Drools User Guide

The Drools Team

Table of Contents

Chapter 1. Drools rule engine

The Drools rule engine stores, processes, and evaluates data to execute the business rules or decision models that you define. The basic function of the Drools rule engine is to match incoming data, or *facts*, to the conditions of rules and determine whether and how to execute the rules.

The Drools rule engine operates using the following basic components:

- **Rules:** Business rules or DMN decisions that you define. All rules must contain at a minimum the conditions that trigger the rule and the actions that the rule dictates.
- **Facts:** Data that enters or changes in the Drools rule engine that the Drools rule engine matches to rule conditions to execute applicable rules.
- **Production memory:** Location where rules are stored in the Drools rule engine.
- **Working memory:** Location where facts are stored in the Drools rule engine.
- **Agenda:** Location where activated rules are registered and sorted (if applicable) in preparation for execution.

When a business user or an automated system adds or updates rule-related information in Drools, that information is inserted into the working memory of the Drools rule engine in the form of one or more facts. The Drools rule engine matches those facts to the conditions of the rules that are stored in the production memory to determine eligible rule executions. (This process of matching facts to rules is often referred to as *pattern matching*.) When rule conditions are met, the Drools rule engine activates and registers rules in the agenda, where the Drools rule engine then sorts prioritized or conflicting rules in preparation for execution.

The following diagram illustrates these basic components of the Drools rule engine:

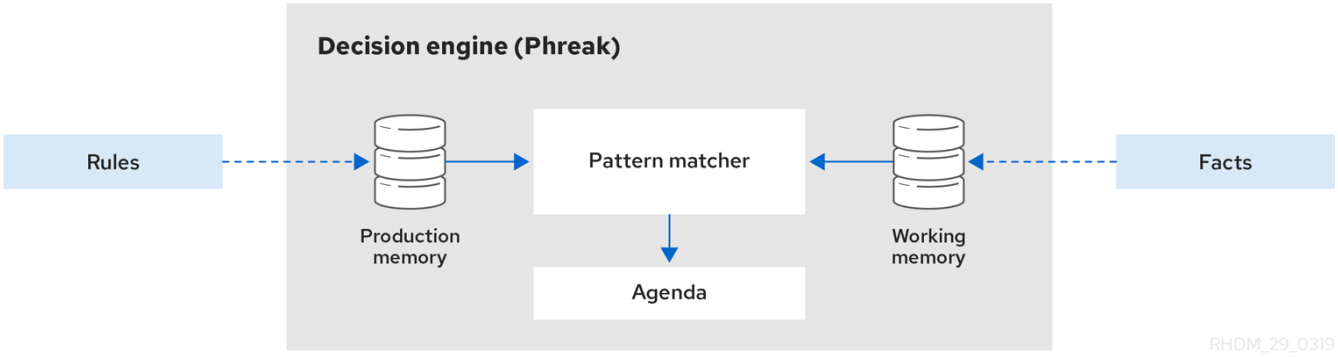


Figure 1. Overview of basic Drools rule engine components

For more details and examples of rule and fact behavior in the Drools rule engine, see Inference and truth maintenance in the Drools rule engine.

These core concepts can help you to better understand other more advanced components, processes, and sub-processes of the Drools rule engine, and as a result, to design more effective business assets in Drools.

1.1. KIE sessions

In Drools, a KIE session stores and executes runtime data. The KIE session is created from a KIE base or directly from a KIE container if you have defined the KIE session in the KIE module descriptor file (`kmodule.xml`) for your project.

Example KIE session configuration in a `kmodule.xml` file

```
<kmodule>
  ...
  <kbase>
    ...
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
    ...
  </kbase>
  ...
</kmodule>
```

A KIE base is a repository that you define in the KIE module descriptor file (`kmodule.xml`) for your project and contains all in Drools, but does not contain any runtime data.

Example KIE base configuration in a `kmodule.xml` file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream" equalsBehavior=
"equality" declarativeAgenda="enabled" packages="org.domain.pkg2, org.domain.pkg3"
includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

A KIE session can be stateless or stateful. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. In a stateful KIE session, that data is retained. The type of KIE session you use depends on your project requirements and how you want data from different asset invocations to be persisted.

1.1.1. Stateless KIE sessions

A stateless KIE session is a session that does not use inference to make iterative changes to facts over time. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations, whereas in a stateful KIE session, that data is retained. A stateless KIE session behaves similarly to a function in that the results that it produces are determined by the contents of the KIE base and by the data that is passed into the KIE session for execution at a specific point in time. The KIE session has no memory of any data that was passed into the KIE session previously.

Stateless KIE sessions are commonly used for the following use cases:

- **Validation**, such as validating that a person is eligible for a mortgage
- **Calculation**, such as computing a mortgage premium
- **Routing and filtering**, such as sorting incoming emails into folders or sending incoming emails to a destination

For example, consider the following driver's license data model and sample DRL rule:

Data model for driver's license application

```
public class Applicant {
  private String name;
  private int age;
  private boolean valid;
  // Getter and setter methods
}
```

Sample DRL rule for driver's license application

```
package com.company.license

rule "Is of valid age"
when
  $a : Applicant(age < 18)
then
  $a.setValid(false);
end
```

The `Is of valid age` rule disqualifies any applicant younger than 18 years old. When the `Applicant` object is inserted into the Drools rule engine, the Drools rule engine evaluates the constraints for each rule and searches for a match. The `"objectType"` constraint is always implied, after which any number of explicit field constraints are evaluated. The variable `$a` is a binding variable that references the matched object in the rule consequence.

The dollar sign (`$`) is optional and helps to differentiate between variable names and field names.

In this example, the sample rule and all other files in the `~/resources` folder of the Drools project are built with the following code:

Create the KIE container

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This code compiles all the rule files found on the class path and adds the result of this compilation, a `KieModule` object, in the `KieContainer`.

Finally, the `StatelessKieSession` object is instantiated from the `KieContainer` and is executed against specified data:

Instantiate the stateless KIE session and enter data

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();

Applicant applicant = new Applicant("Mr John Smith", 16);

assertTrue(applicant.isValid());

ksession.execute(applicant);

assertFalse(applicant.isValid());
```

In a stateless KIE session configuration, the `execute()` call acts as a combination method that instantiates the `KieSession` object, adds all the user data and executes user commands, calls `fireAllRules()`, and then calls `dispose()`. Therefore, with a stateless KIE session, you do not need to call `fireAllRules()` or call `dispose()` after session invocation as you do with a stateful KIE session.

In this case, the specified applicant is under the age of 18, so the application is declined.

For a more complex use case, see the following example. This example uses a stateless KIE session and executes rules against an iterable list of objects, such as a collection.

Expanded data model for driver's license application

```
public class Applicant {
  private String name;
  private int age;
  // Getter and setter methods
}

public class Application {
  private Date dateApplied;
  private boolean valid;
  // Getter and setter methods
}
```

Expanded DRL rule set for driver's license application

```
package com.company.license

rule "Is of valid age"
when
  Applicant(age < 18)
  $a : Application()
then
  $a.setValid(false);
end

rule "Application was made this year"
when
  $a : Application(dateApplied > "01-jan-2009")
then
  $a.setValid(false);
end
```

Expanded Java source with iterable execution in a stateless KIE session

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant("Mr John Smith", 16);
Application application = new Application();

assertTrue(application.isValid());
ksession.execute(Arrays.asList(new Object[] { application, applicant }));  ①
assertFalse(application.isValid());

ksession.execute
  (CommandFactory.newInsertIterable(new Object[] { application, applicant }));  ②

List<Command> cmds = new ArrayList<Command>();  ③
cmds.add(CommandFactory.newInsert(new Person("Mr John Smith"), "mrSmith"));
cmds.add(CommandFactory.newInsert(new Person("Mr John Doe"), "mrDoe"));

BatchExecutionResults results = ksession.execute(CommandFactory.newBatchExecution(
cmds));
assertEquals(new Person("Mr John Smith"), results.getValue("mrSmith"));
```

① Method for executing rules against an iterable collection of objects produced by the `Arrays.asList()` method. Every collection element is inserted before any matched rules are executed. The `execute(Object object)` and `execute(Iterable objects)` methods are wrappers around the `execute(Command command)` method that comes from the `BatchExecutor` interface.

② Execution of the iterable collection of objects using the `CommandFactory` interface.

③ `BatchExecutor` and `CommandFactory` configurations for working with many different commands or result output identifiers. The `CommandFactory` interface supports other commands that you can use in the `BatchExecutor`, such as `StartProcess`, `Query`, and `SetGlobal`.

1.1.1.1. Global variables in stateless KIE sessions

The `StatelessKieSession` object supports global variables (globals) that you can configure to be resolved as session-scoped globals, delegate globals, or execution-scoped globals.

- **Session-scoped globals:** For session-scoped globals, you can use the method `getGlobals()` to return a `Globals` instance that provides access to the KIE session globals. These globals are used for all execution calls. Use caution with mutable globals because execution calls can be executing simultaneously in different threads.

  Session-scoped global

  ```
  import org.kie.api.runtime.StatelessKieSession;

  StatelessKieSession ksession = kbase.newStatelessKieSession();

  // Set a global `myGlobal` that can be used in the rules.
  ksession.setGlobal("myGlobal", "I am a global");

  // Execute while resolving the `myGlobal` identifier.
  ksession.execute(collection);
  ```

- **Delegate globals:** For delegate globals, you can assign a value to a global (with `setGlobal(String, Object)`) so that the value is stored in an internal collection that maps identifiers to values. Identifiers in this internal collection have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) is used.
- **Execution-scoped globals:** For execution-scoped globals, you can use the `Command` object to set a global that is passed to the `CommandExecutor` interface for execution-specific global resolution.

The `CommandExecutor` interface also enables you to export data using out identifiers for globals, inserted facts, and query results:

Out identifiers for globals, inserted facts, and query results

```
import org.kie.api.runtime.ExecutionResults;

// Set up a list of commands.
List cmds = new ArrayList();
cmds.add(CommandFactory.newSetGlobal("list1", new ArrayList(), true));
cmds.add(CommandFactory.newInsert(new Person("jon", 102), "person"));
cmds.add(CommandFactory.newQuery("Get People" "getPeople"));

// Execute the list.
ExecutionResults results = ksession.execute(CommandFactory.newBatchExecution(cmds));

// Retrieve the `ArrayList`.
results.getValue("list1");
// Retrieve the inserted `Person` fact.
results.getValue("person");
// Retrieve the query as a `QueryResults` instance.
results.getValue("Get People");
```

1.1.2. Stateful KIE sessions

A stateful KIE session is a session that uses inference to make iterative changes to facts over time. In a stateful KIE session, data from a previous invocation of the KIE session (the previous session state) is retained between session invocations, whereas in a stateless KIE session, that data is discarded.

Ensure that you call the `dispose()` method after running a stateful KIE session so that no memory leaks occur between session invocations.

Stateful KIE sessions are commonly used for the following use cases:

- **Monitoring**, such as monitoring a stock market and automating the buying process
- **Diagnostics**, such as running fault-finding processes or medical diagnostic processes
- **Logistics**, such as parcel tracking and delivery provisioning
- **Ensuring compliance**, such as verifying the legality of market trades

For example, consider the following fire alarm data model and sample DRL rules:

Data model for sprinklers and fire alarm

```
public class Room {
  private String name;
  // Getter and setter methods
}

public class Sprinkler {
  private Room room;
  private boolean on;
  // Getter and setter methods
}

public class Fire {
  private Room room;
  // Getter and setter methods
}

public class Alarm { }
```

Sample DRL rule set for activating sprinklers and alarm

```
rule "When there is a fire turn on the sprinkler"
when
  Fire($room : room)
  $sprinkler : Sprinkler(room == $room, on == false)
then
  modify($sprinkler) { setOn(true) };
  System.out.println("Turn on the sprinkler for room "+$room.getName());
end

rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end

rule "Cancel the alarm when all the fires have gone"
when
    not Fire()
    $alarm : Alarm()
then
    delete( $alarm );
    System.out.println( "Cancel the alarm" );
end


rule "Status output when things are ok"
when
    not Alarm()
    not Sprinkler( on == true )
then
    System.out.println( "Everything is ok" );
end
```

For the `When there is a fire turn on the sprinkler` rule, when a fire occurs, the instances of the `Fire` class are created for that room and inserted into the KIE session. The rule adds a constraint for the specific `room` matched in the `Fire` instance so that only the sprinkler for that room is checked. When this rule is executed, the sprinkler activates. The other sample rules determine when the alarm is activated or deactivated accordingly.

Whereas a stateless KIE session relies on standard Java syntax to modify a field, a stateful KIE session relies on the `modify` statement in rules to notify the Drools rule engine of changes. The Drools rule engine then reasons over the changes and assesses impact on subsequent rule executions. This process is part of the Drools rule engine ability to use *inference* and *truth maintenance* and is essential in stateful KIE sessions.

In this example, the sample rules and all other files in the `~/resources` folder of the Drools project

are built with the following code:

Create the KIE container

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This code compiles all the rule files found on the class path and adds the result of this compilation, a `KieModule` object, in the `KieContainer`.

Finally, the `KieSession` object is instantiated from the `KieContainer` and is executed against specified data:

Instantiate the stateful KIE session and enter data

```
KieSession ksession = kContainer.newKieSession();

String[] names = new String[]{"kitchen", "bedroom", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();
for( String name: names ){
    Room room = new Room( name );
    name2room.put( name, room );
    ksession.insert( room );
    Sprinkler sprinkler = new Sprinkler( room );
    ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

Console output

```
> Everything is ok
```

With the data added, the Drools rule engine completes all pattern matching but no rules have been executed, so the configured verification message appears. As new data triggers rule conditions, the Drools rule engine executes rules to activate the alarm and later to cancel the alarm that has been activated:

Enter new data to trigger rules

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

Console output

```
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

```
ksession.delete( kitchenFireHandle );
ksession.delete( officeFireHandle );

ksession.fireAllRules();
```

Console output

```
> Cancel the alarm
> Turn off the sprinkler for room office
> Turn off the sprinkler for room kitchen
> Everything is ok
```

In this case, a reference is kept for the returned `FactHandle` object. A fact handle is an internal engine reference to the inserted instance and enables instances to be retracted or modified later.

As this example illustrates, the data and results from previous stateful KIE sessions (the activated alarm) affect the invocation of subsequent sessions (alarm cancellation).

1.1.3. KIE session pools

In use cases with large amounts of KIE runtime data and high system activity, KIE sessions might be created and disposed very frequently. A high turnover of KIE sessions is not always time consuming, but when the turnover is repeated millions of times, the process can become a bottleneck and require substantial clean-up effort.

For these high-volume cases, you can use KIE session pools instead of many individual KIE sessions. To use a KIE session pool, you obtain a KIE session pool from a KIE container, define the initial number of KIE sessions in the pool, and create the KIE sessions from that pool as usual:

Example KIE session pool

```
// Obtain a KIE session pool from the KIE container
KieContainerSessionsPool pool = kContainer.newKieSessionsPool(10);

// Create KIE sessions from the KIE session pool
KieSession kSession = pool.newKieSession();
```

In this example, the KIE session pool starts with 10 KIE sessions in it, but you can specify the number of KIE sessions that you need. This integer value is the number of KIE sessions that are only initially created in the pool. If required by the running application, the number of KIE sessions in the pool can dynamically grow beyond that value.

After you define a KIE session pool, the next time you use the KIE session as usual and call `dispose()` on it, the KIE session is reset and pushed back into the pool instead of being destroyed.

KIE session pools typically apply to stateful KIE sessions, but KIE session pools can also affect stateless KIE sessions that you reuse with multiple `execute()` calls. When you create a stateless KIE session directly from a KIE container, the KIE session continues to internally create a new KIE session for each `execute()` invocation. Conversely, when you create a stateless KIE session from a KIE session pool, the KIE session internally uses only the specific KIE sessions provided by the pool.

When you finish using a KIE session pool, you can call the `shutdown()` method on it to avoid memory leaks. Alternatively, you can call `dispose()` on the KIE container to shut down all the pools created from the KIE container.

1.2. Inference and truth maintenance in the Drools rule engine

The basic function of the Drools rule engine is to match data to business rules and determine whether and how to execute rules. To ensure that relevant data is applied to the appropriate rules, the Drools rule engine makes *inferences* based on existing knowledge and performs the actions based on the inferred information.

For example, the following DRL rule determines the age requirements for adults, such as in a bus pass policy:

Rule to define age requirement

```
rule "Infer Adult"
when
  $p : Person(age >= 18)
then
  insert(new IsAdult($p))
end
```

Based on this rule, the Drools rule engine infers whether a person is an adult or a child and performs the specified action (the `then` consequence). Every person who is 18 years old or older has an instance of `IsAdult` inserted for them in the working memory. This inferred relation of age and bus pass can then be invoked in any rule, such as in the following rule segment:

```
$p : Person()
IsAdult(person == $p)
```

In many cases, new data in a rule system is the result of other rule executions, and this new data can affect the execution of other rules. If the Drools rule engine asserts data as a result of executing a rule, the Drools rule engine uses truth maintenance to justify the assertion and enforce truthfulness when applying inferred information to other rules. Truth maintenance also helps to identify inconsistencies and to handle contradictions. For example, if two rules are executed and result in a contradictory action, the Drools rule engine chooses the action based on assumptions

from previously calculated conclusions.

The Drools rule engine inserts facts using either stated or logical insertions:

- **Stated insertions:** Defined with `insert()`. After stated insertions, facts are generally retracted explicitly. (The term *insertion*, when used generically, refers to *stated insertion*.)
- **Logical insertions:** Defined with `insertLogical()`. 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. The facts are retracted when no condition supports the logical insertion. A fact that is logically inserted is considered to be *justified* by the Drools rule engine.

For example, the following sample DRL rules use stated fact insertion to determine the age requirements for issuing a child bus pass or an adult bus pass:

Rules to issue bus pass, stated insertion

```
rule "Issue Child Bus Pass"
when
  $p : Person(age < 18)
then
  insert(new ChildBusPass($p));
end

rule "Issue Adult Bus Pass"
when
  $p : Person(age >= 18)
then
  insert(new AdultBusPass($p));
end
```

These rules are not easily maintained in the Drools rule engine as bus riders increase in age and move from child to adult bus pass. As an alternative, these rules can be separated into rules for bus rider age and rules for bus pass type using logical fact insertion. The logical insertion of the fact makes the fact dependent on the truth of the `when` clause.

The following DRL rules use logical insertion to determine the age requirements for children and adults:

Children and adult age requirements, logical insertion

```
rule "Infer Child"
when
  $p : Person(age < 18)
then
  insertLogical(new IsChild($p))
end

rule "Infer Adult"
when
  $p : Person(age >= 18)
then
  insertLogical(new IsAdult($p))
end
```

For logical insertions, your fact objects must override the `equals` and `hashCode` methods from the `java.lang.Object` object according to the Java standard. Two objects are equal if their `equals` methods return `true` for each other and if their `hashCode` methods return the same values. For more information, see the Java API documentation for your Java version.

When the condition in the rule is false, the fact is automatically retracted. This behavior is helpful in this example because the two rules are mutually exclusive. In this example, if the person is younger than 18 years old, the rule logically inserts an `IsChild` fact. After the person is 18 years old or older, the `IsChild` fact is automatically retracted and the `IsAdult` fact is inserted.

The following DRL rules then determine whether to issue a child bus pass or an adult bus pass and logically insert the `ChildBusPass` and `AdultBusPass` facts. This rule configuration is possible because the truth maintenance system in the Drools rule engine supports chaining of logical insertions for a cascading set of retracts.

Rules to issue bus pass, logical insertion

```
rule "Issue Child Bus Pass"
when
  $p : Person()
    IsChild(person == $p)
then
  insertLogical(new ChildBusPass($p));
end

rule "Issue Adult Bus Pass"
when
  $p : Person()
    IsAdult(person =$p)
then
  insertLogical(new AdultBusPass($p));
end
```

When a person turns 18 years old, the `IsChild` fact and the person's `ChildBusPass` fact is retracted. To these set of conditions, you can relate another rule that states that a person must return the child pass after turning 18 years old. When the Drools rule engine automatically retracts the `ChildBusPass` object, the following rule is executed to send a request to the person:

Rule to notify bus pass holder of new pass

```
rule "Return ChildBusPass Request"
when
  $p : Person()
    not(ChildBusPass(person == $p))
then
  requestChildBusPass($p);
end
```

The following flowcharts illustrate the life cycle of stated and logical insertions:

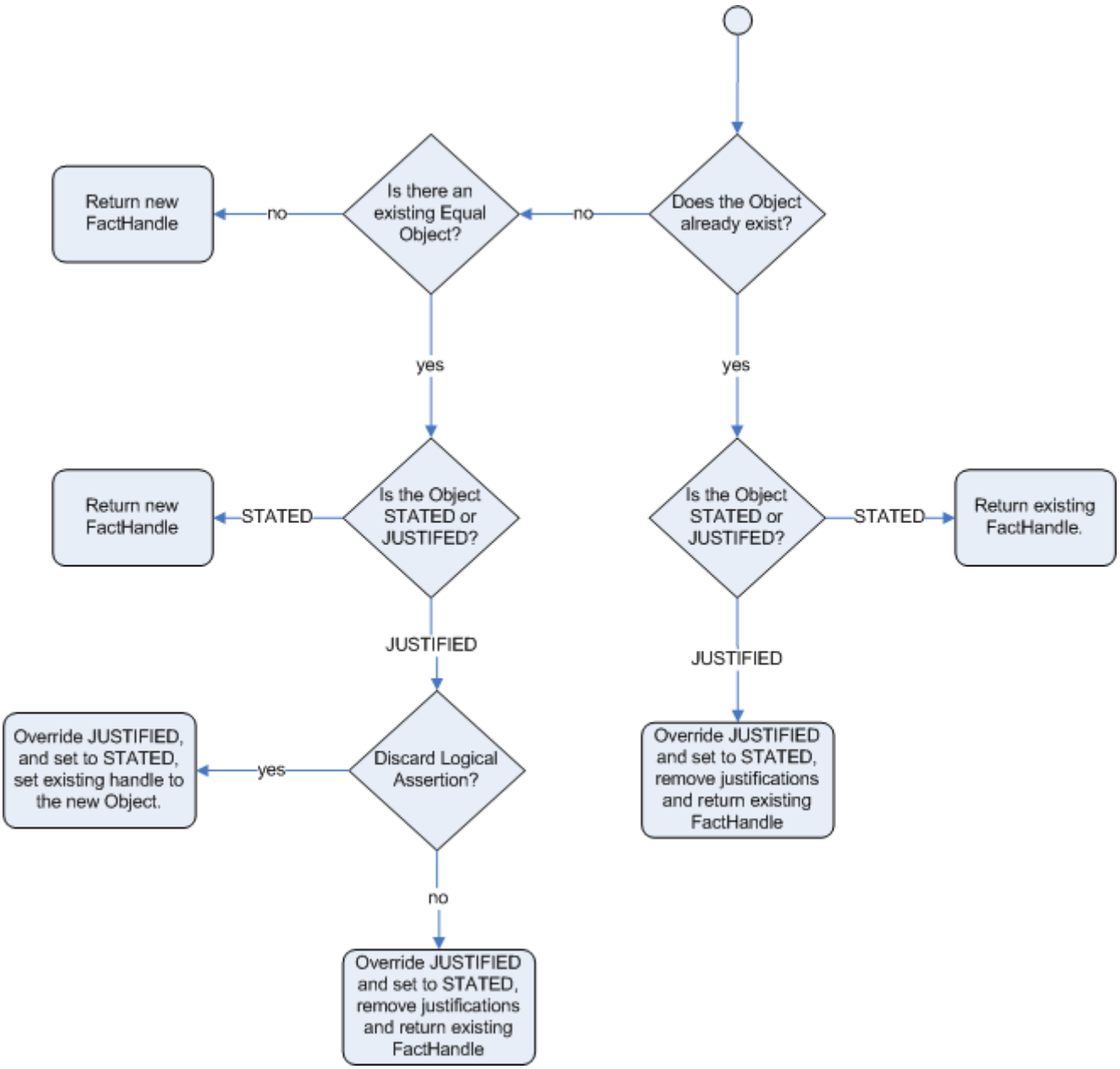


Figure 2. Stated insertion

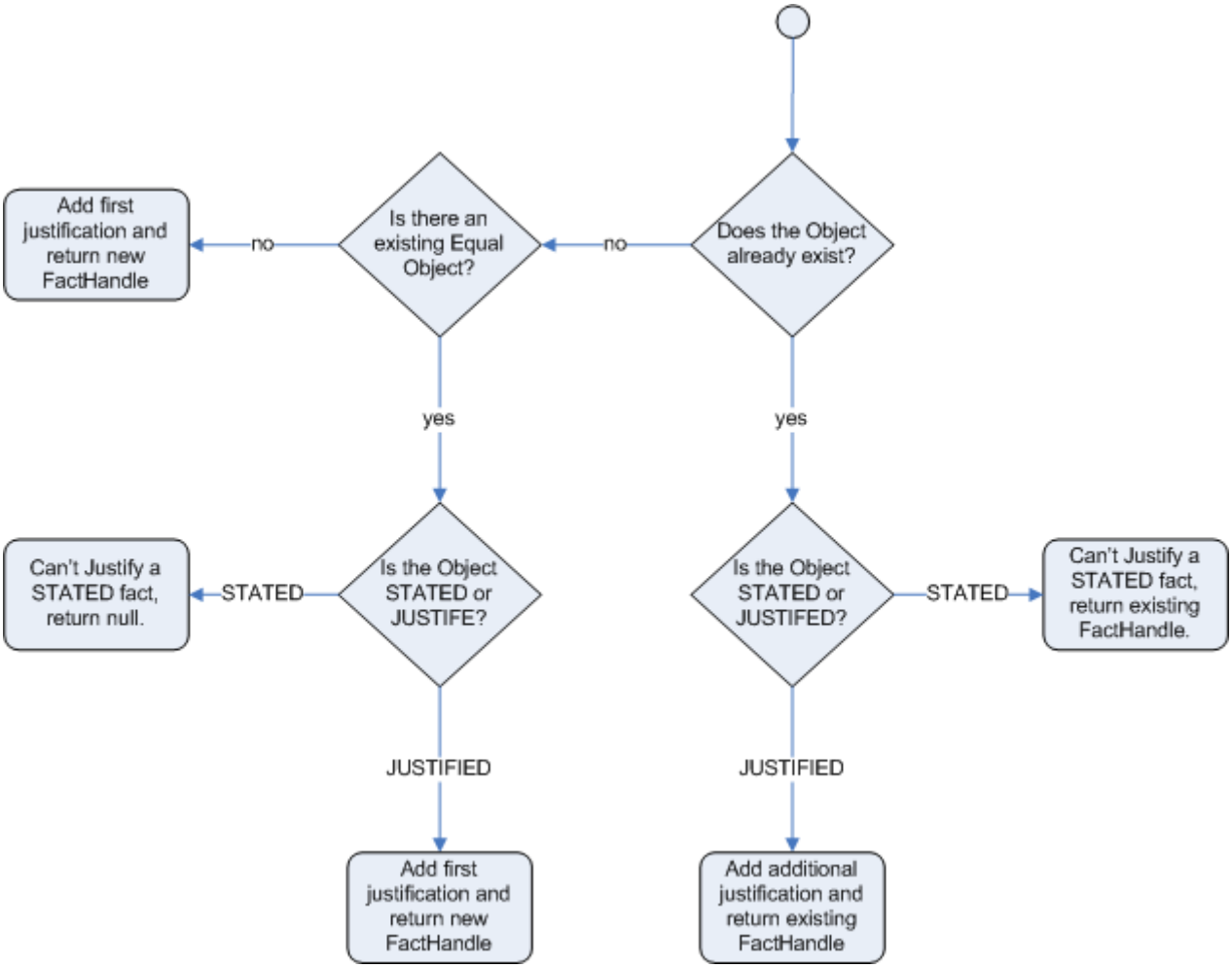


Figure 3. Logical insertion

When the Drools rule engine logically inserts an object during a rule execution, the Drools rule engine *justifies* the object by executing the rule. For each logical insertion, only one equal object can exist, and each subsequent equal logical insertion increases the justification counter for that logical insertion. A justification is removed when the conditions of the rule become untrue. When no more justifications exist, the logical object is automatically retracted.

1.2.1. Government ID example

So now we know what inference is, and have a basic example, how does this facilitate good rule design and maintenance?

Consider a government ID department that is responsible for issuing ID cards when children become adults. They might have a decision table that includes logic like this, which says when an adult living in London is 18 or over, issue the card:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person		
	location	age >= $1	issueIdCard($1)
	Select Person	Select Adults	Issue ID Card

Issue ID Card to Adults	London	18	p

However the ID department does not set the policy on who an adult is. That's done at a central government level. If the central government were to change that age to 21, this would initiate a change management process. Someone would have to liaise with the ID department and make sure their systems are updated, in time for the law going live.

This change management process and communication between departments is not ideal for an agile environment, and change becomes costly and error prone. Also the card department is managing more information than it needs to be aware of with its "monolithic" approach to rules management which is "leaking" information better placed elsewhere. By this I mean that it doesn't care what explicit "age >= 18" information determines whether someone is an adult, only that they are an adult.

In contrast to this, let's pursue an approach where we split (de-couple) the authoring responsibilities, so that both the central government and the ID department maintain their own rules.

It's the central government's job to determine who is an adult. If they change the law they just update their central repository with the new rules, which others use:

	RuleTable Age Policy	
	CONDITION	ACTION
	p : Person	
	age >= $1	insert($1)
	Adult Age Policy	Add Adult Relation
Infer Adult	18	new IsAdult(p)

The IsAdult fact, as discussed previously, is inferred from the policy rules. It encapsulates the seemingly arbitrary piece of logic "age >= 18" and provides semantic abstractions for its meaning. Now if anyone uses the above rules, they no longer need to be aware of explicit information that determines whether someone is an adult or not. They can just use the inferred fact:

	RuleTable ID Card		
	CONDITION	CONDITION	ACTION
	p : Person	isAdult	
	location	person == $1	issueIdCard($1)
	Select Person	Select Adults	Issue ID Card
Issue ID Card to Adults	London	p	p

While the example is very minimal and trivial it illustrates some important points. We started with a monolithic and leaky approach to our knowledge engineering. We created a single decision table that had all possible information in it and that leaks information from central government that the ID department did not care about and did not want to manage.

We first de-coupled the knowledge process so each department was responsible for only what it needed to know. We then encapsulated this leaky knowledge using an inferred fact IsAdult. The use of the term IsAdult also gave a semantic abstraction to the previously arbitrary logic "age >= 18".

So a general rule of thumb when doing your knowledge engineering is:

- **Bad**
  - Monolithic
  - Leaky
- **Good**
  - De-couple knowledge responsibilities
  - Encapsulate knowledge
  - Provide semantic abstractions for those encapsulations

1.2.2. Fact equality modes in the Drools rule engine

The Drools rule engine supports the following fact equality modes that determine how the Drools rule engine stores and compares inserted facts:

- `identity`: (Default) The Drools rule engine uses an `IdentityHashMap` to store all inserted facts. For every new fact insertion, the Drools rule engine returns a new `FactHandle` object. If a fact is inserted again, the Drools rule engine returns the original `FactHandle` object, ignoring repeated insertions for the same fact. In this mode, two facts are the same for the Drools rule engine only if they are the very same object with the same identity.
- `equality`: The Drools rule engine uses a `HashMap` to store all inserted facts. The Drools rule engine returns a new `FactHandle` object only if the inserted fact is not equal to an existing fact, according to the `equals()` method of the inserted fact. In this mode, two facts are the same for the Drools rule engine if they are composed the same way, regardless of identity. Use this mode when you want objects to be assessed based on feature equality instead of explicit identity.

As an illustration of fact equality modes, consider the following example facts:

Example facts

```
Person p1 = new Person("John", 45);
Person p2 = new Person("John", 45);
```

In `identity` mode, facts `p1` and `p2` are different instances of a `Person` class and are treated as separate objects because they have separate identities. In `equality` mode, facts `p1` and `p2` are treated as the same object because they are composed the same way. This difference in behavior affects how you can interact with fact handles.

For example, assume that you insert facts `p1` and `p2` into the Drools rule engine and later you want to retrieve the fact handle for `p1`. In `identity` mode, you must specify `p1` to return the fact handle for that exact object, whereas in `equality` mode, you can specify `p1`, `p2`, or `new Person("John", 45)` to return the fact handle.

Example code to insert a fact and return the fact handle in `identity` mode

```
ksession.insert(p1);

ksession.getFactHandle(p1);
```

Example code to insert a fact and return the fact handle in `equality` mode

```
ksession.insert(p1);

ksession.getFactHandle(p1);

// Alternate option:
ksession.getFactHandle(new Person("John", 45));
```

To set the fact equality mode, use one of the following options:

- Set the system property `drools.equalityBehavior` to `identity` (default) or `equality`.
- Set the equality mode while creating the KIE base programmatically:

  ```
  KieServices ks = KieServices.get();
  KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
  kieBaseConf.setOption(EqualityBehaviorOption.EQUALITY);
  KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
  ```

- Set the equality mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

  ```
  <kmodule>
    ...
    <kbase name="KBase2" default="false" equalsBehavior="equality" packages=
  "org.domain.pkg2, org.domain.pkg3" includes="KBase1">
      ...
    </kbase>
    ...
  </kmodule>
  ```

1.3. Execution control in the Drools rule engine

When new rule data enters the working memory of the Drools rule engine, rules may become fully matched and eligible for execution. A single working memory action can result in multiple eligible rule executions. When a rule is fully matched, the Drools rule engine creates an activation instance, referencing the rule and the matched facts, and adds the activation onto the Drools rule engine agenda. The agenda controls the execution order of these rule activations using a conflict resolution strategy.

After the first call of `fireAllRules()` in the Java application, the Drools rule engine cycles repeatedly through two phases:

- **Agenda evaluation.** In this phase, the Drools rule engine selects all rules that can be executed. If no executable rules exist, the execution cycle ends. If an executable rule is found, the Drools rule engine registers the activation in the agenda and then moves on to the working memory actions phase to perform rule consequence actions.
- **Working memory actions.** In this phase, the Drools rule engine performs the rule consequence actions (the `then` portion of each rule) for all activated rules previously registered in the agenda. After all the consequence actions are complete or the main Java application process calls `fireAllRules()` again, the Drools rule engine returns to the agenda evaluation phase to reassess rules.

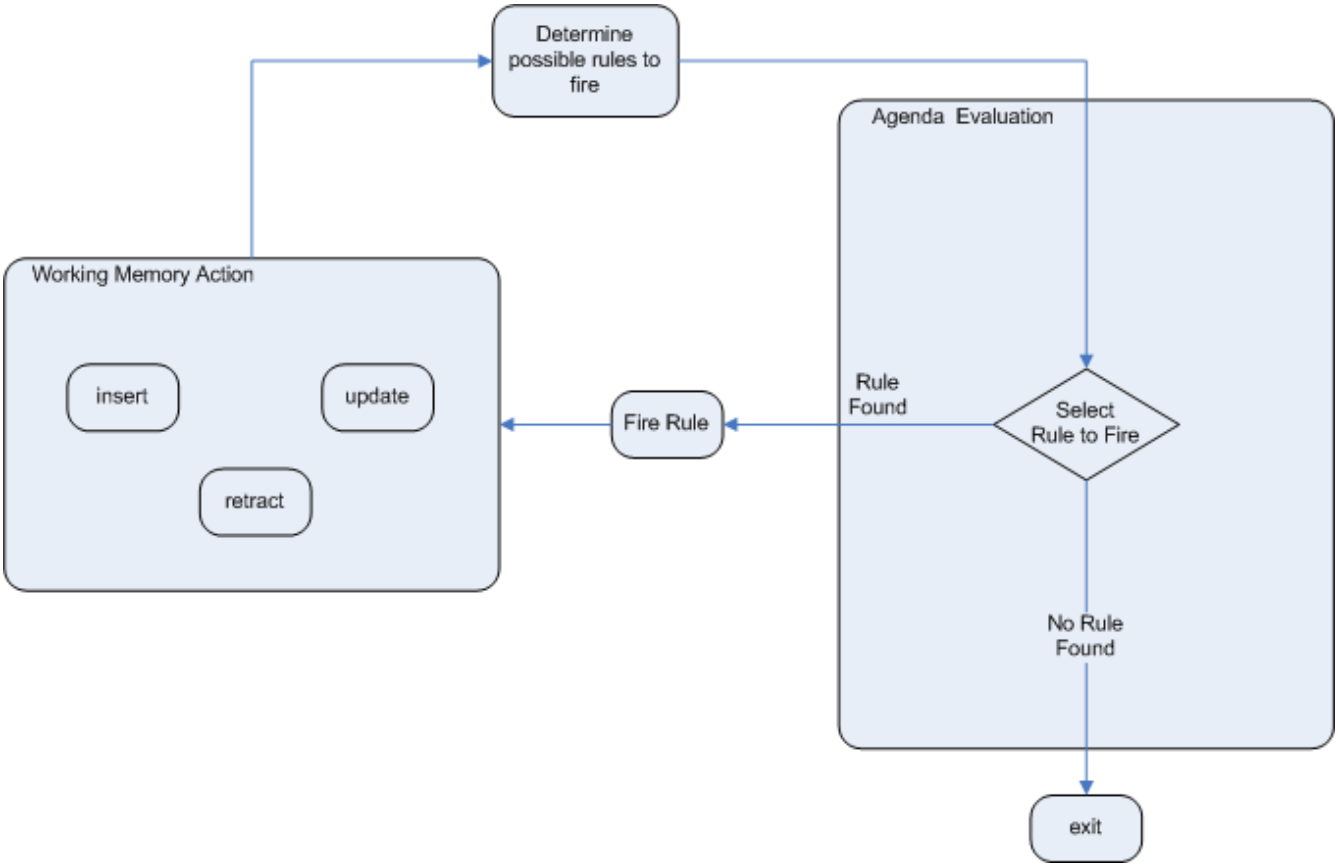


Figure 4. Two-phase execution process in the Drools rule engine

When multiple rules exist on the agenda, the execution of one rule may cause another rule to be removed from the agenda. To avoid this, you can define how and when rules are executed in the Drools rule engine. Some common methods for defining rule execution order are by using rule salience, agenda groups, or activation groups.

1.3.1. Salience for rules

Each rule has an integer `salience` attribute that determines the order of execution. Rules with a higher salience value are given higher priority when ordered in the activation queue. The default salience value for rules is zero, but the salience can be negative or positive.

For example, the following sample DRL rules are listed in the Drools rule engine stack in the order shown:

```
rule "RuleA"
salience 95
when
    $fact : MyFact( field1 == true )
then
    System.out.println("Rule2 : " + $fact);
    update($fact);
end

rule "RuleB"
salience 100
when
   $fact : MyFact( field1 == false )
then
   System.out.println("Rule1 : " + $fact);
   $fact.setField1(true);
   update($fact);
end
```

The `RuleB` rule is listed second, but it has a higher salience value than the `RuleA` rule and is therefore executed first.

1.3.2. Agenda groups for rules

An agenda group is a set of rules bound together by the same `agenda-group` rule attribute. Agenda groups partition rules on the Drools rule engine agenda. At any one time, only one group has a *focus* that gives that group of rules priority for execution before rules in other agenda groups. You determine the focus with a `setFocus()` call for the agenda group. You can also define rules with an `auto-focus` attribute so that the next time the rule is activated, the focus is automatically given to the entire agenda group to which the rule is assigned.

Each time the `setFocus()` call is made in a Java application, the Drools rule engine adds the specified agenda group to the top of the rule stack. The default agenda group `"MAIN"` contains all rules that do not belong to a specified agenda group and is executed first in the stack unless another group has the focus.

For example, the following sample DRL rules belong to specified agenda groups and are listed in the Drools rule engine stack in the order shown:

Sample DRL rules for banking application

```
rule "Increase balance for credits"
  agenda-group "calculation"
when
  ap : AccountPeriod()
  acc : Account( $accountNo : accountNo )
  CashFlow( type == CREDIT,
            accountNo == $accountNo,
            date >= ap.start && <= ap.end,
            $amount : amount )
then
  acc.balance  += $amount;
end
```

```
rule "Print balance for AccountPeriod"
  agenda-group "report"
when
  ap : AccountPeriod()
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

For this example, the rules in the `"report"` agenda group must always be executed first and the rules in the `"calculation"` agenda group must always be executed second. Any remaining rules in other agenda groups can then be executed. Therefore, the `"report"` and `"calculation"` groups must receive the focus to be executed in that order, before other rules can be executed:

Set the focus for the order of agenda group execution

```
Agenda agenda = ksession.getAgenda();
agenda.getAgendaGroup( "report" ).setFocus();
agenda.getAgendaGroup( "calculation" ).setFocus();
ksession.fireAllRules();
```

You can also use the `clear()` method to cancel all the activations generated by the rules belonging to a given agenda group before each has had a chance to be executed:

Cancel all other rule activations

```
ksession.getAgenda().getAgendaGroup( "Group A" ).clear();
```

1.3.3. Activation groups for rules

An activation group is a set of rules bound together by the same `activation-group` rule attribute. In

this group, only one rule can be executed. After conditions are met for a rule in that group to be executed, all other pending rule executions from that activation group are removed from the agenda.

For example, the following sample DRL rules belong to the specified activation group and are listed in the Drools rule engine stack in the order shown:

Sample DRL rules for banking

```
rule "Print balance for AccountPeriod1"
  activation-group "report"
when
  ap : AccountPeriod1()
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

```
rule "Print balance for AccountPeriod2"
  activation-group "report"
when
  ap : AccountPeriod2()
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

For this example, if the first rule in the `"report"` activation group is executed, the second rule in the group and all other executable rules on the agenda are removed from the agenda.

1.3.4. Rule execution modes and thread safety in the Drools rule engine

The Drools rule engine supports the following rule execution modes that determine how and when the Drools rule engine executes rules:

- **Passive mode**: (Default) The Drools rule engine evaluates rules when a user or an application explicitly calls `fireAllRules()`. Passive mode in the Drools rule engine is best for applications that require direct control over rule evaluation and execution, or for complex event processing (CEP) applications that use the pseudo clock implementation in the Drools rule engine.

Example CEP application code with the Drools rule engine in passive mode

```
KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("pseudo") );
KieSession session = kbase.newKieSession( conf, null );
SessionPseudoClock clock = session.getSessionClock();

session.insert( tick1 );
session.fireAllRules();

clock.advanceTime(1, TimeUnit.SECONDS);
session.insert( tick2 );
session.fireAllRules();

clock.advanceTime(1, TimeUnit.SECONDS);
session.insert( tick3 );
session.fireAllRules();

session.dispose();
```

- **Active mode**: If a user or application calls `fireUntilHalt()`, the Drools rule engine starts in active mode and evaluates rules continually until the user or application explicitly calls `halt()`. Active mode in the Drools rule engine is best for applications that delegate control of rule evaluation and execution to the Drools rule engine, or for complex event processing (CEP) applications that use the real-time clock implementation in the Drools rule engine. Active mode is also optimal for CEP applications that use active queries.

Example CEP application code with the Drools rule engine in active mode

```
KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();
config.setOption( ClockTypeOption.get("realtime") );
KieSession session = kbase.newKieSession( conf, null );

new Thread( new Runnable() {
  @Override
  public void run() {
      session.fireUntilHalt();
  }
} ).start();

session.insert( tick1 );

... Thread.sleep( 1000L ); ...

session.insert( tick2 );

... Thread.sleep( 1000L ); ...

session.insert( tick3 );

session.halt();
session.dispose();
```

This example calls `fireUntilHalt()` from a dedicated execution thread to prevent the current thread from being blocked indefinitely while the Drools rule engine continues evaluating rules. The dedicated thread also enables you to call `halt()` at a later stage in the application code.

Although you should avoid using both `fireAllRules()` and `fireUntilHalt()` calls, especially from different threads, the Drools rule engine can handle such situations safely using thread-safety logic and an internal state machine. If a `fireAllRules()` call is in progress and you call `fireUntilHalt()`, the Drools rule engine continues to run in passive mode until the `fireAllRules()` operation is complete and then starts in active mode in response to the `fireUntilHalt()` call. However, if the Drools rule engine is running in active mode following a `fireUntilHalt()` call and you call `fireAllRules()`, the `fireAllRules()` call is ignored and the Drools rule engine continues to run in active mode until you call `halt()`.

For added thread safety in active mode, the Drools rule engine supports a `submit()` method that you can use to group and perform operations on a KIE session in a thread-safe, atomic action:

Example application code with `submit()` method to perform atomic operations in active mode

```
KieSession session = ...;

new Thread( new Runnable() {
  @Override
  public void run() {
      session.fireUntilHalt();
  }
} ).start();

final FactHandle fh = session.insert( fact_a );

... Thread.sleep( 1000L ); ...

session.submit( new KieSession.AtomicAction() {
  @Override
  public void execute( KieSession kieSession ) {
    fact_a.setField("value");
    kieSession.update( fh, fact_a );
    kieSession.insert( fact_1 );
    kieSession.insert( fact_2 );
    kieSession.insert( fact_3 );
  }
} );

... Thread.sleep( 1000L ); ...

session.insert( fact_z );

session.halt();
session.dispose();
```

Thread safety and atomic operations are also helpful from a client-side perspective. For example, you might need to insert more than one fact at a given time, but require the Drools rule engine to consider the insertions as an atomic operation and to wait until all the insertions are complete before evaluating the rules again.

1.3.5. Fact propagation modes in the Drools rule engine

The Drools rule engine supports the following fact propagation modes that determine how the Drools rule engine progresses inserted facts through the engine network in preparation for rule execution:

- **Lazy**: (Default) Facts are propagated in batch collections at rule execution, not in real time as the facts are individually inserted by a user or application. As a result, the order in which the facts are ultimately propagated through the Drools rule engine may be different from the order in which the facts were individually inserted.
- **Immediate**: Facts are propagated immediately in the order that they are inserted by a user or

application.

- **Eager**: Facts are propagated lazily (in batch collections), but before rule execution. The Drools rule engine uses this propagation behavior for rules that have the `no-loop` or `lock-on-active` attribute.

By default, the Phreak rule algorithm in the Drools rule engine uses lazy fact propagation for improved rule evaluation overall. However, in few cases, this lazy propagation behavior can alter the expected result of certain rule executions that may require immediate or eager propagation.

For example, the following rule uses a specified query with a `?` prefix to invoke the query in pull-only or passive fashion:

Example rule with a passive query

```
query Q (Integer i)
    String( this == i.toString() )
end

rule "Rule"
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

For this example, the rule should be executed only when a `String` that satisfies the query is inserted before the `Integer`, such as in the following example commands:

Example commands that should trigger the rule execution

```
KieSession ksession = ...
ksession.insert("1");
ksession.insert(1);
ksession.fireAllRules();
```

However, due to the default lazy propagation behavior in Phreak, the Drools rule engine does not detect the insertion sequence of the two facts in this case, so this rule is executed regardless of `String` and `Integer` insertion order. For this example, immediate propagation is required for the expected rule evaluation.

To alter the Drools rule engine propagation mode to achieve the expected rule evaluation in this case, you can add the `@Propagation(<type>)` tag to your rule and set `<type>` to `LAZY`, `IMMEDIATE`, or `EAGER`.

In the same example rule, the immediate propagation annotation enables the rule to be evaluated only when a `String` that satisfies the query is inserted before the `Integer`, as expected:

Example rule with a passive query and specified propagation mode

```
query Q (Integer i)
    String( this == i.toString() )
end

rule "Rule" @Propagation(IMMEDIATE)
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

1.3.6. Agenda evaluation filters

[AgendaFilter] | *rule-engine/AgendaFilter.png*

Figure 5. AgendaFilters

The Drools rule engine supports an `AgendaFilter` object in the filter interface that you can use to allow or deny the evaluation of specified rules during agenda evaluation. You can specify an agenda filter as part of a `fireAllRules()` call.

The following example code permits only rules ending with the string `"Test"` to be evaluated and executed. All other rules are filtered out of the Drools rule engine agenda.

Example agenda filter definition

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

1.4. Phreak rule algorithm in the Drools rule engine

The Drools rule engine in Drools uses the Phreak algorithm for rule evaluation. Phreak evolved from the Rete algorithm, including the enhanced Rete algorithm ReteOO that was introduced in previous versions of Drools for object-oriented systems. Overall, Phreak is more scalable than Rete and ReteOO, and is faster in large systems.

While Rete is considered eager (immediate rule evaluation) and data oriented, Phreak is considered lazy (delayed rule evaluation) and goal oriented. The Rete algorithm performs many actions during the insert, update, and delete actions in order to find partial matches for all rules. This eagerness of the Rete algorithm during rule matching requires a lot of time before eventually executing rules, especially in large systems. With Phreak, this partial matching of rules is delayed deliberately to handle large amounts of data more efficiently.

The Phreak algorithm adds the following set of enhancements to previous Rete algorithms:

- Three layers of contextual memory: Node, segment, and rule memory types
- Rule-based, segment-based, and node-based linking

- Lazy (delayed) rule evaluation
- Stack-based evaluations with pause and resume
- Isolated rule evaluation
- Set-oriented propagations

1.4.1. Rule evaluation in Phreak

When the Drools rule engine starts, all rules are considered to be *unlinked* from pattern-matching data that can trigger the rules. At this stage, the Phreak algorithm in the Drools rule engine does not evaluate the rules. The `insert`, `update`, and `delete` actions are queued, and Phreak uses a heuristic, based on the rule most likely to result in execution, to calculate and select the next rule for evaluation. When all the required input values are populated for a rule, the rule is considered to be *linked* to the relevant pattern-matching data. Phreak then creates a goal that represents this rule and places the goal into a priority queue that is ordered by rule salience. Only the rule for which the goal was created is evaluated, and other potential rule evaluations are delayed. While individual rules are evaluated, node sharing is still achieved through the process of segmentation.

Unlike the tuple-oriented Rete, the Phreak propagation is collection oriented. For the rule that is being evaluated, the Drools rule engine accesses the first node and processes all queued insert, update, and delete actions. The results are added to a set, and the set is propagated to the child node. In the child node, all queued insert, update, and delete actions are processed, adding the results to the same set. The set is then propagated to the next child node and the same process repeats until it reaches the terminal node. This cycle creates a batch process effect that can provide performance advantages for certain rule constructs.

The linking and unlinking of rules happens through a layered bit-mask system, based on network segmentation. When the rule network is built, segments are created for rule network nodes that are shared by the same set of rules. A rule is composed of a path of segments. In case a rule does not share any node with any other rule, it becomes a single segment.

A bit-mask offset is assigned to each node in the segment. Another bit mask is assigned to each segment in the path of the rule according to these requirements:

- If at least one input for a node exists, the node bit is set to the `on` state.
- If each node in a segment has the bit set to the `on` state, the segment bit is also set to the `on` state.
- If any node bit is set to the `off` state, the segment is also set to the `off` state.
- If each segment in the path of the rule is set to the `on` state, the rule is considered linked, and a goal is created to schedule the rule for evaluation.

The same bit-mask technique is used to track modified nodes, segments, and rules. This tracking ability enables an already linked rule to be unscheduled from evaluation if it has been modified since the evaluation goal for it was created. As a result, no rules can ever evaluate partial matches.

This process of rule evaluation is possible in Phreak because, as opposed to a single unit of memory in Rete, Phreak has three layers of contextual memory with node, segment, and rule memory types. This layering enables much more contextual understanding during the evaluation of a rule.

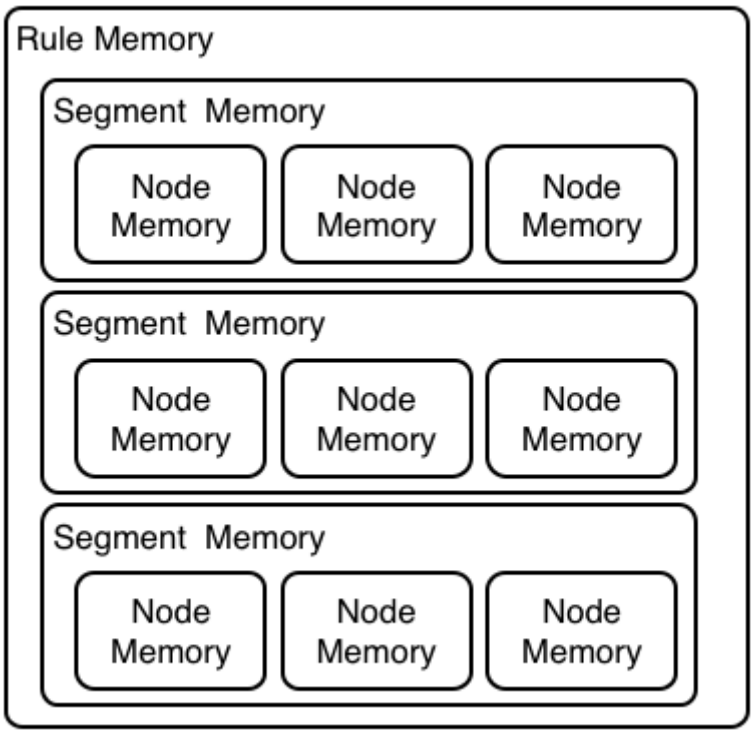


Figure 6. Phreak three-layered memory system

The following examples illustrate how rules are organized and evaluated in this three-layered memory system in Phreak.

Example 1: A single rule (R1) with three patterns: A, B and C. The rule forms a single segment, with bits 1, 2, and 4 for the nodes. The single segment has a bit offset of 1.

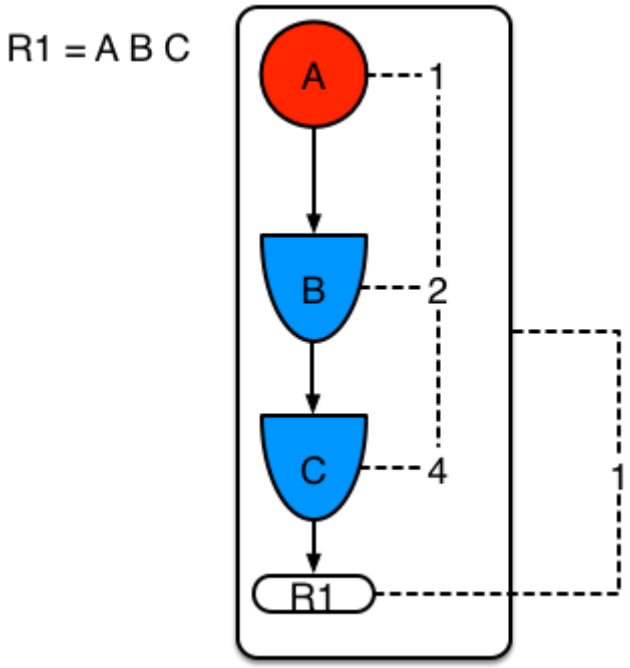


Figure 7. Example 1: Single rule

Example 2: Rule R2 is added and shares pattern A.

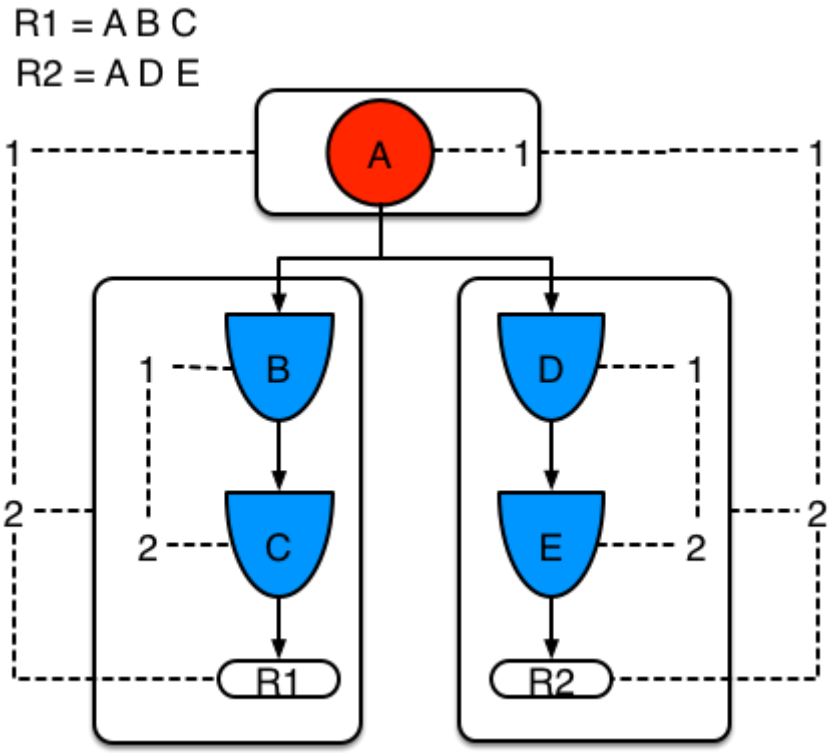


Figure 8. Example 2: Two rules with pattern sharing

Pattern A is placed in its own segment, resulting in two segments for each rule. Those two segments form a path for their respective rules. The first segment is shared by both paths. When pattern A is linked, the segment becomes linked. The segment then iterates over each path that the segment is shared by, setting the bit 1 to on. If patterns B and C are later turned on, the second segment for path R1 is linked, and this causes bit 2 to be turned on for R1. With bit 1 and bit 2 turned on for R1, the rule is now linked and a goal is created to schedule the rule for later evaluation and execution.

When a rule is evaluated, the segments enable the results of the matching to be shared. Each segment has a staging memory to queue all inserts, updates, and deletes for that segment. When R1 is evaluated, the rule processes pattern A, and this results in a set of tuples. The algorithm detects a segmentation split, creates peered tuples for each insert, update, and delete in the set, and adds them to the R2 staging memory. Those tuples are then merged with any existing staged tuples and are executed when R2 is eventually evaluated.

Example 3: Rules R3 and R4 are added and share patterns A and B.

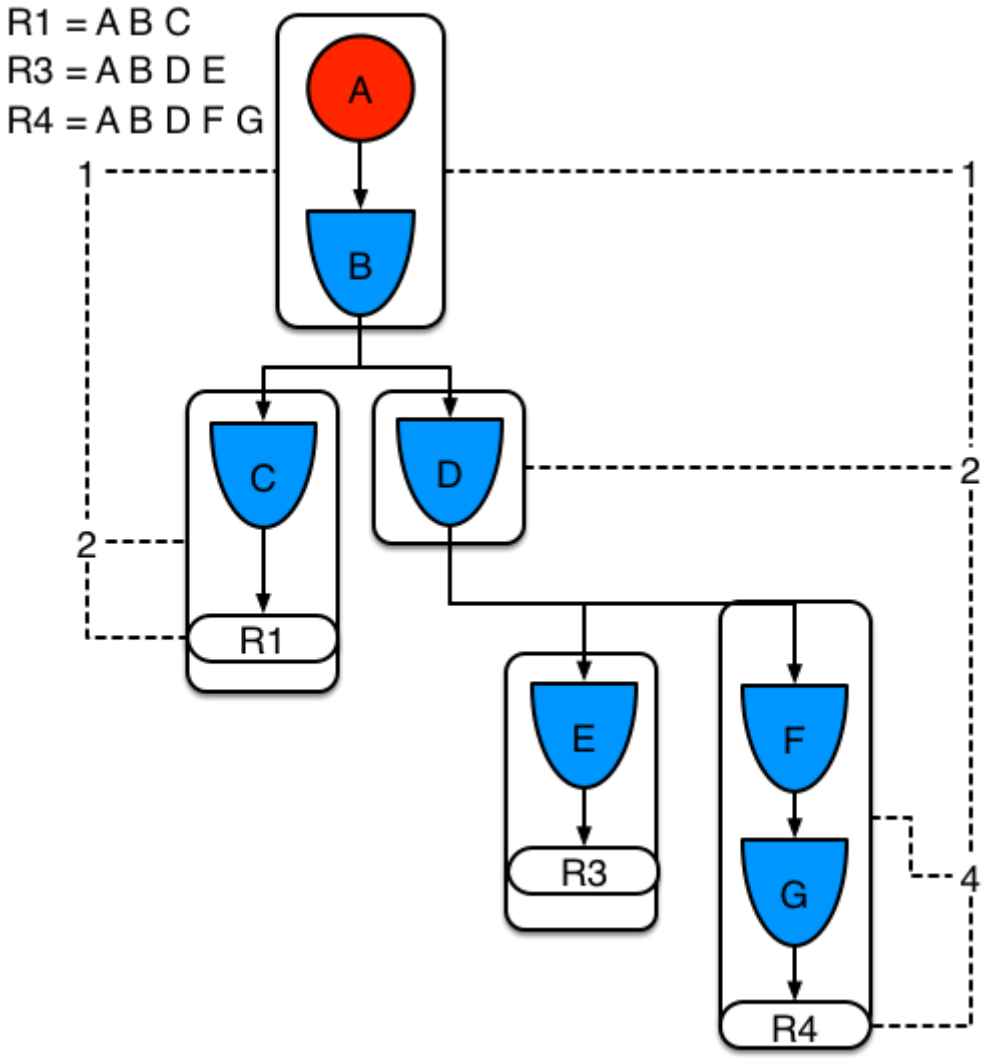


Figure 9. Example 3: Three rules with pattern sharing

Rules R3 and R4 have three segments and R1 has two segments. Patterns A and B are shared by R1, R3, and R4, while pattern D is shared by R3 and R4.

Example 4: A single rule (R1) with a subnetwork and no pattern sharing.

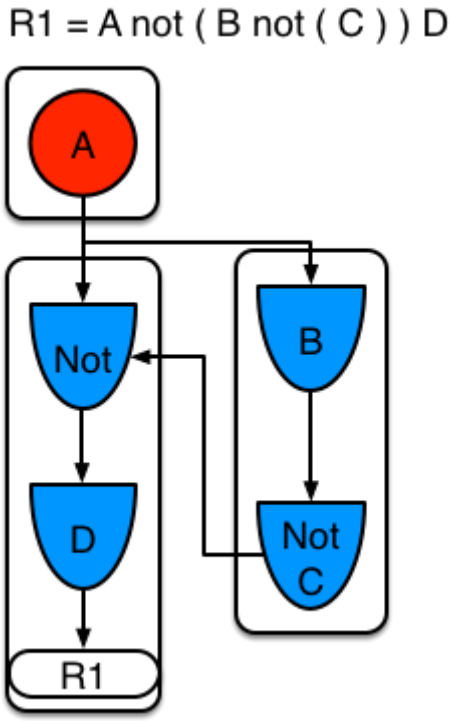


Figure 10. Example 4: Single rule with a subnetwork and no pattern sharing

Subnetworks are formed when a `Not`, `Exists`, or `Accumulate` node contains more than one element. In this example, the element `B not( C )` forms the subnetwork. The element `not( C )` is a single element that does not require a subnetwork and is therefore merged inside of the `Not` node. The subnetwork uses a dedicated segment. Rule R1 still has a path of two segments and the subnetwork forms another inner path. When the subnetwork is linked, it is also linked in the outer segment.

Example 5: Rule R1 with a subnetwork that is shared by rule R2.

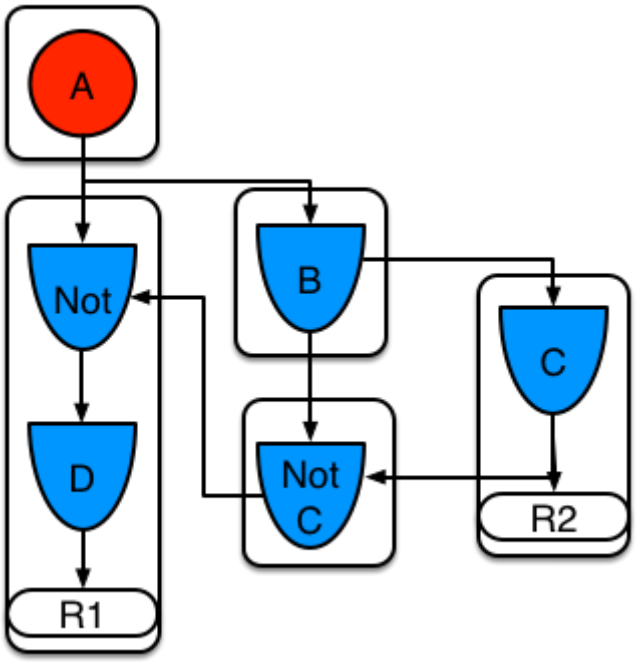


Figure 11. Example 5: Two rules, one with a subnetwork and pattern sharing

The subnetwork nodes in a rule can be shared by another rule that does not have a subnetwork.

This sharing causes the subnetwork segment to be split into two segments.

Constrained `Not` nodes and `Accumulate` nodes can never unlink a segment, and are always considered to have their bits turned on.

The Phreak evaluation algorithm is stack based instead of method-recursion based. Rule evaluation can be paused and resumed at any time when a `StackEntry` is used to represent the node currently being evaluated.

When a rule evaluation reaches a subnetwork, a `StackEntry` object is created for the outer path segment and the subnetwork segment. The subnetwork segment is evaluated first, and when the set reaches the end of the subnetwork path, the segment is merged into a staging list for the outer node that the segment feeds into. The previous `StackEntry` object is then resumed and can now process the results of the subnetwork. This process has the added benefit, especially for `Accumulate` nodes, that all work is completed in a batch, before propagating to the child node.

The same stack system is used for efficient backward chaining. When a rule evaluation reaches a query node, the evaluation is paused and the query is added to the stack. The query is then evaluated to produce a result set, which is saved in a memory location for the resumed `StackEntry` object to pick up and propagate to the child node. If the query itself called other queries, the process repeats, while the current query is paused and a new evaluation is set up for the current query node.

1.4.1.1. Rule evaluation with forward and backward chaining

The Drools rule engine in Drools is a hybrid reasoning system that uses both forward chaining and backward chaining to evaluate rules. A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the Drools rule 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 Drools rule 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 following diagram illustrates how the Drools rule engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

[RuleEvaluation] | *rule-engine/BackwardChaining/RuleEvaluation.png*

Figure 12. Rule evaluation logic using forward and backward chaining

1.4.2. Rule base configuration

Drools contains a `RuleBaseConfiguration.java` object that you can use to configure exception handler settings, multithreaded execution, and sequential mode in the Drools rule engine.

For the rule base configuration options, see the Drools RuleBaseConfiguration.java page in GitHub.

The following rule base configuration options are available for the Drools rule engine:

drools.consequenceExceptionHandler

When configured, this system property defines the class that manages the exceptions thrown by rule consequences. You can use this property to specify a custom exception handler for rule evaluation in the Drools rule engine.

Default value: `org.drools.core.runtime.rule.impl.DefaultConsequenceExceptionHandler`

You can specify the custom exception handler using one of the following options:

- Specify the exception handler in a system property:

```
drools.consequenceExceptionHandler=org.drools.core.runtime.rule.impl.MyCustomCon
sequenceExceptionHandler
```

- Specify the exception handler while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration(); kieBaseConf
.setOption(ConsequenceExceptionHandlerOption.get(MyCustomConsequenceExceptionHan
dler.class));
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

drools.multithreadEvaluation

When enabled, this system property enables the Drools rule engine to evaluate rules in parallel by dividing the Phreak rule network into independent partitions. You can use this property to increase the speed of rule evaluation for specific rule bases.

Default value: `false`

You can enable multithreaded evaluation using one of the following options:

- Enable the multithreaded evaluation system property:

```
drools.multithreadEvaluation=true
```

- Enable multithreaded evaluation while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(MultithreadEvaluationOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

Rules that use queries, salience, or agenda groups are currently not supported by the parallel Drools rule engine. If these rule elements are present in the KIE base, the compiler emits a warning and automatically switches back to single-threaded evaluation. However, in some cases, the Drools rule engine might not detect the unsupported rule elements and rules might be evaluated incorrectly. For example, the Drools rule engine might not detect when rules rely on implicit salience given by rule ordering inside the DRL file, resulting in incorrect evaluation due to the unsupported salience attribute.

drools.sequential

When enabled, this system property enables sequential mode in the Drools rule engine. In sequential mode, the Drools rule engine evaluates rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. This means that the Drools rule engine ignores any `insert`, `modify`, or `update` statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules. You can use this property if you use stateless KIE sessions and you do not want the execution of rules to influence subsequent rules in the agenda. Sequential mode applies to stateless KIE sessions only.

Default value: `false`

You can enable sequential mode using one of the following options:

- Enable the sequential mode system property:

```
drools.sequential=true
```

- Enable sequential mode while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

1.4.3. Sequential mode in Phreak

Sequential mode is an advanced rule base configuration in the Drools rule engine, supported by Phreak, that enables the Drools rule engine to evaluate rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. In sequential mode, the Drools rule engine ignores any `insert`, `modify`, or `update` statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules.

Sequential mode applies to only stateless KIE sessions because stateful KIE sessions inherently use data from previously invoked KIE sessions. If you use a stateless KIE session and you want the execution of rules to influence subsequent rules in the agenda, then do not enable sequential mode. Sequential mode is disabled by default in the Drools rule engine.

To enable sequential mode, use one of the following options:

- Set the system property `drools.sequential` to `true`.
- Enable sequential mode while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

To configure sequential mode to use a dynamic agenda, use one of the following options:

- Set the system property `drools.sequential.agenda` to `dynamic`.
- Set the sequential agenda option while creating the KIE base programmatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialAgendaOption.DYNAMIC);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

When you enable sequential mode, the Drools rule engine evaluates rules in the following way:

1. Rules are ordered by salience and position in the rule set.
2. An element for each possible rule match is created. The element position indicates the execution order.
3. Node memory is disabled, with the exception of the right-input object memory.
4. The left-input adapter node propagation is disconnected and the object with the node is referenced in a `Command` object. The `Command` object is added to a list in the working memory for later execution.
5. All objects are asserted, and then the list of `Command` objects is checked and executed.
6. All matches that result from executing the list are added to elements based on the sequence number of the rule.
7. The elements that contain matches are executed in a sequence. If you set a maximum number of rule executions, the Drools rule engine activates no more than that number of rules in the agenda for execution.

In sequential mode, the `LeftInputAdapterNode` node creates a `Command` object and adds it to a list in the working memory of the Drools rule engine. This `Command` object contains references to the `LeftInputAdapterNode` node and the propagated object. These references stop any left-input propagations at insertion time so that the right-input propagation never needs to attempt to join the left inputs. The references also avoid the need for the left-input memory.

All nodes have their memory turned off, including the left-input tuple memory, but excluding the right-input object memory. After all the assertions are finished and the right-input memory of all the objects is populated, the Drools rule engine iterates over the list of `LeftInputAdatperNode Command` objects. The objects propagate down the network, attempting to join the right-input objects, but they are not retained in the left input.

The agenda with a priority queue to schedule the tuples is replaced by an element for each rule. The sequence number of the `RuleTerminalNode` node indicates the element where to place the match. After all `Command` objects have finished, the elements are checked and existing matches are executed. To improve performance, the first and the last populated cell in the elements are retained.

When the network is constructed, each `RuleTerminalNode` node receives a sequence number based on its salience number and the order in which it was added to the network.

The right-input node memories are typically hash maps for fast object deletion. Because object deletions are not supported, Phreak uses an object list when the values of the object are not indexed. For a large number of objects, indexed hash maps provide a performance increase. If an object has only a few instances, Phreak uses an object list instead of an index.

1.5. Complex event processing (CEP)

In Drools, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing

(CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events because a change has occurred in the state of the application domain at a given time.

The Drools rule engine in Drools uses complex event processing (CEP) to detect and process multiple events within a collection of events, to uncover relationships that exist between events, and to infer new data from the events and their relationships.

CEP use cases share several requirements and goals with business rule use cases.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. In the following examples, events form the basis of business rules:

- In an algorithmic trading application, a rule performs an action if the security price increases by X percent above the day opening price. The price increases are denoted by events on a stock trading application.
- In a monitoring application, a rule performs an action if the temperature in the server room increases X degrees in Y minutes. The sensor readings are denoted by events.

From a technical perspective, business rule evaluation and CEP have the following key similarities:

- Both business rule evaluation and CEP require seamless integration with the enterprise infrastructure and applications. This is particularly important with life-cycle management, auditing, and security.
- Both business rule evaluation and CEP have functional requirements such as pattern matching, and non-functional requirements such as response time limits and query-rule explanations.

CEP scenarios have the following key characteristics:

- Scenarios usually process large numbers of events, but only a small percentage of the events are relevant.
- Events are usually immutable and represent a record of change in state.
- Rules and queries run against events and must react to detected event patterns.
- Related events usually have a strong temporal relationship.
- Individual events are not prioritized. The CEP system prioritizes patterns of related events and the relationships between them.
- Events usually need to be composed and aggregated.

Given these common CEP scenario characteristics, the CEP system in Drools supports the following features and functions to optimize event processing:

- Event processing with proper semantics
- Event detection, correlation, aggregation, and composition
- Event stream processing

- Temporal constraints to model the temporal relationships between events
- Sliding windows of significant events
- Session-scoped unified clock
- Required volumes of events for CEP use cases
- Reactive rules
- Adapters for event input into the Drools rule engine (pipeline)

1.5.1. Events in complex event processing

In Drools, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing (CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events because a change has occurred in the state of the application domain at a given time.

Events have the following key characteristics:

- **Are immutable:** An event is a record of change that has occurred at some time in the past and cannot be changed.

The Drools rule engine does not enforce immutability on the Java objects that represent events. This behavior makes event data enrichment possible. Your application should be able to populate unpopulated event attributes, and these attributes are used by the Drools rule engine to enrich the event with inferred data. However, you should not change event attributes that have already been populated.

- **Have strong temporal constraints:** Rules involving events usually require the correlation of multiple events that occur at different points in time relative to each other.
- **Have managed life cycles:** Because events are immutable and have temporal constraints, they are usually only relevant for a specified period of time. This means that the Drools rule engine can automatically manage the life cycle of events.
- **Can use sliding windows:** You can define sliding windows of time or length with events. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed.

1.5.2. Declaring facts as events

You can declare facts as events in your Java class or DRL rule file so that the Drools rule engine handles the facts as events during complex event processing. You can declare the facts as interval-based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the Drools rule 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.

Procedure

For the relevant fact type in your Java class or DRL rule file, enter the `@role( event )` metadata tag and parameter. The `@role` metadata tag accepts the following two values:

- `fact`: (Default) Declares the type as a regular fact
- `event`: Declares the type as an event

For example, the following snippet declares that the `StockPoint` fact type in a stock broker application must be handled as an event:

Declare fact type as an event

```
import some.package.StockPoint

declare StockPoint
  @role( event )
end
```

If `StockPoint` is a fact type declared in the DRL rule file instead of in a pre-existing class, you can declare the event in-line in your application code:

Declare fact type in-line and assign it to event role

```
declare StockPoint
  @role( event )

  datetime : java.util.Date
  symbol : String
  price : double
end
```

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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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
```

1.5.3. Event processing modes in the Drools rule engine

The Drools rule engine runs in either cloud mode or stream mode. In cloud mode, the Drools rule engine processes facts as facts with no temporal constraints, independent of time, and in no particular order. In stream mode, the Drools rule engine processes facts as events with strong temporal constraints, in real time or near real time. Stream mode uses synchronization to make event processing possible in Drools.

Cloud mode

Cloud mode is the default operating mode of the Drools rule engine. In cloud mode, the Drools rule engine treats events as an unordered cloud. Events still have time stamps, but the Drools rule engine running in cloud mode cannot draw relevance from the time stamp because cloud mode ignores the present time. This mode uses the rule constraints to find the matching tuples to activate and execute rules.

Cloud mode does not impose any kind of additional requirements on facts. However, because

the Drools rule engine in this mode has no concept of time, it cannot use temporal features such as sliding windows or automatic life-cycle management. In cloud mode, events must be explicitly retracted when they are no longer needed.

The following requirements are not imposed in cloud mode:

- No clock synchronization because the Drools rule engine has no notion of time
- No ordering of events because the Drools rule engine processes events as an unordered cloud, against which the Drools rule engine match rules

You can specify cloud mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set cloud mode using system property

```
drools.eventProcessingMode=cloud
```

Set cloud mode using Java client API

```
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.CLOUD);
```

You can also specify cloud mode using the `eventProcessingMode="<mode>"` KIE base attribute in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

Set cloud mode using project `kmodule.xml` file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="cloud" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

Stream mode

Stream mode enables the Drools rule engine to process events chronologically and in real time as they are inserted into the Drools rule engine. In stream mode, the Drools rule engine synchronizes streams of events (so that events in different streams can be processed in chronological order), implements sliding windows of time or length, and enables automatic life-cycle management.

The following requirements apply to stream mode:

- Events in each stream must be ordered chronologically.
- A session clock must be present to synchronize event streams.

Your application does not need to enforce ordering events between streams, but using event streams that have not been synchronized may cause unexpected results.

You can specify stream mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set stream mode using system property

```
drools.eventProcessingMode=stream
```

Set stream mode using Java client API

```
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.STREAM);
```

You can also specify stream mode using the `eventProcessingMode="<mode>"` KIE base attribute in the KIE module descriptor file (`kmodule.xml`) for a specific Drools project:

Set stream mode using project `kmodule.xml` file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream" packages=
"org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

1.5.3.1. Negative patterns in Drools rule engine stream mode

A negative pattern is a pattern for conditions that are not met. For example, the following DRL rule activates a fire alarm if a fire is detected and the sprinkler is not activated:

Fire alarm rule with a negative pattern

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated())
then
  // Sound the alarm.
end
```

In cloud mode, the Drools rule engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately. In stream mode, the Drools rule engine can support temporal constraints on facts to wait for a set time before activating a rule.

The same example rule in stream mode activates the fire alarm as usual, but applies a 10-second delay.

Fire alarm rule with a negative pattern and time delay (stream mode only)

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated(this after[0s,10s] $f))
then
  // Sound the alarm.
end
```

The following modified fire alarm rule expects one `Heartbeat` event to occur every 10 seconds. If the expected event does not occur, the rule is executed. This rule uses the same type of object in both the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait 0 to 10 seconds before executing and excludes the `Heartbeat` event bound to `$h` so that the rule can be executed. The bound event `$h` must be explicitly excluded in order for the rule to be executed because the temporal constraint `[0s, ⋯]` does not inherently exclude that event from being matched again.

Fire alarm rule excluding a bound event in a negative pattern (stream mode only)

```
rule "Sound the alarm"
when
  $h: Heartbeat() from entry-point "MonitoringStream"
  not(Heartbeat(this != $h, this after[0s,10s] $h) from entry-point
"MonitoringStream")
then
  // Sound the alarm.
end
```

1.5.4. Property-change settings and listeners for fact types

By default, the Drools rule 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 Drools rule 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 Drools distribution:

Example property reactivity setting in system properties

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

The Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
    @propertyChangeSupport
end
```

1.5.5. Temporal operators for events

In stream mode, the Drools rule engine supports the following temporal operators for events that are inserted into the working memory of the Drools rule engine. You can use these operators to define the temporal reasoning behavior of the events that you declare in your Java class or DRL rule file. Temporal operators are not supported when the Drools rule engine is running in cloud mode.

- `after`

- `before`
- `coincides`
- `during`
- `includes`
- `finishes`
- `finished by`
- `meets`
- `met by`
- `overlaps`
- `overlapped by`
- `starts`
- `started by`

after

This operator specifies if the current event occurs after the correlated event. This operator can also define an amount of time after which the current event can follow the correlated event, or a delimiting time range during which the current event can follow the correlated event.

For example, the following pattern matches if `$eventA` starts between 3 minutes and 30 seconds and 4 minutes after `$eventB` finishes. If `$eventA` starts earlier than 3 minutes and 30 seconds after `$eventB` finishes, or later than 4 minutes after `$eventB` finishes, then the pattern is not matched.

```
$eventA : EventA(this after[3m30s, 4m] $eventB)
```

You can also express this operator in the following way:

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The `after` operator supports up to two parameter values:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at 1 millisecond and runs indefinitely with no end time.

The `after` operator also supports negative time ranges:

```
$eventA : EventA(this after[-3m30s, -2m] $eventB)
```

If the first value is greater than the second value, the Drools rule engine automatically reverses them. For example, the following two patterns are interpreted by the Drools rule engine in the same way:

```
$eventA : EventA(this after[-3m30s, -2m] $eventB)
$eventA : EventA(this after[-2m, -3m30s] $eventB)
```

before

This operator specifies if the current event occurs before the correlated event. This operator can also define an amount of time before which the current event can precede the correlated event, or a delimiting time range during which the current event can precede the correlated event.

For example, the following pattern matches if `$eventA` finishes between 3 minutes and 30 seconds and 4 minutes before `$eventB` starts. If `$eventA` finishes earlier than 3 minutes and 30 seconds before `$eventB` starts, or later than 4 minutes before `$eventB` starts, then the pattern is not matched.

```
$eventA : EventA(this before[3m30s, 4m] $eventB)
```

You can also express this operator in the following way:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The `before` operator supports up to two parameter values:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at 1 millisecond and runs indefinitely with no end time.

The `before` operator also supports negative time ranges:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
```

If the first value is greater than the second value, the Drools rule engine automatically reverses them. For example, the following two patterns are interpreted by the Drools rule engine in the same way:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
$eventA : EventA(this before[-2m, -3m30s] $eventB)
```

coincides

This operator specifies if the two events occur at the same time, with the same start and end times.

For example, the following pattern matches if both the start and end time stamps of `$eventA` and `$eventB` are identical:

```
$eventA : EventA(this coincides $eventB)
```

The `coincides` operator supports up to two parameter values for the distance between the event start and end times, if they are not identical:

- If only one parameter is given, the parameter is used to set the threshold for both the start and end times of both events.
- If two parameters are given, the first is used as a threshold for the start time and the second is used as a threshold for the end time.

The following pattern uses start and end time thresholds:

```
$eventA : EventA(this coincides[15s, 10s] $eventB)
```

The pattern matches if the following conditions are met:

```
abs($eventA.startTimestamp - $eventB.startTimestamp) <= 15s
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 10s
```

The Drools rule engine does not support negative intervals for the `coincides` operator. If you use negative intervals, the Drools rule engine generates an error.

during

This operator specifies if the current event occurs within the time frame of when the correlated event starts and ends. The current event must start after the correlated event starts and must end before the correlated event ends. (With the `coincides` operator, the start and end times are the same or nearly the same.)

For example, the following pattern matches if `$eventA` starts after `$eventB` starts and ends before `$eventB` ends:

```
$eventA : EventA(this during $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The `during` operator supports one, two, or four optional parameters:

- If one value is defined, this value is the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values are a threshold between which the current event start time and end time must occur in relation to the correlated event start and end times.

  For example, if the values are `5s` and `10s`, the current event must start between 5 and 10 seconds after the correlated event starts and must end between 5 and 10 seconds before the correlated event ends.
- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

includes

This operator specifies if the correlated event occurs within the time frame of when the current event occurs. The correlated event must start after the current event starts and must end before the current event ends. (The behavior of this operator is the reverse of the `during` operator behavior.)

For example, the following pattern matches if `$eventB` starts after `$eventA` starts and ends before `$eventA` ends:

```
$eventA : EventA(this includes $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The `includes` operator supports one, two, or four optional parameters:

- If one value is defined, this value is the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values are a threshold between which the correlated event start time and end time must occur in relation to the current event start and end

times.

For example, if the values are `5s` and `10s`, the correlated event must start between 5 and 10 seconds after the current event starts and must end between 5 and 10 seconds before the current event ends.

- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

finishes

This operator specifies if the current event starts after the correlated event but both events end at the same time.

For example, the following pattern matches if `$eventA` starts after `$eventB` starts and ends at the same time when `$eventB` ends:

```
$eventA : EventA(this finishes $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The `finishes` operator supports one optional parameter that sets the maximum time allowed between the end times of the two events:

```
$eventA : EventA(this finishes[5s] $eventB)
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 5s
```

The Drools rule engine does not support negative intervals for the `finishes` operator. If you use negative intervals, the Drools rule engine generates an error.

finished by

This operator specifies if the correlated event starts after the current event but both events end at the same time. (The behavior of this operator is the reverse of the `finishes` operator behavior.)

For example, the following pattern matches if `$eventB` starts after `$eventA` starts and ends at the same time when `$eventA` ends:

```
$eventA : EventA(this finishedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The `finished by` operator supports one optional parameter that sets the maximum time allowed between the end times of the two events:

```
$eventA : EventA(this finishedby[5s] $eventB)
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 5s
```

The Drools rule engine does not support negative intervals for the `finished by` operator. If you use negative intervals, the Drools rule engine generates an error.

meets

This operator specifies if the current event ends at the same time when the correlated event starts.

For example, the following pattern matches if `$eventA` ends at the same time when `$eventB` starts:

```
$eventA : EventA(this meets $eventB)
```

You can also express this operator in the following way:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) == 0
```

The `meets` operator supports one optional parameter that sets the maximum time allowed between the end time of the current event and the start time of the correlated event:

```
$eventA : EventA(this meets[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

The Drools rule engine does not support negative intervals for the `meets` operator. If you use negative intervals, the Drools rule engine generates an error.

met by

This operator specifies if the correlated event ends at the same time when the current event starts. (The behavior of this operator is the reverse of the `meets` operator behavior.)

For example, the following pattern matches if `$eventB` ends at the same time when `$eventA` starts:

```
$eventA : EventA(this metby $eventB)
```

You can also express this operator in the following way:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) == 0
```

The `met by` operator supports one optional parameter that sets the maximum distance between the end time of the correlated event and the start time of the current event:

```
$eventA : EventA(this metby[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

The Drools rule engine does not support negative intervals for the `met by` operator. If you use negative intervals, the Drools rule engine generates an error.

overlaps

This operator specifies if the current event starts before the correlated event starts and it ends during the time frame that the correlated event occurs. The current event must end between the start and end times of the correlated event.

For example, the following pattern matches if `$eventA` starts before `$eventB` starts and then

ends while `$eventB` occurs, before `$eventB` ends:

```
$eventA : EventA(this overlaps $eventB)
```

The `overlaps` operator supports up to two parameters:

- If one parameter is defined, the value is the maximum distance between the start time of the correlated event and the end time of the current event.
- If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the correlated event and the end time of the current event.

overlapped by

This operator specifies if the correlated event starts before the current event starts and it ends during the time frame that the current event occurs. The correlated event must end between the start and end times of the current event. (The behavior of this operator is the reverse of the `overlaps` operator behavior.)

For example, the following pattern matches if `$eventB` starts before `$eventA` starts and then ends while `$eventA` occurs, before `$eventA` ends:

```
$eventA : EventA(this overlappedby $eventB)
```

The `overlapped by` operator supports up to two parameters:

- If one parameter is defined, the value is the maximum distance between the start time of the current event and the end time of the correlated event.
- If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the current event and the end time of the correlated event.

starts

This operator specifies if the two events start at the same time but the current event ends before the correlated event ends.

For example, the following pattern matches if `$eventA` and `$eventB` start at the same time, and `$eventA` ends before `$eventB` ends:

```
$eventA : EventA(this starts $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

The `starts` operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$eventA : EventA(this starts[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventA.startTimestamp - $eventB.startTimestamp) <= 5s
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

The Drools rule engine does not support negative intervals for the `starts` operator. If you use negative intervals, the Drools rule engine generates an error.

started by

This operator specifies if the two events start at the same time but the correlated event ends before the current event ends. (The behavior of this operator is the reverse of the `starts` operator behavior.)

For example, the following pattern matches if `$eventA` and `$eventB` start at the same time, and `$eventB` ends before `$eventA` ends:

```
$eventA : EventA(this startedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

The `started by` operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$eventA : EventA( this starts[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

The Drools rule engine does not support negative intervals for the `started by` operator. If you use negative intervals, the Drools rule engine generates an error.

1.5.6. Session clock implementations in the Drools rule engine

During complex event processing, events in the Drools rule engine may have temporal constraints and therefore require a session clock that provides the current time. For example, if a rule needs to determine the average price of a given stock over the last 60 minutes, the Drools rule engine must be able to compare the stock price event time stamp with the current time in the session clock.

The Drools rule engine supports a real-time clock and a pseudo clock. You can use one or both clock types depending on the scenario:

- **Rules testing:** Testing requires a controlled environment, and when the tests include rules with temporal constraints, you must be able to control the input rules and facts and the flow of time.
- **Regular execution:** The Drools rule engine reacts to events in real time and therefore requires a real-time clock.
- **Special environments:** Specific environments may have specific time control requirements. For example, clustered environments may require clock synchronization or Java Enterprise Edition (JEE) environments may require a clock provided by the application server.
- **Rules replay or simulation:** In order to replay or simulate scenarios, the application must be able to control the flow of time.

Consider your environment requirements as you decide whether to use a real-time clock or pseudo clock in the Drools rule engine.

Real-time clock

The real-time clock is the default clock implementation in the Drools rule engine and uses the system clock to determine the current time for time stamps. To configure the Drools rule engine to use the real-time clock, set the KIE session configuration parameter to `realtime`:

Configure real-time clock in KIE session

```
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.conf.ClockTypeOption;
import org.kie.api.runtime.KieSessionConfiguration;

KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("realtime"));
```

Pseudo clock

The pseudo clock implementation in the Drools rule engine is helpful for testing temporal rules and it can be controlled by the application. To configure the Drools rule engine to use the pseudo clock, set the KIE session configuration parameter to `pseudo`:

Configure pseudo clock in KIE session

```
import org.kie.api.runtime.conf.ClockTypeOption;
import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;

KieSessionConfiguration config = KieServices.Factory.get()
.newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("pseudo"));
```

You can also use additional configurations and fact handlers to control the pseudo clock:

Control pseudo clock behavior in KIE session

```
import java.util.concurrent.TimeUnit;

import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.KieSession;
import org.drools.core.time.SessionPseudoClock;
import org.kie.api.runtime.rule.FactHandle;
import org.kie.api.runtime.conf.ClockTypeOption;

KieSessionConfiguration conf = KieServices.Factory.get().
newKieSessionConfiguration();

conf.setOption( ClockTypeOption.get("pseudo"));
KieSession session = kbase.newKieSession(conf, null);

SessionPseudoClock clock = session.getSessionClock();

// While inserting facts, advance the clock as necessary.
FactHandle handle1 = session.insert(tick1);
clock.advanceTime(10, TimeUnit.SECONDS);

FactHandle handle2 = session.insert(tick2);
clock.advanceTime(30, TimeUnit.SECONDS);

FactHandle handle3 = session.insert(tick3);
```

1.5.7. Event streams and entry points

The Drools rule engine can process high volumes of events in the form of event streams. In DRL

rule declarations, a stream is also known as an *entry point*. When you declare an entry point in a DRL rule or Java application, the Drools rule engine, at compile time, identifies and creates the proper internal structures to use data from only that entry point to evaluate that rule.

Facts from one entry point, or stream, can join facts from any other entry point in addition to facts already in the working memory of the Drools rule engine. Facts always remain associated with the entry point through which they entered the Drools rule engine. Facts of the same type can enter the Drools rule engine through several entry points, but facts that enter the Drools rule engine through entry point A can never match a pattern from entry point B.

Event streams have the following characteristics:

- Events in the stream are ordered by time stamp. The time stamps may have different semantics for different streams, but they are always ordered internally.
- Event streams usually have a high volume of events.
- Atomic events in streams are usually not useful individually, only collectively in a stream.
- Event streams can be homogeneous and contain a single type of event, or heterogeneous and contain events of different types.

1.5.7.1. Declaring entry points for rule data

You can declare an entry point (event stream) for events so that the Drools rule 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.

Procedure

Use one of the following methods to declare the entry point:

- In the DRL rule file, specify `from entry-point "<name>"` for the inserted fact:

  Authorize withdrawal rule with "ATM Stream" 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
```

Apply fee rule with "Branch Stream" entry point

```
rule "Apply fee on withdraws on branches"
when
  WithdrawRequest($ai : accountId, processed == true) from entry-point "Branch
Stream"
  CheckingAccount(accountId == $ai)
then
  // Apply a $2 fee on the account.
end
```

Both example DRL rules from a banking application insert the event `WithdrawalRequest` with the fact `CheckingAccount`, but from different entry points. At run time, the Drools rule engine evaluates the `Authorize withdrawal` rule using data from only the `"ATM Stream"` entry point, and evaluates the `Apply fee` rule using data from only the `"Branch Stream"` entry point. Any events inserted into the `"ATM Stream"` can never match patterns for the `"Apply fee"` rule, and any events inserted into the `"Branch Stream"` can never match patterns for the `"Authorize withdrawal rule"`.

- In the Java application code, use the `getEntryPoint()` method to specify and obtain an `EntryPoint` object and insert facts into that entry point accordingly:

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);
```

Any DRL rules that specify `from entry-point "ATM Stream"` are then evaluated based on the data in this entry point only.

1.5.8. Sliding windows of time or length

In stream mode, the Drools rule engine can process events from a specified sliding window of time or length. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed. When you declare a sliding window in a DRL rule or Java application, the Drools rule engine, at compile time, identifies and creates the proper internal structures to use data from only that sliding window to evaluate that rule.

For example, the following DRL rule snippets instruct the Drools rule engine to process only the stock points from the last 2 minutes (sliding time window) or to process only the last 10 stock points

(sliding length window):

Process stock points from the last 2 minutes (sliding time window)

```
StockPoint() over window:time(2m)
```

Process the last 10 stock points (sliding length window)

```
StockPoint() over window:length(10)
```

1.5.8.1. Declaring sliding windows for rule data

You can declare a sliding window of time (flow of time) or length (number of occurrences) for events so that the Drools rule engine uses data from only that window to evaluate the rules.

Procedure

In the DRL rule file, specify `over window:<time_or_length>(<value>)` for the inserted fact.

For example, the following two DRL rules activate a fire alarm based on an average temperature. However, the first rule uses a sliding time window to calculate the average over the last 10 minutes while the second rule uses a sliding length window to calculate the average over the last one hundred temperature readings.

Average temperature over sliding time window

```
rule "Sound the alarm if temperature rises above threshold"
when
  TemperatureThreshold($max : max)
  Number(doubleValue > $max) from accumulate(
    SensorReading($temp : temperature) over window:time(10m),
    average($temp))
then
  // Sound the alarm.
end
```

Average temperature over sliding length window

```
rule "Sound the alarm if temperature rises above threshold"
when
  TemperatureThreshold($max : max)
  Number(doubleValue > $max) from accumulate(
    SensorReading($temp : temperature) over window:length(100),
    average($temp))
then
  // Sound the alarm.
end
```

The Drools rule engine discards any `SensorReading` events that are more than 10 minutes old or that

are not part of the last one hundred readings, and continues recalculating the average as the minutes or readings "slide" forward in real time.

The Drools rule engine does not automatically remove outdated events from the KIE session because other rules without sliding window declarations might depend on those events. The Drools rule engine stores events in the KIE session until the events expire either by explicit rule declarations or by implicit reasoning within the Drools rule engine based on inferred data in the KIE base.

1.5.9. Memory management for events

In stream mode, the Drools rule engine uses automatic memory management to maintain events that are stored in KIE sessions. The Drools rule engine can retract from a KIE session any events that no longer match any rule due to their temporal constraints and release any resources held by the retracted events.

The Drools rule engine uses either explicit or inferred expiration to retract outdated events:

- **Explicit expiration:** The Drools rule engine removes events that are explicitly set to expire in rules that declare the `@expires` tag:

  DRL rule snippet with explicit expiration

  ```
  declare StockPoint
    @expires( 30m )
  end
  ```

  This example rule sets any `StockPoint` events to expire after 30 minutes and to be removed from the KIE session if no other rules use the events.

- **Inferred expiration:** The Drools rule engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules:

  DRL rule with temporal constraints

  ```
  rule "Correlate orders"
  when
    $bo : BuyOrder($id : id)
    $ae : AckOrder(id == $id, this after[0,10s] $bo)
  then
    // Perform an action.
  end
  ```

  For this example rule, the Drools rule engine automatically calculates that whenever a `BuyOrder` event occurs, the Drools rule engine needs to store the event for up to 10 seconds and wait for the matching `AckOrder` event. After 10 seconds, the Drools rule engine infers the expiration and removes the event from the KIE session. An `AckOrder` event can only match an existing `BuyOrder` event, so the Drools rule engine infers the expiration if no match occurs and removes the event immediately.

The Drools rule engine analyzes the entire KIE base to find the offset for every event type and to ensure that no other rules use the events that are pending removal. Whenever an implicit expiration clashes with an explicit expiration value, the Drools rule engine uses the greater time frame of the two to store the event longer.

1.6. Drools rule engine queries and live queries

You can use queries with the Drools rule engine to retrieve fact sets based on fact patterns as they are used in rules. The patterns might also use optional parameters.

To use queries with the Drools rule engine, you add the query definitions in DRL files and then obtain the matching results in your application code. While a query iterates over a result collection, you can use any identifier that is bound to the query to access the corresponding fact or fact field by calling the `get()` method with the binding variable name as the argument. If the binding refers to a fact object, you can retrieve the fact handle by calling `getFactHandle()` with the variable name as the parameter.

[QueryResults] | *rule-engine/QueryResults.png*

Figure 13. QueryResults

[QueryResultsRow] | *rule-engine/QueryResultsRow.png*

Figure 14. QueryResultsRow

Example query definition in a DRL file

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

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" );
}
```

Invoking queries and processing the results by iterating over the returned set can be difficult when you are monitoring changes over time. To alleviate this difficulty with ongoing queries, Drools provides *live queries*, which use an attached listener for change events instead of returning an iterable result set. Live queries remain open by creating a view and publishing change events for the contents of this view.

To activate a live query, start your query with parameters and monitor changes in the resulting

view. You can use the `dispose()` method to terminate the query and discontinue this reactive scenario.

Example query definition in a DRL file

```
query colors(String $color1, String $color2)
    TShirt(mainColor = $color1, secondColor = $color2, $price: manufactureCost)
end
```

Example application code with an event listener and a live query

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
 public void rowUpdated(Row row) {
  updated.add( row.get( "$price" ) );
 }

 public void rowRemoved(Row row) {
  removed.add( row.get( "$price" ) );
 }

 public void rowAdded(Row row) {
  added.add( row.get( "$price" ) );
 }
};

// Open the live query:
LiveQuery query = ksession.openLiveQuery( "colors",
                                          new Object[] { "red", "blue" },
                                          listener );
...
...

// Terminate the live query:
query.dispose()
```

For more live query examples, see Glazed Lists examples for Drools Live Queries.

1.7. Drools rule engine event listeners and debug logging

The Drools rule engine generates events when performing activities such as fact insertions and rule executions. If you register event listeners, the Drools rule engine calls every listener when an activity is performed.

Event listeners have methods that correspond to different types of activities. The Drools rule engine passes an event object to each method; this object contains information about the specific activity.

Your code can implement custom event listeners and you can also add and remove registered event listeners. In this way, your code can be notified of Drools rule engine activity, and you can separate logging and auditing work from the core of your application.

The Drools rule engine supports the following event listeners with the following methods:

Agenda event listener

```
public interface AgendaEventListener
    extends
    EventListener {
    void matchCreated(MatchCreatedEvent event);
    void matchCancelled(MatchCancelledEvent event);
    void beforeMatchFired(BeforeMatchFiredEvent event);
    void afterMatchFired(AfterMatchFiredEvent event);
    void agendaGroupPopped(AgendaGroupPoppedEvent event);
    void agendaGroupPushed(AgendaGroupPushedEvent event);
    void beforeRuleFlowGroupActivated(RuleFlowGroupActivatedEvent event);
    void afterRuleFlowGroupActivated(RuleFlowGroupActivatedEvent event);
    void beforeRuleFlowGroupDeactivated(RuleFlowGroupDeactivatedEvent event);
    void afterRuleFlowGroupDeactivated(RuleFlowGroupDeactivatedEvent event);
}
```

Rule runtime event listener

```
public interface RuleRuntimeEventListener extends EventListener {
    void objectInserted(ObjectInsertedEvent event);
    void objectUpdated(ObjectUpdatedEvent event);
    void objectDeleted(ObjectDeletedEvent event);
}
```

For the definitions of event classes, see the GitHub repository.

Drools includes default implementations of these listeners: `DefaultAgendaEventListener` and `DefaultRuleRuntimeEventListener`. You can extend each of these implementations to monitor specific events.

For example, the following code extends `DefaultAgendaEventListener` to monitor the `AfterMatchFiredEvent` event and attaches this listener to a KIE session. The code prints pattern matches when rules are executed (fired):

Example code to monitor and print `AfterMatchFiredEvent` events in the agenda

```
ksession.addEventListener( new DefaultAgendaEventListener() {
   public void afterMatchFired(AfterMatchFiredEvent event) {
       super.afterMatchFired( event );
       System.out.println( event );
   }
});
```

Drools also includes the following Drools rule engine agenda and rule runtime event listeners for debug logging:

- `DebugAgendaEventListener`
- `DebugRuleRuntimeEventListener`

These event listeners implement the same supported event-listener methods and include a debug print statement by default. You can add additional monitoring code for a specific supported event.

For example, the following code uses the `DebugRuleRuntimeEventListener` event listener to monitor and print all working memory (rule runtime) events:

Example code to monitor and print all working memory events

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

1.7.1. Practices for development of event listeners

The Drools rule engine calls event listeners during rule processing. The calls block the execution of the Drools rule engine. Therefore, the event listener can affect the performance of the Drools rule engine.

To ensure minimal disruption, follow the following guidelines:

- Any action must be as short as possible.
- A listener class must not have a state. The Drools rule engine can destroy and re-create a listener class at any time.
- Do not use logic that relies on the order of execution of different event listeners.
- Do not include interactions with different entities outside the Drools rule engine within a listener. For example, do not include REST calls for notification of events. An exception is the output of logging information; however, a logging listener must be as simple as possible.
- You can use a listener to modify the state of the Drools rule engine, for example, to change the values of variables.

1.7.2. Configuring a logging utility in the Drools rule engine

The Drools rule engine uses the Java logging API SLF4J for system logging. You can use one of the following logging utilities with the Drools rule engine to investigate Drools rule engine activity, such

as for troubleshooting or data gathering:

- Logback
- Apache Commons Logging
- Apache Log4j
- `java.util.logging` package

Procedure

For the logging utility that you want to use, add the relevant dependency to your Maven project or save the relevant XML configuration file in the `org.drools` package of your Drools distribution:

Example Maven dependency for Logback

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>
```

Example logback.xml configuration file in org.drools package

```
<configuration>
  <logger name="org.drools" level="debug"/>
  ...
<configuration>
```

Example log4j.xml configuration file in org.drools package

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <category name="org.drools">
    <priority value="debug" />
  </category>
  ...
</log4j:configuration>
```

If you are developing for an ultra light environment, use the `slf4j-nop` or `slf4j-simple` logger.

1.8. Performance tuning considerations with the Drools rule engine

The following key concepts or suggested practices can help you optimize Drools rule 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 Drools.

Use sequential mode for stateless KIE sessions that do not require important Drools rule engine updates

Sequential mode is an advanced rule base configuration in the Drools rule engine that enables the Drools rule engine to evaluate rules one time in the order that they are listed in the Drools rule engine agenda without regard to changes in the working memory. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules. Sequential mode applies to stateless KIE sessions only.

To enable sequential mode, set the system property `drools.sequential` to `true`.

For more information about sequential mode or other options for enabling it, see Sequential mode in Phreak.

Use simple operations with event listeners

Limit the number of event listeners and the type of operations they perform. Use event listeners for simple operations, such as debug logging and setting properties. Complicated operations, such as network calls, in listeners can impede rule execution. After you finish working with a KIE session, remove the attached event listeners so that the session can be cleaned, as shown in the following example:

Example event listener removed after use

```
Listener listener = ...;
StatelessKnowledgeSession ksession = createSession();
try {
    ksession.insert(fact);
    ksession.fireAllRules();
    ...
} finally {
    if (session != null) {
        ksession.detachListener(listener);
        ksession.dispose();
    }
}
```

For information about built-in event listeners and debug logging in the Drools rule engine, see Drools rule engine event listeners and debug logging.

Configure `LambdaIntrospector` cache size for an executable model build

You can configure the size of `LambdaIntrospector.methodFingerprintsMap` cache, which is used in an executable model build. The default size of the cache is `32`. When you configure smaller value for the cache size, it reduces memory usage. For example, you can configure system property `drools.lambda.introspector.cache.size` to `0` for minimum memory usage. Note that smaller cache size also slows down the build performance.

Use lambda externalization for executable model

Enable lambda externalization to optimize the memory consumption during runtime. It rewrites lambdas that are generated and used in the executable model. This enables you to reuse the same lambda multiple times with all the patterns and the same constraint. When the rete or

phreak is instantiated, the executable model becomes garbage collectible.

To enable lambda externalization for the executable model, include the following property:

```
-Ddrools.externaliseCanonicalModelLambda=true
```

Configure alpha node range index threshold

Alpha node range index is used to evaluate the rule constraint. You can configure the threshold of the alpha node range index using the `drools.alphaNodeRangeIndexThreshold` system property. The default value of the threshold is `9`, indicating that the alpha node range index is enabled when a precedent node contains more than nine alpha nodes with inequality constraints. For example, when you have nine rules similar to `Person(age > 10)`, `Person(age > 20)`, ..., `Person(age > 90)`, then you can have similar nine alpha nodes.

The default value of the threshold is based on the related advantages and overhead. However, if you configure a smaller value for the threshold, then the performance can be improved depending on your rules. For example, you can configure the `drools.alphaNodeRangeIndexThreshold` value to `6`, enabling the alpha node range index when you have more than six alpha nodes for a precedent node. You can set a suitable value for the threshold based on the performance test results of your rules.

Enable join node range index

The join node range index feature improves the performance only when there is a large number of facts to be joined, for example, 256*16 combinations. When your application inserts a large number of facts, you can enable the join node range index and evaluate the performance improvement. By default, the join node range index is disabled.

Example `kmodule.xml` file

```
<kbase name="KBase1" betaRangeIndex="enabled">
```

System property for `BetaRangeIndexOption`

```
drools.betaNodeRangeIndexEnabled=true
```

Chapter 2. Rule Language Reference

2.1. 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 {CENTRAL} are ultimately rendered. You can create and manage DRL files within the {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 {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 {CENTRAL}. Assets in the same package are imported by default. Existing assets in other packages can be imported with the DRL rule.

2.1.1. Packages in DRL

A package is a folder of related assets in Drools, 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;
```

The following railroad diagram shows all the components that may make up a package:

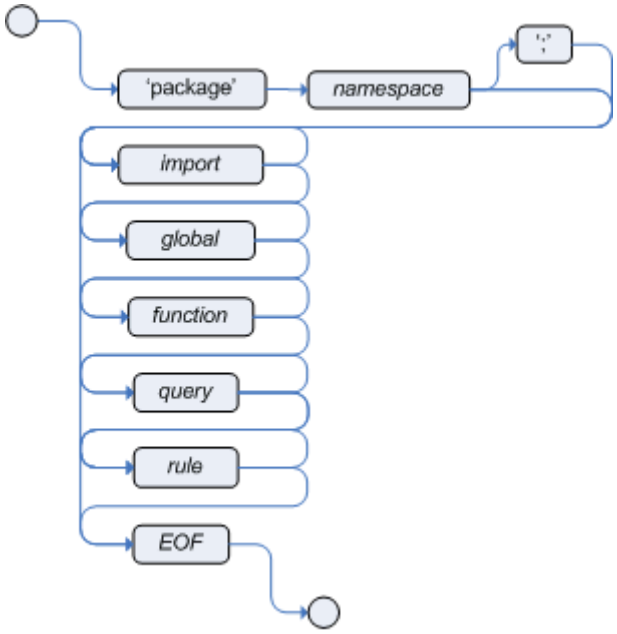


Figure 15. Package

Note that a package *must* have a namespace and be declared using standard Java conventions for package names; i.e., no spaces, unlike rule names which allow spaces. In terms of the order of elements, they can appear in any order in the rule file, with the exception of the `package` statement, which must be at the top of the file. In all cases, the semicolons are optional.

Notice that any rule attribute (as described the section Rule Attributes) may also be written at package level, superseding the attribute's default value. The modified default may still be replaced by an attribute setting within a rule.

2.1.2. Import statements in DRL

Figure 16. Import

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 Drools rule 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;
```

2.1.3. Functions in DRL

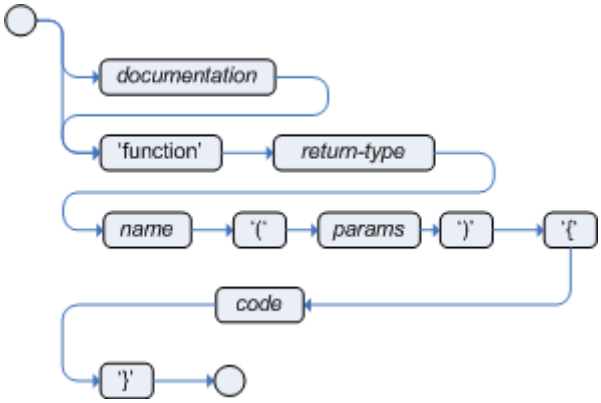


Figure 17. Function

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
```

2.1.4. Queries in DRL

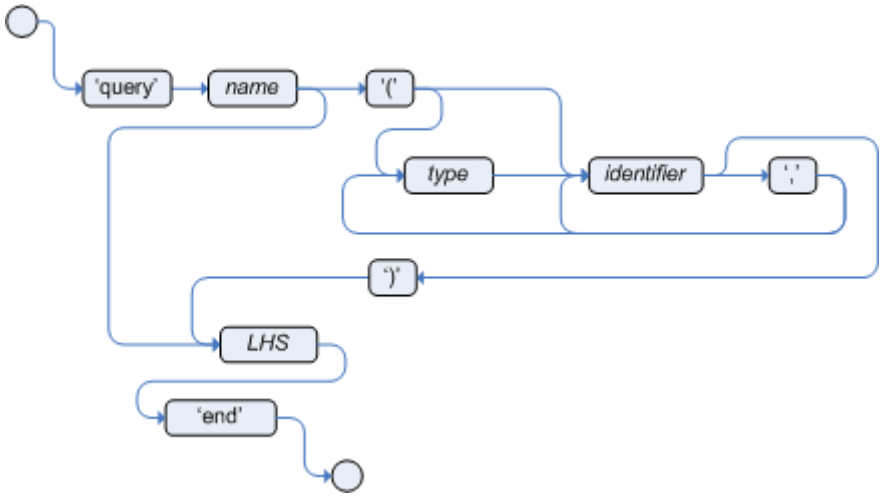


Figure 18. Query

Queries in DRL files search the working memory of the Drools rule 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" );
}
```

Support for positional syntax has been added for more compact code. By default the declared type order in the type declaration matches the argument position. But it possible to override these using the @position annotation. This allows patterns to be used with positional arguments, instead of the more verbose named arguments.

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
end
```

The @Position annotation, in the org.drools.definition.type package, can be used to annotate original pojos on the classpath. Currently only fields on classes can be annotated. Inheritance of classes is supported, but not interfaces or methods. The isContainedIn query below demonstrates the use of positional arguments in a pattern; `Location(x, y;)` instead of `Location( thing == x, location == y)`.

Queries can now call other queries, this combined with optional query arguments provides derivation query style backward chaining. Positional and named syntax is supported for arguments. It is also possible to mix both positional and named, but positional must come first, separated by a semi colon. Literal expressions can be passed as query arguments, but at this stage you cannot mix expressions with variables. Here is an example of a query that calls another query. Note that 'z' here will always be an 'out' variable. The '?' symbol means the query is pull only, once the results are returned you will not receive further results as the underlying data changes.

```
declare Location
    thing : String
    location : String
end

query isContainedIn( String x, String y )
    Location(x, y;)
    or
    ( Location(z, y;) and ?isContainedIn(x, z;) )
end
```

As previously mentioned you can use live "open" queries to reactively receive changes over time from the query results, as the underlying data it queries against changes. Notice the "look" rule calls the query without using '?'.

```
query isContainedIn( String x, String y )
    Location(x, y;)
    or
    ( Location(z, y;) and isContainedIn(x, z;) )
end

rule look when
    Person( $l : likes )
    isContainedIn( $l, 'office'; )
then
   insertLogical( $l 'is in the office' );
end
```

Drools supports unification for derivation queries, in short this means that arguments are optional. It is possible to call queries from Java leaving arguments unspecified using the static field org.drools.core.runtime.rule.Variable.v - note you must use 'v' and not an alternative instance of Variable. These are referred to as 'out' arguments. Note that the query itself does not declare at compile time whether an argument is in or an out, this can be defined purely at runtime on each use. The following example will return all objects contained in the office.

```
results = ksession.getQueryResults( "isContainedIn", new Object[] {  Variable.v,
"office" } );
l = new ArrayList<List<String>>();
for ( QueryResultsRow r : results ) {
    l.add( Arrays.asList( new String[] { (String) r.get( "x" ), (String) r.get( "y" )
} ) );
}
```

The algorithm uses stacks to handle recursion, so the method stack will not blow up.

It is also possible to use as input argument for a query both the field of a fact as in:

```
query contains(String $s, String $c)
    $s := String( this.contains( $c ) )
end

rule PersonNamesWithA when
    $p : Person()
    contains( $p.name, "a"; )
then
end
```

and more in general any kind of valid expression like in:

```
query checkLength(String $s, int $l)
    $s := String( length == $l )
end

rule CheckPersonNameLength when
    $i : Integer()
    $p : Person()
    checkLength( $p.name, 1 + $i + $p.age; )
then
end
```

The following is not yet supported:

- List and Map unification
- Expression unification - pred(X, X + 1, X * Y / 7)

2.1.5. Type declarations and metadata in DRL

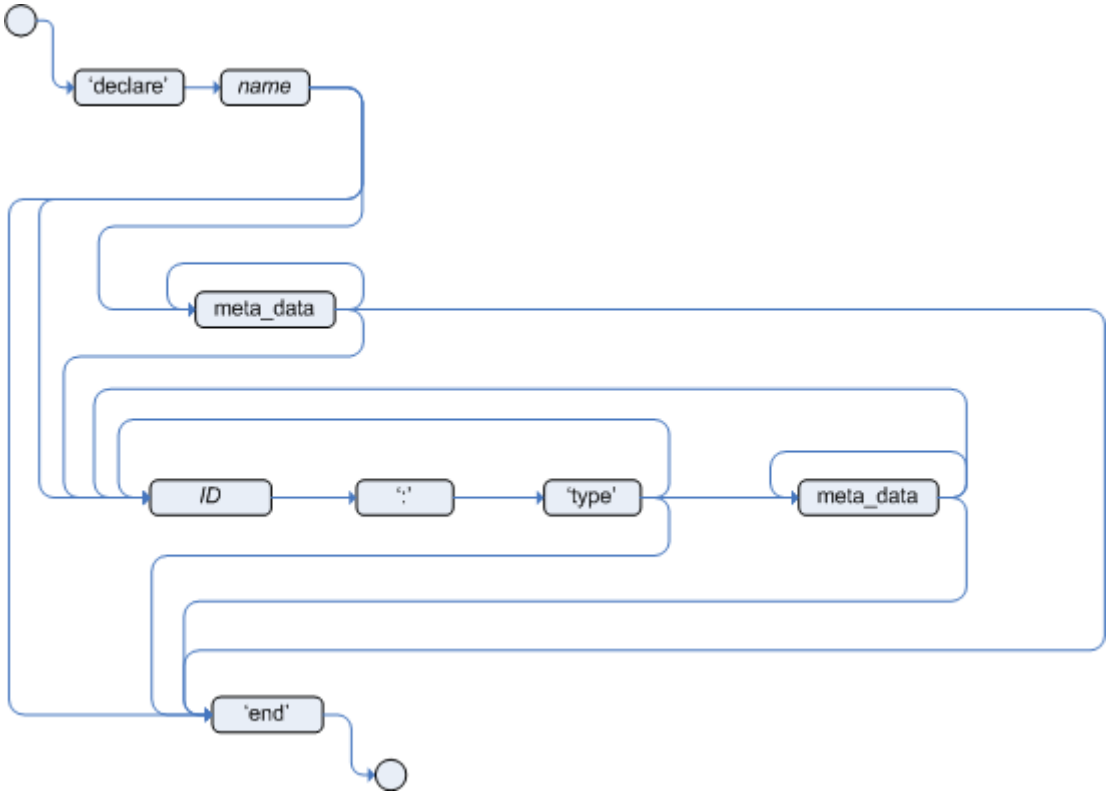


Figure 19. Type declaration

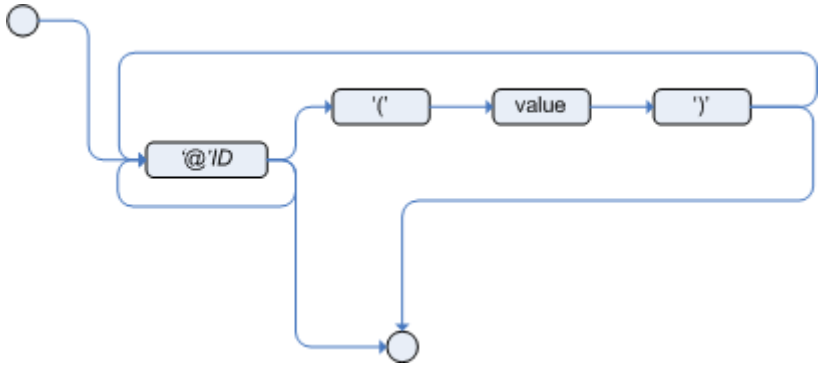


Figure 20. Metadata

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 Drools 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 Drools rule 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 Drools rule engine and used in the reasoning process.

2.1.5.1. 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 Drools rule 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 Drools rule 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

```
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
```

2.1.5.2. 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("Frida
y"),SAT("Saturday");

   fullName : String
end

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

2.1.5.3. 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
```

2.1.5.4. 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 Drools rule 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
```

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

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

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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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)

2.1.5.6. Property-change settings and listeners for fact types

By default, the Drools rule 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 Drools rule 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 Drools distribution:

Example property reactivity setting in system properties

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

The Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 Drools rule engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
    @propertyChangeSupport
end
```

2.1.5.7. 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 Drools rule 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 Drools. 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.

2.1.6. Global variables in DRL

Figure 21. Global

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 Drools rule 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 Drools rule engine and the corresponding global variable definition in the DRL file:

Example global list configuration for the Drools rule 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
```

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 Drools rule engine, so the Drools rule 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 Drools rule engine.

A use case for a global variable might be an instance of an email service. In your integration code that is calling the Drools rule engine, you obtain your `emailService` object and then set it in the working memory of the Drools rule 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.

2.1.7. Rule attributes in DRL

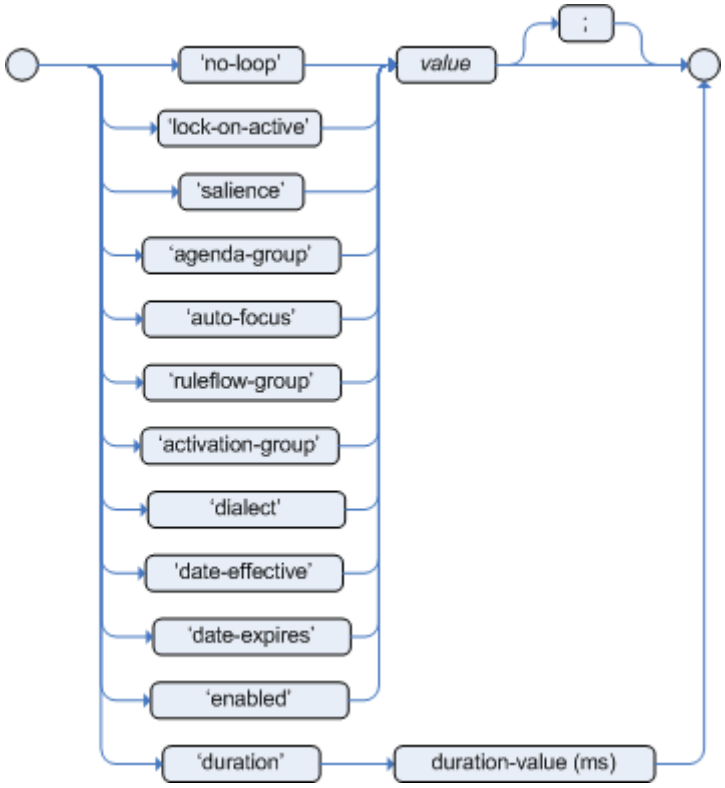


Figure 22. Rule attributes

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 1. Rule attributes

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`

Attribute	Value
`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"` When you use Drools without the executable model, the `dialect "JAVA"` rule consequences support only Java 5 syntax. For more information about executable models, see [executable-model-con_packaging-deploying].

2.1.7.1. 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 Drools rule engine handles rules with timers depends on whether the Drools rule engine is in *active mode* or in *passive mode*.

By default, the Drools rule 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 Drools rule engine starts in *active mode* and evaluates rules continually until the user or application explicitly calls `halt()`.

When the Drools rule engine is in active mode, rule consequences are executed even after control

returns from a call to `fireUntilHalt()` and the Drools rule 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 Drools rule engine is not continually *active*, but is active only after a rule is executed. Therefore, the Drools rule 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 Drools rule 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
```

2.1.8. Rule conditions in DRL (WHEN)

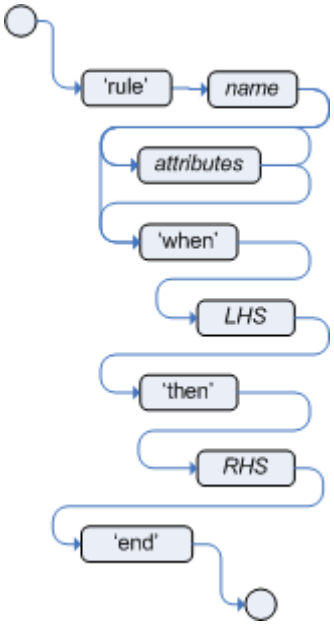


Figure 23. Rule

Figure 24. Conditional element in a rule

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 )`.

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 Drools rule engine. This is useful if you want to use rules to set up the Drools rule 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
```

2.1.8.1. Patterns and constraints

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

The railroad diagram below shows the syntax for this:

Figure 25. Pattern

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 Drools rule 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 Drools rule 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 )
```

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

Do not use property accessors to change the state of the object in a way that might affect the rules because the Drools rule 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 Drools rule engine.

Example pattern with nested property access

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

// This is the same as the following format:

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

In stateful KIE sessions, use nested accessors carefully because the working memory of the Drools rule 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 )
```

Do not use constraints to change the state of the object in a way that might affect the rules because the Drools rule 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 Drools rule 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 )
```

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 Drools rule 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 )
```

2.1.8.2. 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
```

Constraint binding considers only the first atomic expression that follows it. In the following example the pattern only binds the age of the person to the variable `$a`:

```
Person( $a : age * 2 < 100 )
```

For clearer and more efficient rule definitions, separate constraint bindings and constraint expressions. Although mixed bindings and expressions are supported, which can complicate patterns and affect evaluation efficiency.

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

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

In the preceding example, if you want to bind to the variable `$a` the double of the person's age, you must make it an atomic expression by wrapping it in parentheses as shown in the following example:

```
Person( $a : (age * 2) )
```

The Drools rule 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.

2.1.8.3. 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") )
```

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`.

2.1.8.4. Date literal in constraints

By default, the Drools rule 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" )
```

2.1.8.5. Auto-boxing and primitive types

Drools attempts to preserve numbers in their primitive or object wrapper form, so a variable bound to an int primitive when used in a code block or expression will no longer need manual unboxing; unlike early Drools versions where all primitives were autoboxed, requiring manual unboxing. A variable bound to an object wrapper will remain as an object; the existing JDK 1.5 and JDK 5 rules to handle auto-boxing and unboxing apply in this case. When evaluating field constraints, the system attempts to coerce one of the values into a comparable format; so a primitive is comparable to an object wrapper.

2.1.8.6. 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") )
```

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")
```

Figure 26. Abbreviated combined relation condition

Figure 27. Abbreviated combined relation condition withparentheses

`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 )
```

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 `!=`.

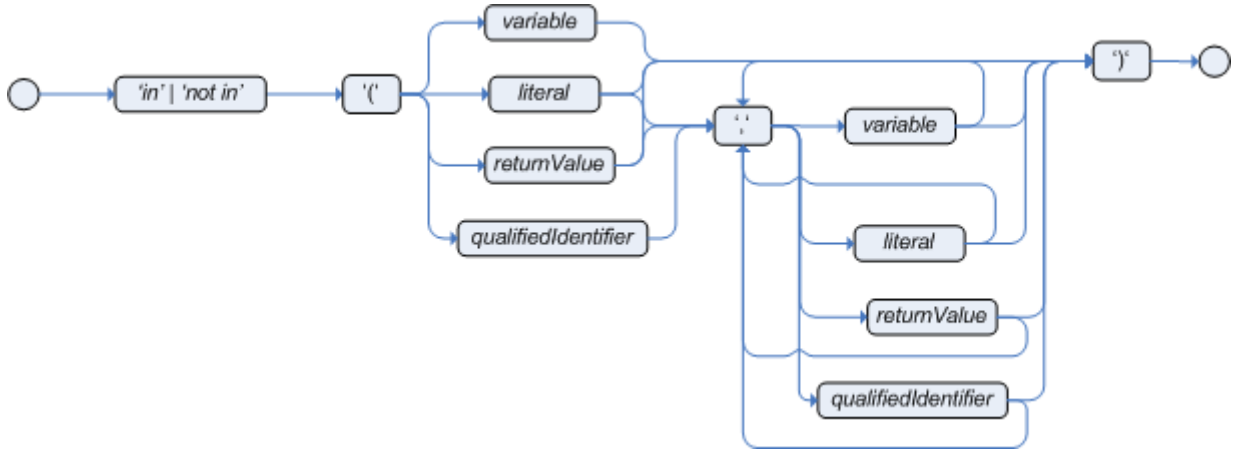


Figure 28. compoundValueRestriction

Example constraints with `in` and `notin`

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

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

2.1.8.7. 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. 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
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`	`^`	

Operator type	Operators	Notes
Non-short-circuiting inclusive OR	|	
Logical AND	&&	
Logical OR	||	
Ternary	? :	
Comma-separated AND	,	Not standard Java semantics

2.1.8.8. 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.

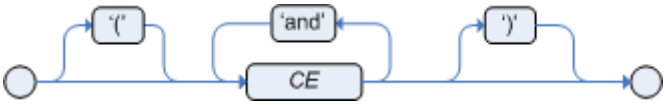


Figure 29. infixAnd

Figure 30. prefixAnd

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 )
```

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.

Figure 31. infixOr

Figure 32. prefixOr

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 Drools rule 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 `()`.

Figure 33. Exists

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 `()`.

Figure 34. Not

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.

Figure 35. Forall

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 Drools rule 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
```

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 Drools rule 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.

Figure 36. from

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
```

For large collections of objects, instead of adding an object with a large graph that the Drools rule 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
```

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 Drools rule 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 Drools rule 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`

```
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 Drools rule 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.

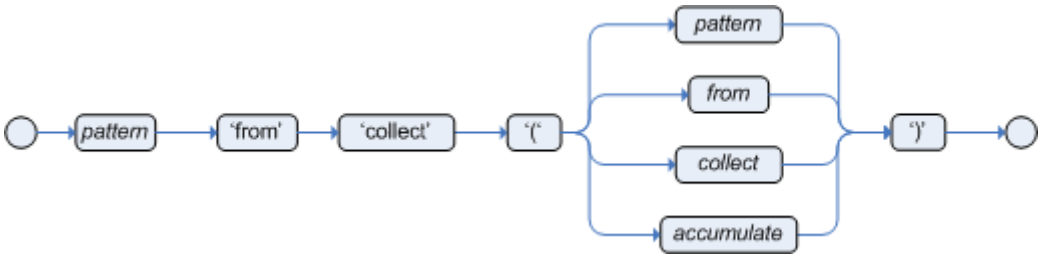


Figure 37. Collect

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 Drools rule 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>] )
```

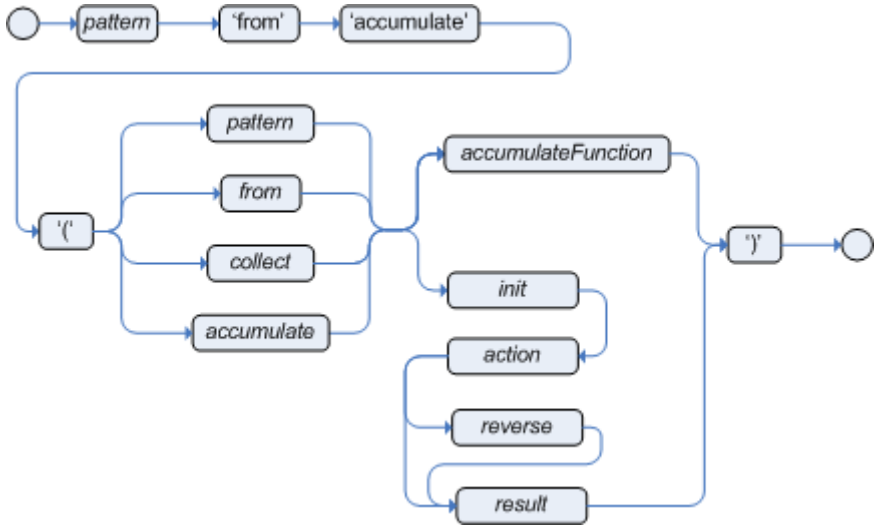


Figure 38. Accumulate

Although the Drools rule engine supports alternate formats for the `accumulate` element for backward compatibility, this format is preferred for optimal performance in rules and applications.

The Drools rule 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();
        }

        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
```

For backward compatibility, the Drools rule engine also supports the configuration of `accumulate` functions through configuration files and system properties, but this is a deprecated method. To configure the `average` function from the previous example using the configuration file or system property, set a property as shown in the following example:

```
drools.accumulate.function.average =
AverageAccumulateFunction.AverageData
```

Note that `drools.accumulate.function` is a required prefix, `average` is how the function is used in the DRL files, and `AverageAccumulateFunction.AverageData` is the fully qualified name of the class that implements the function behavior.

`accumulate` alternate syntax for a single function with return type

The accumulate syntax evolved over time with the goal of becoming more compact and expressive. Nevertheless, Drools still supports previous syntaxes for backward compatibility purposes.

In case the rule is using a single accumulate function on a given accumulate, the author may add a pattern for the result object and use the "from" keyword to link it to the accumulate result.

Example: a rule to apply a 10% discount on orders over $100 could be written in the following way:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              sum( $value ) )
then
    // apply discount to $order
end
```

In the previous example, the accumulate element is using only one function (sum), and so, the rules author opted to explicitly write a pattern for the result type of the accumulate function (Number) and write the constraints inside it. There are no problems in using this syntax over the compact syntax presented before, except that is is a bit more verbose. Also note that it is not allowed to use both the return type and the functions binding in the same accumulate statement.

Compile-time checks are performed in order to ensure the pattern used with the "`from`" keyword is assignable from the result of the accumulate function used.

With this syntax, the "`from`" binds to the single result returned by the accumulate function, and it does not iterate.

In the previous example, "`$total`" is bound to the result returned by the accumulate sum() function.

As another example however, if the result of the accumulate function is a collection, "`from`" still binds to the single result and it does not iterate:

```
rule "Person names"
when
  $x : Object() from accumulate(MyPerson( $val : name );
                                collectList( $val ) )
then
  // $x is a List
end
```

The bound "`$x : Object()`" is the List itself, returned by the collectList accumulate function used.

This is an important distinction to highlight, as the "`from`" keyword can also be used separately of accumulate, to iterate over the elements of a collection:

```
rule "Iterate the numbers"
when
    $xs : List()
    $x : Integer() from $xs
then
  // $x matches and binds to each Integer in the collection
end
```

While this syntax is still supported for backward compatibility purposes, for this and other reasons we encourage rule authors to make use instead of the preferred `accumulate` syntax (described previously), to avoid any potential pitfalls.

`accumulate` with inline custom code

Another possible syntax for the `accumulate` is to define inline custom code, instead of using accumulate functions.

The use of accumulate with inline custom code is not a good practice for several reasons, including difficulties on maintaining and testing rules that use them, as well as the inability of reusing that code. Implementing your own accumulate functions is very simple and straightforward, they are easy to unit test and to use. This form of accumulate is supported for backward compatibility only.

Only limited support for inline accumulate is provided while using the executable model. For example, you cannot use an external binding in the code while using the MVEL dialect:

```
rule R
dialect "mvel"
when
    String( $l : length )
    $sum : Integer() from accumulate (
                            Person( age > 18, $age : age ),
                            init( int sum = 0 * $l; ),
                            action( sum += $age; ),
                            reverse( sum -= $age; ),
                            result( sum )
                        )
```

The general syntax of the `accumulate` CE with inline custom code is:

```
<result pattern> from accumulate( <source pattern>,
                                  init( <init code> ),
                                  action( <action code> ),
                                  reverse( <reverse code> ),
                                  result( <result expression> ) )
```

The meaning of each of the elements is the following:

- <*source pattern*>: the source pattern is a regular pattern that the Drools rule engine will try to match against each of the source objects.
- <*init code*>: this is a semantic block of code in the selected dialect that will be executed once for each tuple, before iterating over the source objects.
- <*action code*>: this is a semantic block of code in the selected dialect that will be executed for each of the source objects.
- <*reverse code*>: this is an optional semantic block of code in the selected dialect that if present will be executed for each source object that no longer matches the source pattern. The objective of this code block is to undo any calculation done in the <*action code*> block, so that the Drools rule engine can do decremental calculation when a source object is modified or deleted, hugely improving performance of these operations.
- <*result expression*>: this is a semantic expression in the selected dialect that is executed after all source objects are iterated.
- <*result pattern*>: this is a regular pattern that the Drools rule engine tries to match against the object returned from the <*result expression*>. If it matches, the `accumulate` conditional element evaluates to *true* and the Drools rule engine proceeds with the evaluation of the next CE in the rule. If it does not matches, the `accumulate` CE evaluates to *false* and the Drools rule engine stops evaluating CEs for that rule.

It is easier to understand if we look at an example:

```
rule "Apply 10% discount to orders over US$ 100,00"
when
    $order : Order()
    $total : Number( doubleValue > 100 )
             from accumulate( OrderItem( order == $order, $value : value ),
                              init( double total = 0; ),
                              action( total += $value; ),
                              reverse( total -= $value; ),
                              result( total ) )
then
    // apply discount to $order
end
```

In the previous example, for each `Order` in the Working Memory, the Drools rule engine will execute the *init code* initializing the total variable to zero. Then it will iterate over all `OrderItem` objects for that order, executing the *action* for each one (in the example, it will sum the value of

all items into the total variable). After iterating over all `OrderItem` objects, it will return the value corresponding to the *result expression* (in the previous example, the value of variable `total`). Finally, the Drools rule engine will try to match the result with the `Number` pattern, and if the double value is greater than 100, the rule will fire.

The example used Java as the semantic dialect, and as such, note that the usage of the semicolon as statement delimiter is mandatory in the init, action and reverse code blocks. The result is an expression and, as such, it does not admit ';'. If the user uses any other dialect, he must comply to that dialect's specific syntax.

As mentioned before, the *reverse code* is optional, but it is strongly recommended that the user writes it in order to benefit from the *improved performance on update and delete*.

The `accumulate` CE can be used to execute any action on source objects. The following example instantiates and populates a custom object:

```
rule "Accumulate using custom objects"
when
    $person   : Person( $likes : likes )
    $cheesery : Cheesery( totalAmount > 100 )
                from accumulate( $cheese : Cheese( type == $likes ),
                                 init( Cheesery cheesery = new Cheesery(); ),
                                 action( cheesery.addCheese( $cheese ); ),
                                 reverse( cheesery.removeCheese( $cheese ); ),
                                 result( cheesery ) );
then
    // do something
end
```

`eval`

The conditional element `eval` is essentially a catch-all which allows any semantic code (that returns a primitive boolean) to be executed. This code can refer to variables that were bound in the conditions of the rule and functions in the rule package. Overuse of `eval` reduces the declarativeness of your rules and can result in a poorly performing Drools rule engine. While `eval` can be used anywhere in the patterns, it is typically added as the last conditional element in the conditions of a rule.

Figure 39. Eval

Instances of `eval` cannot be indexed and thus are not as efficient as Field Constraints. However this makes them ideal for being used when functions return values that change over time, which is not allowed within Field Constraints.

For those who are familiar with Drools 2.x lineage, the old Drools parameter and condition tags are equivalent to binding a variable to an appropriate type, and then using it in an `eval` node.

```
p1 : Parameter()
p2 : Parameter()
eval( p1.getList().containsKey( p2.getItem() ) )
```

```
p1 : Parameter()
p2 : Parameter()
// call function isValid in the LHS
eval( isValid( p1, p2 ) )
```

2.1.8.9. 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 Drools rule 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 Drools rule 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 Drools rule engine when one of the fields has been updated, as shown in the following example:

Example object method to notify the Drools rule 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 Drools rule 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 Drools. 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
    }
  }
```

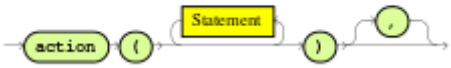
① Uses the `ReactiveList` instance for reactive support over the standard Java `List` instance.

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

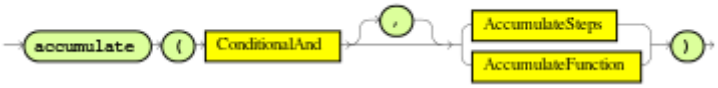
③ The `children` field is a `ReactiveList` instance, so the `notifyModification()` method call is not required. The notification is handled automatically, like all other mutating operations performed over the `children` field.

2.1.8.10. Railroad diagrams for rule condition elements in DRL

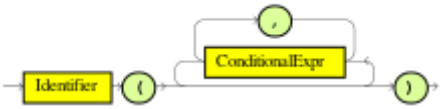
AccumulateAction



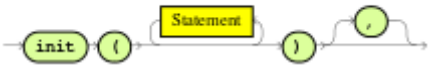
AccumulateClause



AccumulateFunction

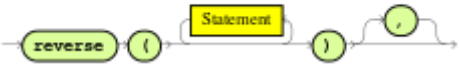


AccumulateInit



AccumulateResult

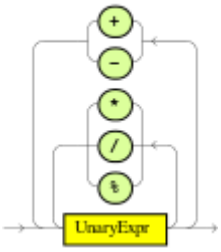
AccumulateReverse



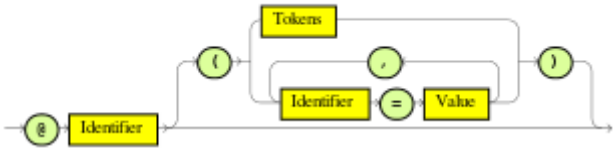
AccumulateSteps

Accumulations

AdditiveExpr

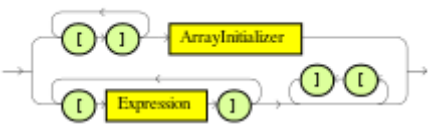


Annotation

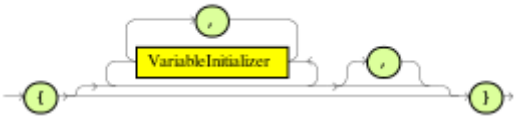


Arguments

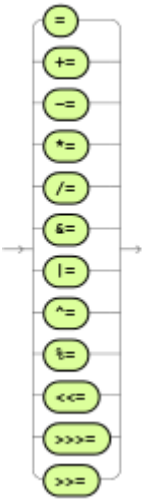
ArrayCreatorRest



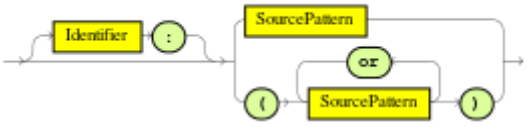
ArrayInitializer



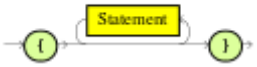
AssignmentOperator



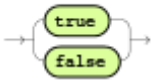
BindingPattern



Block

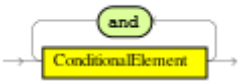


BooleanLiteral



CompilationUnit

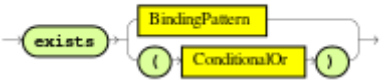
ConditionalAnd



ConditionalElementAccumulate

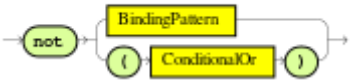
ConditionalElementEval

ConditionalElementExists

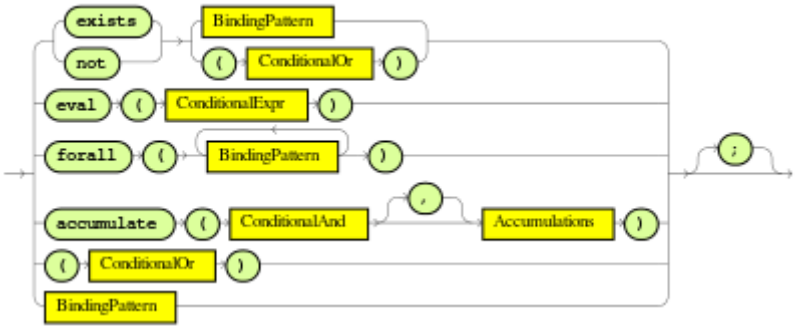


ConditionalElementForall

ConditionalElementNot

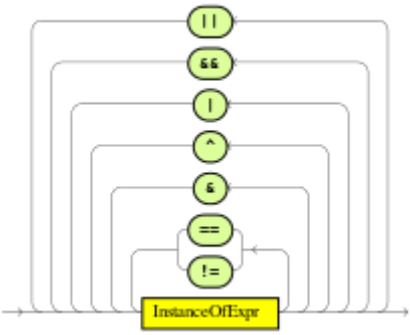


ConditionalElement

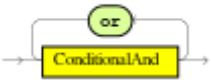


ConditionalExpr

ConditionalOrExpr

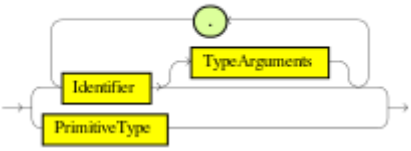


ConditionalOr

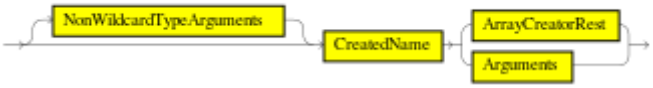


Constraints

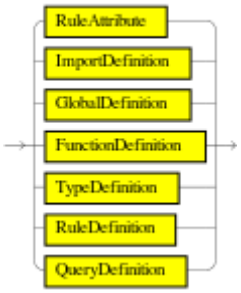
CreatedName



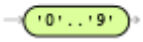
Creator



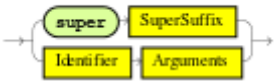
Definition



Digit

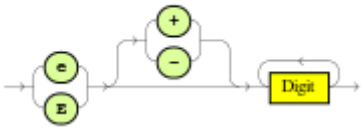


ExplicitGenericInvocationSuffix

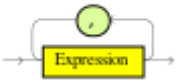


ExplicitGenericInvocation

Exponent



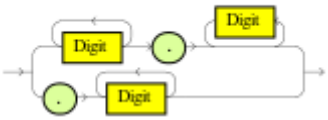
ExpressionList



Expression

Field

Fraction



FromAccumulateClause

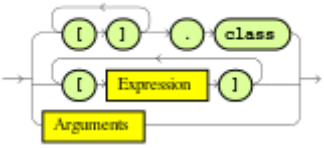
FromClause

FromCollectClause

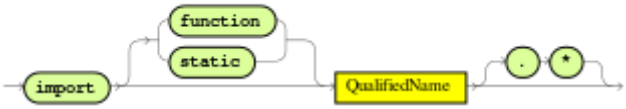
FunctionDefinition

GlobalDefinition

IdentifierSuffix



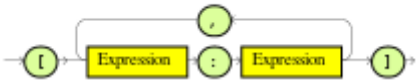
ImportDefinition



InExpr

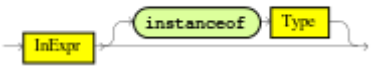
InlineListExpr

InlineMapExpr

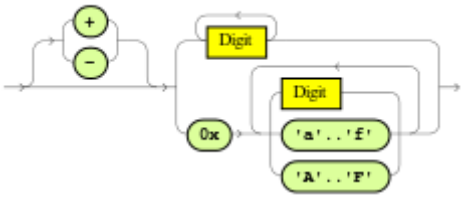


InnerCreator

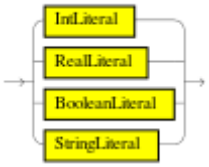
InstanceOfExpr



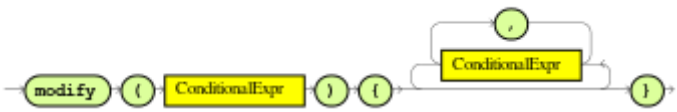
IntLiteral



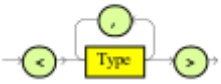
Literal



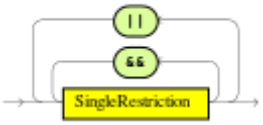
ModifyStatement



NonWildcardTypeArguments

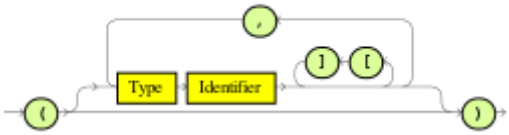


OrRestriction

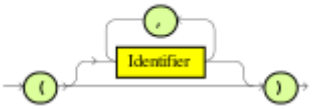


OverClause

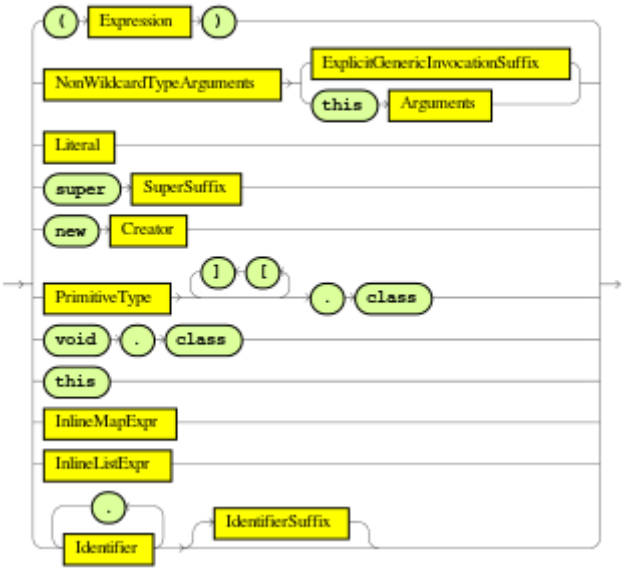
Parameters



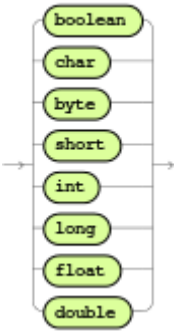
Placeholders



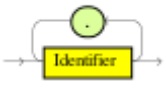
Primary



PrimitiveType



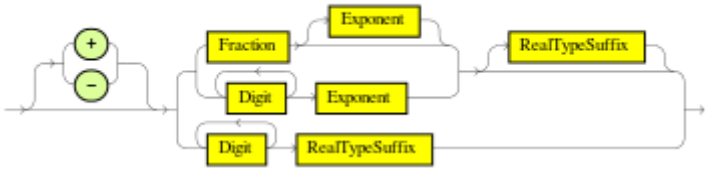
QualifiedName



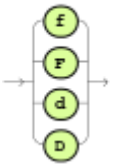
QueryDefinition

QueryOptions

RealLiteral

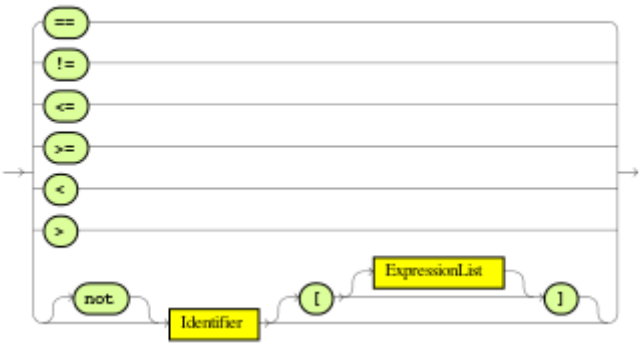


RealTypeSuffix



RelationalExpr

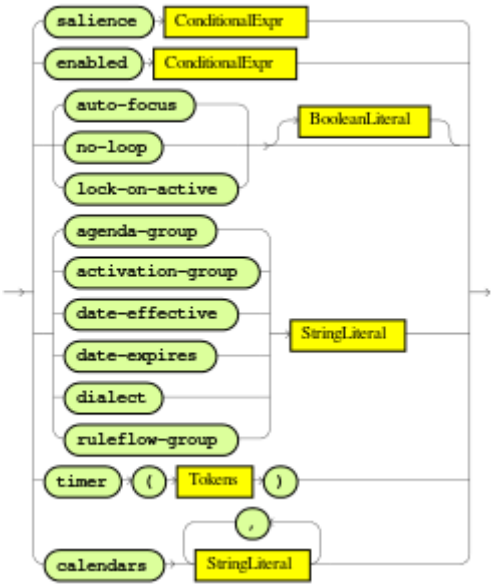
RelationalOperator



RhsStatement

RuleAttributes

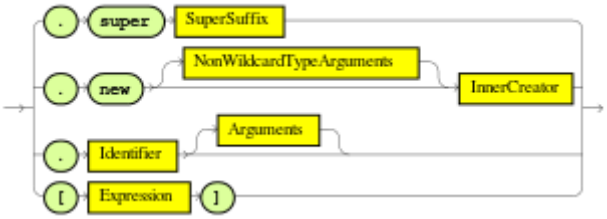
RuleAttribute



RuleDefinition

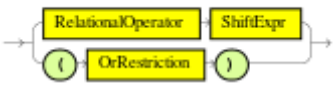
RuleOptions

Selector

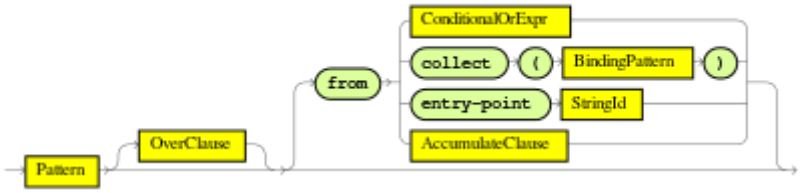


ShiftExpr

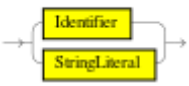
SingleRestriction



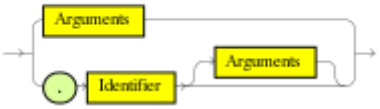
SourcePattern



StringId

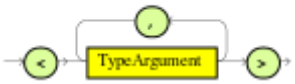


SuperSuffix

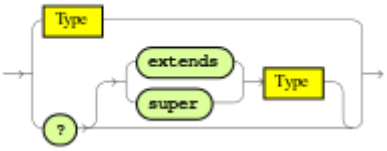


ThenPart

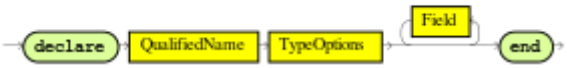
TypeArguments



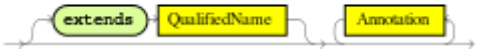
TypeArgument



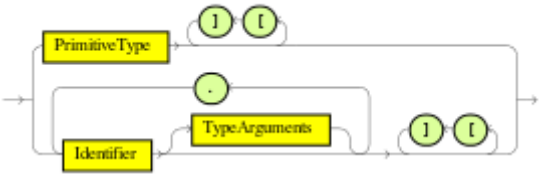
TypeDefinition



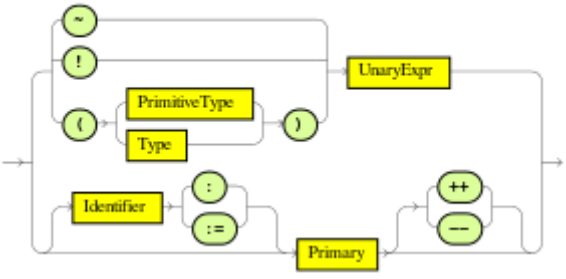
TypeOptions



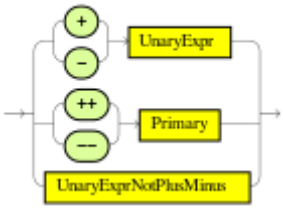
Type



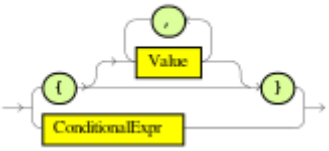
UnaryExprNotPlusMinus

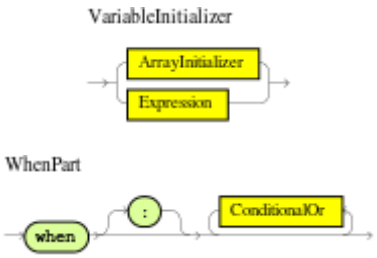


UnaryExpr



Value





2.1.9. 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 be 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 insert, delete, or modify data in the working memory of the Drools rule 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
```

2.1.9.1. 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 Drools rule engine without having to first reference a working memory instance. These methods act as shortcuts to the methods provided by the `RuleContext` class in your Drools distribution.

For all rule action methods, see the Drools RuleContext.java page in GitHub.

`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 Drools rule 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 Drools rule 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 Drools rule 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 );
```

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 [property-change-listeners-con_decision-engine].

`insert`

Use this to insert a `new` fact into the working memory of the Drools rule 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 Drools rule engine. The Drools rule 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 Drools rule 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 );
```

2.1.9.2. Other rule action methods from `drools` variable

In addition to the standard rule action methods, the Drools rule engine supports methods in conjunction with the predefined `drools` variable that you can also use in rule actions.

You can use the `drools` variable to call methods from the `org.kie.api.runtime.rule.RuleContext` class in your Drools distribution, which is also the class that the standard rule action methods are based on. For all `drools` rule action options, see the Drools RuleContext.java page in GitHub.

The `drools` variable contains methods that provide information about the firing rule and the set of

facts that activated the firing rule:

- `drools.getRule().getName()`: Returns the name of the currently firing rule.
- `drools.getMatch()`: Returns the `Match` that activated the currently firing rule. It contains information that is useful for logging and debugging purposes, for instance `drools.getMatch().getObjects()` returns the list of objects, enabling rule to fire in the proper tuple order.

From the `drools` variable, you can also obtain a reference to the `KieRuntime` providing useful methods to interact with the running session, for example:

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

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

+ This example sets the focus to a specified agenda group to which the rule belongs.

- `drools.getKieRuntime().setGlobal()`, `~.getGlobal()`, `~.getGlobals()`: Sets or retrieves global variables.
- `drools.getKieRuntime().getEnvironment()`: Returns the runtime `Environment`, similar to your operating system environment.
- `drools.getKieRuntime().getQueryResults(<string> query)`: Runs a query and returns the results.

2.1.9.3. 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 Drools rule 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.

2.1.10. 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 Drools rule 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
```

Figure 40. Multi-line comment

The hash symbol # is not supported for DRL comments.

2.1.11. Error messages for DRL troubleshooting

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

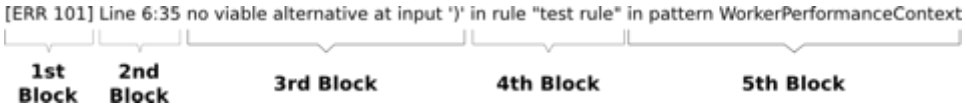


Figure 41. Error message format for DRL file problems

- **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)

Drools 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
```

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
```

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.

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.

2.1.12. Rule units in DRL rule sets

Rule units are experimental in Drools 7. Only supported in {KOGITO}.

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 Drools rule 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 Drools rule engine.

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

Table 3. Rule unit life-cycle methods

Method	Invoked when
`onStart()`	Rule unit execution starts
`onEnd()`	Rule unit execution ends

Method	Invoked when
`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
```

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
```

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 Drools rule 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 Drools rule 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();
```

2.1.12.1. 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 Drools rule 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 );
```

2.1.12.2. 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 Drools rule 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 Drools rule 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() );
```

2.1.12.3. 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.

2.1.13. Performance tuning considerations with DRL

The following key concepts or suggested practices can help you optimize DRL rules and Drools rule 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 Drools.

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 Drools rule engine performance.

For more information about DRL patterns and constraints, see Rule conditions in DRL (WHEN).

Use equality operators more than other operator types in pattern constraints when possible

Although the Drools rule 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 Drools rule 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 Drools rule 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 Drools rule 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 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 Drools rule 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 Drools rule 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 Drools rule 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 Supported rule condition elements in DRL (keywords).

Use Drools rule engine event listeners instead of `System.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 Drools rule engine event listeners when possible. If these listeners do not meet your requirements, use a system logging utility supported by the Drools rule engine, such as Logback, Apache Commons Logging, or Apache Log4j.

For more information about supported Drools rule engine event listeners and logging utilities, see [engine-event-listeners-con_decision-engine].

Use the `drools-metric` module to identify the obstruction in your rules

You can use the `drools-metric` module to identify slow rules especially when you process many rules. The `drools-metric` module can also assist in analyzing the Drools rule engine performance. Note that the `drools-metric` module is not for production environment use. However, you can perform the analysis in your test environment.

To analyze the Drools rule engine performance using `drools-metric`, first add `drools-metric` to your project dependencies:

Example project dependency for `drools-metric`

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-metric</artifactId>
</dependency>
```

If you want to use `drools-metric` to enable trace logging, configure a logger for `org.drools.metric.util.MetricLogUtils` as shown in the following example:

Example logback.xml configuration file

```
<configuration>
  <logger name="org.drools.metric.util.MetricLogUtils" level="trace"/>
  ...
<configuration>
```

Alternatively, you can use `drools-metric` to expose the data using Micrometer. To expose the data, enable the Micrometer registry of your choice as shown in the following example:

Example project dependency for Micrometer

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-jmx</artifactId> <!-- Discover more registries at
micrometer.io. -->
</dependency>
```

Example Java code for Micrometer

```
  Metrics.addRegitry(new JmxMeterRegistry(s -> null, Clock.SYSTEM));
```

Regardless of whether you want to use logging or Micrometer, you need to enable `MetricLogUtils` by setting the system property `drools.metric.logger.enabled` to `true`. Optionally, you can change the microseconds threshold of metric reporting by setting the `drools.metric.logger.threshold` system property.

Only node executions exceeding the threshold are reported. The default value is `500`.

After configuring the `drools-metric` to use logging, rule execution produces logs as shown in the following example:

Example rule execution output

```
TRACE [JoinNode(6) - [ClassObjectType class=com.sample.Order]], evalCount:1000,
elapsedMicro:5962
TRACE [JoinNode(7) - [ClassObjectType class=com.sample.Order]], evalCount:100000,
elapsedMicro:95553
TRACE [ AccumulateNode(8) ], evalCount:4999500, elapsedMicro:2172836
TRACE [EvalConditionNode(9)]:
cond=com.sample.Rule_Collect_expensive_orders_combination930932360Eval1Invoker@ee2a
6922], evalCount:49500, elapsedMicro:18787
```

This example includes the following key parameters:

- `evalCount` is the number of constraint evaluations against inserted facts during the node execution. When `evalCount` is used with Micrometer, a counter with the data is called `org.drools.metric.evaluation.count`.
- `elapsedMicro` is the elapsed time of the node execution in microseconds. When `elapsedMicro` is used with Micrometer, look for a timer called `org.drools.metric.elapsed.time`.

If you find an outstanding `evalCount` or `elapsedMicro` log, correlate the node name with `ReteDumper.dumpAssociatedRulesRete()` output to identify the rule associated with the node.

Example ReteDumper usage

```
ReteDumper.dumpAssociatedRulesRete(kbase);
```

Example ReteDumper output

```
[ AccumulateNode(8) ] : [Collect expensive orders combination]
...
```

2.2. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. Given a DSL, you write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files, and you can use any text editor to create and modify them. But there are also DSL and DSLR editors, both in the IDE as well as in the web based BRMS, and you can use those as well, although they may not provide you with the full DSL functionality.

2.2.1. When to Use a DSL

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modelling of domain object and the Drools rule engine's native language and methods. If your rules need to be read and validated by domain experts (such as business analysts, for instance) who are not programmers, you should consider using a DSL; it hides implementation details and focuses on the rule logic proper. DSL sentences can also act as "templates" for conditional elements and consequence actions that are used repeatedly in your rules, possibly with minor variations. You may define DSL sentences as being mapped to these repeated phrases, with parameters providing a means for accommodating those variations.

DSLs have no impact on the Drools rule engine at runtime, they are just a compile time feature, requiring a special parser and transformer.

2.2.2. DSL Basics

The Drools DSL mechanism allows you to customise conditional expressions and consequence actions. A global substitution mechanism ("keyword") is also available.

Example 1. Example DSL mapping

```
[when]Something is {colour}=Something(colour=="{colour}")
```

In the preceding example, `[when]` indicates the scope of the expression, i.e., whether it is valid for the LHS or the RHS of a rule. The part after the bracketed keyword is the expression that you use in the rule; typically a natural language expression, but it doesn't have to be. The part to the right of the equal sign ("=") is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it ought to be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation. First, it extracts the string values appearing where the expression contains variable names in braces (here: `{colour}`). Then, the values obtained from these captures are then interpolated wherever that name, again enclosed in braces, occurs on the right hand side of the mapping. Finally, the interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

Note that the expressions (i.e., the strings on the left hand side of the equal sign) are used as regular expressions in a pattern matching operation against a line of the DSL rule file, matching all or part of a line. This means you can use (for instance) a '?' to indicate that the preceding character is optional. One good reason to use this is to overcome variations in natural language phrases of your DSL. But, given that these expressions are regular expression patterns, this also means that all "magic" characters of Java's pattern syntax have to be escaped with a preceding backslash ('\').

It is important to note that the compiler transforms DSL rule files line by line. In the previous example, all the text after "Something is " to the end of the line is captured as the replacement value for "{colour}", and this is used for interpolating the target string. This may not be exactly what you

want. For instance, when you intend to merge different DSL expressions to generate a composite DRL pattern, you need to transform a DSLR line in several independent operations. The best way to achieve this is to ensure that the captures are surrounded by characteristic text - words or even single characters. As a result, the matching operation done by the parser plucks out a substring from somewhere within the line. In the example below, quotes are used as distinctive characters. Note that the characters that surround the capture are not included during interpolation, just the contents between them.

As a rule of thumb, use quotes for textual data that a rule editor may want to enter. You can also enclose the capture with words to ensure that the text is correctly matched. Both is illustrated by the following example. Note that a single line such as `Something is "green" and another solid thing` is now correctly expanded.

Example 2. Example with quotes

```
[when]something is "{colour}"=Something(colour=="{colour}")
[when]another {state} thing=OtherThing(state=="{state}")"
```

It is a good idea to avoid punctuation (other than quotes or apostrophes) in your DSL expressions as much as possible. The main reason is that punctuation is easy to forget for rule authors using your DSL. Another reason is that parentheses, the period and the question mark are magic characters, requiring escaping in the DSL definition.

In a DSL mapping, the braces "{" and "}" should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\"):

```
[then]do something= if (foo) \{ doSomething(); \}
```

If braces "{" and "}" should appear in the replacement string of a DSL definition, escape them with a backslash ('\').

Example 3. Examples of DSL mapping entries

```
# This is a comment to be ignored.
[when]There is a person with name of "{name}"=Person(name=="{name}")
[when]Person is at least {age} years old and lives in "{location}"=
      Person(age >= {age}, location=="{location}")
[then]Log "{message}"=System.out.println("{message}");
[when]And = and
```

Given the above DSL examples, the following examples show the expansion of various DSLR snippets:

Example 4. Examples of DSL expansions

```
There is a person with name of "Kitty"
   ==> Person(name="Kitty")
Person is at least 42 years old and lives in "Atlanta"
   ==> Person(age >= 42, location="Atlanta")
Log "boo"
   ==> System.out.println("boo");
There is a person with name of "Bob" And Person is at least 30 years old and lives
in "Utah"
   ==> Person(name="Bob") and Person(age >= 30, location="Utah")
```

Don't forget that if you are capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

You can chain DSL expressions together on one line, as long as it is clear to the parser where one ends and the next one begins and where the text representing a parameter ends. (Otherwise you risk getting all the text until the end of the line as a parameter value.) The DSL expressions are tried, one after the other, according to their order in the DSL definition file. After any match, all remaining DSL expressions are investigated, too.

The resulting DRL text may consist of more than one line. Line ends are in the replacement text are written as `\n`.

2.2.3. Adding Constraints to Facts

A common requirement when writing rule conditions is to be able to add an arbitrary combination of constraints to a pattern. Given that a fact type may have many fields, having to provide an individual DSL statement for each combination would be plain folly.

The DSL facility allows you to add constraints to a pattern by a simple convention: if your DSL expression starts with a hyphen (minus character, "-") it is assumed to be a field constraint and, consequently, is is added to the last pattern line preceding it.

For an example, lets take look at class `Cheese`, with the following fields: type, price, age and country. We can express some LHS condition in normal DRL like the following

```
Cheese(age < 5, price == 20, type=="stilton", country=="ch")
```

The DSL definitions given below result in three DSL phrases which may be used to create any combination of constraint involving these fields.

```
[when]There is a Cheese with=Cheese()
[when]- age is less than {age}=age<{age}
[when]- type is '{type}'=type=='{type}'
[when]- country equal to '{country}'=country=='{country}'
```

You can then write rules with conditions like the following:

```
There is a Cheese with
        - age is less than 42
        - type is 'stilton'
```

```
 The parser will pick up a line beginning with "-" and add it as a constraint to  the
preceding pattern, inserting a comma when it is required.
For the preceding example, the resulting DRL is:
```

```
Cheese(age<42, type=='stilton')
```

Combining all numeric fields with all relational operators (according to the DSL expression "age is less than..." in the preceding example) produces an unwieldy amount of DSL entries. But you can define DSL phrases for the various operators and even a generic expression that handles any field constraint, as shown below. (Notice that the expression definition contains a regular expression in addition to the variable name.)

```
[when][]is less than or equal to=<=
[when][]is less than=<
[when][]is greater than or equal to=>=
[when][]is greater than=>
[when][]is equal to===
[when][]equals===
[when][]There is a Cheese with=Cheese()
[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}
```

Given these DSL definitions, you can write rules with conditions such as:

```
There is a Cheese with
   - age is less than 42
   - rating is greater than 50
   - type equals 'stilton'
```

In this specific case, a phrase such as "is less than" is replaced by <, and then the line matches the last DSL entry. This removes the hyphen, but the final result is still added as a constraint to the preceding pattern. After processing all of the lines, the resulting DRL text is:

```
Cheese(age<42, rating > 50, type=='stilton')
```

The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.

2.2.4. Developing a DSL

A good way to get started is to write representative samples of the rules your application requires, and to test them as you develop. This will provide you with a stable framework of conditional elements and their constraints. Rules, both in DRL and in DSLR, refer to entities according to the data model representing the application data that should be subject to the reasoning process defined in rules. Notice that writing rules is generally easier if most of the data model's types are facts.

Given an initial set of rules, it should be possible to identify recurring or similar code snippets and to mark variable parts as parameters. This provides reliable leads as to what might be a handy DSL entry. Also, make sure you have a full grasp of the jargon the domain experts are using, and base your DSL phrases on this vocabulary.

You may postpone implementation decisions concerning conditions and actions during this first design phase by leaving certain conditional elements and actions in their DRL form by prefixing a line with a greater sign (">"). (This is also handy for inserting debugging statements.)

During the next development phase, you should find that the DSL configuration stabilizes pretty quickly. New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.

Try to keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints. But do not exaggerate: authors using the DSL should still be able to identify DSL phrases by some fixed text.

2.2.5. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file according to DRL syntax.

- A line starting with " " or "//" (with or without preceding white space) is treated as a comment. A comment line starting with "/" is scanned for words requesting a debug option, see below.
- Any line starting with an opening bracket ("[") is assumed to be the first line of a DSL entry definition.
- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

A DSL entry consists of the following four parts:

- A scope definition, written as one of the keywords "when" or "condition", "then" or "consequence", "*" and "keyword", enclosed in brackets ("[" and "]"). This indicates whether the

DSL entry is valid for the condition or the consequence of a rule, or both. A scope indication of "keyword" means that the entry has global significance, i.e., it is recognized anywhere in a DSLR file.

- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the next part begins with an opening bracket. An empty pair of brackets is valid, too.
- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions*, terminated by an equal sign ("="). A variable definition is enclosed in braces ("{" and "}"). It consists of a variable name and two optional attachments, separated by colons (":"). If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable; if there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

  Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash ("\") if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in braces. Optionally, the variable name may be followed by an exclamation mark ("!") and a transformation function, see below.

  Note that braces ("{" and "}") must be escaped with a preceding backslash ("\") if they should occur literally within the replacement string.

Debugging of DSL expansion can be turned on, selectively, by using a comment line starting with "#/" which may contain one or more words from the table presented below. The resulting output is written to standard output.

Table 4. Debug options for DSL expansion

Word	**Description**
result	Prints the resulting DRL text, with line numbers.
steps	Prints each expansion step of condition and consequence lines.
keyword	Dumps the internal representation of all DSL entries with scope "keyword".
when	Dumps the internal representation of all DSL entries with scope "when" or "*".
then	Dumps the internal representation of all DSL entries with scope "then" or "*".
usage	Displays a usage statistic of all DSL entries.

Below are some sample DSL definitions, with comments describing the language features they illustrate.

```
# Comment: DSL examples

#/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
[keyword][]regula=rule

# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
        ${entity!lc}: {entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][]update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.
2. Each of the "keyword" entries is applied to the entire text. First, the regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default (".*?"). Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.
3. Sections of the DSLR text between "when" and "then", and "then" and "end", respectively, are located and processed in a uniform manner, line by line, as described below.

   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ".*?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

   Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

   If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, i.e., a type name followed by a pair of parentheses. if this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma (",") is inserted beforehand.

If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a "modify" statement, ending in a pair of braces ("{" and "}"). If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma (",") is inserted beforehand.

It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., "accumulate") or it may only work for the first insertion (e.g., "eval").

All string transformation functions are described in the following table.

Table 5. String transformation functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.
ucfirst	Converts the first letter to upper case, and all other letters to lower case.
num	Extracts all digits and "-" from the string. If the last two digits in the original string are preceded by "." or ",", a decimal period is inserted in the corresponding position.
a?*b*/*c*	Compares the string with string *a*, and if they are equal, replaces it with *b*, otherwise with *c*. But *c* can be another triplet *a*, *b*, *c*, so that the entire structure is, in fact, a translation table.

The following DSL examples show how to use string transformation functions.

```
# definitions for conditions
[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()
[when][]- with an? {attr} greater than {amount}={attr} <= {amount!num}
[when][]- with a {what} {attr}={attr} {what!positive?>0/negative?%lt;0/zero?==0/ERROR}
```

A file containing a DSL definition has to be put under the resources folder or any of its subfolders like any other drools artifact. It must have the extension `.dsl`, or alternatively be marked with type `ResourceType.DSL`. when programmatically added to a `KieFileSystem`. For a file using DSL definition, the extension `.dslr` should be used, while it can be added to a `KieFileSystem` with type `ResourceType.DSLR`.

For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. Thus, the parser can "recognize" the DSL expressions and transform them into native rule language expressions.