. Rule unit identity conflicts

In rule unit execution scenarios with guarded rule units, a rule can guard multiple rule units and at the same time a rule unit can be guarded and then activated by multiple rules. For these two-way guarding scenarios, rule units must have a clearly defined identity to avoid identity conflicts.

By default, the identity of a rule unit is the rule unit class name and is treated as a singleton class by the **RuleUnitExecutor**. This identification behavior is encoded in the **getUnitIdentity()** default method of the **RuleUnit** interface:

Default identity method in the RuleUnit interface

```
default Identity getUnitIdentity() {
    return new Identity( getClass() );
}
```

In some cases, you may need to override this default identification behavior to avoid conflicting identities between rule units.

For example, the following **RuleUnit** class contains a **DataSource** definition that accepts any kind of object:

Example Unit0 rule unit class

```
public class Unit0 implements RuleUnit {
    private DataSource<Object> input;

    public DataSource<Object> getInput() {
        return input;
    }
}
```

This rule unit contains the following DRL rule that guards another rule unit based on two conditions (in OOPath notation):

Example GuardAgeCheck DRL rule in the rule unit

```
package org.mypackage.myunit
unit Unit0

rule GuardAgeCheck
  when
    $i: /input#Integer
    $s: /input#String
```

```
  then
    drools.guard( new AgeCheckUnit($i) );
    drools.guard( new AgeCheckUnit($s.length()) );
  end
```

The guarded **AgeCheckUnit** rule unit verifies the age of a set of **persons**. The **AgeCheckUnit** contains a **DataSource** definition of the **persons** to check, a **minAge** variable that it verifies against, and a **List** for gathering the results:

Example **AgeCheckUnit** rule unit

```
public class AgeCheckUnit implements RuleUnit {
    private final int minAge;
    private DataSource<Person> persons;
    private List<String> results;

    public AgeCheckUnit( int minAge ) {
        this.minAge = minAge;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public int getMinAge() {
        return minAge;
    }

    public List<String> getResults() {
        return results;
    }
}
```

The **AgeCheckUnit** rule unit contains the following DRL rule that performs the verification of the **persons** in the data source:

Example **CheckAge** DRL rule in the rule unit

```
package org.mypackage.myunit
unit AgeCheckUnit

rule CheckAge
  when
    $p : /persons{ age > minAge }
  then
    results.add($p.getName() + ">" + minAge);
end
```

This example creates a **RuleUnitExecutor** class, binds the class to the KIE base that contains these two rule units, and creates the two **DataSource** definitions for the same rule units:

Example executor and data source definitions

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
```

```
DataSource<Object> input = executor.newDataSource( "input" );
DataSource<Person> persons = executor.newDataSource( "persons",
                            new Person( "John", 42 ),
                            new Person( "Sally", 4 ) );

List<String> results = new ArrayList<>();
executor.bindVariable( "results", results );
```

You can now insert some objects into the input data source and execute the **Unit0** rule unit:

Example rule unit execution with inserted objects

```
ds.insert("test");
ds.insert(3);
ds.insert(4);
executor.run(Unit0.class);
```

Example results list from the execution

```
[Sally>3, John>3]
```

In this example, the rule unit named **AgeCheckUnit** is considered a singleton class and then executed only once, with the **minAge** variable set to **3**. Both the String **"test"** and the Integer **4** inserted into the input data source can also trigger a second execution with the **minAge** variable set to **4**. However, the second execution does not occur because another rule unit with the same identity has already been evaluated.

To resolve this rule unit identity conflict, override the **getUnitIdentity()** method in the **AgeCheckUnit** class to include also the **minAge** variable in the rule unit identity:

Modified AgeCheckUnit rule unit to override the getUnitIdentity() method

```
public class AgeCheckUnit implements RuleUnit {

    ...

    @Override
    public Identity getUnitIdentity() {
        return new Identity(getClass(), minAge);
    }
}
```

With this override in place, the previous example rule unit execution produces the following output:

Example results list from executing the modified rule unit

```
[John>4, Sally>3, John>3]
```

The rule units with **minAge** set to **3** and **4** are now considered two different rule units and both are executed.

CHAPTER 3. DATA OBJECTS

Data objects are the building blocks for the rule assets that you create. Data objects are custom data types implemented as Java objects in specified packages of your project. For example, you might create a **Person** object with data fields **Name**, **Address**, and **DateOfBirth** to specify personal details for loan application rules. These custom data types determine what data your assets and your decision services are based on.

3.1. CREATING DATA OBJECTS

The following procedure is a generic overview of creating data objects. It is not specific to a particular business asset.

Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.

2. Click **Add Asset → Data Object**.

3. Enter a unique **Data Object** name and select the **Package** where you want the data object to be available for other rule assets. Data objects with the same name cannot exist in the same package. In the specified DRL file, you can import a data object from any package.
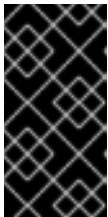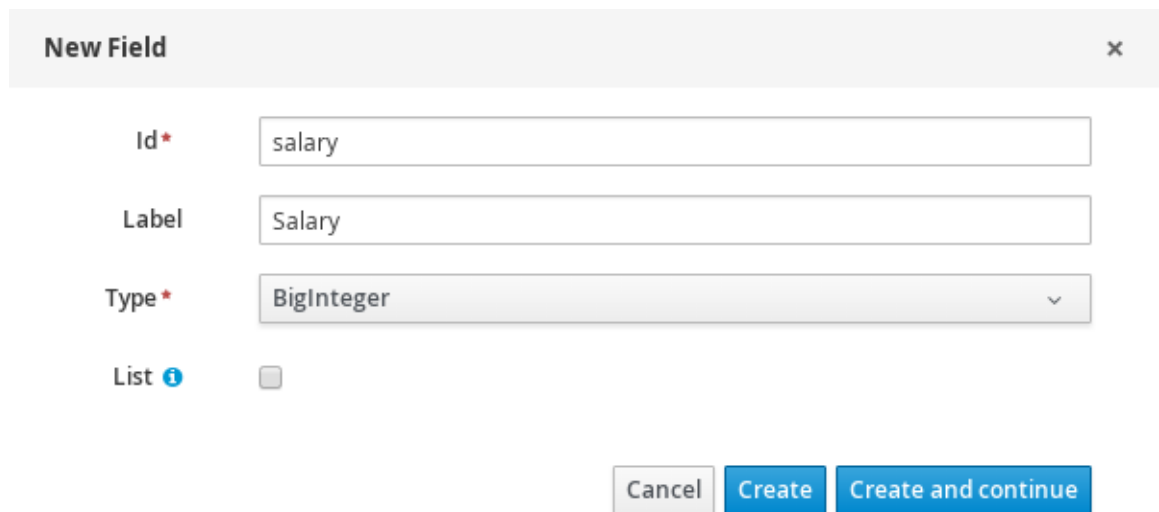

   IMPORTING DATA OBJECTS FROM OTHER PACKAGES

   You can import an existing data object from another package directly into the asset designers like guided rules or guided decision table designers. Select the relevant rule asset within the project and in the asset designer, go to **Data Objects → New item** to select the object to be imported.

4. To make your data object persistable, select the **Persistable** checkbox. Persistable data objects are able to be stored in a database according to the JPA specification. The default JPA is Hibernate.

5. Click **Ok**.

6. In the data object designer, click **add field** to add a field to the object with the attributes **Id**, **Label**, and **Type**. Required attributes are marked with an asterisk (*).

   - **Id:** Enter the unique ID of the field.

   - **Label:** (Optional) Enter a label for the field.

   - **Type:** Enter the data type of the field.

   - **List:** (Optional) Select this check box to enable the field to hold multiple items for the specified type.

Figure 3.1. Add data fields to a data object

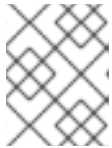7. Click **Create** to add the new field, or click **Create and continue** to add the new field and continue adding other fields.

> **NOTE**
>
> To edit a field, select the field row and use the **general properties** on the right side of the screen.

CHAPTER 4. CREATING DRL RULES IN BUSINESS CENTRAL

You can create and manage DRL rules for your project in Business Central. In each DRL rule file, you define rule conditions, actions, and other components related to the rule, based on the data objects you create or import in the package.

Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.

2. Click **Add Asset → DRL file**.

3. Enter an informative **DRL file** name and select the appropriate **Package**. The package that you specify must be the same package where the required data objects have been assigned or will be assigned.
   You can also select **Show declared DSL sentences** if any domain specific language (DSL) assets have been defined in your project. These DSL assets will then become usable objects for conditions and actions that you define in the DRL designer.

4. Click **Ok** to create the rule asset.
   The new DRL file is now listed in the **DRL** panel of the **Project Explorer**, or in the **DSLR** panel if you selected the **Show declared DSL sentences** option. The package to which you assigned this DRL file is listed at the top of the file.

5. In the **Fact types** list in the left panel of the DRL designer, confirm that all data objects and data object fields (expand each) required for your rules are listed. If not, you can either import relevant data objects from other packages by using **import** statements in the DRL file, or create data objects within your package.

6. After all data objects are in place, return to the **Model** tab of the DRL designer and define the DRL file with any of the following components:

Components in a DRL file

```
package

import

function  // Optional

query  // Optional

declare  // Optional

global  // Optional

rule "rule name"
   // Attributes
   when
      // Conditions
   then
      // Actions
end
```

```
rule "rule2 name"

...
```

- **package**: (automatic) This was defined for you when you created the DRL file and selected the package.

- **import**: Use this to identify the data objects from either this package or another package that you want to use in the DRL file. Specify the package and data object in the format **packageName.objectName**, with multiple imports on separate lines.

  Importing data objects

  ```
  import org.mortgages.LoanApplication;
  ```

- **function**: (optional) Use this to include a function to be used by rules in the DRL file. Functions in DRL files put semantic code in your rule source file instead of in Java classes. Functions are especially useful if an action (**then**) part of a rule is used repeatedly and only the parameters differ for each rule. Above the rules in the DRL file, you can declare the function or import a static method from a helper class as a function, and then use the function by name in an action (**then**) part of the rule.

  Declaring and using a function with a rule (option 1)

  ```
  function String hello(String applicantName) {
      return "Hello " + applicantName + "!";
  }

  rule "Using a function"
    when
      // Empty
    then
      System.out.println( hello( "James" ) );
  end
  ```

  Importing and using the function with a rule (option 2)

  ```
  import function my.package.applicant.hello;

  rule "Using a function"
    when
      // Empty
    then
      System.out.println( hello( "James" ) );
  end
  ```

- **query**: (optional) Use this to search the decision engine for facts related to the rules in the DRL file. You add the query definitions in DRL files and then obtain the matching results in your application code. Queries search for a set of defined conditions and do not require **when** or **then** specifications. Query names are global to the KIE base and therefore must be unique among all other rule queries in the project. To return the results of a query, construct a traditional **QueryResults** definition using **ksession.getQueryResults("name")**, where

"name" is the query name. This returns a list of query results, which enable you to retrieve the objects that matched the query. Define the query and query results parameters above the rules in the DRL file.

Example query definition in a DRL file

```
query "people under the age of 21"
  $person : Person( age < 21 )
end
```

Example application code to obtain query results

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age  of 21" );
```

- **declare**: (optional) Use this to declare a new fact type to be used by rules in the DRL file. The default fact type in the **java.lang** package of Red Hat Decision Manager is **Object**, but you can declare other types in DRL files as needed. Declaring fact types in DRL files enables you to define a new fact model directly in the decision engine, without creating models in a lower-level language like Java.

Declaring and using a new fact type

```
declare Person
  name : String
  dateOfBirth : java.util.Date
  address : Address
end

rule "Using a declared type"
  when
    $p : Person( name == "James" )
  then   // Insert Mark, who is a customer of James.
    Person mark = new Person();
    mark.setName( "Mark" );
    insert( mark );
end
```

- **global**: (optional) Use this to include a global variable to be used by rules in the DRL file. Global variables typically provide data or services for the rules, such as application services used in rule consequences, and return data from rules, such as logs or values added in rule consequences. Set the global value in the working memory of the decision engine through a KIE session configuration or REST operation, declare the global variable above the rules in the DRL file, and then use it in an action (**then**) part of the rule. For multiple global variables, use separate lines in the DRL file.

Setting the global list configuration for the decision engine

```
List<String> list = new ArrayList<>();
KieSession kieSession = kiebase.newKieSession();
kieSession.setGlobal( "myGlobalList", list );
```

Defining the global list in a rule

```
global java.util.List myGlobalList;

rule "Using a global"
  when
    // Empty
  then
    myGlobalList.add( "My global list" );
end
```

> **WARNING**
>
> Do not use global variables to establish conditions in rules unless a global variable has a constant immutable value. Global variables are not inserted into the working memory of the decision engine, so the decision engine cannot track value changes of variables.
>
> Do not use global variables to share data between rules. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory of the decision engine.

- **rule**: Use this to define each rule in the DRL file. Rules consist of a rule name in the format **rule "name"**, followed by optional attributes that define rule behavior (such as **salience** or **no-loop**), followed by **when** and **then** definitions. Each rule must have a unique name within the rule package. The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition for an **"Underage"** rule would be **Applicant(age < 21)**. The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when the loan applicant is under 21 years old, the **then** action would be **setApproved(false)**, declining the loan because the applicant is under age.

Rule for loan application age limit

```
rule "Underage"
  salience 15
  when
    $application : LoanApplication()
    Applicant( age < 21 )
  then
    $application.setApproved( false );
    $application.setExplanation( "Underage" );
end
```

At a minimum, each DRL file must specify the **package**, **import**, and **rule** components. All other components are optional.

The following is an example DRL file in a loan application decision service:

Example DRL file for a loan application

```
package org.mortgages;

import org.mortgages.LoanApplication;
import org.mortgages.Bankruptcy;
import org.mortgages.Applicant;

rule "Bankruptcy history"
 salience 10
 when
  $a : LoanApplication()
  exists (Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000 ))
 then
  $a.setApproved( false );
  $a.setExplanation( "has been bankrupt" );
  delete( $a );
end

rule "Underage"
 salience 15
 when
  $application : LoanApplication()
  Applicant( age < 21 )
 then
  $application.setApproved( false );
  $application.setExplanation( "Underage" );
  delete( $application );
end
```

Figure 4.1. Example DRL file for a loan application in Business Central
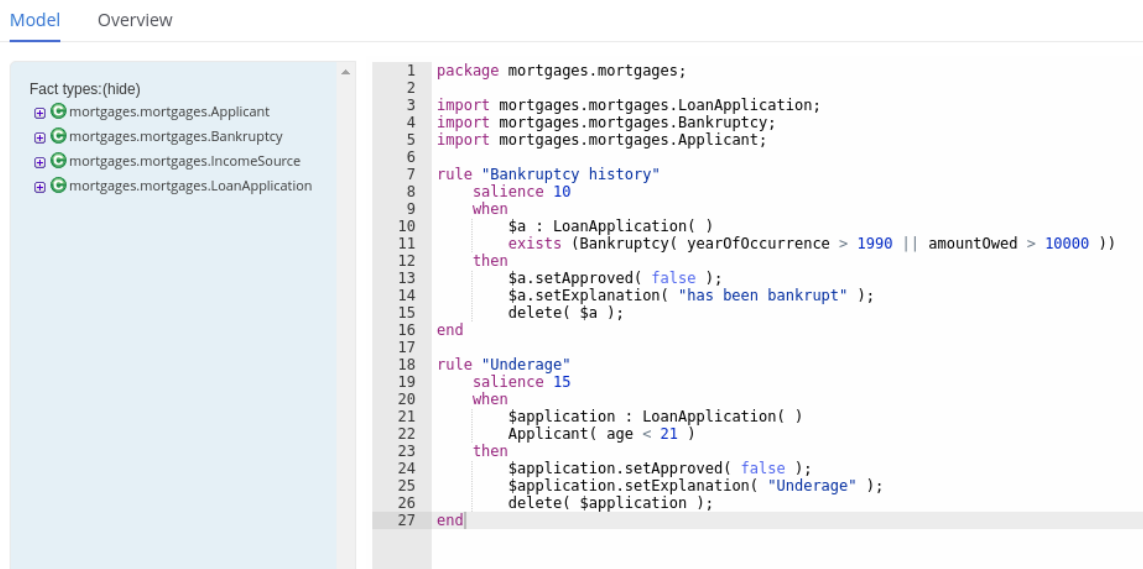
7. After you define all components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

8. Click **Save** in the DRL designer to save your work.

4.1. ADDING WHEN CONDITIONS IN DRL RULES

The **when** part of the rule contains the conditions that must be met to execute an action. For example, if a bank requires loan applicants to have over 21 years of age, then the **when** condition of an **"Underage"** rule would be **Applicant(age < 21)**. Conditions consist of a series of stated patterns and constraints, with optional bindings and other supported DRL elements, based on the available data objects in the package.

Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.

- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Business Central.

- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

Procedure

1. In the DRL designer, enter **when** within the rule to begin adding condition statements. The **when** section consists of zero or more fact patterns that define conditions for the rule.
   If the **when** section is empty, then the conditions are considered to be true and the actions in the **then** section are executed the first time a **fireAllRules()** call is made in the decision engine. This is useful if you want to use rules to set up the decision engine state.

   Example rule without conditions

   ```
   rule "Always insert applicant"
     when
       // Empty
     then   // Actions to be executed once
       insert( new Applicant() );
   end

   // The rule is internally rewritten in the following way:

   rule "Always insert applicant"
     when
       eval( true )
     then
       insert( new Applicant() );
   end
   ```

2. Enter a pattern for the first condition to be met, with optional constraints, bindings, and other supported DRL elements. A basic pattern format is **<patternBinding> : <patternType> (<constraints>)**. Patterns are based on the available data objects in the package and define the conditions to be met in order to trigger actions in the **then** section.

   - **Simple pattern:** A simple pattern with no constraints matches against a fact of the given type. For example, the following condition is only that the applicant exists.

     ```
     when
       Applicant()
     ```

- **Pattern with constraints:** A pattern with constraints matches against a fact of the given type and the additional restrictions in parentheses that are true or false. For example, the following condition is that the applicant is under the age of 21.

```
when
  Applicant( age < 21 )
```

- **Pattern with binding:** A binding on a pattern is a shorthand reference that other components of the rule can use to refer back to the defined pattern. For example, the following binding **a** on **LoanApplication** is used in a related action for underage applicants.

```
when
  $a : LoanApplication()
  Applicant( age < 21 )
then
  $a.setApproved( false );
  $a.setExplanation( "Underage" )
```

3. Continue defining all condition patterns that apply to this rule. The following are some of the keyword options for defining DRL conditions:

- **and**: Use this to group conditional components into a logical conjunction. Infix and prefix **and** are supported. By default, all listed patterns are combined with **and** when no conjunction is specified.

```
// All of the following examples are interpreted the same way:
$a : LoanApplication() and Applicant( age < 21 )

$a : LoanApplication()
and Applicant( age < 21 )

$a : LoanApplication()
Applicant( age < 21 )

(and $a : LoanApplication() Applicant( age < 21 ))
```

- **or**: Use this to group conditional components into a logical disjunction. Infix and prefix **or** are supported.

```
// All of the following examples are interpreted the same way:
Bankruptcy( amountOwed == 100000 ) or IncomeSource( amount == 20000 )

Bankruptcy( amountOwed == 100000 )
or IncomeSource( amount == 20000 )

(or Bankruptcy( amountOwed == 100000 ) IncomeSource( amount == 20000 ))
```

- **exists**: Use this to specify facts and constraints that must exist. This option is triggered on only the first match, not subsequent matches. If you use this element with multiple patterns, enclose the patterns with parentheses **()**.

```
exists ( Bankruptcy( yearOfOccurrence > 1990 || amountOwed > 10000 ) )
```

- **not**: Use this to specify facts and constraints that must not exist.

> not (Applicant(age < 21))

- **forall**: Use this to verify whether all facts that match the first pattern match all the remaining patterns. When a **forall** construct is satisfied, the rule evaluates to **true**.

> forall($app : Applicant(age < 21)
>   Applicant(this == $app, status = 'underage'))

- **from**: Use this to specify a data source for a pattern.

> Applicant(ApplicantAddress : address)
> Address(zipcode == "23920W") from ApplicantAddress

- **entry-point**: Use this to define an **Entry Point** corresponding to a data source for the pattern. Typically used with **from**.

> Applicant() from entry-point "LoanApplication"

- **collect**: Use this to define a collection of objects that the rule can use as part of the condition. In the example, all pending applications in the decision engine for each given mortgage are grouped in a **List**. If three or more pending applications are found, the rule is executed.

> $m : Mortgage()
> $a : List(size >= 3)
>   from collect(LoanApplication(Mortgage == $m, status == 'pending'))

- **accumulate**: Use this to iterate over a collection of objects, execute custom actions for each of the elements, and return one or more result objects (if the constraints evaluate to **true**). This option is a more flexible and powerful form of **collect**. Use the format **accumulate(<source pattern>; <functions> [;<constraints>])**. In the example, **min**, **max**, and **average** are accumulate functions that calculate the minimum, maximum, and average temperature values over all the readings for each sensor. Other supported functions include **count**, **sum**, **variance**, **standardDeviation**, **collectList**, and **collectSet**.

> $s : Sensor()
> accumulate(Reading(sensor == $s, $temp : temperature);
> $min : min($temp),
> $max : max($temp),
> $avg : average($temp);
> $min < 20, $avg > 70)

> **NOTE**
>
> For more information about DRL rule conditions, see Section 2.8, "Rule conditions in DRL (WHEN)".

4. After you define all condition components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

5. Click **Save** in the DRL designer to save your work.

4.2. ADDING THEN ACTIONS IN DRL RULES

The **then** part of the rule contains the actions to be performed when the conditional part of the rule has been met. For example, when a loan applicant is under 21 years old, the **then** action of an **"Underage"** rule would be **setApproved(false)**, declining the loan because the applicant is under age. Actions consist of one or more methods that execute consequences based on the rule conditions and on available data objects in the package. The main purpose of rule actions is to to insert, delete, or modify data in the working memory of the decision engine.

Prerequisites

- The **package** is defined at the top of the DRL file. This should have been done for you when you created the file.

- The **import** list of data objects used in the rule is defined below the **package** line of the DRL file. Data objects can be from this package or from another package in Business Central.

- The **rule** name is defined in the format **rule "name"** below the **package**, **import**, and other lines that apply to the entire DRL file. The same rule name cannot be used more than once in the same package. Optional rule attributes (such as **salience** or **no-loop**) that define rule behavior are below the rule name, before the **when** section.

Procedure

1. In the DRL designer, enter **then** after the **when** section of the rule to begin adding action statements.

2. Enter one or more actions to be executed on fact patterns based on the conditions for the rule. The following are some of the keyword options for defining DRL actions:

   - **set**: Use this to set the value of a field.

     ```
     $application.setApproved ( false );
     $application.setExplanation( "has been bankrupt" );
     ```

   - **modify**: Use this to specify fields to be modified for a fact and to notify the decision engine of the change. This method provides a structured approach to fact updates. It combines the **update** operation with setter calls to change object fields.

     ```
     modify( LoanApplication ) {
         setAmount( 100 ),
         setApproved ( true )
     }
     ```

   - **update**: Use this to specify fields and the entire related fact to be updated and to notify the decision engine of the change. After a fact has changed, you must call **update** before changing another fact that might be affected by the updated values. To avoid this added step, use the **modify** method instead.

     ```
     LoanApplication.setAmount( 100 );
     update( LoanApplication );
     ```

   - **insert**: Use this to insert a **new** fact into the decision engine.

```
insert( new Applicant() );
```

- **insertLogical**: Use this to insert a **new** fact logically into the decision engine. The decision engine is responsible for logical decisions on insertions and retractions of facts. After regular or stated insertions, facts must be retracted explicitly. After logical insertions, the facts that were inserted are automatically retracted when the conditions in the rules that inserted the facts are no longer true.

```
insertLogical( new Applicant() );
```

- **delete**: Use this to remove an object from the decision engine. The keyword **retract** is also supported in DRL and executes the same action, but **delete** is typically preferred in DRL code for consistency with the keyword **insert**.

```
delete( Applicant );
```
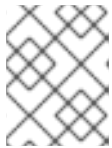
> **NOTE**
>
> For more information about DRL rule actions, see Section 2.9, "Rule actions in DRL (THEN)".

3. After you define all action components of the rule, click **Validate** in the upper-right toolbar of the DRL designer to validate the DRL file. If the file validation fails, address any problems described in the error message, review all syntax and components in the DRL file, and try again to validate the file until the file passes.

4. Click **Save** in the DRL designer to save your work.

CHAPTER 5. EXECUTING RULES

After you identify example rules or create your own rules in Business Central, you can build and deploy the associated project and execute rules locally or on Decision Server to test the rules.

Prerequisites

- Business Central and Decision Server are installed and running. For installation options, see *Planning a Red Hat Decision Manager installation* .

Procedure

1. In Business Central, go to **Menu → Design → Projects** and click the project name.

2. In the upper-right corner of the project **Assets** page, click **Deploy** to build the project and deploy it to Decision Server. If the build fails, address any problems described in the **Alerts** panel at the bottom of the screen.
   For more information about project deployment options, see *Packaging and deploying a Red Hat Decision Manager project*.

3. Create a Maven or Java project outside of Business Central, if not created already, that you can use for executing rules locally or that you can use as a client application for executing rules on Decision Server. The project must contain a **pom.xml** file and any other required components for executing the project resources.
   For example test projects, see "Other methods for creating and executing DRL rules".

4. Open the **pom.xml** file of your test project or client application and add the following dependencies, if not added already:

   - **kie-ci**: Enables your client application to load Business Central project data locally using **ReleaseId**

   - **kie-server-client**: Enables your client application to interact remotely with assets on Decision Server

   - **slf4j**: (Optional) Enables your client application to use Simple Logging Facade for Java (SLF4J) to return debug logging information after you interact with Decision Server

   Example dependencies for Red Hat Decision Manager 7.6 in a client application **pom.xml** file:

   ```
   <!-- For local execution -->
   <dependency>
     <groupId>org.kie</groupId>
     <artifactId>kie-ci</artifactId>
     <version>7.30.0.Final-redhat-00003</version>
   </dependency>

   <!-- For remote execution on Decision Server -->
   <dependency>
     <groupId>org.kie.server</groupId>
     <artifactId>kie-server-client</artifactId>
     <version>7.30.0.Final-redhat-00003</version>
   </dependency>

   <!-- For debug logging (optional) -->
   <dependency>
   ```

```
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.7.25</version>
  </dependency>
```

For available versions of these artifacts, search the group ID and artifact ID in the Nexus Repository Manager online.

> **NOTE**
>
> Instead of specifying a Red Hat Decision Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.
>
> Example BOM dependency:
>
> ```
> <dependency>
>   <groupId>com.redhat.ba</groupId>
>   <artifactId>ba-platform-bom</artifactId>
>   <version>7.6.0.redhat-00004</version>
>   <scope>import</scope>
>   <type>pom</type>
> </dependency>
> ```
>
> For more information about the Red Hat Business Automation BOM, see What is the mapping between Red Hat Decision Manager and the Maven library version?.

5. Ensure that the dependencies for artifacts containing model classes are defined in the client application **pom.xml** file exactly as they appear in the **pom.xml** file of the deployed project. If dependencies for model classes differ between the client application and your projects, execution errors can occur.
   To access the project **pom.xml** file in Business Central, select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

   For example, the following **Person** class dependency appears in both the client and deployed project **pom.xml** files:

   ```
   <dependency>
     <groupId>com.sample</groupId>
     <artifactId>Person</artifactId>
     <version>1.0.0</version>
   </dependency>
   ```

6. If you added the **slf4j** dependency to the client application **pom.xml** file for debug logging, create a **simplelogger.properties** file on the relevant classpath (for example, in **src/main/resources/META-INF** in Maven) with the following content:

   ```
   org.slf4j.simpleLogger.defaultLogLevel=debug
   ```

7. In your client application, create a **.java** main class containing the necessary imports and a **main()** method to load the KIE base, insert facts, and execute the rules.

For example, a **Person** object in a project contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person. The following **Wage** rule in a project calculates the wage and hourly rate values and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

To test this rule locally outside of Decision Server (if needed), configure the **.java** class to import KIE services, a KIE container, and a KIE session, and then use the **main()** method to fire all rules against a defined fact model:

Executing rules locally

```
import org.kie.api.KieServices;
import org.kie.api.builder.ReleaseId;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.drools.compiler.kproject.ReleaseIdImpl;

public class RulesTest {

  public static final void main(String[] args) {
    try {
      // Identify the project in the local repository:
      ReleaseId rid = new ReleaseIdImpl("com.myspace", "MyProject", "1.0.0");

      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.newKieContainer(rid);
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);
```

```
    // Fire all rules:
    kSession.fireAllRules();
    kSession.dispose();
  }

  catch (Throwable t) {
    t.printStackTrace();
  }
 }
}
```

To test this rule on Decision Server, configure the **.java** class with the imports and rule execution information similarly to the local example, and additionally specify KIE services configuration and KIE services client details:

Executing rules on Decision Server

```java
package com.sample;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import org.kie.api.command.BatchExecutionCommand;
import org.kie.api.command.Command;
import org.kie.api.KieServices;
import org.kie.api.runtime.ExecutionResults;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;
import org.kie.server.api.marshalling.MarshallingFormat;
import org.kie.server.api.model.ServiceResponse;
import org.kie.server.client.KieServicesClient;
import org.kie.server.client.KieServicesConfiguration;
import org.kie.server.client.KieServicesFactory;
import org.kie.server.client.RuleServicesClient;

import com.sample.Person;

public class RulesTest {

  private static final String containerName = "testProject";
  private static final String sessionName = "myStatelessSession";

  public static final void main(String[] args) {
   try {
     // Define KIE services configuration and client:
     Set<Class<?>> allClasses = new HashSet<Class<?>>();
     allClasses.add(Person.class);
     String serverUrl = "http://$HOST:$PORT/kie-server/services/rest/server";
     String username = "$USERNAME";
     String password = "$PASSWORD";
     KieServicesConfiguration config =
       KieServicesFactory.newRestConfiguration(serverUrl,
                             username,
```

```
                        password);
      config.setMarshallingFormat(MarshallingFormat.JAXB);
      config.addExtraClasses(allClasses);
      KieServicesClient kieServicesClient =
        KieServicesFactory.newKieServicesClient(config);

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert Person into the session:
      KieCommands kieCommands = KieServices.Factory.get().getCommands();
      List<Command> commandList = new ArrayList<Command>();
      commandList.add(kieCommands.newInsert(p, "personReturnId"));

      // Fire all rules:
      commandList.add(kieCommands.newFireAllRules("numberOfFiredRules"));
      BatchExecutionCommand batch = kieCommands.newBatchExecution(commandList,
    sessionName);

      // Use rule services client to send request:
      RuleServicesClient ruleClient =
    kieServicesClient.getServicesClient(RuleServicesClient.class);
      ServiceResponse<ExecutionResults> executeResponse =
    ruleClient.executeCommandsWithResults(containerName, batch);
      System.out.println("number of fired rules:" +
    executeResponse.getResult().getValue("numberOfFiredRules"));
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
   }
 }
```

8. Run the configured **.java** class from your project directory. You can run the file in your development platform (such as Red Hat CodeReady Studio) or in the command line. Example Maven execution (within project directory):

```
mvn clean install exec:java -Dexec.mainClass="com.sample.app.RulesTest"
```

Example Java execution (within project directory)

```
javac -classpath "./$DEPENDENCIES/*:." RulesTest.java
java -classpath "./$DEPENDENCIES/*:." RulesTest
```

9. Review the rule execution status in the command line and in the server log. If any rules do not execute as expected, review the configured rules in the project and the main class configuration to validate the data provided.

CHAPTER 6. OTHER METHODS FOR CREATING AND EXECUTING DRL RULES

As an alternative to creating and managing DRL rules within the Business Central interface, you can create DRL rule files externally as part of a Maven or Java project using Red Hat CodeReady Studio or another integrated development environment (IDE). These standalone projects can then be integrated as knowledge JAR (KJAR) dependencies in existing Red Hat Decision Manager projects in Business Central. The DRL files in your standalone project must contain at a minimum the required **package** specification, **import** lists, and **rule** definitions. Any other DRL components, such as global variables and functions, are optional. All data objects related to a DRL rule must be included with your standalone DRL project or deployment.

You can also use executable rule models in your Maven or Java projects to provide a Java-based representation of a rule set for execution at build time. The executable model is a more efficient alternative to the standard asset packaging in Red Hat Decision Manager and enables KIE containers and KIE bases to be created more quickly, especially when you have large lists of DRL (Drools Rule Language) files and other Red Hat Decision Manager assets.

6.1. CREATING AND EXECUTING DRL RULES IN RED HAT CODEREADY STUDIO

You can use Red Hat CodeReady Studio to create DRL files with rules and integrate the files with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use Red Hat CodeReady Studio for your decision service and want to continue with the same workflow. If you do not already use this method, then the Business Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Prerequisites

- Red Hat CodeReady Studio has been installed from the Red Hat Customer Portal .

Procedure

1. In the Red Hat CodeReady Studio, click **File → New → Project**.

2. In the **New Project** window that opens, select **Drools → Drools Project** and click **Next**.

3. Click the second icon to **Create a project and populate it with some example files to help you get started quickly**. Click **Next**.

4. Enter a **Project name** and select the **Maven** radio button as the project building option. The GAV values are generated automatically. You can update these values as needed for your project:

   - Group ID: **com.sample**

   - Artifact ID: **my-project**

   - Version: **1.0.0-SNAPSHOT**

5. Click **Finish** to create the project.
   This configuration sets up a basic project structure, class path, and sample rules. The following is an overview of the project structure:

```
my-project
 `-- src/main/java
   | `-- com.sample
   |    `-- DecisionTableTest.java
   |    `-- DroolsTest.java
   |    `-- ProcessTest.java
   |
 `-- src/main/resources
   | `-- dtables
   |    `-- Sample.xls
   | `-- process
   |    `-- sample.bpmn
   | `-- rules
   |    `-- Sample.drl
   | `-- META-INF
   |
 `-- JRE System Library
   |
 `-- Maven Dependencies
   |
 `-- Drools Library
   |
 `-- src
   |
 `-- target
   |
 `-- pom.xml
```

Notice the following elements:

- A **Sample.drl** rule file in the **src/main/resources** directory, containing an example **Hello World** and **GoodBye** rules.

- A **DroolsTest.java** file under the **src/main/java** directory in the **com.sample** package. The **DroolsTest** class can be used to execute the **Sample.drl** rule.

- The **Drools Library** directory, which acts as a custom class path containing JAR files necessary for execution.

You can edit the existing **Sample.drl** file and **DroolsTest.java** files with new configurations as needed, or create new rule and object files. In this procedure, you are creating a new rule and new Java objects.

6. Create a Java object on which the rule or rules will operate.
   In this example, a **Person.java** file is created in **my-project/src/main/java/com.sample**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

   ```java
   public class Person {
     private String firstName;
     private String lastName;
     private Integer hourlyRate;
     private Integer wage;

     public String getFirstName() {
       return firstName;
   ```

```java
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public Integer getHourlyRate() {
    return hourlyRate;
  }

  public void setHourlyRate(Integer hourlyRate) {
    this.hourlyRate = hourlyRate;
  }

  public Integer getWage(){
    return wage;
  }

  public void setWage(Integer wage){
    this.wage = wage;
  }
}
```

7. Click **File → Save** to save the file.

8. Create a rule file in **.drl** format in **my-project/src/main/resources/rules**. The DRL file must contain at a minimum a package specification, an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.
   The following **Wage.drl** file contains a **Wage** rule that imports the **Person** class, calculates the wage and hourly rate values, and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

9. Click **File → Save** to save the file.

10. Create a main class and save it to the same directory as the Java object that you created. The main class will load the KIE base and execute rules.

> **NOTE**
>
> You can also add the **main()** method and **Person** class within a single Java object file, similar to the **DroolsTest.java** sample file.

11. In the main class, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **main()** method that passes the fact model to the rule.
    In this example, a **RulesTest.java** file is created in **my-project/src/main/java/com.sample** with the required imports and **main()** method:

```java
package com.sample;

import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {
  public static final void main(String[] args) {
    try {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      KieSession kSession = kContainer.newKieSession();

      // Set up the fact model:
      Person p = new Person();
      p.setWage(12);
      p.setFirstName("Tom");
      p.setLastName("Summers");
      p.setHourlyRate(10);

      // Insert the person into the session:
      kSession.insert(p);

      // Fire all rules:
      kSession.fireAllRules();
      kSession.dispose();
    }

    catch (Throwable t) {
      t.printStackTrace();
    }
  }
}
```

12. Click **File → Save** to save the file.

13. After you create and save all DRL assets in your project, right-click your project folder and select **Run As → Java Application** to build the project. If the project build fails, address any problems described in the **Problems** tab of the lower window in CodeReady Studio, and try again to validate the project until the project builds.

IF THE RUN AS → JAVA APPLICATION OPTION IS NOT AVAILABLE

If **Java Application** is not an option when you right-click your project and select **Run As**, then go to **Run As → Run Configurations**, right-click **Java Application**, and click **New**. Then in the **Main** tab, browse for and select your **Project** and the associated **Main class**. Click **Apply** and then click **Run** to test the project. The next time you right-click your project folder, the **Java Application** option will appear.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central. To access the project **pom.xml** file in Business Central, you can select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

6.2. CREATING AND EXECUTING DRL RULES USING JAVA

You can use Java objects to create DRL files with rules and integrate the objects with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use external Java objects for your decision service and want to continue with the same workflow. If you do not already use this method, then the Business Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Procedure

1. Create a Java object on which the rule or rules will operate.
   In this example, a **Person.java** file is created in a directory **my-project**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```java
public class Person {
  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public Integer getHourlyRate() {
    return hourlyRate;
  }
```

```
    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

2. Create a rule file in **.drl** format under the **my-project** directory. The DRL file must contain at a minimum a package specification (if applicable), an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.
   The following **Wage.drl** file contains a **Wage** rule that calculates the wage and hourly rate values and displays a message based on the result:

```
package com.sample;

import com.sample.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello" + " " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

3. Create a main class and save it to the same directory as the Java object that you created. The main class will load the KIE base and execute rules.

4. In the main class, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **main()** method that passes the fact model to the rule.
   In this example, a **RulesTest.java** file is created in **my-project** with the required imports and **main()** method:

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class RulesTest {
  public static final void main(String[] args) {
    try {
      // Load the KIE base:
      KieServices ks = KieServices.Factory.get();
      KieContainer kContainer = ks.getKieClasspathContainer();
      KieSession kSession = kContainer.newKieSession();
```

```
  // Set up the fact model:
  Person p = new Person();
  p.setWage(12);
  p.setFirstName("Tom");
  p.setLastName("Summers");
  p.setHourlyRate(10);

  // Insert the person into the session:
  kSession.insert(p);

  // Fire all rules:
  kSession.fireAllRules();
  kSession.dispose();
  }

 catch (Throwable t) {
  t.printStackTrace();
 }
 }
}
```

5. Download the **Red Hat Decision Manager 7.6.0 Source Distribution** ZIP file from the Red Hat Customer Portal and extract it under **my-project/dm-engine-jars/**.

6. In the **my-project/META-INF** directory, create a **kmodule.xml** metadata file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

This **kmodule.xml** file is a KIE module descriptor that selects resources to KIE bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced **kmodule.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
    <ksession name="KSession1_1" type="stateful" default="true" />
    <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtms" />
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
      <fileLogger file="debugInfo" threaded="true" interval="10" />
      <workItemHandlers>
        <workItemHandler name="name" type="new org.domain.WorkItemHandler()" />
      </workItemHandlers>
      <listeners>
```

```
            <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener" />
            <agendaEventListener type="org.domain.FirstAgendaListener" />
            <agendaEventListener type="org.domain.SecondAgendaListener" />
            <processEventListener type="org.domain.ProcessListener" />
        </listeners>
      </ksession>
    </kbase>
  </kmodule>
```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. The KIE session from **KBase2** is a **stateless** KIE session, which means that data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

7. After you create and save all DRL assets in your Java object, navigate to the **my-project** directory in the command line and run the following command to build your Java files. Replace **RulesTest.java** with the name of your Java main class.

   ```
   javac -classpath "./dm-engine-jars/*:." RulesTest.java
   ```

   If the build fails, address any problems described in the command line error messages and try again to validate the Java object until the object passes.

8. After your Java files build successfully, run the following command to execute the rules locally. Replace **RulesTest** with the prefix of your Java main class.

   ```
   java -classpath "./dm-engine-jars/*:." RulesTest
   ```

9. Review the rules to ensure that they executed properly, and address any needed changes in the Java files.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Java project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central. To access the project **pom.xml** file in Business Central, you can select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

6.3. CREATING AND EXECUTING DRL RULES USING MAVEN

You can use Maven archetypes to create DRL files with rules and integrate the archetypes with your Red Hat Decision Manager decision service. This method of creating DRL rules is helpful if you already use external Maven archetypes for your decision service and want to continue with the same workflow. If you do not already use this method, then the Business Central interface of Red Hat Decision Manager is recommended for creating DRL files and other rule assets.

Procedure

1. Navigate to a directory where you want to create a Maven archetype and run the following command:

   ```
   mvn archetype:generate -DgroupId=com.sample.app -DartifactId=my-app -
   DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
   ```

This creates a directory **my-app** with the following structure:

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- sample
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- sample
                    `-- app
                        `-- AppTest.java
```

The **my-app** directory contains the following key components:

- A **src/main** directory for storing the application sources

- A **src/test** directory for storing the test sources

- A **pom.xml** file with the project configuration

2. Create a Java object on which the rule or rules will operate within the Maven archetype. In this example, a **Person.java** file is created in the directory **my-app/src/main/java/com/sample/app**. The **Person** class contains getter and setter methods to set and retrieve the first name, last name, hourly rate, and the wage of a person:

```java
package com.sample.app;

public class Person {

  private String firstName;
  private String lastName;
  private Integer hourlyRate;
  private Integer wage;

  public String getFirstName() {
    return firstName;
  }

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }
```

```
    public Integer getHourlyRate() {
      return hourlyRate;
    }

    public void setHourlyRate(Integer hourlyRate) {
      this.hourlyRate = hourlyRate;
    }

    public Integer getWage(){
      return wage;
    }

    public void setWage(Integer wage){
      this.wage = wage;
    }
  }
```

3. Create a rule file in **.drl** format in **my-app/src/main/resources/rules**. The DRL file must contain at a minimum a package specification, an import list of data objects to be used by the rule or rules, and one or more rules with **when** conditions and **then** actions.

   The following **Wage.drl** file contains a **Wage** rule that imports the **Person** class, calculates the wage and hourly rate values, and displays a message based on the result:

```
package com.sample.app;

import com.sample.app.Person;

dialect "java"

rule "Wage"
  when
    Person(hourlyRate * wage > 100)
    Person(name : firstName, surname : lastName)
  then
    System.out.println("Hello " + name + " " + surname + "!");
    System.out.println("You are rich!");
end
```

4. In the **my-app/src/main/resources/META-INF** directory, create a **kmodule.xml** metadata file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://www.drools.org/xsd/kmodule">
</kmodule>
```

This **kmodule.xml** file is a KIE module descriptor that selects resources to KIE bases and configures sessions. This file enables you to define and configure one or more KIE bases, and to include DRL files from specific **packages** in a specific KIE base. You can also create one or more KIE sessions from each KIE base.

The following example shows a more advanced **kmodule.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://www.drools.org/xsd/kmodule">
```

```xml
    <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg1">
      <ksession name="KSession1_1" type="stateful" default="true" />
      <ksession name="KSession1_2" type="stateful" default="true" beliefSystem="jtms" />
    </kbase>
    <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
      <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
        <fileLogger file="debugInfo" threaded="true" interval="10" />
        <workItemHandlers>
          <workItemHandler name="name" type="new org.domain.WorkItemHandler()" />
        </workItemHandlers>
        <listeners>
          <ruleRuntimeEventListener type="org.domain.RuleRuntimeListener" />
          <agendaEventListener type="org.domain.FirstAgendaListener" />
          <agendaEventListener type="org.domain.SecondAgendaListener" />
          <processEventListener type="org.domain.ProcessListener" />
        </listeners>
      </ksession>
    </kbase>
  </kmodule>
```

This example defines two KIE bases. Two KIE sessions are instantiated from the **KBase1** KIE base, and one KIE session from **KBase2**. The KIE session from **KBase2** is a **stateless** KIE session, which means that data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. Specific **packages** of rule assets are included with both KIE bases. When you specify packages in this way, you must organize your DRL files in a folder structure that reflects the specified packages.

5. In the **my-app**/**pom.xml** configuration file, specify the libraries that your application requires. Provide the Red Hat Decision Manager dependencies as well as the **group ID**, **artifact ID**, and **version** (GAV) of your application.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>com.sample.app</groupId>
<artifactId>my-app</artifactId>
<version>1.0.0</version>
<repositories>
  <repository>
    <id>jboss-ga-repository</id>
    <url>http://maven.repository.redhat.com/ga/</url>
  </repository>
</repositories>
<dependencies>
  <dependency>
    <groupId>org.drools</groupId>
    <artifactId>drools-compiler</artifactId>
    <version>VERSION</version>
  </dependency>
  <dependency>
```

```
      <groupId>org.kie</groupId>
      <artifactId>kie-api</artifactId>
      <version>VERSION</version>
    </dependency>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

For information about Maven dependencies and the BOM (Bill of Materials) in Red Hat Decision Manager, see What is the mapping between Red Hat Decision Manager and Maven library version?.

6. Use the **testApp** method in **my-app/src/test/java/com/sample/app/AppTest.java** to test the rule. The **AppTest.java** file is created by Maven by default.

7. In the **AppTest.java** file, add the required **import** statements to import KIE services, a KIE container, and a KIE session. Then load the KIE base, insert facts, and execute the rule from the **testApp()** method that passes the fact model to the rule.

```
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public void testApp() {

  // Load the KIE base:
  KieServices ks = KieServices.Factory.get();
  KieContainer kContainer = ks.getKieClasspathContainer();
  KieSession kSession = kContainer.newKieSession();

  // Set up the fact model:
  Person p = new Person();
  p.setWage(12);
  p.setFirstName("Tom");
  p.setLastName("Summers");
  p.setHourlyRate(10);

  // Insert the person into the session:
  kSession.insert(p);

  // Fire all rules:
  kSession.fireAllRules();
  kSession.dispose();
}
```

8. After you create and save all DRL assets in your Maven archetype, navigate to the **my-app** directory in the command line and run the following command to build your files:

```
mvn clean install
```

If the build fails, address any problems described in the command line error messages and try again to validate the files until the build is successful.

9. After your files build successfully, run the following command to execute the rules locally. Replace **com.sample.app** with your package name.

```
mvn exec:java -Dexec.mainClass="com.sample.app"
```

10. Review the rules to ensure that they executed properly, and address any needed changes in the files.

To integrate the new rule assets with an existing project in Red Hat Decision Manager, you can compile the new Maven project as a knowledge JAR (KJAR) and add it as a dependency in the **pom.xml** file of the project in Business Central. To access the project **pom.xml** file in Business Central, you can select any existing asset in the project and then in the **Project Explorer** menu on the left side of the screen, click the **Customize View** gear icon and select **Repository View → pom.xml**.

6.4. EXECUTABLE RULE MODELS

Executable rule models are embeddable models that provide a Java-based representation of a rule set for execution at build time. The executable model is a more efficient alternative to the standard asset packaging in Red Hat Decision Manager and enables KIE containers and KIE bases to be created more quickly, especially when you have large lists of DRL (Drools Rule Language) files and other Red Hat Decision Manager assets. The model is low level and enables you to provide all necessary execution information, such as the lambda expressions for the index evaluation.

Executable rule models provide the following specific advantages for your projects:

- **Compile time:** Traditionally, a packaged Red Hat Decision Manager project (KJAR) contains a list of DRL files and other Red Hat Decision Manager artifacts that define the rule base together with some pre-generated classes implementing the constraints and the consequences. Those DRL files must be parsed and compiled when the KJAR is downloaded from the Maven repository and installed in a KIE container. This process can be slow, especially for large rule sets. With an executable model, you can package within the project KJAR the Java classes that implement the executable model of the project rule base and re-create the KIE container and its KIE bases out of it in a much faster way. In Maven projects, you use the **kie-maven-plugin** to automatically generate the executable model sources from the DRL files during the compilation process.

- **Run time:** In an executable model, all constraints are defined as Java lambda expressions. The same lambda expressions are also used for constraints evaluation, so you no longer need to use **mvel** expressions for interpreted evaluation nor the just-in-time (JIT) process to transform the **mvel**-based constraints into bytecode. This creates a quicker and more efficient run time.

- **Development time:** An executable model enables you to develop and experiment with new features of the decision engine without needing to encode elements directly in the DRL format or modify the DRL parser to support them.

NOTE

For query definitions in executable rule models, you can use up to 10 arguments only.

For variables within rule consequences in executable rule models, you can use up to 24 bound variables only (including the built-in **drools** variable). For example, the following rule consequence uses more than 24 bound variables and creates a compilation error:

```
...
then
  $input.setNo25Count(functions.sumOf(new Object[]{$no1Count_1, $no2Count_1,
$no3Count_1, ..., $no25Count_1}).intValue());
  $input.getFirings().add("fired");
  update($input);
```

6.4.1. Embedding an executable rule model in a Maven project

You can embed an executable rule model in your Maven project to compile your rule assets more efficiently at build time.

Prerequisites

- You have a Mavenized project that contains Red Hat Decision Manager business assets.

Procedure

1. In the **pom.xml** file of your Maven project, ensure that the packaging type is set to **kjar** and add the **kie-maven-plugin** build component:

```
<packaging>kjar</packaging>
...
<build>
 <plugins>
  <plugin>
   <groupId>org.kie</groupId>
   <artifactId>kie-maven-plugin</artifactId>
   <version>${rhdm.version}</version>
   <extensions>true</extensions>
  </plugin>
 </plugins>
</build>
```

The **kjar** packaging type activates the **kie-maven-plugin** component to validate and pre-compile artifact resources. The **<version>** is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.30.0.Final-redhat-00003). These settings are required to properly package the Maven project.

NOTE

Instead of specifying a Red Hat Decision Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.

Example BOM dependency:

```
<dependency>
  <groupId>com.redhat.ba</groupId>
  <artifactId>ba-platform-bom</artifactId>
  <version>7.6.0.redhat-00004</version>
  <scope>import</scope>
  <type>pom</type>
</dependency>
```

For more information about the Red Hat Business Automation BOM, see What is the mapping between RHDM product and maven library version?.

2. Add the following dependencies to the **pom.xml** file to enable rule assets to be built from an executable model:

- **drools-canonical-model**: Enables an executable canonical representation of a rule set model that is independent from Red Hat Decision Manager

- **drools-model-compiler**: Compiles the executable model into Red Hat Decision Manager internal data structures so that it can be executed by the decision engine

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-canonical-model</artifactId>
  <version>${rhdm.version}</version>
</dependency>

<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-model-compiler</artifactId>
  <version>${rhdm.version}</version>
</dependency>
```

3. In a command terminal, navigate to your Maven project directory and run the following command to build the project from an executable model:

```
mvn clean install -DgenerateModel=<VALUE>
```

The **-DgenerateModel=<VALUE>** property enables the project to be built as a model-based KJAR instead of a DRL-based KJAR.

Replace **<VALUE>** with one of three values:

- **YES**: Generates the executable model corresponding to the DRL files in the original project and excludes the DRL files from the generated KJAR.

- **WITHDRL**: Generates the executable model corresponding to the DRL files in the original project and also adds the DRL files to the generated KJAR for documentation purposes (the KIE base is built from the executable model regardless).

- **NO**: Does not generate the executable model.

Example build command:

```
mvn clean install -DgenerateModel=YES
```

For more information about packaging Maven projects, see *Packaging and deploying a Red Hat Decision Manager project*.

6.4.2. Embedding an executable rule model in a Java application

You can embed an executable rule model programmatically within your Java application to compile your rule assets more efficiently at build time.

Prerequisites

- You have a Java application that contains Red Hat Decision Manager business assets.

Procedure

1. Add the following dependencies to the relevant classpath for your Java project:

   - **drools-canonical-model**: Enables an executable canonical representation of a rule set model that is independent from Red Hat Decision Manager

   - **drools-model-compiler**: Compiles the executable model into Red Hat Decision Manager internal data structures so that it can be executed by the decision engine

   ```
   <dependency>
     <groupId>org.drools</groupId>
     <artifactId>drools-canonical-model</artifactId>
     <version>${rhdm.version}</version>
   </dependency>

   <dependency>
     <groupId>org.drools</groupId>
     <artifactId>drools-model-compiler</artifactId>
     <version>${rhdm.version}</version>
   </dependency>
   ```

   The **<version>** is the Maven artifact version for Red Hat Decision Manager currently used in your project (for example, 7.30.0.Final-redhat-00003).

> **NOTE**
>
> Instead of specifying a Red Hat Decision Manager **<version>** for individual dependencies, consider adding the Red Hat Business Automation bill of materials (BOM) dependency to your project **pom.xml** file. The Red Hat Business Automation BOM applies to both Red Hat Decision Manager and Red Hat Process Automation Manager. When you add the BOM files, the correct versions of transitive dependencies from the provided Maven repositories are included in the project.
>
> Example BOM dependency:
>
> ```
> <dependency>
>   <groupId>com.redhat.ba</groupId>
>   <artifactId>ba-platform-bom</artifactId>
>   <version>7.6.0.redhat-00004</version>
>   <scope>import</scope>
>   <type>pom</type>
> </dependency>
> ```
>
> For more information about the Red Hat Business Automation BOM, see What is the mapping between RHDM product and maven library version?.

2. Add rule assets to the KIE virtual file system **KieFileSystem** and use **KieBuilder** with **buildAll(ExecutableModelProject.class)** specified to build the assets from an executable model:

```
import org.kie.api.KieServices;
import org.kie.api.builder.KieFileSystem;
import org.kie.api.builder.KieBuilder;

KieServices ks = KieServices.Factory.get();
KieFileSystem kfs = ks.newKieFileSystem()
kfs.write("src/main/resources/KBase1/ruleSet1.drl", stringContainingAValidDRL)
.write("src/main/resources/dtable.xls",
  kieServices.getResources().newInputStreamResource(dtableFileStream));

KieBuilder kieBuilder = ks.newKieBuilder( kfs );
// Build from an executable model
kieBuilder.buildAll( ExecutableModelProject.class )
assertEquals(0, kieBuilder.getResults().getMessages(Message.Level.ERROR).size());
```

After **KieFileSystem** is built from the executable model, the resulting **KieSession** uses constraints based on lambda expressions instead of less-efficient **mvel** expressions. If **buildAll()** contains no arguments, the project is built in the standard method without an executable model.

As a more manual alternative to using **KieFileSystem** for creating executable models, you can define a **Model** with a fluent API and create a **KieBase** from it:

```
Model model = new ModelImpl().addRule( rule );
KieBase kieBase = KieBaseBuilder.createKieBaseFromModel( model );
```

For more information about packaging projects programmatically within a Java application, see *Packaging and deploying a Red Hat Decision Manager project* .

CHAPTER 7. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE

Red Hat Decision Manager provides example decisions distributed as Java classes that you can import into your integrated development environment (IDE). You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

The following example decision sets are some of the examples available in Red Hat Decision Manager:

- **Hello World example**: Demonstrates basic rule execution and use of debug output

- **State example**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

- **Fibonacci example**: Demonstrates recursion and conflict resolution through rule salience

- **Banking example**: Demonstrates pattern matching, basic sorting, and calculation

- **Pet Store example**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

- **Sudoku example**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

- **House of Doom example**: Demonstrates backward chaining and recursion

> **NOTE**
>
> For optimization examples provided with Red Hat Business Optimizer, see *Getting started with Red Hat Business Optimizer*.

7.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE

You can import Red Hat Decision Manager example decisions into your integrated development environment (IDE) and execute them to explore how the rules and code function. You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

Prerequisites

- Java 8 or later is installed.

- Maven 3.5.x or later is installed.

- An IDE is installed, such as Red Hat CodeReady Studio.

Procedure

1. Download and unzip the **Red Hat Decision Manager 7.6.0 Source Distribution** from the Red Hat Customer Portal to a temporary directory, such as **/rhdm-7.6.0-sources**.

2. Open your IDE and select **File → Import → Maven → Existing Maven Projects**, or the equivalent option for importing a Maven project.

3. Click **Browse**, navigate to ~/**rhdm-7.6.0-sources/src/drools-$VERSION/drools-examples** (or, for the Conway's Game of Life example, ~/**rhdm-7.6.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples**), and import the project.

4. Navigate to the example package that you want to run and find the Java class with the **main** method.

5. Right-click the Java class and select **Run As → Java Application** to run the example.
   To run all examples through a basic user interface, run the **DroolsExamplesApp.java** class (or, for Conway's Game of Life, the **DroolsJbpmIntegrationExamplesApp.java** class) in the **org.drools.examples** main class.

Figure 7.1. Interface for all examples in drools-examples (DroolsExamplesApp.java)
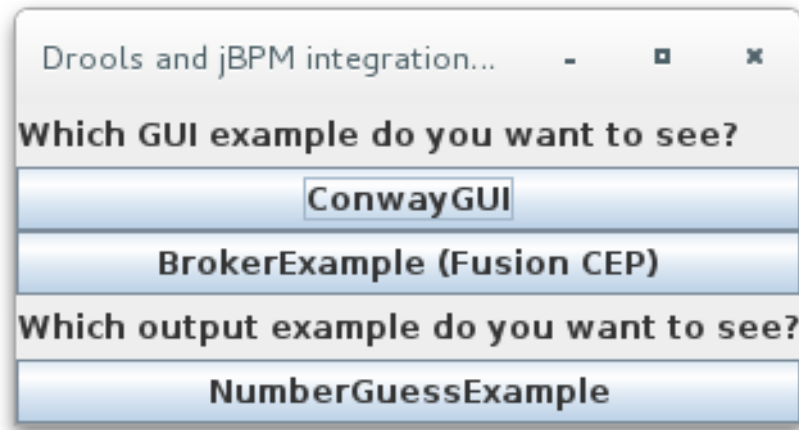
7.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING)

The Hello World example decision set demonstrates how to insert objects into the decision engine working memory, how to match the objects using rules, and how to configure logging to trace the internal activity of the decision engine.

The following is an overview of the Hello World example:

- Name: **helloworld**

- Main class: **org.drools.examples.helloworld.HelloWorldExample** (in **src**/**main**/**java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.helloworld.HelloWorld.drl** (in **src**/**main**/**resources**)

- Objective: Demonstrates basic rule execution and use of debug output

In the Hello World example, a KIE session is generated to enable rule execution. All rules require a KIE session for execution.

KIE session for rule execution

```
KieServices ks = KieServices.Factory.get(); ❶
KieContainer kc = ks.getKieClasspathContainer(); ❷
KieSession ksession = kc.newKieSession("HelloWorldKS"); ❸
```

❶ Obtains the **KieServices** factory. This is the main interface that applications use to interact with the decision engine.

❷ Creates a **KieContainer** from the project class path. This detects a **/META-INF/kmodule.xml** file from which it configures and instantiates a **KieContainer** with a **KieModule**.

❸ Creates a **KieSession** based on the **"HelloWorldKS"** KIE session configuration defined in the **/META-INF/kmodule.xml** file.

> **NOTE**
>
> For more information about Red Hat Decision Manager project packaging, see *Packaging and deploying a Red Hat Decision Manager project*.

Red Hat Decision Manager has an event model that exposes internal engine activity. Two default debug listeners, **DebugAgendaEventListener** and **DebugRuleRuntimeEventListener**, print debug event information to the **System.err** output. The **KieRuntimeLogger** provides execution auditing, the result of which you can view in a graphical viewer.

Debug listeners and audit loggers

```
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession,
"./target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "./target/helloworld",
1000 );
```

The logger is a specialized implementation built on the **Agenda** and **RuleRuntime** listeners. When the decision engine has finished executing, **logger.close()** is called.

The example creates a single **Message** object with the message **"Hello World"**, inserts the status **HELLO** into the **KieSession**, executes rules with **fireAllRules()**.

Data insertion and execution

```
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

Rule execution uses a data model to pass data as inputs and outputs to the **KieSession**. The data model in this example has two fields: the **message**, which is a **String**, and the **status**, which can be **HELLO** or **GOODBYE**.

Data model class

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String        message;
```

```
    private int        status;
    ...
}
```

The two rules are located in the file **src/main/resources/org/drools/examples/helloworld/HelloWorld.drl**.

The **when** condition of the **"Hello World"** rule states that the rule is activated for each **Message** object inserted into the KIE session that has the status **Message.HELLO**. Additionally, two variable bindings are created: the variable **message** is bound to the **message** attribute and the variable **m** is bound to the matched **Message** object itself.

The **then** action of the rule specifies to print the content of the bound variable **message** to **System.out**, and then changes the values of the **message** and **status** attributes of the **Message** object bound to **m**. The rule uses the **modify** statement to apply a block of assignments in one statement and to notify the decision engine of the changes at the end of the block.

"Hello World" rule

```
rule "Hello World"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbye cruel world",
            status = Message.GOODBYE };
end
```

The **"Good Bye"** rule is similar to the **"Hello World"** rule except that it matches **Message** objects that have the status **Message.GOODBYE**.

"Good Bye" rule

```
rule "Good Bye"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
end
```

To execute the example, run the **org.drools.examples.helloworld.HelloWorldExample** class as a Java application in your IDE. The rule writes to **System.out**, the debug listener writes to **System.err**, and the audit logger creates a log file in **target/helloworld.log**.

System.out output in the IDE console

```
Hello World
Goodbye cruel world
```

System.err output in the IDE console

```
==>[ActivationCreated(0): rule=Hello World;
        tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```

```
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
            object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
            tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;
            tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
            old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;
            new_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
            tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

To better understand the execution flow of this example, you can load the audit log file from **target/helloworld.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit view** shows that the object is inserted, which creates an activation for the **"Hello World"** rule. The activation is then executed, which updates the **Message** object and causes the **"Good Bye"** rule to activate. Finally, the **"Good Bye"** rule is executed. When you select an event in the **Audit View**, the origin event, which is the **"Activation created"** event in this example, is highlighted in green.

Figure 7.3. Hello World example Audit View
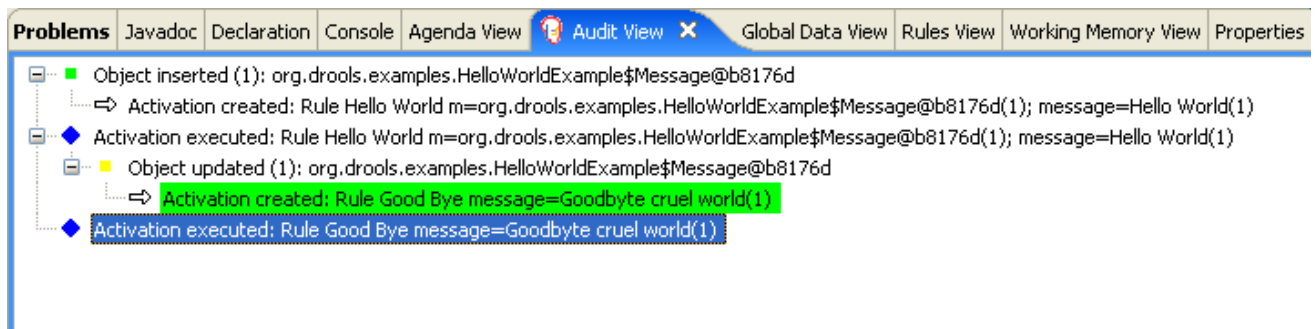
7.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION)

The State example decision set demonstrates how the decision engine uses forward chaining and any changes to facts in the working memory to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values or through agenda groups that you can define in rules.

The following is an overview of the State example:

- Name: **state**

- Main classes: **org.drools.examples.state.StateExampleUsingSalience**, **org.drools.examples.state.StateExampleUsingAgendaGroup** (in **src/main/java**)

- Module: **drools-examples**

- Type: Java application

- Rule files: **org.drools.examples.state.*.drl** (in **src/main/resources**)

- **Objective**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

In contrast, a backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 7.4. Rule evaluation logic using forward and backward chaining
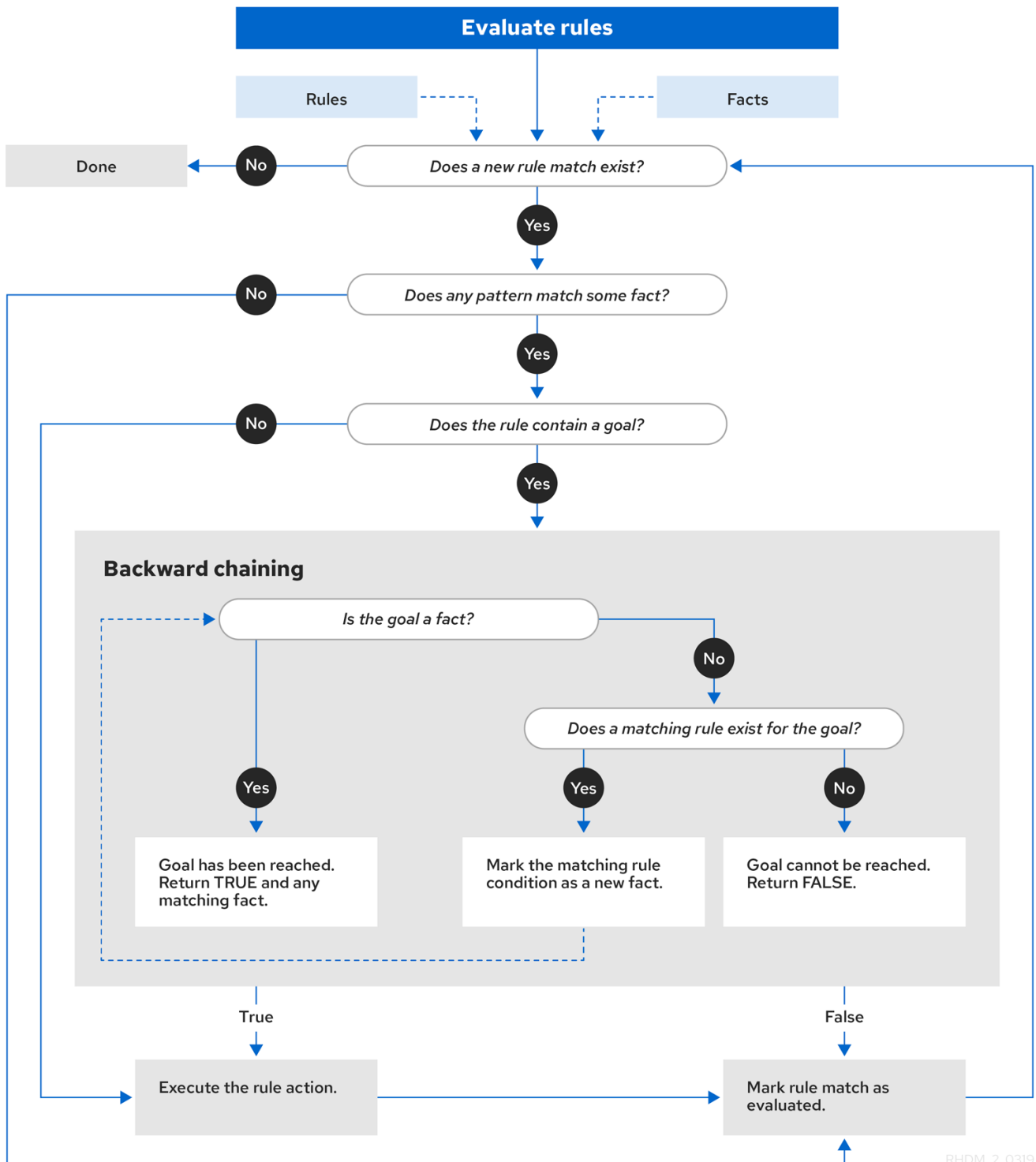
In the State example, each **State** class has fields for its name and its current state (see the class **org.drools.examples.state.State**). The following states are the two possible states for each object:

- **NOTRUN**

- **FINISHED**

State class

```
public class State {
    public static final int NOTRUN   = 0;
```

```java
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

The State example contains two versions of the same example to resolve rule execution conflicts:

- A **StateExampleUsingSalience** version that resolves conflicts by using rule salience

- A **StateExampleUsingAgendaGroups** version that resolves conflicts by using rule agenda groups

Both versions of the state example involve four **State** objects: **A**, **B**, **C**, and **D**. Initially, their states are set to **NOTRUN**, which is the default value for the constructor that the example uses.

State example using salience

The **StateExampleUsingSalience** version of the State example uses salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

The example inserts each **State** instance into the KIE session and then calls **fireAllRules()**.

Salience State example execution

```java
final State a = new State( "A" );
final State b = new State( "B" );
final State c = new State( "C" );
final State d = new State( "D" );

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingSalience** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Salience State example output in the IDE console

```
A finished
B finished
C finished
```

> D finished

Four rules are present.

First, the **"Bootstrap"** rule fires, setting **A** to state **FINISHED**, which then causes **B** to change its state to **FINISHED**. Objects **C** and **D** are both dependent on **B**, causing a conflict that is resolved by the salience values.

To better understand the execution flow of this example, you can load the audit log file from **target/state.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows that the assertion of the object **A** in the state **NOTRUN** activates the **"Bootstrap"** rule, while the assertions of the other objects have no immediate effect.

**Figure 7.5. Salience State example Audit View**
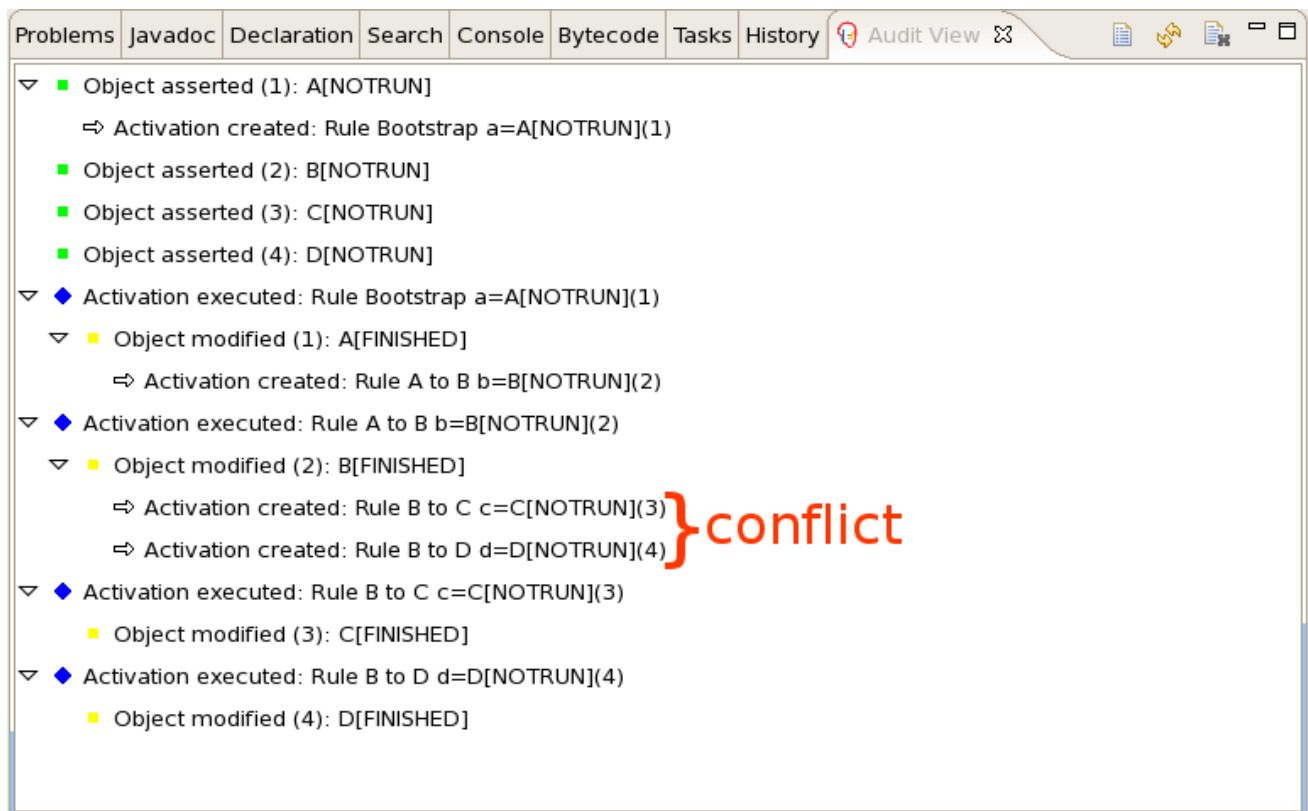


## Rule "Bootstrap" in salience State example

```
rule "Bootstrap"
  when
    a : State(name == "A", state == State.NOTRUN )
  then
    System.out.println(a.getName() + " finished" );
    a.setState( State.FINISHED );
end
```

The execution of the **"Bootstrap"** rule changes the state of **A** to **FINISHED**, which activates rule **"A to B"**.

Rule "A to B" in salience State example

```
rule "A to B"
  when
    State(name == "A", state == State.FINISHED )
    b : State(name == "B", state == State.NOTRUN )
  then
    System.out.println(b.getName() + " finished" );
    b.setState( State.FINISHED );
end
```

The execution of rule **"A to B"** changes the state of **B** to **FINISHED**, which activates both rules **"B to C"** and **"B to D"**, placing their activations onto the decision engine agenda.

Rules "B to C" and "B to D" in salience State example

```
rule "B to C"
    salience 10
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
end

rule "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

From this point on, both rules may fire and, therefore, the rules are in conflict. The conflict resolution strategy enables the decision engine agenda to decide which rule to fire. Rule **"B to C"** has the higher salience value (**10** versus the default salience value of **0**), so it fires first, modifying object **C** to state **FINISHED**.

The **Audit View** in your IDE shows the modification of the **State** object in the rule **"A to B"**, which results in two activations being in conflict.

You can also use the **Agenda View** in your IDE to investigate the state of the decision engine agenda. In this example, the **Agenda View** shows the breakpoint in the rule **"A to B"** and the state of the agenda with the two conflicting rules. Rule **"B to D"** fires last, modifying object **D** to state **FINISHED**.

Figure 7.6. Salience State example Agenda View

State example using agenda groups

The **StateExampleUsingAgendaGroups** version of the State example uses agenda groups in rules to resolve rule execution conflicts. Agenda groups enable you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

In this example, the **auto-focus** attribute enables rule **"B to C"** to fire before **"B to D"**.

Rule "B to C" in agenda group State example

```
rule "B to C"
   agenda-group "B to C"
   auto-focus true
 when
   State(name == "B", state == State.FINISHED )
   c : State(name == "C", state == State.NOTRUN )
 then
   System.out.println(c.getName() + " finished" );
   c.setState( State.FINISHED );
   kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
end
```

The rule **"B to C"** calls **setFocus()** on the agenda group **"B to D"**, enabling its active rules to fire, which then enables the rule **"B to D"** to fire.

Rule "B to D" in agenda group State example

```
rule "B to D"
   agenda-group "B to D"
 when
   State(name == "B", state == State.FINISHED )
   d : State(name == "D", state == State.NOTRUN )
 then
   System.out.println(d.getName() + " finished" );
   d.setState( State.FINISHED );
end
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingAgendaGroups** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window (same as the salience version of the State example):

Agenda group State example output in the IDE console

```
A finished
B finished
C finished
D finished
```

Dynamic facts in the State example

Another notable concept in this State example is the use of *dynamic facts*, based on objects that implement a **PropertyChangeListener** object. In order for the decision engine to see and react to changes of fact properties, the application must notify the decision engine that changes occurred. You

can configure this communication explicitly in the rules by using the **modify** statement, or implicitly by specifying that the facts implement the **PropertyChangeSupport** interface as defined by the JavaBeans specification.

This example demonstrates how to use the **PropertyChangeSupport** interface to avoid the need for explicit **modify** statements in the rules. To make use of this interface, ensure that your facts implement **PropertyChangeSupport** in the same way that the class **org.drools.example.State** implements it, and then use the following code in the DRL rule file to configure the decision engine to listen for property changes on those facts:

Declaring a dynamic fact

```
declare type State
  @propertyChangeSupport
end
```

When you use **PropertyChangeListener** objects, each setter must implement additional code for the notification. For example, the following setter for **state** is in the class **org.drools.examples**:

Setter example with PropertyChangeSupport

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                        oldState,
                        newState );
}
```

7.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION)

The Fibonacci example decision set demonstrates how the decision engine uses recursion to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values that you can define in rules.

The following is an overview of the Fibonacci example:

- Name: **fibonacci**

- Main class: **org.drools.examples.fibonacci.FibonacciExample** (in **src/main/java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.fibonacci.Fibonacci.drl** (in **src/main/resources**)

- Objective: Demonstrates recursion and conflict resolution through rule salience

The Fibonacci Numbers form a sequence starting with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The Fibonacci example uses the single fact class **Fibonacci** with the following two fields:

- **sequence**

- **value**

The **sequence** field indicates the position of the object in the Fibonacci number sequence. The **value** field shows the value of that Fibonacci object for that sequence position, where **-1** indicates a value that still needs to be computed.

Fibonacci class

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

To execute the example, run the **org.drools.examples.fibonacci.FibonacciExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Fibonacci example output in the IDE console

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

To achieve this behavior in Java, the example inserts a single **Fibonacci** object with a sequence field of **50**. The example then uses a recursive rule to insert the other 49 **Fibonacci** objects.

Instead of implementing the **PropertyChangeSupport** interface to use dynamic facts, this example uses the MVEL dialect **modify** keyword to enable a block setter action and notify the decision engine of changes.

Fibonacci example execution

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

This example uses the following three rules:

- **"Recurse"**

- **"Bootstrap"**

- **"Calculate"**

The rule **"Recurse"** matches each asserted **Fibonacci** object with a value of **-1**, creating and asserting a new **Fibonacci** object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to **1** does not exist, the rule re-matches and fires again. The **not** conditional element is used to stop the rule matching once you have all 50 Fibonacci objects in memory. The rule also has a **salience** value because you need to have all 50 **Fibonacci** objects asserted before you execute the **"Bootstrap"** rule.

Rule "Recurse"

```
rule "Recurse"
    salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
  end
```

To better understand the execution flow of this example, you can load the audit log file from **target/fibonacci.log** into your IDE debug view or **Audit View**, if available (for example, in **Window →Show View** in some IDEs).

In this example, the **Audit View** shows the original assertion of the **Fibonacci** object with a **sequence** field of **50**, done from Java code. From there on, the **Audit View** shows the continual recursion of the rule, where each asserted **Fibonacci** object causes the **"Recurse"** rule to become activated and to fire again.

Figure 7.7. Rule "Recurse" in Audit View

When a **Fibonacci** object with a **sequence** field of **2** is asserted, the **"Bootstrap"** rule is matched and activated along with the **"Recurse"** rule. Notice the multiple restrictions on field **sequence** that test for equality with **1** or **2**:

Rule "Bootstrap"

```
rule "Bootstrap"
  when
    f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction
  then
    modify ( f ){ value = 1 };
    System.out.println( f.sequence + " == " + f.value );
end
```

You can also use the **Agenda View** in your IDE to investigate the state of the decision engine agenda. The **"Bootstrap"** rule does not fire yet because the **"Recurse"** rule has a higher salience value.

Figure 7.8. Rules "Recurse" and "Bootstrap" in Agenda View 1
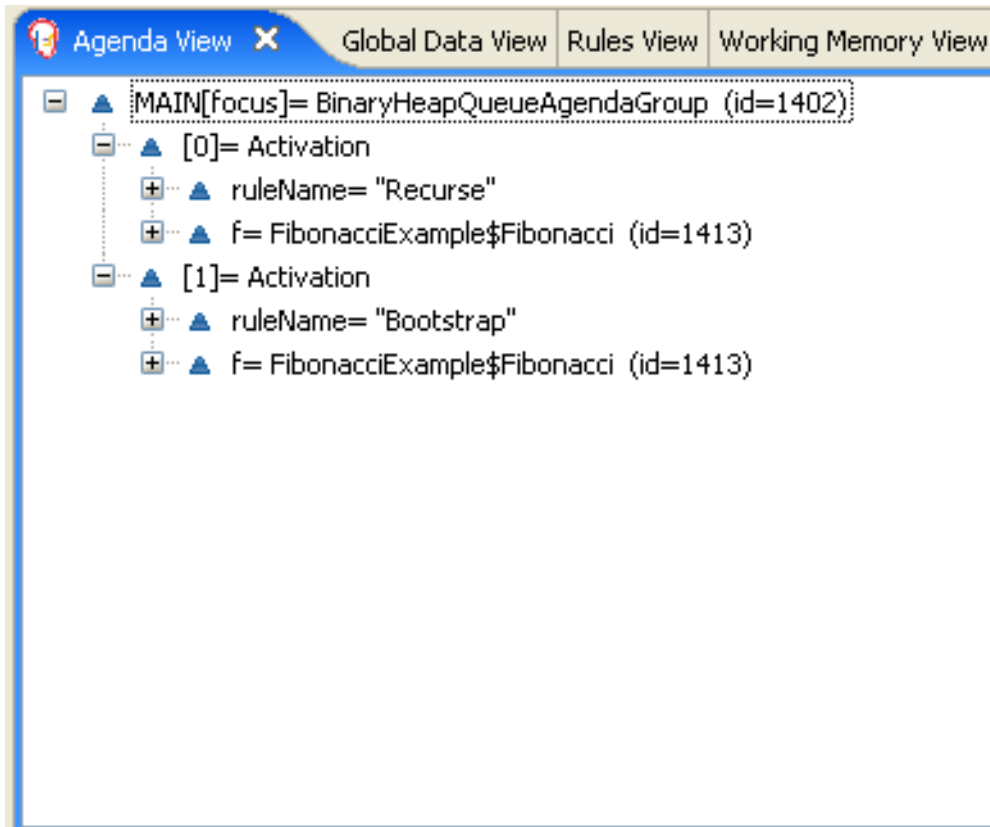
When a **Fibonacci** object with a **sequence** of **1** is asserted, the **"Bootstrap"** rule is matched again, causing two activations for this rule. The **"Recurse"** rule does not match and activate because the **not** conditional element stops the rule matching as soon as a **Fibonacci** object with a **sequence** of **1** exists.

Figure 7.9. Rules "Recurse" and "Bootstrap" in Agenda View 2
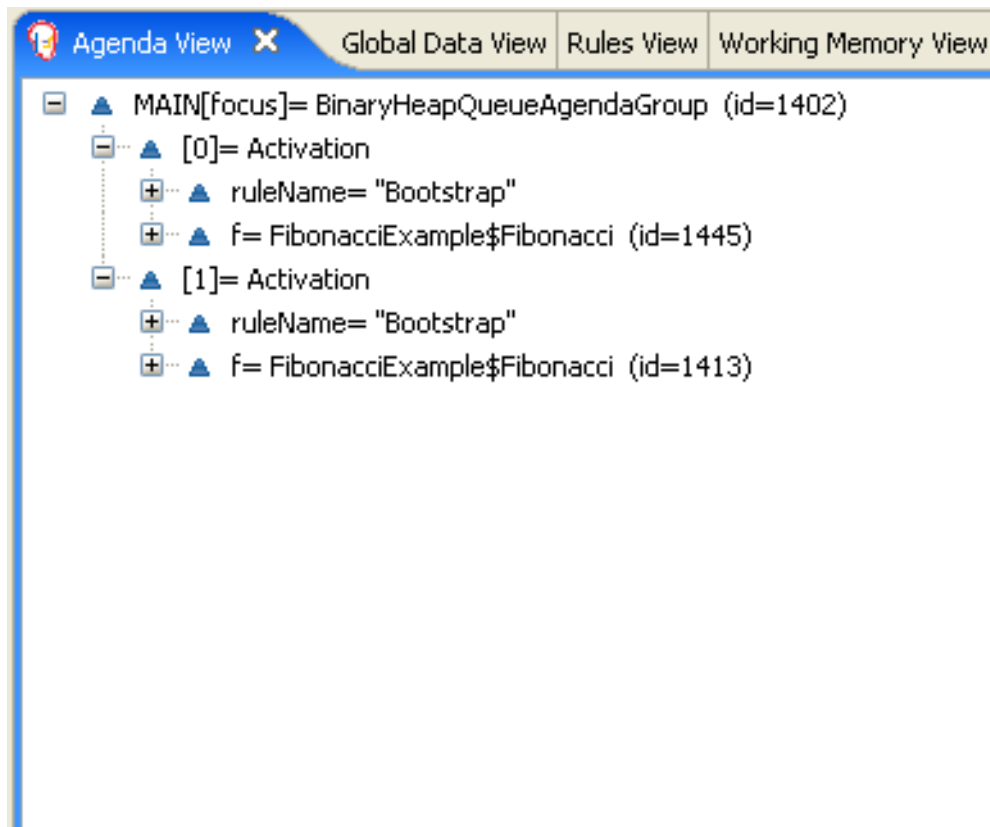
The **"Bootstrap"** rule sets the objects with a **sequence** of **1** and **2** to a value of **1**. Now that you have two **Fibonacci** objects with values not equal to **-1**, the **"Calculate"** rule is able to match.

At this point in the example, nearly 50 **Fibonacci** objects exist in the working memory. You need to select a suitable triple to calculate each of their values in turn. If you use three Fibonacci patterns in a rule without field constraints to confine the possible cross products, the result would be 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect.

The **"Calculate"** rule uses field constraints to evaluate the three Fibonacci patterns in the correct order. This technique is called *cross-product matching*.

The first pattern finds any **Fibonacci** object with a value **!= -1** and binds both the pattern and the field. The second **Fibonacci** object does the same thing, but adds an additional field constraint to ensure that its sequence is greater by one than the **Fibonacci** object bound to **f1**. When this rule fires for the first time, you know that only sequences **1** and **2** have values of **1**, and the two constraints ensure that **f1** references sequence **1** and that **f2** references sequence **2**.

The final pattern finds the **Fibonacci** object with a value equal to **-1** and with a sequence one greater than **f2**.

At this point in the example, three **Fibonacci** objects are correctly selected from the available cross products, and you can calculate the value for the third **Fibonacci** object that is bound to **f3**.

Rule "Calculate"

```
rule "Calculate"
  when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
  then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
  end
```

The **modify** statement updates the value of the **Fibonacci** object bound to **f3**. This means that you now have another new **Fibonacci** object with a value not equal to **-1**, which allows the **"Calculate"** rule to re-match and calculate the next Fibonacci number.

The debug view or **Audit View** of your IDE shows how the firing of the last **"Bootstrap"** rule modifies the **Fibonacci** object, enabling the **"Calculate"** rule to match, which then modifies another **Fibonacci** object that enables the **"Calculate"** rule to match again. This process continues until the value is set for all **Fibonacci** objects.

Figure 7.10. Rules in Audit View
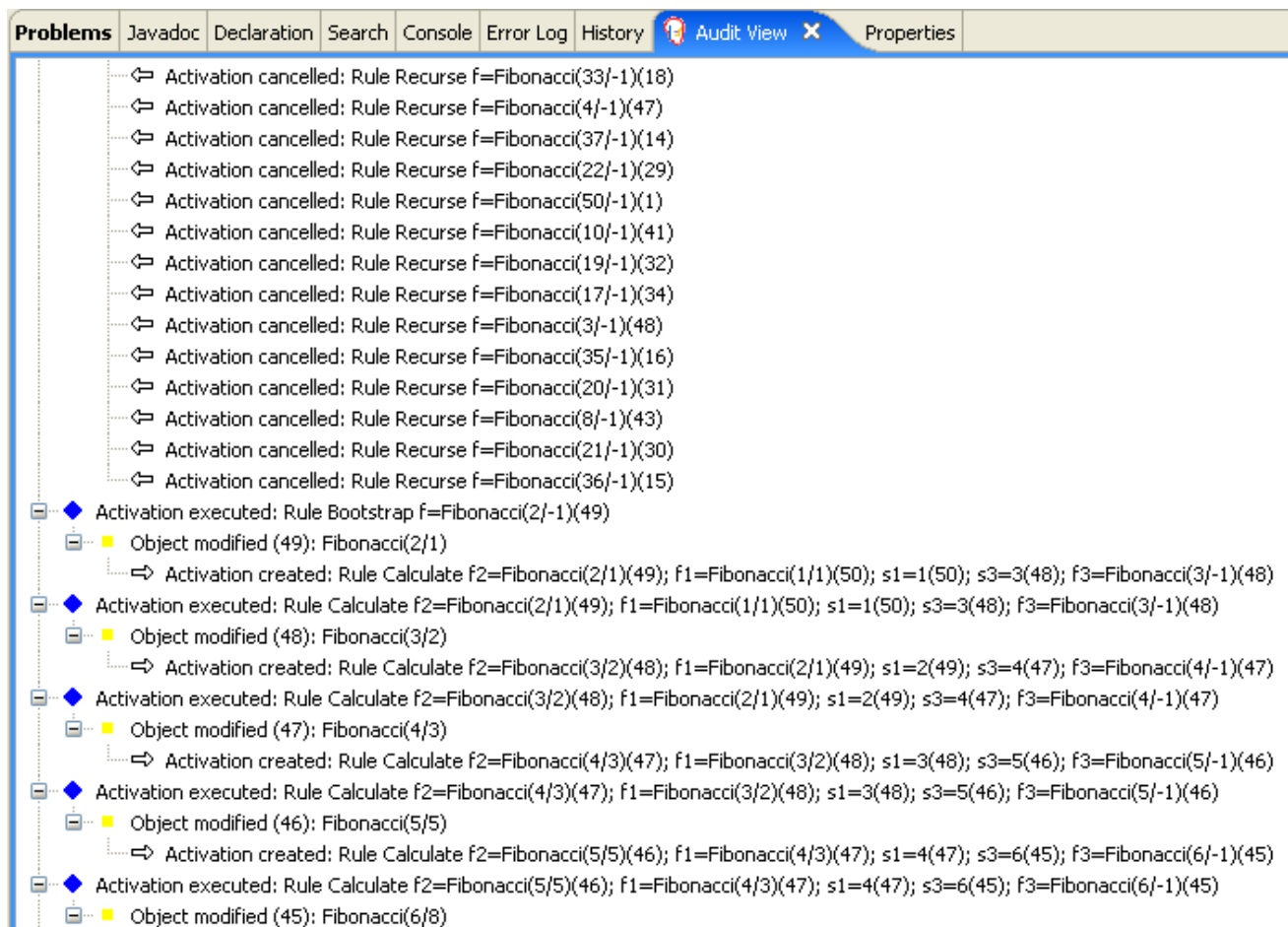
7.5. PRICING EXAMPLE DECISIONS (DECISION TABLES)

The Pricing example decision set demonstrates how to use a spreadsheet decision table for calculating the retail cost of an insurance policy in tabular format instead of directly in a DRL file.

The following is an overview of the Pricing example:

- Name: **decisiontable**

- Main class: **org.drools.examples.decisiontable.PricingRuleDTExample** (in **src**/**main**/**java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.decisiontable.ExamplePolicyPricing.xls** (in **src**/**main**/**resources**)

- Objective: Demonstrates use of spreadsheet decision tables to define rules

Spreadsheet decision tables are XLS or XLSX spreadsheets that contain business rules defined in a tabular format. You can include spreadsheet decision tables with standalone Red Hat Decision Manager projects or upload them to projects in Business Central. Each row in a decision table is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your decision tables into your Red Hat Decision Manager project, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

The purpose of the Pricing example is to provide a set of business rules to calculate the base price and a discount for a car driver applying for a specific type of insurance policy. The driver's age and history and the policy type all contribute to calculate the basic premium, and additional rules calculate potential discounts for which the driver might be eligible.

To execute the example, run the **org.drools.examples.decisiontable.PricingRuleDTExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to execute the example follows the typical execution pattern: the rules are loaded, the facts are inserted, and a stateless KIE session is created. The difference in this example is that the rules are defined in an **ExamplePolicyPricing.xls** file instead of a DRL file or other source. The spreadsheet file is loaded into the decision engine using templates and DRL rules.

Spreadsheet decision table setup
The **ExamplePolicyPricing.xls** spreadsheet contains two decision tables in the first tab:

- **Base pricing rules**

- **Promotional discount rules**

As the example spreadsheet demonstrates, you can use only the first tab of a spreadsheet to create decision tables, but multiple tables can be within a single tab. Decision tables do not necessarily follow top-down logic, but are more of a means to capture data resulting in rules. The evaluation of the rules is not necessarily in the given order, because all of the normal mechanics of the decision engine still apply. This is why you can have multiple decision tables in the same tab of a spreadsheet.

The decision tables are executed through the corresponding rule template files **BasePricing.drt** and **PromotionalPricing.drt**. These template files reference the decision tables through their template parameter and directly reference the various headers for the conditions and actions in the decision tables.

BasePricing.drt rule template file

```
template header
age[]
profile
priorClaims
policyType
base
reason

package org.drools.examples.decisiontable;

template "Pricing bracket"
age
policyType
base

rule "Pricing bracket_@{row.rowNumber}"
    when
```

```
    Driver(age >= @{age0}, age <= @{age1}
        , priorClaims == "@{priorClaims}"
        , locationRiskProfile == "@{profile}"
    )
    policy: Policy(type == "@{policyType}")
  then
    policy.setBasePrice(@{base});
    System.out.println("@{reason}");
end
end template
```

PromotionalPricing.drt rule template file

```
template header
age[]
priorClaims
policyType
discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_@{row.rowNumber}"
  when
    Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}")
    policy: Policy(type == "@{policyType}")
  then
    policy.applyDiscount(@{discount});
end
end template
```

The rules are executed through the **kmodule.xml** reference of the KIE Session **DTableWithTemplateKB**, which specifically mentions the **ExamplePolicyPricing.xls** spreadsheet and is required for successful execution of the rules. This execution method enables you to execute the rules as a standalone unit (as in this example) or to include the rules in a packaged knowledge JAR (KJAR) file, so that the spreadsheet is packaged along with the rules for execution.

The following section of the **kmodule.xml** file is required for the execution of the rules and spreadsheet to work successfully:

```
<kbase name="DecisionTableKB" packages="org.drools.examples.decisiontable">
  <ksession name="DecisionTableKS" type="stateless"/>
</kbase>

<kbase name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
  <ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
              template="org/drools/examples/decisiontable-template/BasePricing.drt"
              row="3" col="3"/>
  <ruleTemplate dtable="org/drools/examples/decisiontable-
```

```
template/ExamplePolicyPricingTemplateData.xls"
            template="org/drools/examples/decisiontable-template/PromotionalPricing.drt"
            row="18" col="3"/>
    <ksession name="DTableWithTemplateKS"/>
  </kbase>
```

As an alternative to executing the decision tables using rule template files, you can use the **DecisionTableConfiguration** object and specify an input spreadsheet as the input type, such as **DecisionTableInputType.xls**:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
        dtableconfiguration.setInputType( DecisionTableInputType.XLS );

        KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

        Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls",
                                        getClass() );
        kbuilder.add( xlsRes,
                ResourceType.DTABLE,
                dtableconfiguration );
```

The Pricing example uses two fact types:

- **Driver**

- **Policy**.

The example sets the default values for both facts in their respective Java classes **Driver.java** and **Policy.java**. The **Driver** is 30 years old, has had no prior claims, and currently has a risk profile of **LOW**. The **Policy** that the driver is applying for is **COMPREHENSIVE**.

In any decision table, each row is considered a different rule and each column is a condition or an action. Each row is evaluated in a decision table unless the agenda is cleared upon execution.

Decision table spreadsheets (XLS or XLSX) require two key areas that define rule data:

- A **RuleSet** area

- A **RuleTable** area

The **RuleSet** area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The **RuleTable** area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A decision table spreadsheet can contain multiple **RuleTable** areas, but only one **RuleSet** area.

Figure 7.11. Decision table configuration

The **RuleTable** area also defines the objects to which the rule attributes apply, in this case **Driver** and **Policy**, followed by constraints on the objects. For example, the **Driver** object constraint that defines the **Age Bracket** column is **age >= $1, age <= $2**, where the comma-separated range is defined in the table column values, such as **18,24**.

Base pricing rules

The **Base pricing rules** decision table in the Pricing example evaluates the age, risk profile, number of claims, and policy type of the driver and produces the base price of the policy based on these conditions.

Figure 7.12. Base price calculation

The **Driver** attributes are defined in the following table columns:

- **Age Bracket**: The age bracket has a definition for the condition **age >=$1, age <=$2**, which defines the condition boundaries for the driver's age. This condition column highlights the use of **$1 and $2**, which is comma delimited in the spreadsheet. You can write these values as **18,24** or **18, 24** and both formats work in the execution of the business rules.

- **Location risk profile**: The risk profile is a string that the example program passes always as **LOW** but can be changed to reflect **MED** or **HIGH**.

- **Number of prior claims**: The number of claims is defined as an integer that the condition column must exactly equal to trigger the action. The value is not a range, only exact matches.

The **Policy** of the decision table is used in both the conditions and the actions of the rule and has attributes defined in the following table columns:

- **Policy type applying for**: The policy type is a condition that is passed as a string that defines the type of coverage: **COMPREHENSIVE**, **FIRE_THEFT**, or **THIRD_PARTY**.

- **Base $ AUD**: The **basePrice** is defined as an **ACTION** that sets the price through the constraint **policy.setBasePrice($param);** based on the spreadsheet cells corresponding to this value. When you execute the corresponding DRL rule for this decision table, the **then** portion of the rule executes this action statement on the true conditions matching the facts and sets the base price to the corresponding value.

- **Record Reason**: When the rule successfully executes, this action generates an output message to the **System.out** console reflecting which rule fired. This is later captured in the application and printed.

The example also uses the first column on the left to categorize rules. This column is for annotation only and has no affect on rule execution.

Promotional discount rules

The **Promotional discount rules** decision table in the Pricing example evaluates the age, number of prior claims, and policy type of the driver to generate a potential discount on the price of the insurance policy.

Figure 7.13. Discount calculation

29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30		18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32	Rewards for safe drivers	25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20
35					

This decision table contains the conditions for the discount for which the driver might be eligible. Similar to the base price calculation, this table evaluates the **Age**, **Number of prior claims** of the driver, and the **Policy type applying for** to determine a **Discount %** rate to be applied. For example, if the driver is 30 years old, has no prior claims, and is applying for a **COMPREHENSIVE** policy, the driver is given a discount of **20** percent.

7.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION)

The Pet Store example decision set demonstrates how to use agenda groups and global variables in rules and how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based desktop application. The example also demonstrates how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Pet Store example:

- Name: **petstore**

- Main class: **org.drools.examples.petstore.PetStoreExample** (in **src/main/java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.petstore.PetStore.drl** (in **src/main/resources**)

- Objective: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

In the Pet Store example, the sample **PetStoreExample.java** class defines the following principal classes (in addition to several classes to handle Swing events):

- **Petstore** contains the **main()** method.

- **PetStoreUI** is responsible for creating and displaying the Swing-based GUI. This class contains several smaller classes, mainly for responding to various GUI events, such as user mouse clicks.

- **TableModel** holds the table data. This class is essentially a JavaBean that extends the Swing class **AbstractTableModel**.

- **CheckoutCallback** enables the GUI to interact with the rules.

- **Ordershow** keeps the items that you want to buy.

- **Purchase** stores details of the order and the products that you are buying.

- **Product** is a JavaBean containing details of the product available for purchase and its price.

Much of the Java code in this example is either plain JavaBean or Swing based. For more information about Swing components, see the Java tutorial on Creating a GUI with JFC/Swing .

Rule execution behavior in the Pet Store example

Unlike other example decision sets where the facts are asserted and fired immediately, the Pet Store example does not execute the rules until more facts are gathered based on user interaction. The example executes rules through a **PetStoreUI** object, created by a constructor, that accepts the **Vector** object **stock** for collecting the products. The example then uses an instance of the **CheckoutCallback** class containing the rule base that was previously loaded.

Pet Store KIE container and fact execution setup

```
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );
```

```
// A callback is responsible for populating the working memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                    new CheckoutCallback( kc ) );
ui.createAndShowGUI();
```

The Java code that fires the rules is in the **CheckoutCallBack.checkout()** method. This method is triggered when the user clicks **Checkout** in the UI.

Rule execution from CheckoutCallBack.checkout()

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user interaction.

    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

The example code passes two elements into the **CheckoutCallBack.checkout()** method. One element is the handle for the **JFrame** Swing component surrounding the output text frame, found at the bottom of the GUI. The second element is a list of order items, which comes from the **TableModel** that stores the information from the **Table** area at the upper-right section of the GUI.

The **for** loop transforms the list of order items coming from the GUI into the   **Order** JavaBean, also contained in the file **PetStoreExample.java**.

In this case, the rule is firing in a stateless KIE session because all of the data is stored in Swing components and is not executed until the user clicks **Checkout** in the UI. Each time the user clicks **Checkout**, the content of the list is moved from the Swing   **TableModel** into the KIE session working memory and is then executed with the **ksession.fireAllRules()** method.

Within this code, there are nine calls to **KieSession**. The first of these creates a new **KieSession** from the **KieContainer** (the example passed in this **KieContainer** from the **CheckoutCallBack** class in the **main()** method). The next two calls pass in the two objects that hold the global variables in the rules: the Swing text area and the Swing frame used for writing messages. More inserts put information on products into the **KieSession**, as well as the order list. The final call is the standard **fireAllRules()**.

Pet Store rule file imports, global variables, and Java functions

The **PetStore.drl** file contains the standard package and import statements to make various Java classes available to the rules. The rule file also includes *global variables* to be used within the rules, defined as **frame** and **textArea**. The global variables hold references to the Swing components **JFrame** and **JTextArea** components that were previously passed on by the Java code that called the **setGlobal()** method. Unlike standard variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the KIE session. This means the contents of these global variables are available for evaluation on all subsequent rules.

PetStore.drl package, imports, and global variables

```
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;

import javax.swing.JFrame;

global JFrame frame
global javax.swing.JTextArea textArea
```

The **PetStore.drl** file also contains two functions that the rules in the file use:

PetStore.drl Java functions

```
function void doCheckout(JFrame frame, KieRuntime krt) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
                        "Would you like to checkout?",
                        "",
                        JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE,
                        null,
                        options,
                        options[0]);

    if (n == 0) {
        krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
    }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total)
{
```

```
        Object[] options = {"Yes",
                    "No"};

        int n = JOptionPane.showOptionDialog(frame,
                            "Would you like to buy a tank for your " + total + " fish?",
                            "Purchase Suggestion",
                            JOptionPane.YES_NO_OPTION,
                            JOptionPane.QUESTION_MESSAGE,
                            null,
                            options,
                            options[0]);

        System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                    + total + " fish? - " );

        if (n == 0) {
            Purchase purchase = new Purchase( order, fishTank );
            krt.insert( purchase );
            order.addItem( purchase );
            System.out.println( "Yes" );
        } else {
            System.out.println( "No" );
        }
        return true;
    }
```

The two functions perform the following actions:

- **doCheckout()** displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire.

- **requireTank()** displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory.

> **NOTE**
>
> For this example, all rules and functions are within the same rule file for efficiency. In a production environment, you typically separate the rules and functions in different files or build a static Java method and import the files using the import function, such as **import function my.package.name.hello**.

Pet Store rules with agenda groups

Most of the rules in the Pet Store example use agenda groups to control rule execution. Agenda groups allow you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

The Pet Store example uses the following agenda groups for rules:

- **"init"**

- **"evaluate"**

- **"show items"**

- **"checkout"**

For example, the sample rule **"Explode Cart"** uses the **"init"** agenda group to ensure that it has the option to fire and insert shopping cart items into the KIE session working memory:

Rule "Explode Cart"

```
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
  agenda-group "init"
  auto-focus true
  salience 10
when
  $order : Order( grossTotal == -1 )
  $item : Purchase() from $order.items
then
  insert( $item );
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();
end
```

This rule matches against all orders that do not yet have their **grossTotal** calculated. The execution loops for each purchase item in that order.

The rule uses the following features related to its agenda group:

- **agenda-group "init"** defines the name of the agenda group. In this case, only one rule is in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the **auto-focus** attribute for its chance to fire.

- **auto-focus true** ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when **fireAllRules()** is called from the Java code.

- **kcontext….setFocus()** sets the focus to the **"show items"** and **"evaluate"** agenda groups, enabling their rules to fire. In practice, you loop through all items in the order, insert them into memory, and then fire the other rules after each insertion.

The **"show items"** agenda group contains only one rule, **"Show Items"**. For each purchase in the order currently in the KIE session working memory, the rule logs details to the text area at the bottom of the GUI, based on the **textArea** variable defined in the rule file.

Rule "Show Items"

```
rule "Show Items"
  agenda-group "show items"
when
  $order : Order()
  $p : Purchase( order == $order )
```

```
    then
      textArea.append( $p.product + "\n");
    end
```

The **"evaluate"** agenda group also gains focus from the **"Explode Cart"** rule. This agenda group contains two rules, **"Free Fish Food Sample"** and **"Suggest Tank"**, which are executed in that order.

Rule "Free Fish Food Sample"

```
  // Free fish food sample when users buy a goldfish if they did not already buy
  // fish food and do not already have a fish food sample.
  rule "Free Fish Food Sample"
    agenda-group "evaluate"  1
   when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p ) )  2
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product == $p ) )  3
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p ) )  4
    $fishFoodSample : Product( name == "Fish Food Sample" );
   then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
  end
```

The rule **"Free Fish Food Sample"** fires only if all of the following conditions are true:

1   The agenda group **"evaluate"** is being evaluated in the rules execution.

2   User does not already have fish food.

3   User does not already have a free fish food sample.

4   User has a goldfish in the order.

If the order facts meet all of these requirements, then a new product is created (Fish Food Sample) and is added to the order in working memory.

Rule "Suggest Tank"

```
  // Suggest a fish tank if users buy more than five goldfish and
  // do not already have a tank.
  rule "Suggest Tank"
    agenda-group "evaluate"
   when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p ) )  1
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) )  2
    $fishTank : Product( name == "Fish Tank" )
   then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
  end
```

The rule **"Suggest Tank"** fires only if the following conditions are true:

1 User does not have a fish tank in the order.

2 User has more than five fish in the order.

When the rule fires, it calls the **requireTank()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory. When the rule calls the **requireTank()** function, the rule passes the **frame** global variable so that the function has a handle for the Swing GUI.

The **"do checkout"** rule in the Pet Store example has no agenda group and no **when** conditions, so the rule is always executed and considered part of the default **MAIN** agenda group.

Rule "do checkout"

```
rule "do checkout"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

When the rule fires, it calls the **doCheckout()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire. When the rule calls the **doCheckout()** function, the rule passes the **frame** global variable so that the function has a handle for the Swing GUI.

> **NOTE**
>
> This example also demonstrates a troubleshooting technique if results are not executing as you expect: You can remove the conditions from the **when** statement of a rule and test the action in the **then** statement to verify that the action is performed correctly.

The **"checkout"** agenda group contains three rules for processing the order checkout and applying any discounts: **"Gross Total"**, **"Apply 5% Discount"**, and **"Apply 10% Discount"**.

Rules "Gross Total", "Apply 5% Discount", and "Apply 10% Discount"

```
rule "Gross Total"
  agenda-group "checkout"
  when
    $order : Order( grossTotal == -1 )
    Number( total : doubleValue ) from accumulate( Purchase( $price : product.price ),
                                   sum( $price ) )
  then
    modify( $order ) { grossTotal = total }
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
  agenda-group "checkout"
  when
    $order : Order( grossTotal >= 10 && < 20 )
```

```
  then
    $order.discountedTotal = $order.grossTotal * 0.95;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end

rule "Apply 10% Discount"
  agenda-group "checkout"
  when
    $order : Order( grossTotal >= 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end
```

If the user has not already calculated the gross total, the **Gross Total** accumulates the product prices into a total, puts this total into the KIE session, and displays it through the Swing **JTextArea** using the **textArea** global variable.

If the gross total is between **10** and **20** (currency units), the **"Apply 5% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

If the gross total is not less than **20**, the **"Apply 10% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

Pet Store example execution

Similar to other Red Hat Decision Manager decision examples, you execute the Pet Store example by running the **org.drools.examples.petstore.PetStoreExample** class as a Java application in your IDE.

When you execute the Pet Store example, the **Pet Store Demo** GUI window appears. This window displays a list of available products (upper left), an empty list of selected products (upper right), **Checkout** and **Reset** buttons (middle), and an empty system messages area (bottom).

Figure 7.14. Pet Store example GUI after launch
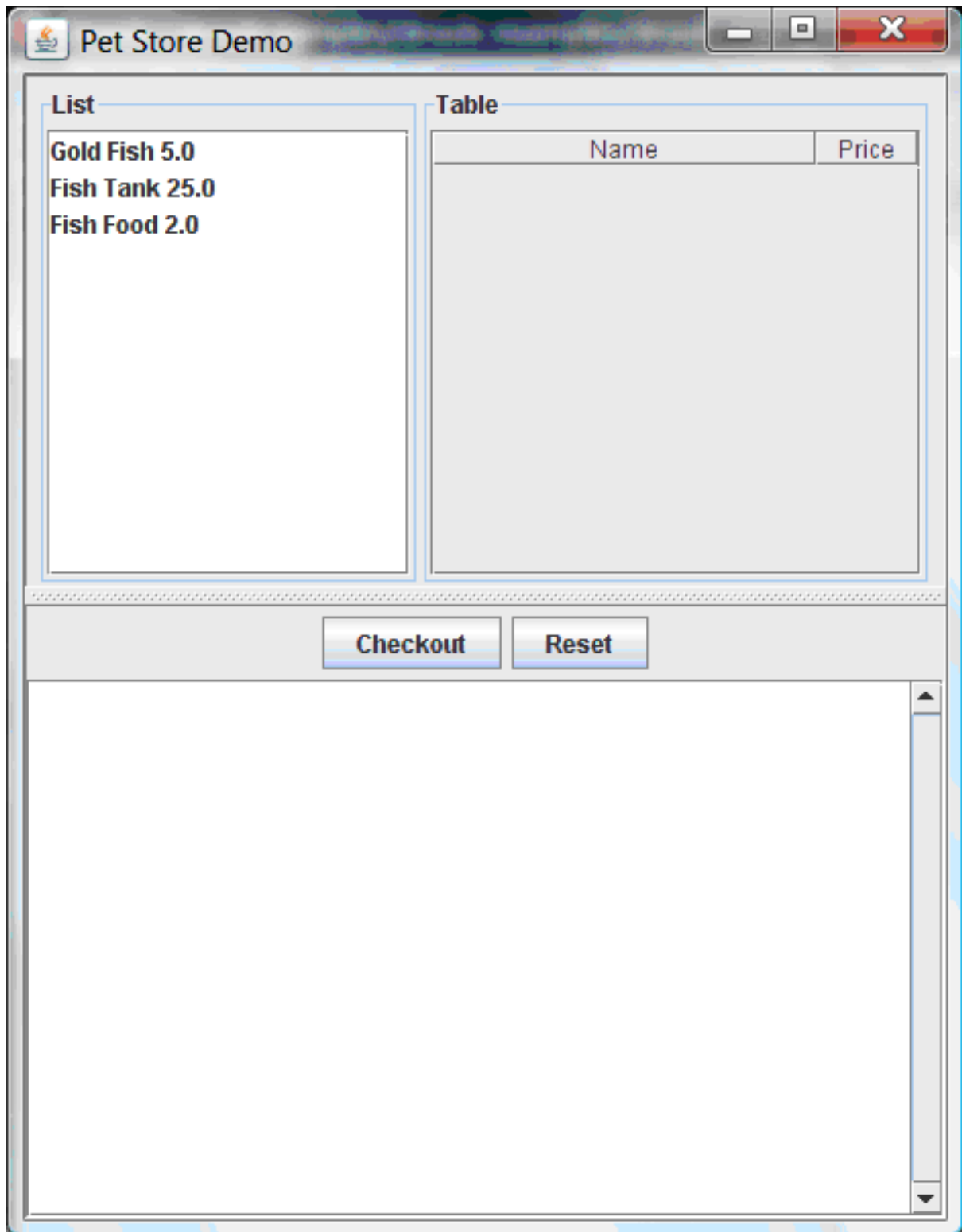
The following events occurred in this example to establish this execution behavior:

1. The **main()** method has run and loaded the rule base but has not yet fired the rules. So far, this is the only code in connection with rules that has been run.

2. A new **PetStoreUI** object has been created and given a handle for the rule base, for later use.

3. Various Swing components have performed their functions, and the initial UI screen is displayed and waits for user input.

You can click various products from the list to explore the UI setup:

Figure 7.15. Explore the Pet Store example GUI
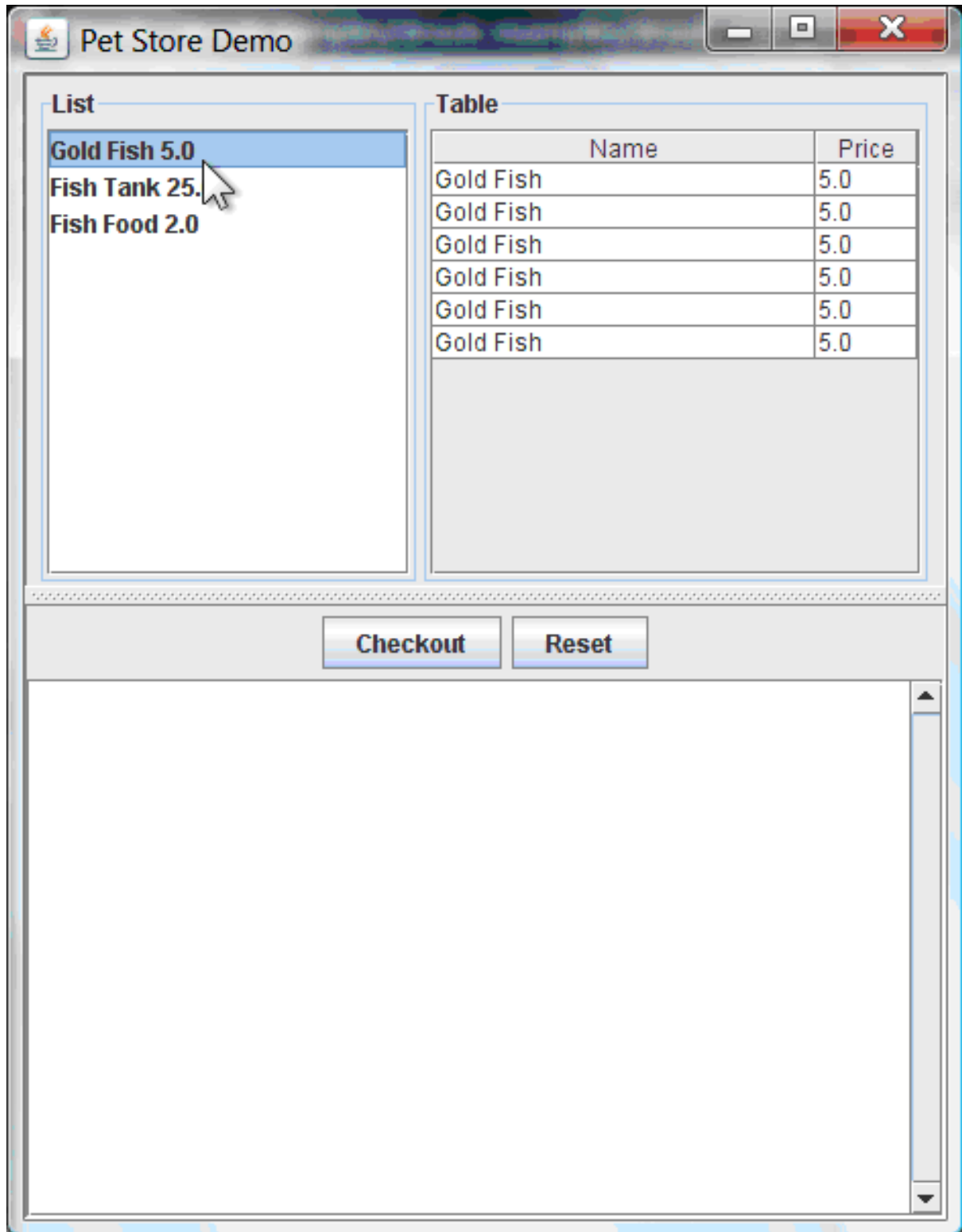
No rules code has been fired yet. The UI uses Swing code to detect user mouse clicks and add selected products to the **TableModel** object for display in the upper-right corner of the UI. This example illustrates the Model-View-Controller design pattern.

When you click **Checkout**, the rules are then fired in the following way:

1. Method **CheckOutCallBack.checkout()** is called (eventually) by the Swing class waiting for a user to click **Checkout**. This inserts the data from the **TableModel** object (upper-right corner of the UI) into the KIE session working memory. The method then fires the rules.

2. The **"Explode Cart"** rule is the first to fire, with the **auto-focus** attribute set to **true**. The rule loops through all of the products in the cart, ensures that the products are in the working memory, and then gives the **"show Items"** and **"evaluate"** agenda groups the option to fire. The rules in these groups add the contents of the cart to the text area (bottom of the UI), evaluate if you are eligible for free fish food, and determine whether to ask if you want to buy a fish tank.

Figure 7.16. Fish tank qualification
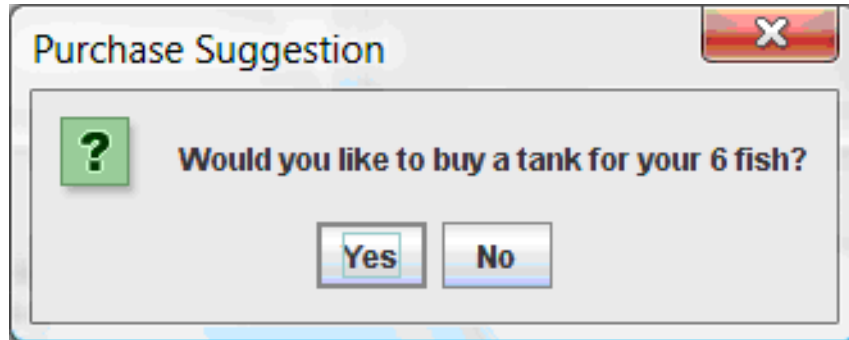
3. The **"do checkout"** rule is the next to fire because no other agenda group currently has focus and because it is part of the default **MAIN** agenda group. This rule always calls the **doCheckout()** function, which asks you if you want to check out.

4. The **doCheckout()** function sets the focus to the **"checkout"** agenda group, giving the rules in that group the option to fire.

5. The rules in the **"checkout"** agenda group display the contents of the cart and apply the appropriate discount.

6. Swing then waits for user input to either select more products (and cause the rules to fire again) or to close the UI.

Figure 7.17. Pet Store example GUI after all rules have fired
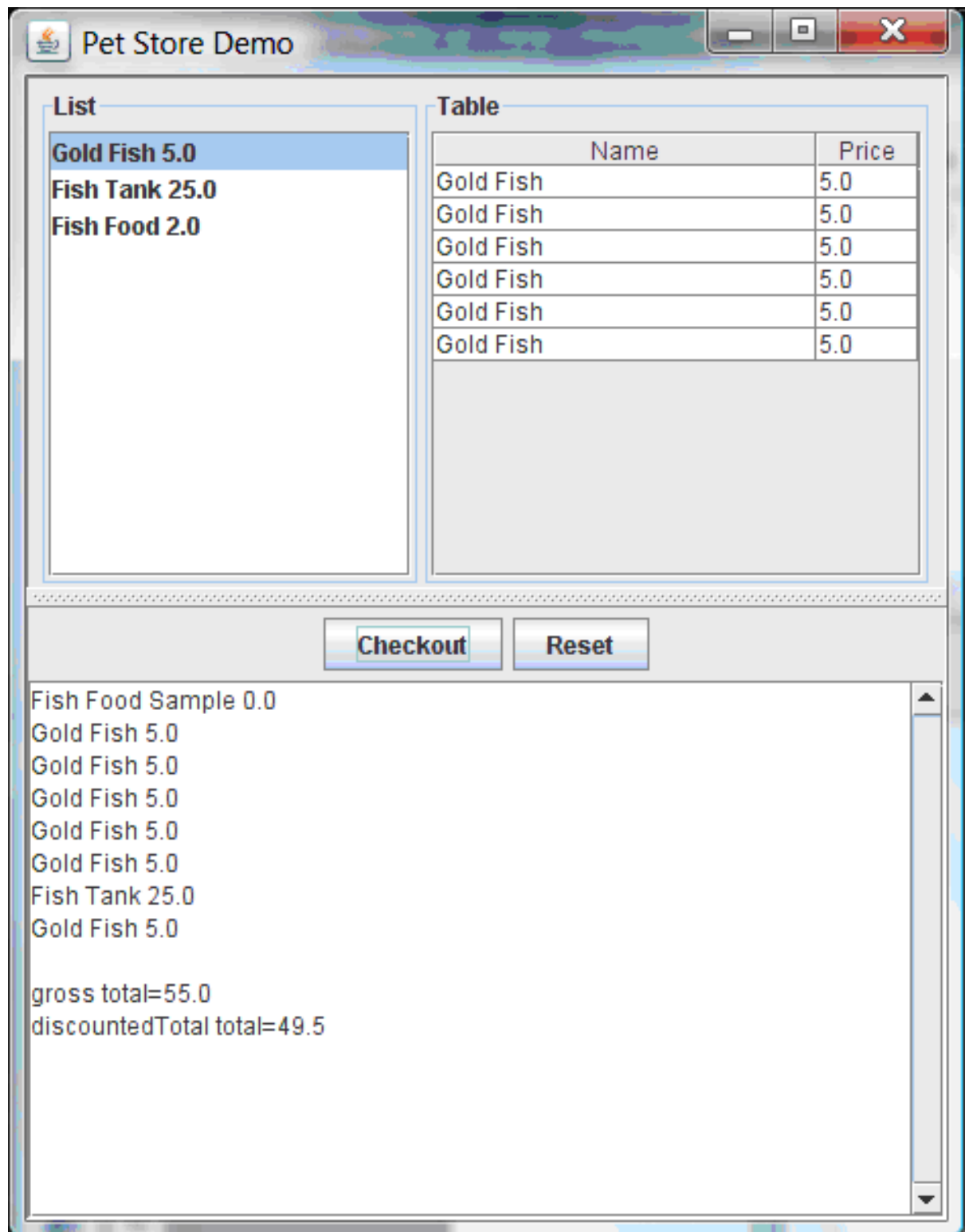
You can add more **System.out** calls to demonstrate this flow of events in your IDE console:

System.out output in the IDE console

> Adding free Fish Food Sample to cart
> SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes

7.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE)

The Honest Politician example decision set demonstrates the concept of truth maintenance with logical insertions and the use of salience in rules.

The following is an overview of the Honest Politician example:

- Name: **honestpolitician**

- Main class: **org.drools.examples.honestpolitician.HonestPoliticianExample** (in **src/main/java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.honestpolitician.HonestPolitician.drl** (in **src/main/resources**)

- Objective: Demonstrates the concept of truth maintenance based on the logical insertion of facts and the use of salience in rules

The basic premise of the Honest Politician example is that an object can only exist while a statement is true. A rule consequence can logically insert an object with the **insertLogical()** method. This means the object remains in the KIE session working memory as long as the rule that logically inserted it remains true. When the rule is no longer true, the object is automatically retracted.

In this example, rule execution causes a group of politicians to change from being honest to being dishonest as a result of a corrupt corporation. As each politician is evaluated, they start out with their honesty attribute being set to **true**, but a rule fires that makes the politicians no longer honest. As they switch their state from being honest to dishonest, they are then removed from the working memory. The rule salience notifies the decision engine how to prioritize any rules that have a salience defined for them, otherwise utilizing the default salience value of **0**. Rules with a higher salience value are given higher priority when ordered in the activation queue.

Politician and Hope classes

The sample class **Politician** in the example is configured for an honest politician. The **Politician** class is made up of a String item **name** and a Boolean item **honest**:

Politician class

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

The **Hope** class determines if a **Hope** object exists. This class has no meaningful members, but is present in the working memory as long as society has hope.

Hope class

```
public class Hope {

    public Hope() {

    }
}
```

Rule definitions for politician honesty

In the Honest Politician example, when at least one honest politician exists in the working memory, the **"We have an honest Politician"** rule logically inserts a new **Hope** object. As soon as all politicians become dishonest, the **Hope** object is automatically retracted. This rule has a **salience** attribute with a value of **10** to ensure that it fires before any other rule, because at that stage the **"Hope is Dead"** rule is true.

Rule "We have an honest politician"

```
rule "We have an honest Politician"
  salience 10
 when
  exists( Politician( honest == true ) )
 then
  insertLogical( new Hope() );
end
```

As soon as a **Hope** object exists, the **"Hope Lives"** rule matches and fires. This rule also has a **salience** value of **10** so that it takes priority over the **"Corrupt the Honest"** rule.

Rule "Hope Lives"

```
rule "Hope Lives"
  salience 10
 when
  exists( Hope() )
 then
  System.out.println("Hurrah!!! Democracy Lives");
end
```

Initially, four honest politicians exist so this rule has four activations, all in conflict. Each rule fires in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted, no politicians have the property **honest == true**. The rule **"We have an honest Politician"** is no longer true and the object it logically inserted (due to the last execution of **new Hope()**) is automatically retracted.

Rule "Corrupt the Honest"

```
rule "Corrupt the Honest"
  when
   politician : Politician( honest == true )
   exists( Hope() )
  then
   System.out.println( "I'm an evil corporation and I have corrupted " + politician.getName() );
   modify ( politician ) { honest = false };
 end
```

With the **Hope** object automatically retracted through the truth maintenance system, the conditional element **not** applied to **Hope** is no longer true so that the **"Hope is Dead"** rule matches and fires.

Rule "Hope is Dead"

```
rule "Hope is Dead"
  when
```

```
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
  end
```

Example execution and audit trail

In the **HonestPoliticianExample.java** class, the four politicians with the honest state set to **true** are inserted for evaluation against the defined business rules:

HonestPoliticianExample.java class execution

```java
public static void execute( KieContainer kc ) {
    KieSession ksession = kc.newKieSession("HonestPoliticianKS");

    final Politician p1 = new Politician( "President of Umpa Lumpa", true );
    final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
    final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
    final Politician p4 = new Politician( "Omnipotence Om", true );

    ksession.insert( p1 );
    ksession.insert( p2 );
    ksession.insert( p3 );
    ksession.insert( p4 );

    ksession.fireAllRules();

    ksession.dispose();
}
```

To execute the example, run the **org.drools.examples.honestpolitician.HonestPoliticianExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

The output shows that, while there is at least one honest politician, democracy lives. However, as each politician is corrupted by some corporation, all politicians become dishonest, and democracy is dead.

To better understand the execution flow of this example, you can modify the **HonestPoliticianExample.java** class to include a **DebugRuleRuntimeEventListener** listener and an audit logger to view execution details:

HonestPoliticianExample.java class with an audit logger

```java
package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
```

```
import org.kie.api.event.rule.DebugAgendaEventListener; 1
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
     KieServices ks = KieServices.Factory.get(); 2
    //ks = KieServices.Factory.get();
        KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        //execute( kc );
        execute( ks, kc); 3
    }

    public static void execute( KieServices ks, KieContainer kc ) { 4
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        // The application can also setup listeners 5
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger( ksession, "./target/honestpolitician" ); 6

        ksession.fireAllRules();

        ksession.dispose();
    }

}
```

1 Adds to your imports the packages that handle the **DebugAgendaEventListener** and **DebugRuleRuntimeEventListener**

2 Creates a **KieServices Factory** and a **ks** element to produce the logs because this audit log is not available at the **KieContainer** level

3 Modifies the **execute** method to use both **KieServices** and **KieContainer**

4 Modifies the **execute** method to pass in **KieServices** in addition to the **KieContainer**

5　Creates the listeners

6　Builds the log that can be passed into the debug view or **Audit View** or your IDE after executing of the rules

When you run the Honest Politician with this modified logging capability, you can load the audit log file from **target/honestpolitician.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows the flow of executions, insertions, and retractions as defined in the example classes and rules:

Figure 7.18. Honest Politician example Audit View
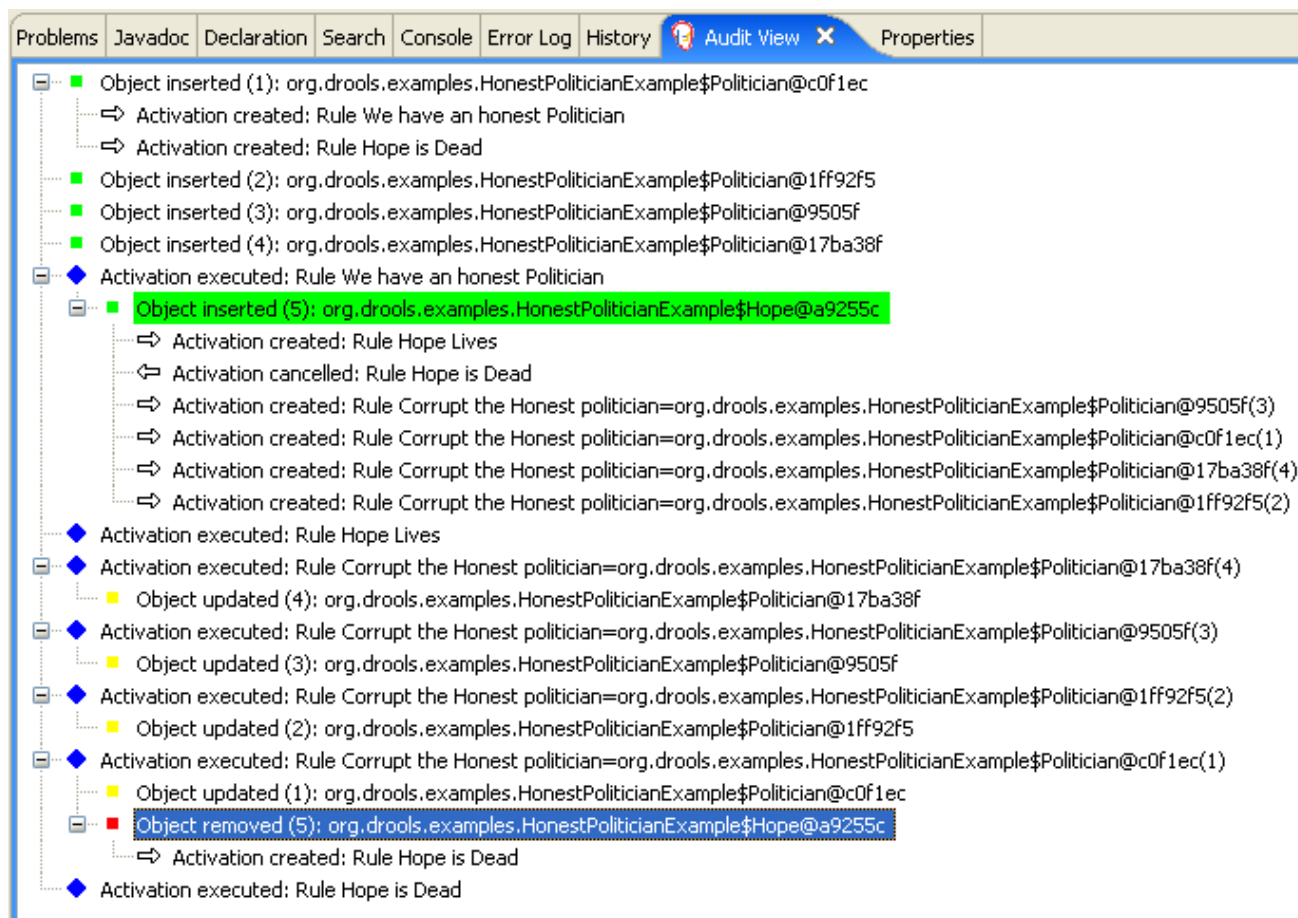
When the first politician is inserted, two activations occur. The rule **"We have an honest Politician"** is activated only one time for the first inserted politician because it uses an **exists** conditional element, which matches when at least one politician is inserted. The rule **"Hope is Dead"** is also activated at this stage because the **Hope** object is not yet inserted. The rule **"We have an honest Politician"** fires first because it has a higher **salience** value than the rule **"Hope is Dead"**, and inserts the **Hope** object (highlighted in green). The insertion of the **Hope** object activates the rule **"Hope Lives"** and deactivates the rule **"Hope is Dead"**. The insertion also activates the rule **"Corrupt the Honest"** for each inserted honest politician. The rule **"Hope Lives"** is executed and prints **"Hurrah!!! Democracy Lives"**.

Next, for each politician, the rule **"Corrupt the Honest"** fires, printing **"I'm an evil corporation and I have corrupted X"**, where **X** is the name of the politician, and modifies the politician honesty value to **false**. When the last honest politician is corrupted, **Hope** is automatically retracted by the truth maintenance system (highlighted in blue). The green highlighted area shows the origin of the currently selected blue highlighted area. After the **Hope** fact is retracted, the rule **"Hope is dead"** fires, printing **"We are all Doomed!!! Democracy is Dead"**.

7.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION)

The Sudoku example decision set, based on the popular number puzzle Sudoku, demonstrates how to use rules in Red Hat Decision Manager to find a solution in a large potential solution space based on various constraints. This example also shows how to integrate Red Hat Decision Manager rules into a graphical user interface (GUI), in this case a Swing-based desktop application, and how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Sudoku example:

- Name: **sudoku**

- Main class: **org.drools.examples.sudoku.SudokuExample** (in **src**/**main**/**java**)

- Module: **drools-examples**

- Type: Java application

- Rule files: **org.drools.examples.sudoku.*.drl** (in **src**/**main**/**resources**)

- Objective: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 only one time. The puzzle setter provides a partially completed grid and the puzzle solver's task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number, it must be unique in its particular 3x3 zone, row, and column. This Sudoku example decision set uses Red Hat Decision Manager rules to solve Sudoku puzzles from a range of difficulty levels, and to attempt to resolve flawed puzzles that contain invalid entries.

Sudoku example execution and interaction

Similar to other Red Hat Decision Manager decision examples, you execute the Sudoku example by running the **org.drools.examples.sudoku.SudokuExample** class as a Java application in your IDE.

When you execute the Sudoku example, the **Drools Sudoku Example** GUI window appears. This window contains an empty grid, but the program comes with various grids stored internally that you can load and solve.

Click **File → Samples → Simple** to load one of the examples. Notice that all buttons are disabled until a grid is loaded.

Figure 7.19. Sudoku example GUI after launch
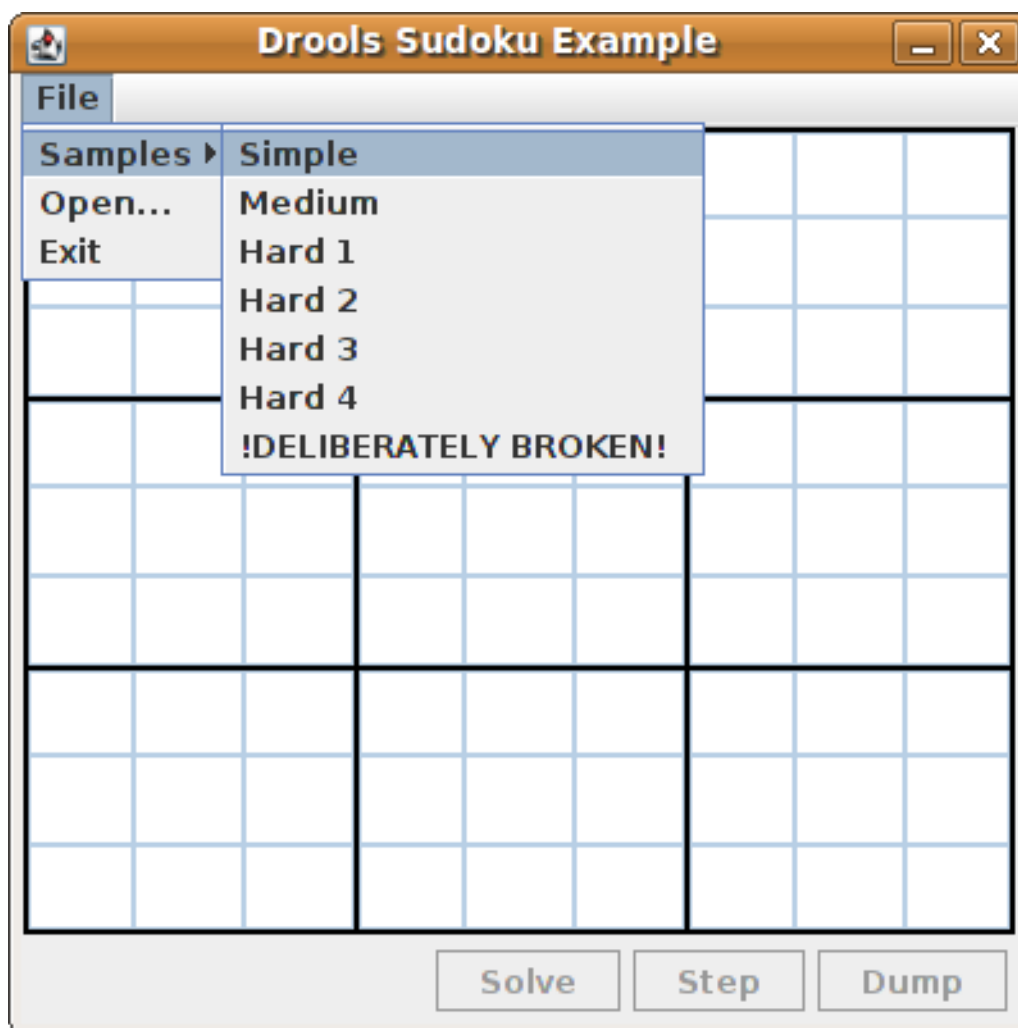
When you load the **Simple** example, the grid is filled according to the puzzle's initial state.

Figure 7.20. Sudoku example GUI after loading Simple sample
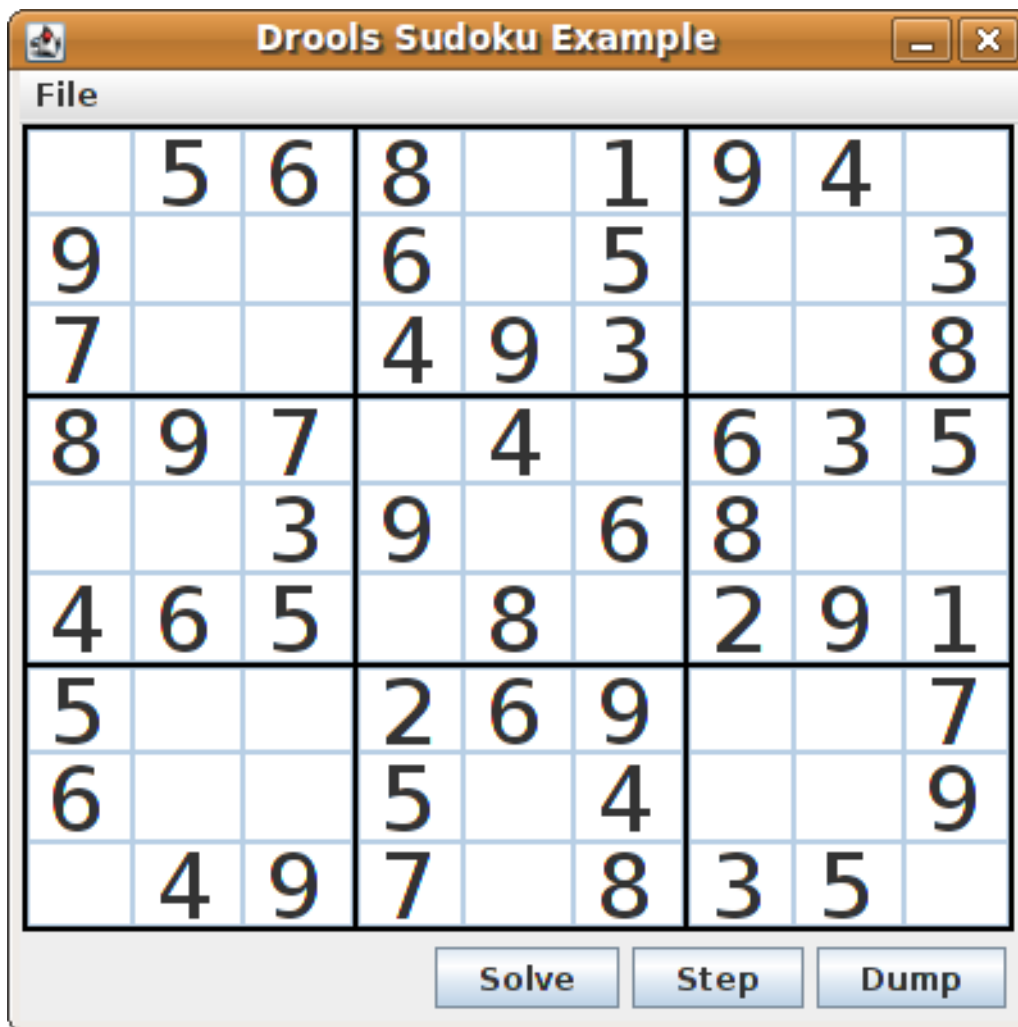
Choose from the following options:

- Click **Solve** to fire the rules defined in the Sudoku example that fill out the remaining values and that make the buttons inactive again.

Figure 7.21. Simple sample solved

- Click **Step** to see the next digit found by the rule set. The console window in your IDE displays detailed information about the rules that are executing to solve the step.

Step execution output in the IDE console

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```
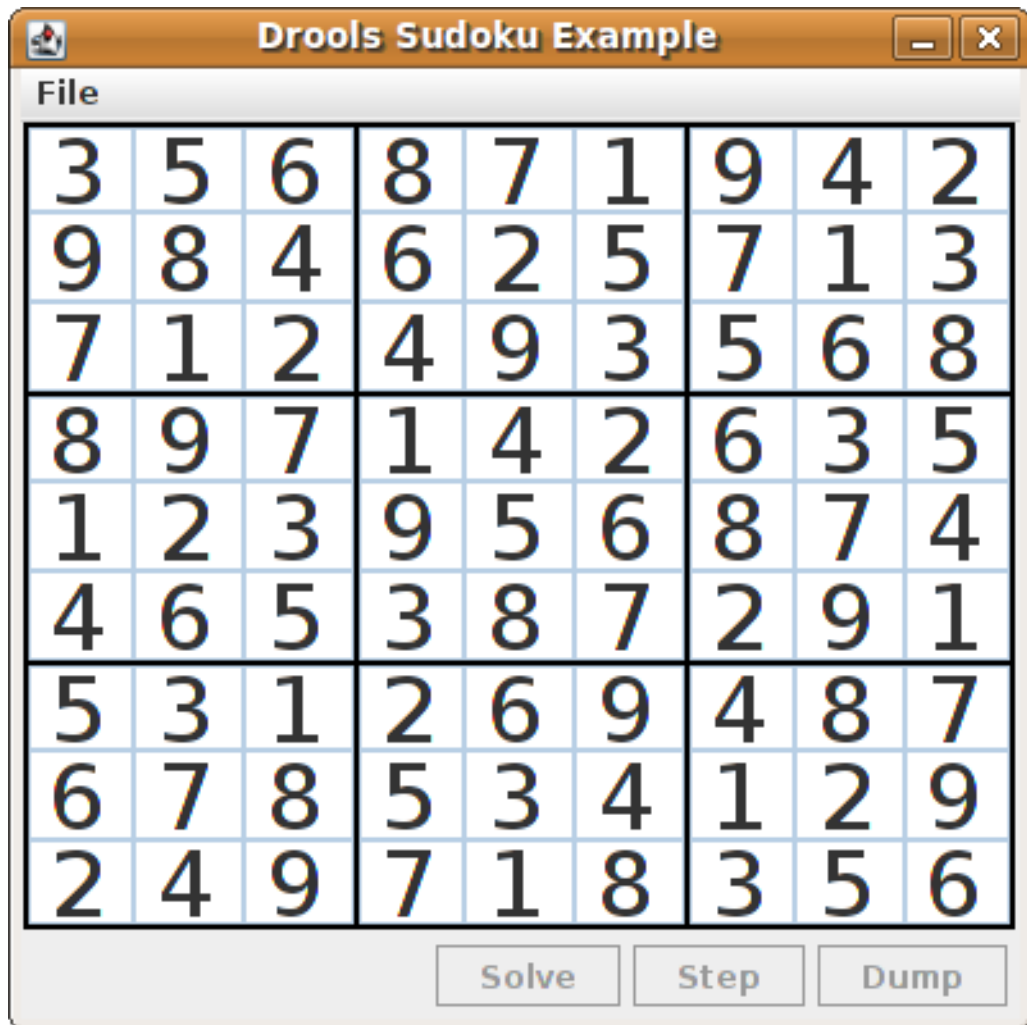
- Click **Dump** to see the state of the grid, with cells showing either the established value or the remaining possibilities.

Dump execution output in the IDE console

```
      Col: 0    Col: 1    Col: 2    Col: 3    Col: 4    Col: 5    Col: 6    Col: 7    Col: 8
Row 0:  123456789  --- 5 ---  --- 6 ---  --- 8 ---  123456789  --- 1 ---  --- 9 ---  --- 4 ---
123456789
Row 1:  --- 9 ---  123456789  123456789  --- 6 ---  123456789  --- 5 ---  123456789
123456789  --- 3 ---
Row 2:  --- 7 ---  123456789  123456789  --- 4 ---  --- 9 ---  --- 3 ---  123456789  123456789
--- 8 ---
```

```
Row 3:  --- 8 ---  --- 9 ---  --- 7 ---  123456789  --- 4 ---  123456789  --- 6 ---  --- 3 ---  --- 5 ---
Row 4:  123456789  123456789  --- 3 ---  --- 9 ---  123456789  --- 6 ---  --- 8 ---  123456789
123456789
Row 5:  --- 4 ---  --- 6 ---  --- 5 ---  123456789  --- 8 ---  123456789  --- 2 ---  --- 9 ---  --- 1 ---
Row 6:  --- 5 ---  123456789  123456789  --- 2 ---  --- 6 ---  --- 9 ---  123456789  123456789
--- 7 ---
Row 7:  --- 6 ---  123456789  123456789  --- 5 ---  123456789  --- 4 ---  123456789
123456789  --- 9 ---
Row 8:  123456789  --- 4 ---  --- 9 ---  --- 7 ---  123456789  --- 8 ---  --- 3 ---  --- 5 ---
123456789
```

The Sudoku example includes a deliberately broken sample file that the rules defined in the example can resolve.

Click **File → Samples → !DELIBERATELY BROKEN!** to load the broken sample. The grid starts with some issues, for example, the value **5** appears two times in the first row, which is not allowed.

Figure 7.22. Broken Sudoku example initial state
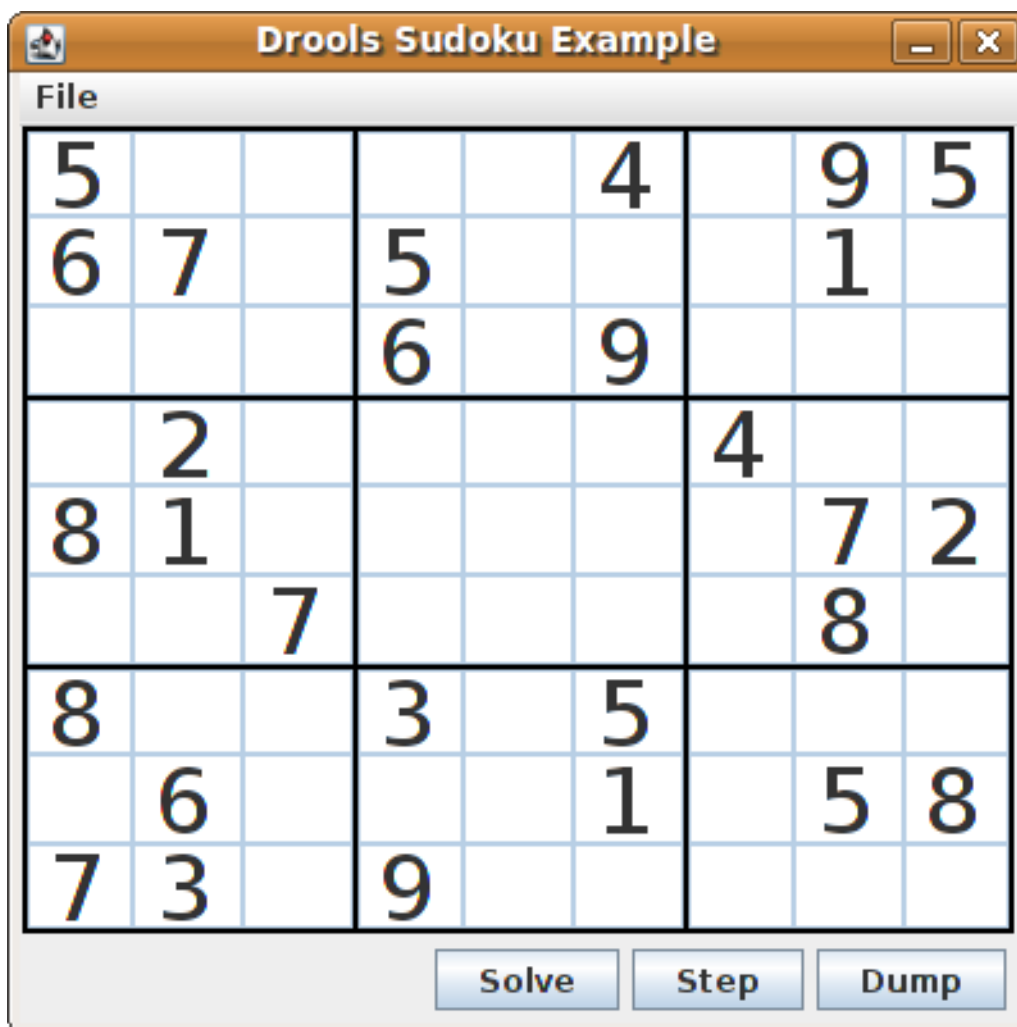
Click **Solve** to apply the solving rules to this invalid grid. The associated solving rules in the Sudoku example detect the issues in the sample and attempts to solve the puzzle as far as possible. This process does not complete and leaves some cells empty.
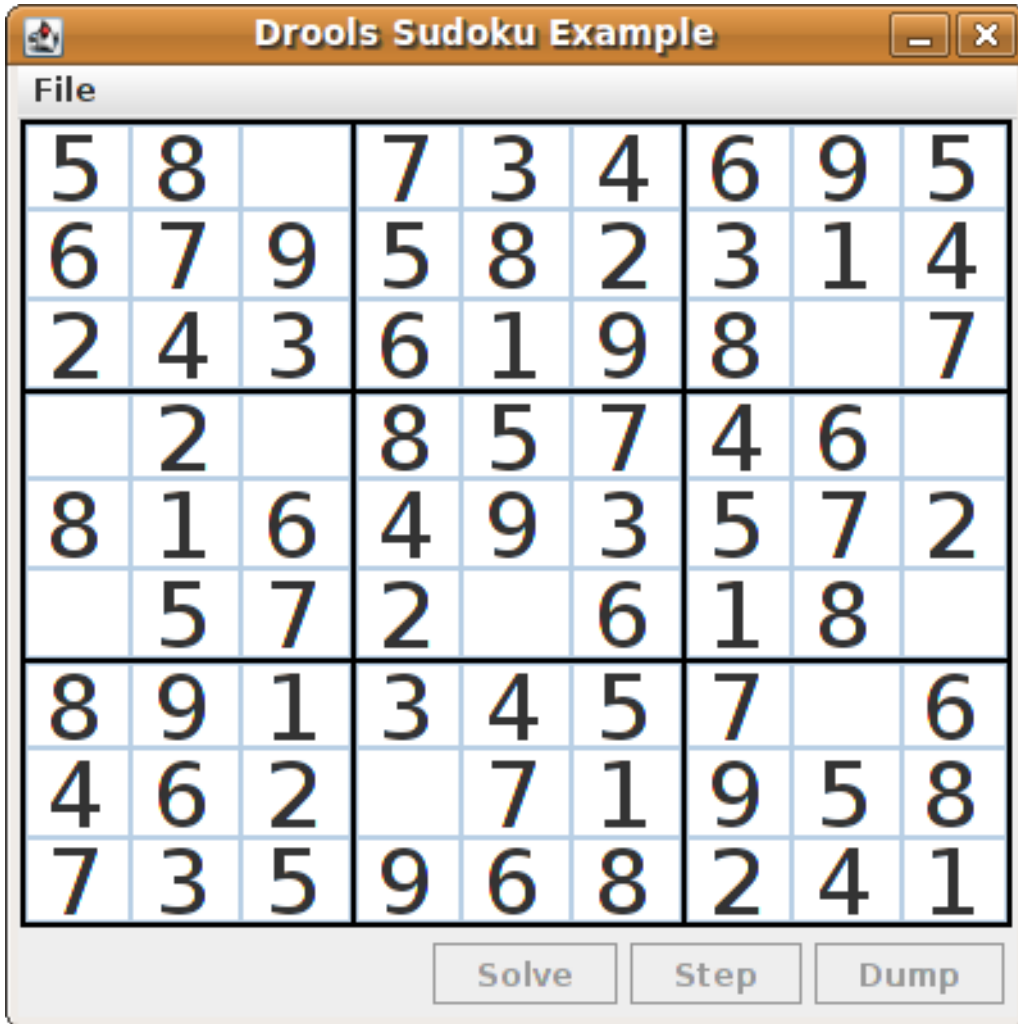
The solving rule activity is displayed in the IDE console window:

Detected issues in the broken sample

> cell [0,8]: 5 has a duplicate in row 0
> cell [0,0]: 5 has a duplicate in row 0
> cell [6,0]: 8 has a duplicate in col 0
> cell [4,0]: 8 has a duplicate in col 0
> Validation complete.

Figure 7.23. Broken sample solution attempt

The sample Sudoku files labeled **Hard** are more complex and the solving rules might not be able to solve them. The unsuccessful solution attempt is displayed in the IDE console window:

Hard sample unresolved

> Validation complete.
> ...
> Sorry - can't solve this grid.

The rules that work to solve the broken sample implement standard solving techniques based on the sets of values that are still candidates for a cell. For example, if a set contains a single value, then this is the value for the cell. For a single occurrence of a value in one of the groups of nine cells, the rules insert a fact of type **Setting** with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to and the value is retracted.

Other rules in the example reduce the permissible values for some cells. The rules **"naked pair"**, **"hidden pair in row"**, **"hidden pair in column"**, and **"hidden pair in square"** eliminate possibilities but do not establish solutions. The rules **"X-wings in rows"**, "`X-wings in columns"`, **"intersection removal**

row", and **"intersection removal column"** perform more sophisticated eliminations.

Sudoku example classes

The package **org.drools.examples.sudoku.swing** contains the following core set of classes that implement a framework for Sudoku puzzles:

- The **SudokuGridModel** class defines an interface that is implemented to store a Sudoku puzzle as a 9x9 grid of **Cell** objects.

- The **SudokuGridView** class is a Swing component that can visualize any implementation of the **SudokuGridModel** class.

- The **SudokuGridEvent** and **SudokuGridListener** classes communicate state changes between the model and the view. Events are fired when a cell value is resolved or changed.

- The **SudokuGridSamples** class provides partially filled Sudoku puzzles for demonstration purposes.
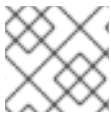
> **NOTE**
>
> This package does not have any dependencies on Red Hat Decision Manager libraries.

The package **org.drools.examples.sudoku** contains the following core set of classes that implement the elementary **Cell** object and its various aggregations:

- The **CellFile** class, with subtypes **CellRow**, **CellCol**, and **CellSqr**, all of which are subtypes of the **CellGroup** class.

- The **Cell** and **CellGroup** subclasses of **SetOfNine**, which provides a property **free** with the type **Set<Integer>**. For a **Cell** class, the set represents the individual candidate set. For a **CellGroup** class, the set is the union of all candidate sets of its cells (the set of digits that still need to be allocated).
  In the Sudoku example are 81 **Cell** and 27 **CellGroup** objects and a linkage provided by the **Cell** properties **cellRow**, **cellCol**, and **cellSqr**, and by the **CellGroup** property **cells** (a list of **Cell** objects). With these components, you can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.

- The **Setting** class is used to trigger the operations that accompany the allocation of a value. The presence of a **Setting** fact is used in all rules that detect a new situation in order to avoid reactions to inconsistent intermediary states.

- The **Stepping** class is used in a low priority rule to execute an emergency halt when a **"Step"** does not terminate regularly. This behavior indicates that the program cannot solve the puzzle.

- The main class **org.drools.examples.sudoku.SudokuExample** implements a Java application combining all of these components.

Sudoku validation rules (validate.drl)

The **validate.drl** file in the Sudoku example contains validation rules that detect duplicate numbers in cell groups. They are combined in a **"validate"** agenda group that enables the rules to be explicitly activated after a user loads the puzzle.

The **when** conditions of the three rules **"duplicate in cell …"** all function in the following ways:

- The first condition in the rule locates a cell with an allocated value.

- The second condition in the rule pulls in any of the three cell groups to which the cell belongs.

- The final condition finds a cell (other than the first one) with the same value as the first cell and in the same row, column, or square, depending on the rule.

Rules "duplicate in cell ..."

```
rule "duplicate in cell row"
  when
    $c: Cell( $v: value != null )
    $cr: CellRow( cells contains $c )
    exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
end

rule "duplicate in cell col"
  when
    $c: Cell( $v: value != null )
    $cc: CellCol( cells contains $c )
    exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
end

rule "duplicate in cell sqr"
  when
    $c: Cell( $v: value != null )
    $cs: CellSqr( cells contains $c )
    exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has duplicate in its square of nine." );
end
```

The rule **"terminate group"** is the last to fire. This rule prints a message and stops the sequence.

Rule "terminate group"

```
rule "terminate group"
    salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
end
```

Sudoku solving rules (sudoku.drl)
The **sudoku.drl** file in the Sudoku example contains three types of rules: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

The rules **"set a value"**, **"eliminate a value from Cell"**, and **"retract setting"** depend on the presence of a **Setting** object. The first rule handles the assignment to the cell and the operations for removing the value from the **free** sets of the three groups of the cell. This group also reduces a counter that, when zero, returns control to the Java application that has called **fireUntilHalt()**.

The purpose of the rule **"eliminate a value from Cell"** is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, the rule **"retract setting"** retracts the triggering **Setting** fact.

Rules "set a value", "eliminate a value from a Cell", and "retract setting"

```
// A Setting object is inserted to define the value of a Cell.
// Rule for updating the cell and all cell groups that contain it
rule "set a value"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // A matching Cell, with no value set
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,
          $cr: cellRow, $cc: cellCol, $cs: cellSqr )

    // Count down
    $ctr: Counter( $count: count )
  then
    // Modify the Cell by setting its value.
    modify( $c ){ setValue( $v ) }
    // System.out.println( "set cell " + $c.toString() );
    modify( $cr ){ blockValue( $v ) }
    modify( $cc ){ blockValue( $v ) }
    modify( $cs ){ blockValue( $v ) }
    modify( $ctr ){ setCount( $count - 1 ) }
end

// Rule for removing a value from all cells that are siblings
// in one of the three cell groups
rule "eliminate a value from Cell"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )

    // For all Cells that are associated with the updated cell
    $c: Cell( free contains $v ) from $exCells
  then
    // System.out.println( "clear " + $v + " from cell " + $c.posAsString()  );
    // Modify a related Cell by blocking the assigned value.
    modify( $c ){ blockValue( $v ) }
end

// Rule for eliminating the Setting fact
rule "retract setting"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    $c: Cell( rowNo == $rn, colNo == $cn, value == $v )
```

```
    // This is the negation of the last pattern in the previous rule.
    // Now the Setting fact can be safely retracted.
    not( $x: Cell( free contains $v )
        and
        Cell( this == $c, exCells contains $x ) )
  then
    // System.out.println( "done setting cell " + $c.toString() );
    // Discard the Setter fact.
    delete( $s );
    // Sudoku.sudoku.consistencyCheck();
  end
```

Two solving rules detect a situation where an allocation of a number to a cell is possible. The rule **"single"** fires for a **Cell** with a candidate set containing a single number. The rule **"hidden single"** fires when no cell exists with a single candidate, but when a cell exists containing a candidate, this candidate is absent from all other cells in one of the three groups to which the cell belongs. Both rules create and insert a **Setting** fact.

Rules "single" and "hidden single"

```
    // Detect a set of candidate values with cardinality 1 for some Cell.
    // This is the value to be set.
    rule "single"
      when
        // Currently no setting underway
        not Setting()

        // One element in the "free" set
        $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )
      then
        Integer i = $c.getFreeValue();
        if (explain) System.out.println( "single " + i + " at " + $c.posAsString() );
        // Insert another Setter fact.
        insert( new Setting( $rn, $cn, i ) );
    end

    // Detect a set of candidate values with a value that is the only one
    // in one of its groups. This is the value to be set.
    rule "hidden single"
      when
        // Currently no setting underway
        not Setting()
        not Cell( freeCount == 1 )

        // Some integer
        $i: Integer()

        // The "free" set contains this number
        $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )

        // A cell group contains this cell $c.
        $cg: CellGroup( cells contains $c )
        // No other cell from that group contains $i.
        not ( Cell( this != $c, free contains $i ) from $cg.getCells() )
      then
        if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
```

```
      // Insert another Setter fact.
      insert( new Setting( $rn, $cn, $i ) );
   end
```

Rules from the largest group, either individually or in groups of two or three, implement various solving techniques used for solving Sudoku puzzles manually.

The rule **"naked pair"** detects identical candidate sets of size **2** in two cells of a group. These two values may be removed from all other candidate sets of that group.

Rule "naked pair"

```
   // A "naked pair" is two cells in some cell group with their sets of
   // permissible values being equal with cardinality 2. These two values
   // can be removed from all other candidate lists in the group.
   rule "naked pair"
     when
       // Currently no setting underway
       not Setting()
       not Cell( freeCount == 1 )

       // One cell with two candidates
       $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )

       // The containing cell group
       $cg: CellGroup( freeCount > 2, cells contains $c1 )

       // Another cell with two candidates, not the one we already have
       $c2: Cell( this != $c1, free == $f1 /*** , rowNo >= $rn1, colNo >= $cn1 ***/ ) from $cg.cells

       // Get one of the "naked pair".
       Integer( $v: intValue ) from $c1.getFree()

       // Get some other cell with a candidate equal to one from the pair.
       $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
     then
       if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair
   at " + $c1.posAsString() + " and " + $c2.posAsString() );
       // Remove the value.
       modify( $c3 ){ blockValue( $v ) }
   end
```

The three rules **"hidden pair in …"** functions similarly to the rule **"naked pair"**. These rules detect a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of the group. This means that all other candidates can be eliminated from the two cells harboring the hidden pair.

Rules "hidden pair in ..."

```
   // If two cells within the same cell group contain candidate sets with more than
   // two values, with two values being in both of them but in none of the other
   // cells, then we have a "hidden pair". We can remove all other candidates from
   // these two cells.
   rule "hidden pair in row"
     when
```

```
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Establish a pair of Integer facts.
    $i1: Integer()
    $i2: Integer( this > $i1 )

    // Look for a Cell with these two among its candidates. (The upper bound on
    // the number of candidates avoids a lot of useless work during startup.)
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellRow: cellRow )

    // Get another one from the same row, with the same pair among its candidates.
    $c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free contains $i1 && contains $i2 )

    // Ascertain that no other cell in the group has one of these two values.
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellRow.getCells() )
  then
    if( explain) System.out.println( "hidden pair in row at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    // Set the candidate lists of these two Cells to the "hidden pair".
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellCol: cellCol )
    $c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellCol.getCells() )
  then
    if (explain) System.out.println( "hidden pair in column at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
          $cellSqr: cellSqr )
    $c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellSqr.getCells() )
  then
```

```
   if (explain) System.out.println( "hidden pair in square " + $c1.posAsString() + " and " +
$c2.posAsString() );
   modify( $c1 ){ blockExcept( $i1, $i2 ) }
   modify( $c2 ){ blockExcept( $i1, $i2 ) }
end
```

Two rules deal with **"X-wings"** in rows and columns. When only two possible cells for a value exist in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. When you follow the pattern sequence in one of these rules, notice how the conditions that are conveniently expressed by words such as **same** or **only** result in patterns with suitable constraints or that are prefixed with **not**.

Rules "X-wings in ..."

```
rule "X-wings in rows"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
            $ra: cellRow, $rano: rowNo,        $c1: cellCol,        $c1no: colNo )
    $cb1: Cell( freeCount > 1, free contains $i,
            $rb: cellRow, $rbno: rowNo > $rano,      cellCol == $c1 )
    not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )

    $ca2: Cell( freeCount > 1, free contains $i,
            cellRow == $ra, $c2: cellCol,        $c2no: colNo > $c1no )
    $cb2: Cell( freeCount > 1, free contains $i,
            cellRow == $rb,      cellCol == $c2 )
    not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )

    $cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
            freeCount > 1, free contains $i )
  then
    if (explain) {
       System.out.println( "X-wing with " + $i + " in rows " +
          $ca1.posAsString() + " - " + $cb1.posAsString() +
          $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
            $c1: cellCol, $c1no: colNo,        $ra: cellRow,        $rano: rowNo )
    $ca2: Cell( freeCount > 1, free contains $i,
            $c2: cellCol, $c2no: colNo > $c1no,      cellRow == $ra )
    not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

    $cb1: Cell( freeCount > 1, free contains $i,
```

```
              cellCol == $c1, $rb: cellRow,  $rbno: rowNo > $rano )
    $cb2: Cell( freeCount > 1, free contains $i,
            cellCol == $c2,     cellRow == $rb )
    not( Cell( this != $cb1 && != $cb2, free contains $i ) from $rb.getCells() )

    $cx: Cell( colNo == $c1no || == $c2no, rowNo != $rano && != $rbno,
            freeCount > 1, free contains $i )
  then
    if (explain) {
      System.out.println( "X-wing with " + $i + " in columns " +
          $ca1.posAsString() + " - " + $ca2.posAsString() +
          $cb1.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString()  );
    }
    modify( $cx ){ blockValue( $i ) }
  end
```

The two rules **"intersection removal …"** are based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column and can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside of the square and within the same cell file.

Rules "intersection removal …"

```
rule "intersection removal column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
    // Does not occur in another cell of the same square and a different column
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc )

    // A cell exists in the same column and another square containing this value.
    $cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr != $cs )
  then
    // Remove the value from that other cell.
    if (explain) {
      System.out.println( "column elimination due to " + $c.posAsString() +
                ": remove " + $i + " from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
  end

rule "intersection removal row"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    // Occurs in a Cell
    $c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
    // Does not occur in another cell of the same square and a different row.
    not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr )
```

```
   // A cell exists in the same row and another square containing this value.
   $cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr != $cs )
then
   // Remove the value from that other cell.
   if (explain) {
      System.out.println( "row elimination due to " + $c.posAsString() +
                    ": remove " + $i + " from " + $cx.posAsString() );
   }
   modify( $cx ){ blockValue( $i ) }
end
```

These rules are sufficient for many but not all Sudoku puzzles. To solve very difficult grids, the rule set requires more complex rules. (Ultimately, some puzzles can be solved only by trial and error.)

7.9. CONWAY'S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION)

The Conway's Game of Life example decision set, based on the famous cellular automaton by John Conway, demonstrates how to use ruleflow groups in rules to control rule execution. The example also demonstrates how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based implementation of Conway's Game of Life.

The following is an overview of the Conway's Game of Life (Conway) example:

- Name: **conway**

- Main classes: **org.drools.examples.conway.ConwayRuleFlowGroupRun**, **org.drools.examples.conway.ConwayAgendaGroupRun** (in **src/main/java**)

- Module: **droolsjbpm-integration-examples**

- Type: Java application

- Rule files: **org.drools.examples.conway.*.drl** (in **src/main/resources**)

- Objective: Demonstrates ruleflow groups and GUI integration

> **NOTE**
>
> The Conway's Game of Life example is separate from most of the other example decision sets in Red Hat Decision Manager and is located in **~/rhdm-7.6.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples** of the **Red Hat Decision Manager 7.6.0 Source Distribution** from the Red Hat Customer Portal.

In Conway's Game of Life, a user interacts with the game by creating an initial configuration or an advanced pattern with defined properties and then observing how the initial state evolves. The objective of the game is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells.

The following basic rules govern what the next generation looks like:

- If a live cell has fewer than two live neighbors, it dies of loneliness.

- If a live cell has more than three live neighbors, it dies from overcrowding.

- If a dead cell has exactly three live neighbors, it comes to life.

Any cell that does not meet any of those criteria is left as is for the next generation.

The Conway's Game of Life example uses Red Hat Decision Manager rules with **ruleflow-group** attributes to define the pattern implemented in the game. The example also contains a version of the decision set that achieves the same behavior using agenda groups. Agenda groups enable you to partition the decision engine agenda to provide execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

This overview does not explore the version of the Conway example using agenda groups. For more information about agenda groups, see the Red Hat Decision Manager example decision sets that specifically address agenda groups.

Conway example execution and interaction

Similar to other Red Hat Decision Manager decision examples, you execute the Conway ruleflow example by running the **org.drools.examples.conway.ConwayRuleFlowGroupRun** class as a Java application in your IDE.

When you execute the Conway example, the **Conway's Game of Life** GUI window appears. This window contains an empty grid, or "arena" where the life simulation takes place. Initially the grid is empty because no live cells are in the system yet.

Figure 7.24. Conway example GUI after launch
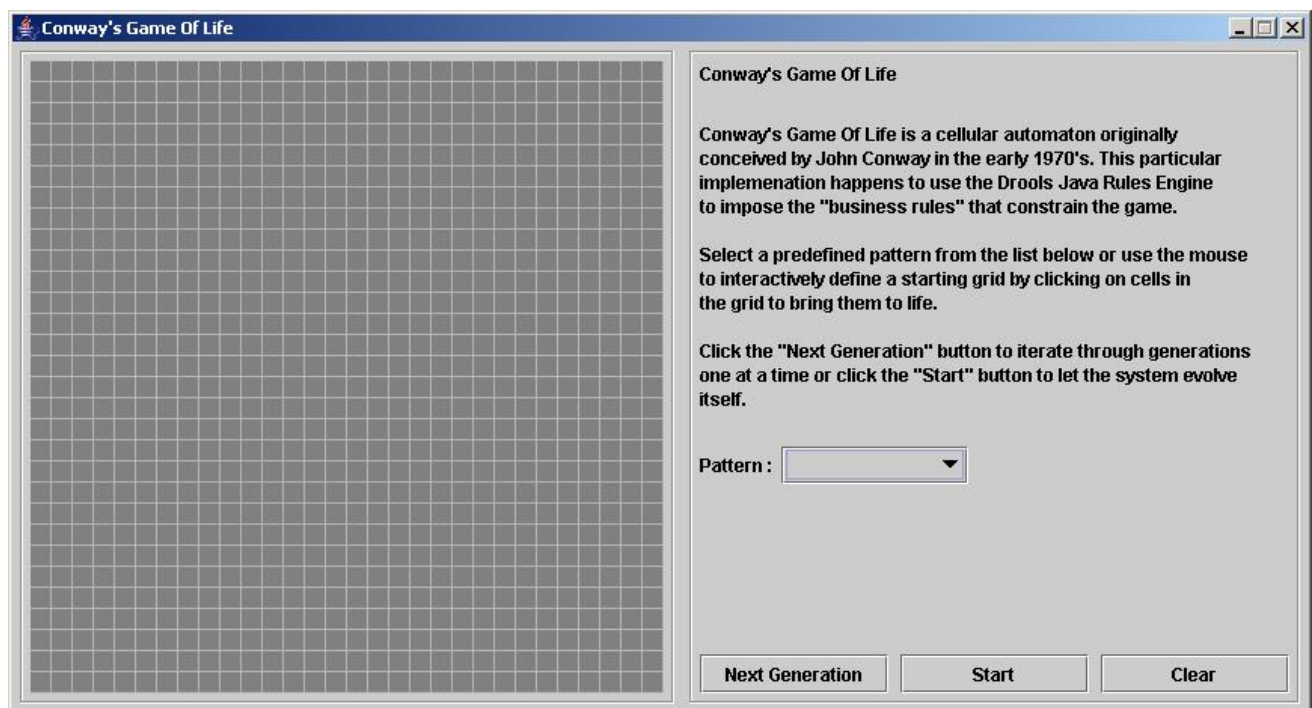
Select a predefined pattern from the **Pattern** drop-down menu and click **Next Generation** to click through each population generation. Each cell is either alive or dead, where live cells contain a green ball. As the population evolves from the initial pattern, cells live or die relative to neighboring cells, according to the rules of the game.

Figure 7.25. Generation evolution in Conway example
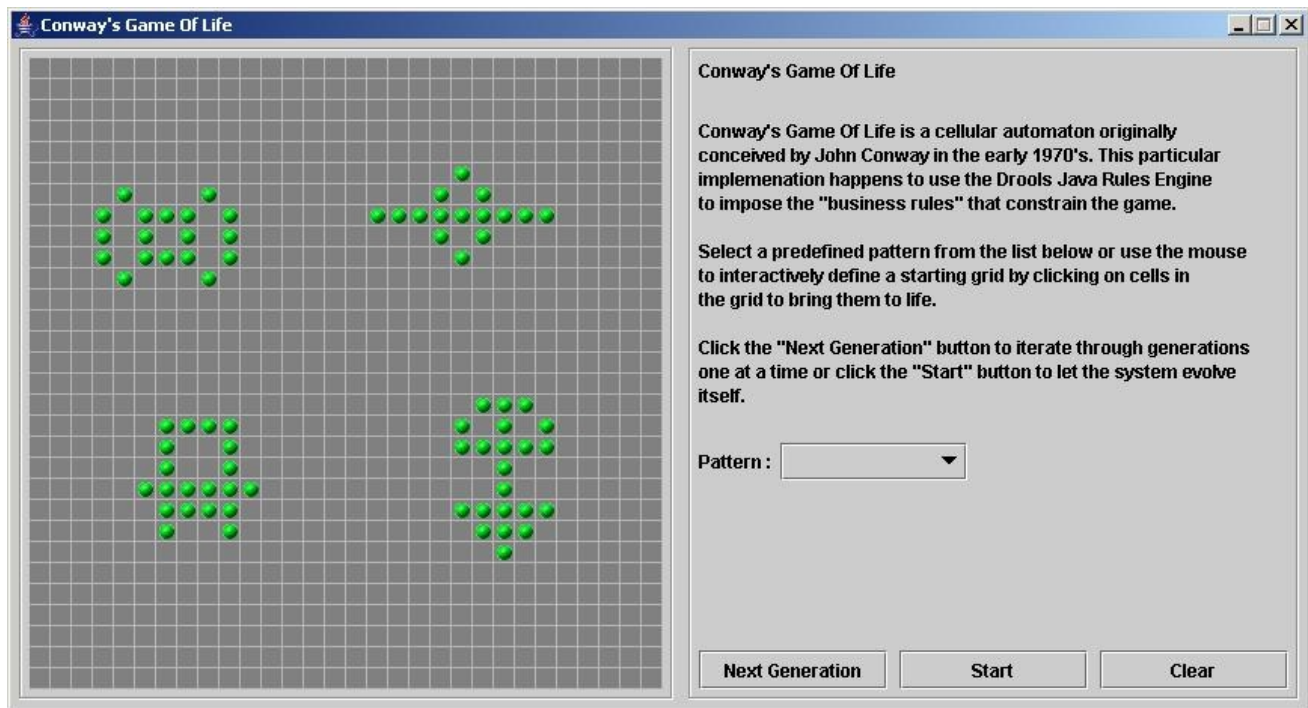
Neighbors include not only cells to the left, right, top, and bottom but also cells that are connected diagonally, so that each cell has a total of eight neighbors. Exceptions are the corner cells, which have only three neighbors, and the cells along the four borders, with five neighbors each.

You can manually intervene to create or kill cells by clicking the cell.

To run through an evolution automatically from the initial pattern, click **Start**.

Conway example rules with ruleflow groups

The rules in the **ConwayRuleFlowGroupRun** example use ruleflow groups to control rule execution. A ruleflow group is a group of rules associated by the **ruleflow-group** rule attribute. These rules can only fire when the group is activated. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group.

The Conway example uses the following ruleflow groups for rules:

- **"register neighbor"**

- **"evaluate"**

- **"calculate"**

- **"reset calculate"**

- **"birth"**

- **"kill"**

- **"kill all"**

All of the **Cell** objects are inserted into the KIE session and the **"register ..."** rules in the ruleflow group **"register neighbor"** are allowed to execute by the ruleflow process. This group of four rules creates **Neighbor** relations between some cell and its northeastern, northern, northwestern, and western neighbors.

This relation is bidirectional and handles the other four directions. Border cells do not require any special treatment. These cells are not paired with neighboring cells where there is not any.

By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Rules "register ..."

```
rule "register north east"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row  == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

rule "register north west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row  == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row  == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end
```
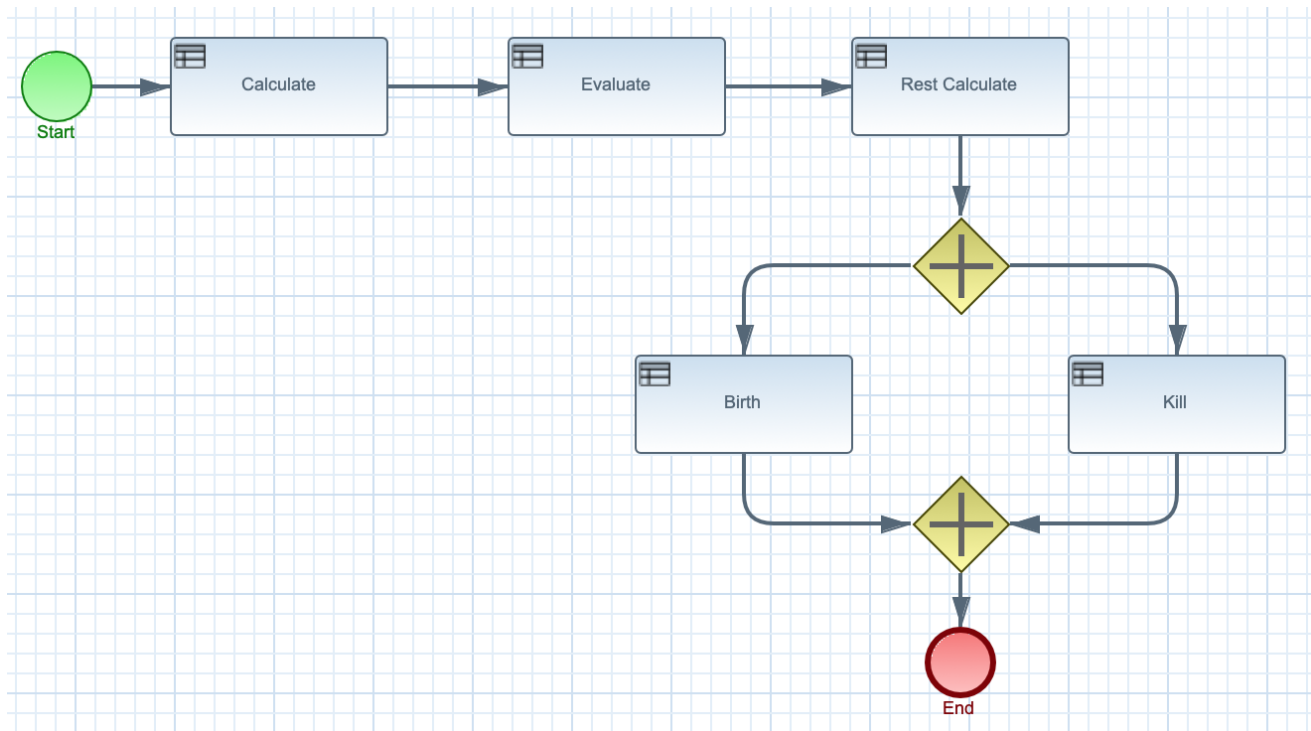
After all the cells are inserted, some Java code applies the pattern to the grid, setting certain cells to **Live**. Then, when the user clicks **Start** or **Next Generation**, the example executes the **Generation** ruleflow. This ruleflow manages all changes of cells in each generation cycle.

Figure 7.26. Generation ruleflow

The ruleflow process enters the **"evaluate"** ruleflow group and any active rules in the group can fire. The rules **"Kill the …"** and **"Give Birth"** in this group apply the game rules to birth or kill cells. The example uses the **phase** attribute to drive the reasoning of the **Cell** object by specific groups of rules. Typically, the phase is tied to a ruleflow group in the ruleflow process definition.

Notice that the example does not change the state of any **Cell** objects at this point because it must complete the full evaluation before those changes can be applied. The example sets the cell to a **phase** that is either **Phase.KILL** or **Phase.BIRTH**, which is used later to control actions applied to the **Cell** object.

Rules "Kill the …" and "Give Birth"

```
rule "Kill The Lonely"
   ruleflow-group "evaluate"
   no-loop
  when
   // A live cell has fewer than 2 live neighbors.
   theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE,
            phase == Phase.EVALUATE )
  then
   modify( theCell ){
      setPhase( Phase.KILL );
   }
 end

rule "Kill The Overcrowded"
   ruleflow-group "evaluate"
   no-loop
  when
   // A live cell has more than 3 live neighbors.
   theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE,
            phase == Phase.EVALUATE )
  then
```

```
    modify( theCell ){
        setPhase( Phase.KILL );
    }
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
  when
    // A dead cell has 3 live neighbors.
    theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,
             phase == Phase.EVALUATE )
  then
    modify( theCell ){
        theCell.setPhase( Phase.BIRTH );
    }
end
```

After all **Cell** objects in the grid have been evaluated, the example uses the **"reset calculate"** rule to clear any activations in the **"calculate"** ruleflow group. The example then enters a split in the ruleflow that enables the rules **"kill"** and **"birth"** to fire, if the ruleflow group is activated. These rules apply the state change.

Rules "reset calculate", "kill", and "birth"

```
rule "reset calculate"
    ruleflow-group "reset calculate"
  when
  then
    WorkingMemory wm = drools.getWorkingMemory();
    wm.clearRuleFlowGroup( "calculate" );
end

rule "kill"
    ruleflow-group "kill"
    no-loop
  when
    theCell: Cell( phase == Phase.KILL )
  then
    modify( theCell ){
        setCellState( CellState.DEAD ),
        setPhase( Phase.DONE );
    }
end

rule "birth"
    ruleflow-group "birth"
    no-loop
  when
    theCell: Cell( phase == Phase.BIRTH )
  then
    modify( theCell ){
        setCellState( CellState.LIVE ),
        setPhase( Phase.DONE );
    }
end
```

At this stage, several **Cell** objects have been modified with the state changed to either **LIVE** or **DEAD**. When a cell becomes live or dead, the example uses the **Neighbor** relation in the rules **"Calculate …"** to iterate over all surrounding cells, increasing or decreasing the **liveNeighbor** count. Any cell that has its count changed is also set to the **EVALUATE** phase to make sure it is included in the reasoning during the evaluation stage of the ruleflow process.

After the live count has been determined and set for all cells, the ruleflow process ends. If the user initially clicked **Start**, the decision engine restarts the ruleflow at that point. If the user initially clicked **Next Generation**, the user can request another generation.

Rules "Calculate …"

```
rule "Calculate Live"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.LIVE )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
        setPhase( Phase.EVALUATE );
    }
end

rule "Calculate Dead"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.DEAD )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
        setPhase( Phase.EVALUATE );
    }
end
```

7.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION)

The House of Doom example decision set demonstrates how the decision engine uses backward chaining and recursion to reach defined goals or subgoals in a hierarchical system.

The following is an overview of the House of Doom example:

- Name: **backwardchaining**

- Main class: **org.drools.examples.backwardchaining.HouseOfDoomMain** (in **src/main/java**)

- Module: **drools-examples**

- Type: Java application

- Rule file: **org.drools.examples.backwardchaining.BC-Example.drl** (in **src/main/resources**)

- **Objective**: Demonstrates backward chaining and recursion

A backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

In contrast, a forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 7.27. Rule evaluation logic using forward and backward chaining
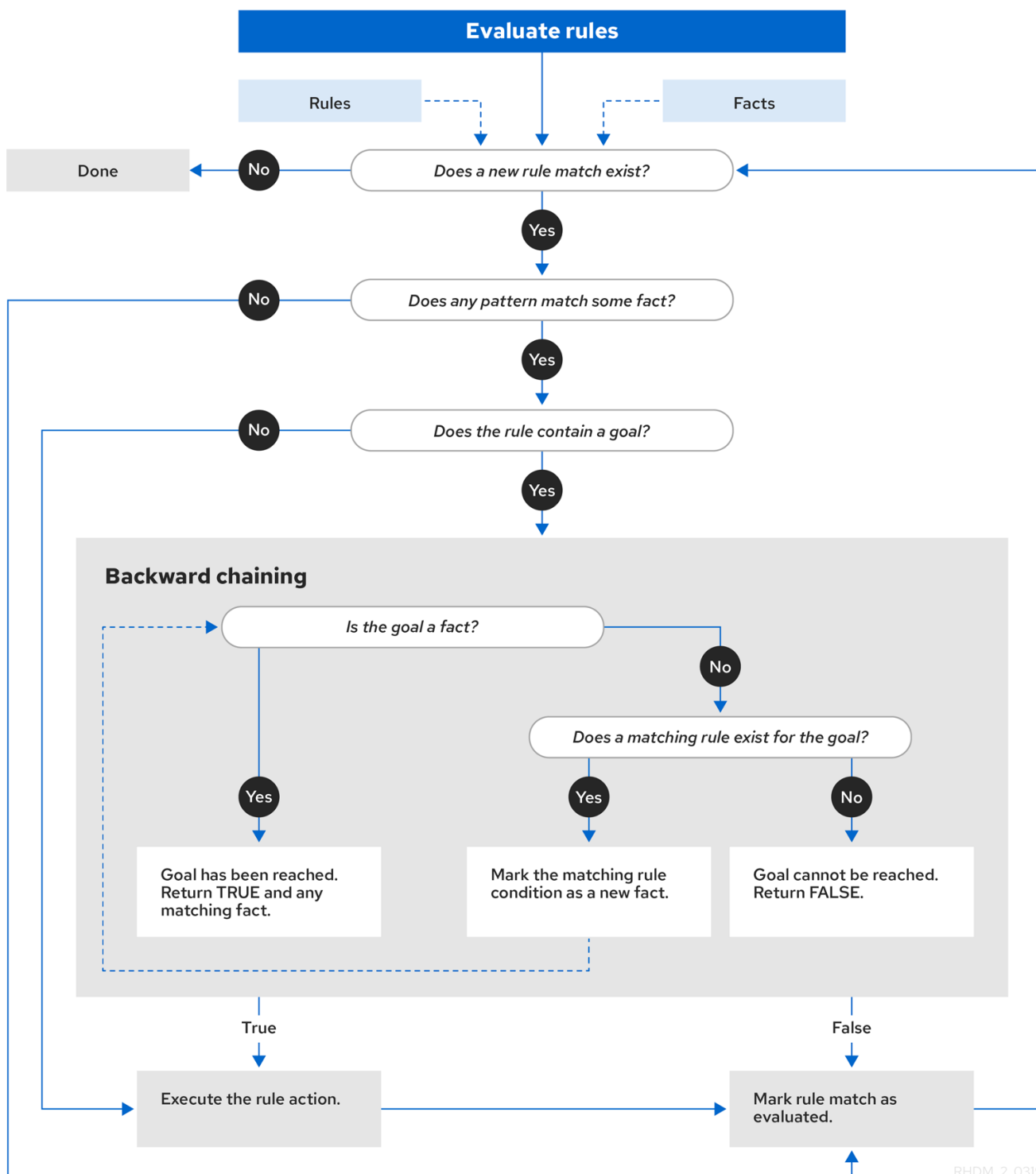
The House of Doom example uses rules with various types of queries to find the location of rooms and items within the house. The sample class **Location.java** contains the **item** and **location** elements used in the example. The sample class **HouseOfDoomMain.java** inserts the items or rooms in their respective locations in the house and executes the rules.

Items and locations in HouseOfDoomMain.java class

```
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
```

```
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

The example rules rely on backward chaining and recursion to determine the location of all items and rooms in the house structure.

The following diagram illustrates the structure of the House of Doom and the items and rooms within it:

Figure 7.28. House of Doom structure
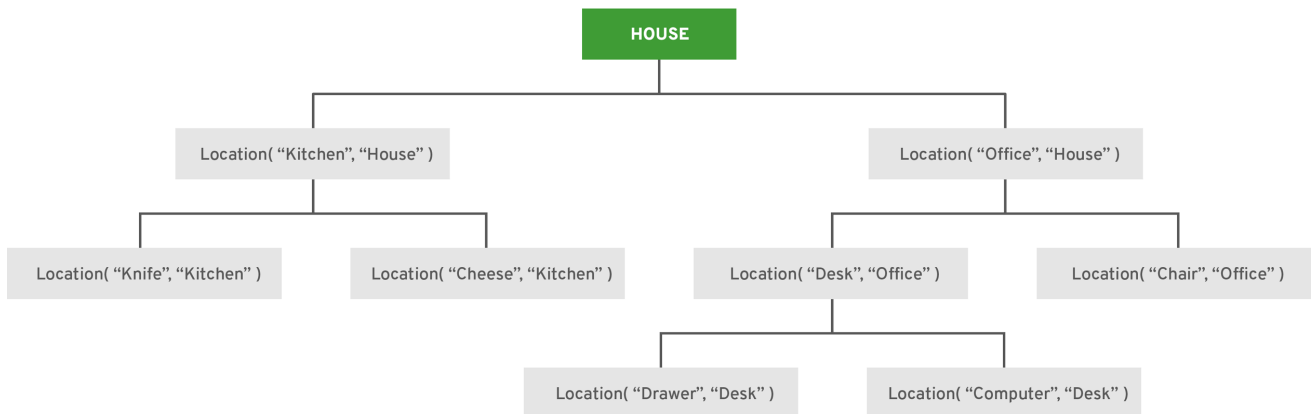
RHDM_2_0319

To execute the example, run the **org.drools.examples.backwardchaining.HouseOfDoomMain** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
```

> Kitchen is in House
> Cheese is in Kitchen
> Knife is in Kitchen
> Computer is in Desk
> Office is in House
> Key is in Office
> Drawer is in House
> Computer is in House
> Key is in House
> Desk is in House
> Chair is in House
> Knife is in House
> Cheese is in House
> Computer is in Office
> Drawer is in Office
> Key is in Desk

All rules in the example have fired to detect the location of all items in the house and to print the location of each in the output.

Recursive query and related rules

A recursive query repeatedly searches through the hierarchy of a data structure for relationships between elements.

In the House of Doom example, the **BC-Example.drl** file contains an **isContainedIn** query that most of the rules in the example use to recursively evaluate the house data structure for data inserted into the decision engine:

Recursive query in BC-Example.drl

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

The rule **"go"** prints every string inserted into the system to determine how items are implemented, and the rule **"go1"** calls the query **isContainedIn**:

Rules "go" and "go1"

```
rule "go" salience 10
  when
    $s : String()
  then
    System.out.println( $s );
end

rule "go1"
  when
    String( this == "go1" )
    isContainedIn("Office", "House"; )
  then
    System.out.println( "Office is in the House" );
end
```

The example inserts the **"go1"** string into the decision engine and activates the **"go1"** rule to detect that item **Office** is in the location **House**:

Insert string and fire rules

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

Rule "go1" output in the IDE console

```
go1
Office is in the House
```

Transitive closure rule

Transitive closure is a relationship between an element contained in a parent element that is multiple levels higher in a hierarchical structure.

The rule **"go2"** identifies the transitive closure relationship of the **Drawer** and the **House**: The **Drawer** is in the **Desk** in the **Office** in the **House**.

```
rule "go2"
  when
    String( this == "go2" )
    isContainedIn("Drawer", "House"; )
  then
    System.out.println( "Drawer is in the House" );
end
```

The example inserts the **"go2"** string into the decision engine and activates the **"go2"** rule to detect that item **Drawer** is ultimately within the location **House**:

Insert string and fire rules

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

Rule "go2" output in the IDE console

```
go2
Drawer is in the House
```

The decision engine determines this outcome based on the following logic:

1. The query recursively searches through several levels in the house to detect the transitive closure between **Drawer** and **House**.

2. Instead of using **Location(x, y;)**, the query uses the value of **(z, y;)** because **Drawer** is not directly in **House**.

3. The **z** argument is currently unbound, which means it has no value and returns everything that is in the argument.

4. The **y** argument is currently bound to **House**, so **z** returns **Office** and **Kitchen**.

5. The query gathers information from the **Office** and checks recursively if the **Drawer** is in the **Office**. The query line **isContainedIn(x, z;)** is called for these parameters.

6. No instance of **Drawer** exists directly in **Office**, so no match is found.

7. With **z** unbound, the query returns data within the **Office** and determines that **z == Desk**.

> isContainedIn(x==drawer, z==desk)

8. The **isContainedIn** query recursively searches three times, and on the third time, the query detects an instance of **Drawer** in **Desk**.

> Location(x==drawer, y==desk)

9. After this match on the first location, the query recursively searches back up the structure to determine that the **Drawer** is in the **Desk**, the **Desk** is in the **Office**, and the **Office** is in the **House**. Therefore, the **Drawer** is in the **House** and the rule is satisfied.

Reactive query rule

A reactive query searches through the hierarchy of a data structure for relationships between elements and is dynamically updated when elements in the structure are modified.

The rule **"go3"** functions as a reactive query that detects if a new item **Key** ever becomes present in the **Office** by transitive closure: A **Key** in the **Drawer** in the **Office**.

Rule "go3"

```
rule "go3"
  when
    String( this == "go3" )
    isContainedIn("Key", "Office"; )
  then
    System.out.println( "Key is in the Office" );
end
```

The example inserts the **"go3"** string into the decision engine and activates the **"go3"** rule. Initially, this rule is not satisfied because no item **Key** exists in the house structure, so the rule produces no output.

Insert string and fire rules

```
ksession.insert( "go3" );
ksession.fireAllRules();
```

Rule "go3" output in the IDE console (unsatisfied)

```
go3
```

The example then inserts a new item **Key** in the location **Drawer**, which is in **Office**. This change satisfies the transitive closure in the **"go3"** rule and the output is populated accordingly.

Insert new item location and fire rules

```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

Rule "go3" output in the IDE console (satisfied)

Key is in the Office

This change also adds another level in the structure that the query includes in subsequent recursive searches.

Queries with unbound arguments in rules

A query with one or more unbound arguments returns all undefined (unbound) items within a defined (bound) argument of the query. If all arguments in a query are unbound, then the query returns all items within the scope of the query.

The rule **"go4"** uses an unbound argument **thing** to search for all items within the bound argument **Office**, instead of using a bound argument to search for a specific item in the **Office**:

Rule "go4"

```
rule "go4"
  when
    String( this == "go4" )
    isContainedIn(thing, "Office"; )
  then
    System.out.println( thing + "is in the Office" );
end
```

The example inserts the **"go4"** string into the decision engine and activates the **"go4"** rule to return all items in the **Office**:

Insert string and fire rules

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

Rule "go4" output in the IDE console

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

The rule **"go5"** uses both unbound arguments **thing** and **location** to search for all items and their locations in the entire **House** data structure:

Rule "go5"

```
rule "go5"
  when
    String( this == "go5" )
```

```
    isContainedIn(thing, location; )
  then
    System.out.println(thing + " is in " + location );
  end
```

The example inserts the **"go5"** string into the decision engine and activates the **"go5"** rule to return all items and their locations in the **House** data structure:

Insert string and fire rules

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

Rule "go5" output in the IDE console

```
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

CHAPTER 8. PERFORMANCE TUNING CONSIDERATIONS WITH DRL

The following key concepts or suggested practices can help you optimize DRL rules and decision engine performance. These concepts are summarized in this section as a convenience and are explained in more detail in the cross-referenced documentation, where applicable. This section will expand or change as needed with new releases of Red Hat Decision Manager.

Define the property and value of pattern constraints from left to right

In DRL pattern constraints, ensure that the fact property name is on the left side of the operator and that the value (constant or a variable) is on the right side. The property name must always be the key in the index and not the value. For example, write **Person(firstName == "John")** instead of **Person("John" == firstName)**. Defining the constraint property and value from right to left can hinder decision engine performance.

For more information about DRL patterns and constraints, see Section 2.8, "Rule conditions in DRL (WHEN)".

Use equality operators more than other operator types in pattern constraints when possible

Although the decision engine supports many DRL operator types that you can use to define your business rule logic, the equality operator **==** is evaluated most efficiently by the decision engine. Whenever practical, use this operator instead of other operator types. For example, the pattern **Person(firstName == "John")** is evaluated more efficiently than **Person(firstName != "OtherName")**. In some cases, using only equality operators might be impractical, so consider all of your business logic needs and options as you use DRL operators.

List the most restrictive rule conditions first

For rules with multiple conditions, list the conditions from most to least restrictive so that the decision engine can avoid assessing the entire set of conditions if the more restrictive conditions are not met.

For example, the following conditions are part of a travel-booking rule that applies a discount to travelers who book both a flight and a hotel together. In this scenario, customers rarely book hotels with flights to receive this discount, so the hotel condition is rarely met and the rule is rarely executed. Therefore, the first condition ordering is more efficient because it prevents the decision engine from evaluating the flight condition frequently and unnecessarily when the hotel condition is not met.

Preferred condition order: hotel and flight

```
when
  $h:hotel() // Rarely booked
  $f:flight()
```

Inefficient condition order: flight and hotel

```
when
  $f:flight()
  $h:hotel() // Rarely booked
```

For more information about DRL patterns and constraints, see Section 2.8, "Rule conditions in DRL (WHEN)".

Avoid iterating over large collections of objects with excessive `from` clauses

Avoid using the **from** condition element in DRL rules to iterate over large collections of objects, as shown in the following example:

Example conditions with from clause

```
when
  $c: Company()
  $e : Employee ( salary > 100000.00) from $c.employees
```

In such cases, the decision engine iterates over the large graph every time the rule condition is evaluated and impedes rule evaluation.

Alternatively, instead of adding an object with a large graph that the decision engine must iterate over frequently, add the collection directly to the KIE session and then join the collection in the condition, as shown in the following example:

Example conditions without from clause

```
when
  $c: Company();
  Employee (salary > 100000.00, company == $c)
```

In this example, the decision engine iterates over the list only one time and can evaluate rules more efficiently.

For more information about the **from** element or other DRL condition elements, see Section 2.8.7, "Supported rule condition elements in DRL (keywords)".

Use decision engine event listeners instead ofSystem.out.println statements in rules for debug logging

You can use **System.out.println** statements in your rule actions for debug logging and console output, but doing this for many rules can impede rule evaluation. As a more efficient alternative, use the built-in decision engine event listeners when possible. If these listeners do not meet your requirements, use a system logging utility supported by the decision engine, such as Logback, Apache Commons Logging, or Apache Log4j.

For more information about supported decision engine event listeners and logging utilities, see *Decision engine in Red Hat Decision Manager* .

CHAPTER 9. NEXT STEPS

- *Testing a decision service using test scenarios*

- *Packaging and deploying a Red Hat Decision Manager project*

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Wednesday, June 10, 2020.