JBoss Enterprise SOA Platform 5

JBoss Rules 5 Reference Guide

This guide is for developers

Edition 5.3.1

Last Updated: 2017-10-27

JBoss Enterprise SOA Platform 5 JBoss Rules 5 Reference Guide

This guide is for developers
Edition 5.3.1

David Le Sage
Red Hat Engineering Content Services

Suzanne Dorfield
Red Hat Engineering Content Services

Abstract

This guide teaches developers how to use JBoss Rules.

Table of Contents

PREFACE

CHAPTER 1. PREFACE

1.1. BUSINESS INTEGRATION

In order to provide a dynamic and competitive business infrastructure, it is crucial to have a service-oriented architecture in place that enables your disparate applications and data sources to communicate with each other with minimum overhead.

The JBoss Enterprise SOA Platform is a framework capable of orchestrating business services without the need to constantly reprogram them to fit changes in business processes. By using its business rules and message transformation and routing capabilities, JBoss Enterprise SOA Platform enables you to manipulate business data from multiple sources.

Report a bug

1.2. WHAT IS A SERVICE-ORIENTED ARCHITECTURE?

Introduction

A *Service Oriented Architecture* (SOA) is not a single program or technology. Think of it, rather, as a software design paradigm.

As you may already know, a *hardware bus* is a physical connector that ties together multiple systems and subsystems. If you use one, instead of having a large number of point-to-point connectors between pairs of systems, you can simply connect each system to the central bus. An *enterprise service bus* (ESB) does exactly the same thing in software.

The ESB sits in the architectural layer above a messaging system. This messaging system facilitates *asynchronous communications* between services through the ESB. In fact, when you are using an ESB, everything is, conceptually, either a *service* (which, in this context, is your application software) or a *message* being sent between services. The services are listed as connection addresses (known as *end-points references*.)

It is important to note that, in this context, a "service" is not necessarily always a web service. Other types of applications, using such transports as File Transfer Protocol and the Java Message Service, can also be "services."

NOTE

At this point, you may be wondering if an enterprise service bus is the same thing as a service-oriented architecture. The answer is, "Not exactly." An ESB does not form a service-oriented architecture of itself. Rather, it provides many of the tools than can be used to build one. In particular, it facilitates the *loose-coupling* and *asynchronous message passing* needed by a SOA. Always think of a SOA as being more than just software: it is a series of principles, patterns and best practices.

Report a bug

1.3. KEY POINTS OF A SERVICE-ORIENTED ARCHITECTURE

These are the key components of a service-oriented architecture:

1. the *messages* being exchanged
2. the *agents* that act as service requesters and providers
3. the *shared transport mechanisms* that allow the messages to flow back and forth.

Report a bug

1.4. WHAT IS THE JBOSS ENTERPRISE SOA PLATFORM?

The JBoss Enterprise SOA Platform is a framework for developing enterprise application integration (EAI) and service-oriented architecture (SOA) solutions. It is made up of an enterprise service bus (JBoss ESB) and some business process automation infrastructure. It allows you to build, deploy, integrate and orchestrate business services.

Report a bug

1.5. THE SERVICE-ORIENTED ARCHITECTURE PARADIGM

The service-oriented architecture (SOA) consists of three roles: requester, provider, and broker.

Service Provider
: A service provider allows access to services, creates a description of a service and publishes it to the service broker.

Service Requester
: A service requester is responsible for discovering a service by searching through the service descriptions given by the service broker. A requester is also responsible for binding to services provided by the service provider.

Service Broker
: A service broker hosts a registry of service descriptions. It is responsible for linking a requester to a service provider.

Report a bug

1.6. CORE AND COMPONENTS

The JBoss Enterprise SOA Platform provides a comprehensive server for your data integration needs. On a basic level, it is capable of updating business rules and routing messages through an Enterprise Service Bus.

The heart of the JBoss Enterprise SOA Platform is the Enterprise Service Bus. JBoss (ESB) creates an environment for sending and receiving messages. It is able to apply “actions” to messages to transform them and route them between services.

There are a number of components that make up the JBoss Enterprise SOA Platform. Along with the ESB, there is a registry (jUDDI), transformation engine (Smooks), message queue (HornetQ) and BPEL engine (Riftsaw).

Report a bug

1.7. COMPONENTS OF THE JBOSS ENTERPRISE SOA PLATFORM

- A full Java EE-compliant application server (the JBoss Enterprise Application Platform)
- an enterprise service bus (JBoss ESB)
- a business process management system (jBPM)
- a business rules engine (JBoss Rules)
- support for the optional JBoss Enterprise Data Services (EDS) product.

Report a bug

1.8. JBOSS ENTERPRISE SOA PLATFORM FEATURES

The JBoss Enterprise Service Bus (ESB)

The ESB sends messages between services and transforms them so that they can be processed by different types of systems.

Business Process Execution Language (BPEL)

You can use web services to orchestrate business rules using this language. It is included with SOA for the simple execution of business process instructions.

Java Universal Description, Discovery and Integration (jUDDI)

This is the default service registry in SOA. It is where all the information pertaining to services on the ESB are stored.

Smooks

This transformation engine can be used in conjunction with SOA to process messages. It can also be used to split messages and send them to the correct destination.

JBoss Rules

This is the rules engine that is packaged with SOA. It can infer data from the messages it receives to determine which actions need to be performed.

Report a bug

1.9. FEATURES OF THE JBOSS ENTERPRISE SOA PLATFORM'S JBOSSESB COMPONENT

The JBoss Enterprise SOA Platform's JBossESB component supports:

- Multiple transports and protocols
- A listener-action model (so that you can loosely-couple services together)

- Content-based routing (through the JBoss Rules engine, XPath, Regex and Smooks)
- Integration with the JBoss Business Process Manager (jBPM) in order to provide service orchestration functionality
- Integration with JBoss Rules in order to provide business rules development functionality.
- Integration with a BPEL engine.

Furthermore, the ESB allows you to integrate legacy systems in new deployments and have them communicate either synchronously or asynchronously.

In addition, the enterprise service bus provides an infrastructure and set of tools that can:

- Be configured to work with a wide variety of transport mechanisms (such as e-mail and JMS),
- Be used as a general-purpose object repository,
- Allow you to implement pluggable data transformation mechanisms,
- Support logging of interactions.

IMPORTANT

There are two trees within the source code: **`org.jboss.internal.soa.esb`** and **`org.jboss.soa.esb`**. Use the contents of the **`org.jboss.internal.soa.esb`** package sparingly because they are subject to change without notice. By contrast, everything within the **`org.jboss.soa.esb`** package is covered by Red Hat's deprecation policy.

Report a bug

1.10. TASK MANAGEMENT

JBoss SOA simplifies tasks by designating tasks to be performed universally across all systems it affects. This means that the user does not have to configure the task to run separately on each terminal. Users can connect systems easily by using web services.

Businesses can save time and money by using JBoss SOA to delegate their transactions once across their networks instead of multiple times for each machine. This also decreases the chance of errors ocurring.

Report a bug

1.11. INTEGRATION USE CASE

Acme Equity is a large financial service. The company possesses many databases and systems. Some are older, COBOL-based legacy systems and some are databases obtained through the acquisition of smaller companies in recent years. It is challenging and expensive to integrate these databases as business rules frequently change. The company wants to develop a new series of client-facing e-commerce websites, but these may not synchronise well with the existing systems as they currently stand.

The company wants an inexpensive solution but one that will adhere to the strict regulations and security requirements of the financial sector. What the company does not want to do is to have to write and maintain “glue code” to connect their legacy databases and systems.

The JBoss Enterprise SOA Platform was selected as a middleware layer to integrate these legacy systems with the new customer websites. It provides a bridge between front-end and back-end systems. Business rules implemented with the JBoss Enterprise SOA Platform can be updated quickly and easily.

As a result, older systems can now synchronise with newer ones due to the unifying methods of SOA. There are no bottlenecks, even with tens of thousands of transactions per month. Various integration types, such as XML, JMS and FTP, are used to move data between systems. Any one of a number of enterprise-standard messaging systems can be plugged into JBoss Enterprise SOA Platform providing further flexibility.

An additional benefit is that the system can now be scaled upwards easily as more servers and databases are added to the existing infrastructure.

Report a bug

1.12. UTILISING THE JBOSS ENTERPRISE SOA PLATFORM IN A BUSINESS ENVIRONMENT

Cost reduction can be achieved due to the implementation of services that can quickly communicate with each other with less chance of error messages occurring. Through enhanced productivity and sourcing options, ongoing costs can be reduced.

Information and business processes can be shared faster because of the increased connectivity. This is enhanced by web services, which can be used to connect clients easily.

Legacy systems can be used in conjunction with the web services to allow different systems to "speak" the same language. This reduces the amount of upgrades and custom code required to make systems synchronise.

Report a bug

CHAPTER 2. INTRODUCTION

2.1. INTENDED AUDIENCE

This book is aimed at system administrators who wish to learn how to utilize the tools in JBoss Rules. It explains how to create new projects, debug projects and how to use editors.

Report a bug

2.2. AIM OF THE GUIDE

This guide aims to give users an overview of how to use JBoss Rules. Users will be taken through basic terminologies and learn how to create rules from scratch. There are also a number of tutorials to assist in creating rules for projects.

Report a bug

CHAPTER 3. QUICK START

3.1. JBOSS RULES

JBoss Rules is the name of the business rule engine provided as part of the JBoss Enterprise SOA Platform product.

Report a bug

3.2. THE JBOSS RULES ENGINE

The JBoss Rules engine is the computer program that applies rules and delivers Knowledge Representation and Reasoning (KRR) functionality to the developer.

Report a bug

3.3. PRODUCTION RULES

A *production rule* is a two-part structure that uses first order logic to represent knowledge. It takes the following form:

```
when
 <conditions>
then
 <actions>
```

Report a bug

3.4. THE INFERENCE ENGINE

The *inference engine* is the part of the JBoss Rules engine which matches production facts and data to rules. It will then perform actions based on what it infers from the information. A production rules system's inference engine is *stateful* and is responsible for *truth maintenance*.

Report a bug

3.5. RETEOO

The Rete implementation used in JBoss Rules is called *ReteOO*. It is an enhanced and optimized implementation of the Rete algorithm specifically for object oriented systems.

Report a bug

3.6. PRODUCTION MEMORY

The **`production memory`** is where rules are stored.

Report a bug

3.7. WORKING MEMORY

The **`working memory`** is the part of the JBoss Rules engine where facts are asserted. From here, the facts can be modified or retracted.

Report a bug

3.8. CONFLICT RESOLUTION STRATEGY

Conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on the working memory, the rule engine needs to know in what order the rules should fire (for instance, firing ruleA may cause ruleB to be removed from the agenda).

Report a bug

3.9. HYBRID RULE SYSTEMS

A hybrid rule system pertains to using both forward-chaining and backward-chaining rule systems to process rules.

Report a bug

3.10. FORWARD-CHAINING

Forward-chaining is a production rule system. It is data-driven which means it reacts to the data it is presented. Facts are inserted into the working memory which results in one or more rules being true. They are then placed on the schedule to be executed by the agenda.

JBoss Rules is a forward-chaining engine.

Report a bug

3.11. BACKWARD-CHAINING

A backward-chaining rule system is goal-driven. This means the system starts with a *conclusion* which the engine tries to satisfy. If it cannot do so it searches for sub-goals, that is, conclusions that will complete part of the current goal. It continues this process until either the initial conclusion is satisfied or there are no more unsatisfied sub-goals. **`Prolog`** is an example of a backward-chaining engine.

IMPORTANT

Backward-chaining was implemented in JBoss BRMS 5.2.

Report a bug

3.12. REASONING CAPABILITIES

JBoss Rules uses backward-chaining *reasoning capabilities* to help infer which rules to apply from the data.

Report a bug

3.13. EXPERT SYSTEMS

An *expert system* is said to be formed when a framework uses ontological model to represent a domain and contains facilities for knowledge acquisition and explanation.

Report a bug

3.14. THE RETE ROOT NODE

When using Rete00, the root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode.

Report a bug

3.15. THE OBJECTTYPENODE

The *ObjectTypeNode* helps to reduce the workload of the rules engine. If there are several objects and the rules engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the ObjectTypeNode is used so the engine only passes objects to the nodes that match the object's type. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object.

In JBoss Rules, an object which has been asserted will retrieve a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class. If this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables JBoss Rules to match against any Class type that matches with an `instanceof` check.

Report a bug

3.16. ALPHANODES

AlphaNodes are used to evaluate literal conditions. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode.

AlphaNodes are propagated using ObjectTypeNodes.

Report a bug

3.17. HASHING

JBoss Rules uses *hashing* to extend Rete by optimizing the propagation from ObjectTypeNode to AlphaNode. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap, thereby avoiding unnecessary literal checks.

Report a bug

3.18. BETANODES

BetaNodes are used to compare two objects and their fields. The objects may be the same or different types.

Report a bug

3.19. ALPHA MEMORY

Alpha memory refers to the right input on a BetaNode. In JBoss Rules, this input remembers all incoming objects.

Report a bug

3.20. BETA MEMORY

Beta memory is the term used to refer to the left input of a BetaNode. It remembers all incoming tuples.

Report a bug

3.21. LOOKUPS WITH BETANODES

When facts enter from one side, you can do a hash lookup returning potentially valid candidates (referred to as indexing). At any point a valid join is found, the Tuple will join with the Object (referred to as a partial match) and then propagate to the next node.

Report a bug

3.22. LEFTINPUTNODEADAPTERS

A *LeftInputNodeAdapter* takes an Object as an input and propagates a single Object Tuple.

Report a bug

3.23. TERMINAL NODES

Terminal nodes are used to indicate when a single rule has matched all its conditions (that is, the rule has a full match). A rule with an 'or' conditional disjunctive connective will result in a sub-rule generation for each possible logically branch. Because of this, one rule can have multiple terminal nodes.

Report a bug

3.24. NODE SHARING

Node sharing is used to prevent unnecessary redundancy. Because many rules repeat the same patterns, node sharing allows users to collapse those patterns so they don't have to be re-evaluated for every single instance.

Report a bug

3.25. NODE SHARING EXAMPLE

The following two rules share the first pattern, but not the last:

```
rule
when
    vehicle( $car : name == "car" )
    $driver: Driver( typeCar == $sedan )
then
    System.out.println( $driver.getName() + " drives sedan" );
end
```

```
rule
when
    Vehicle( $sedan : name == "sedan" )
    $driver : Driver( typeCar != $sedan )
then
    System.out.println( $driver.getName() + " does not drive sedan" );
end
```

Report a bug

3.26. LOOSE COUPLING

Loose coupling involves "loosely" linking rules so that the execution of one rule will not lead to the execution of another.

Generally, a design exhibiting loose coupling is preferable because it allows for more flexibility. If the rules are all strongly coupled, they are likely to be inflexible. More significantly, it indicates that deploying a rule engine is overkill for the situation.

Report a bug

3.27. STRONG COUPLING

Strong coupling is a way of linking rules. If rules are strongly-coupled, it means executing one rule will directly result in the execution of another. In other words, there is a clear chain of logic. (A clear chain can be hard-coded, or implemented using a decision tree.)

Report a bug

3.28. DECLARATIVE PROGRAMMING

Declarative programming refers to the way the rule engine allows users to declare "what to do" as opposed to "how to do it". The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified. Rules are much easier to read than code.

Report a bug

3.29. LOGIC AND DATA SEPARATION

Logic and Data separation refers to the process of de-coupling logic and data components. Using this method, the logic can be spread across many domain objects or controllers and it can all be organized in one or more discrete rules files.

Report a bug

3.30. KNOWLEDGE BASE

A knowledge base is a collection of rules which have been compiled by the **`KnowledgeBuilder`**. It is a repository of all the application's knowledge definitions. It may contain rules, processes, functions, and type models. The Knowledge Base itself does not contain instance data (known as facts). Instead, sessions are created from the Knowledge Base into which data can be inserted and where process instances may be started. It is recommended that Knowledge Bases be cached where possible to allow for repeated session creation.

Report a bug

CHAPTER 4. USER GUIDE

4.1. STATELESS KNOWLEDGE SESSIONS

A *stateless knowledge session* is a session without inference. A stateless session can be called like a function in that you can use it to pass data and then receive the result back.

Stateless knowledge sessions are useful in situations requiring validation, calculation, routing and filtering.

Report a bug

4.2. CONFIGURING RULES IN A STATELESS SESSION

Procedure 4.1. Task

1. Create a data model like the driver's license example below:

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

2. Write the first rule. In this example, a rule is added to disqualify any applicant younger than 18:

```
package com.company.license

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

3. When the **`Applicant`** object is inserted into the rule engine, each rule's constraints evaluate it and search for a match. (There is always an implied constraint of "object type" after which there can be any number of explicit field constraints.)

   In the **`Is of valid age`** rule there are two constraints:

   - The fact being matched must be of type Applicant
   - The value of Age must be less than eighteen.

   `$a` is a binding variable. It exists to make possible a reference to the matched object in the rule's consequence (from which place the object's properties can be updated).

NOTE

Use of the dollar sign (**$**) is optional. It helps to differentiate between variable names and field names.

NOTE

If the rules are in the same folder as the classes, the classpath resource loader can be used to build the first knowledge base.

4. Use the KnowledgeBuilder to to compile the list of rules into a knowledge base as shown:

```
KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource(
"licenseApplication.drl",
              getClass() ), ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
    System.err.println( kbuilder.getErrors().toString() );
}
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

   The above code snippet looks on the classpath for the **`licenseApplication.drl`** file, using the method **`newClassPathResource()`**. (The resource type is DRL, short for "Drools Rule Language".)

5. Check the KnowledgeBuilder for any errors. If there are none, you can build the session.

6. Execute the data against the rules. (Since the applicant is under the age of eighteen, their application will be marked as "invalid.")

```
StatelessKnowledgeSession ksession =
kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant( "Mr John Smith", 16 );
assertTrue( applicant.isValid() );
ksession.execute( applicant );
assertFalse( applicant.isValid() );
```

Result

The preceding code executes the data against the rules. Since the applicant is under the age of 18, the application is marked as invalid.

Report a bug

4.3. CONFIGURING RULES WITH MULTIPLE OBJECTS

Procedure 4.2. Task

1. To execute rules against any object-implementing **`iterable`** (such as a collection), add another class as shown in the example code below:

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

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

2. In order to check that the application was made within a legitimate time-frame, add this rule:

```
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
```

3. Use the JDK converter to implement the iterable interface. (This method commences with the line **`Arrays.asList(...)`**.) The code shown below executes rules against an iterable list. Every collection element is inserted before any matched rules are fired:

```
StatelessKnowledgeSession 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 } ) );
assertFalse( application.isValid() );
```

NOTE

The **`execute(Object object)`** and **`execute(Iterable objects)`** methods are actually "wrappers" around a further method called **`execute(Command command)`** which comes from the **`BatchExecutor`** interface.

4. Use the **`CommandFactory`** to create instructions, so that the following is equivalent to **`execute( Iterable it )`**:

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

5. Use the **`BatchExecutor`** and **`CommandFactory`** when working with many different commands or result output identifiers:

```
List<Command> cmds = new ArrayList<Command>();
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" ) );
```

NOTE

`CommandFactory` supports many other commands that can be used in the **`BatchExecutor`**. Some of these are **`StartProcess`**, **`Query`** and **`SetGlobal`**.

Report a bug

4.4. STATEFUL SESSIONS

A *stateful session* allow you to make iterative changes to facts over time. As with the **`StatelessKnowledgeSession`**, the **`StatefulKnowledgeSession`** supports the **`BatchExecutor`** interface. The only difference is the **`FireAllRules`** command is not automatically called at the end.

WARNING

Ensure that the **`dispose()`** method is called after running a stateful session. This is to ensure that there are no memory leaks. This is due to the fact that knowledge bases will obtain references to stateful knowledge sessions when they are created.

Report a bug

4.5. COMMON USE CASES FOR STATEFUL SESSIONS

Monitoring

For example, you can monitor a stock market and automate the buying process.

Diagnostics

Stateful sessions can be used to run fault-finding processes. They could also be used for medical diagnostic processes.

Logistical

For example, they could be applied to problems involving parcel tracking and delivery provisioning.

Ensuring compliance

For example, to validate the legality of market trades.

Report a bug

4.6. STATEFUL SESSION MONITORING EXAMPLE

Procedure 4.3. Task

1. Create a model of what you want to monitor. In this example involving fire alarms, the rooms in a house have been listed. Each has one sprinkler. A fire can start in any of the rooms:

   ```
   public class Room
   {
    private String name
     // getter and setter methods here
   }

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

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

   public class Alarm
   {
   }
   ```

2. The rules must express the relationships between multiple objects (to define things such as the presence of a sprinkler in a certain room). To do this, use a *binding variable* as a constraint in a pattern. This results in a cross-product.

3. Create an instance of the **`Fire`** class and insert it into the session.

   The rule below adds a binding to **`Fire`** object's room field to constrain matches. This so that only the sprinkler for that room is checked. When this rule fires and the consequence executes, the sprinkler activates:

```
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
```

Whereas the stateless session employed standard Java syntax to modify a field, the rule above uses the **`modify`** statement. (It acts much like a "with" statement.)

Report a bug

4.7. FIRST ORDER LOGIC

First order logic allows you to look at sets of data instead of individual instances.

Report a bug

4.8. CONFIGURING RULES WITH FIRST ORDER LOGIC

Procedure 4.4. Task

1. Configure a pattern featuring the keyword **`Not`**. First order logic ensures rules will only be matched when no other keywords are present. In this example, the rule turns the sprinkler off when the fire is extinguished:

```
rule "When the fire is gone turn off the sprinkler"
when
    $room : Room( )
    $sprinkler : Sprinkler( room == $room, on == true )
    not Fire( room == $room )
then
    modify( $sprinkler ) { setOn( false ) };
    System.out.println("Turn off the sprinkler for room
"+$room.getName());
end
```

2. An **`Alarm`** object is created when there is a fire, but only one **`Alarm`** is needed for the entire building no matter how many fires there might be. **`Not`**'s complement, **`exists`** can now be introduced. It matches one or more instances of a category:

```
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
```

▪

3. If there are no more fires, the alarm must be deactivated. To turn it off, use **`Not`** again:

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

4. Use this code to print a general health status message when the application first starts and also when the alarm and all of the sprinklers have been deactivated:

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

5. Store the rules in a file called **`fireAlarm.drl`**. Save this file in a sub-directory on the class-path.

6. Finally, build a **`knowledge base`**, using the new name, **`fireAlarm.drl`**:

```
KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "fireAlarm.drl",
          getClass() ), ResourceType.DRL );

if ( kbuilder.hasErrors() )
 System.err.println( kbuilder.getErrors().toString() );

StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();
```

Report a bug

4.9. RULE SYSTEM SAMPLE CONFIGURATION

Procedure 4.5. Task

1. Insert **`ksession.fireAllRules()`**. This grants the matched rules permission to run but, since there is no fire in this example, they will merely produce the health message:

```
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();
```

The resulting message reads:

```
> Everything is okay
```

2. Create and insert two fires. (A fact handle will be kept.)

3. With the fires now in the engine, call **`fireAllRules()`**. The alarm will be raised and the respective sprinklers will be turned on:

```
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();
```

The resulting message reads:

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

4. When the fires are extinguished, the fire objects are retracted and the sprinklers are turned off. At this point in time, the alarm is canceled and the health message displays once more:

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

ksession.fireAllRules();
```

The resulting message reads:

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

Report a bug

4.10. METHODS IN JBOSS RULES

Methods are different to rules. They are called directly and are used to pass specific instances. A single call results in a single execution.

Report a bug

4.11. METHOD EXAMPLE

A method looks like this:

```
public void helloWorld(Person person) {
    if ( person.getName().equals( "Chuck" ) ) {
        System.out.println( "Hello Chuck" );
    }
}
```

Report a bug

4.12. RULE EXAMPLE

A rule looks like this:

```
rule "Hello World"
    when
        Person( name == "Chuck" )
    then
        System.out.println( "Hello Chuck" );
end
```

Report a bug

4.13. CROSS-PRODUCTS

When two or more sets of data are combined, the result is called a *cross-product*.

Report a bug

4.14. CROSS-PRODUCT CONSTRAINING

Procedure 4.6. Task

- To prevent a rule from outputting a huge amount of cross-products, you should constrain the cross-products themselves. Do this using the variable constraint seen below:

  ```
  rule
  when
  ```

```
    $room : Room()
    $sprinkler : Sprinkler( room == $room )
then
    System.out.println( "room:" + $room.getName() +
                        " sprinkler:" +
$sprinkler.getRoom().getName() );
end
```

The following output will be displayed:

```
room:office sprinkler:office
room:kitchen sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

Result

Only four rows are outputted with the correct sprinkler for each room. Without this variable, every row in the Room table would have been joined with every row in the Sprinkler table resulting in many lines of output.

Report a bug

4.15. THE INFERENCE ENGINE

The *inference engine* is the part of the JBoss Rules engine which matches production facts and data to rules. It will then perform actions based on what it infers from the information. A production rules system's inference engine is *stateful* and is responsible for *truth maintenance*.

Report a bug

4.16. INFERENCE EXAMPLE

In this example, a Person fact with an age field and a rule that provides age policy control is used. Inference is used to determine if a Person is an adult or a minor, then act on the result:

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

In the above snippet, every Person who is 18 or over will have an instance of IsAdult inserted for them. This fact is special in that it is known as a relation. We can use this inferred relation in any rule:

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

Report a bug

4.17. LOGICAL ASSERTIONS

After a standard object insertion, you have to retract facts explicitly. With *logical* assertions, the fact that was asserted will be automatically retracted when the conditions that asserted it in the first place are no longer true. It will be retracted only if there isn't any single condition that supports the logical assertion.

Report a bug

4.18. STATED INSERTIONS

Normal insertions are said to be *stated*, as in "stating a fact". Using a **`HashMap`** and a counter, you can track how many times a particular equality is *stated*. This means you can count how many different instances are equal.

Report a bug

4.19. JUSTIFIED INSERTIONS

When an object is logically inserted, it is said to be *justified*. It is considered to be justified by the firing rule. For each logical insertion there can only be one equal object, and each subsequent equal logical insertion increases the justification counter for this logical assertion. A justification is removed when the creating rule's LHS becomes untrue, and the counter is decreased accordingly. As soon as there are no more justifications, the logical object is automatically retracted.

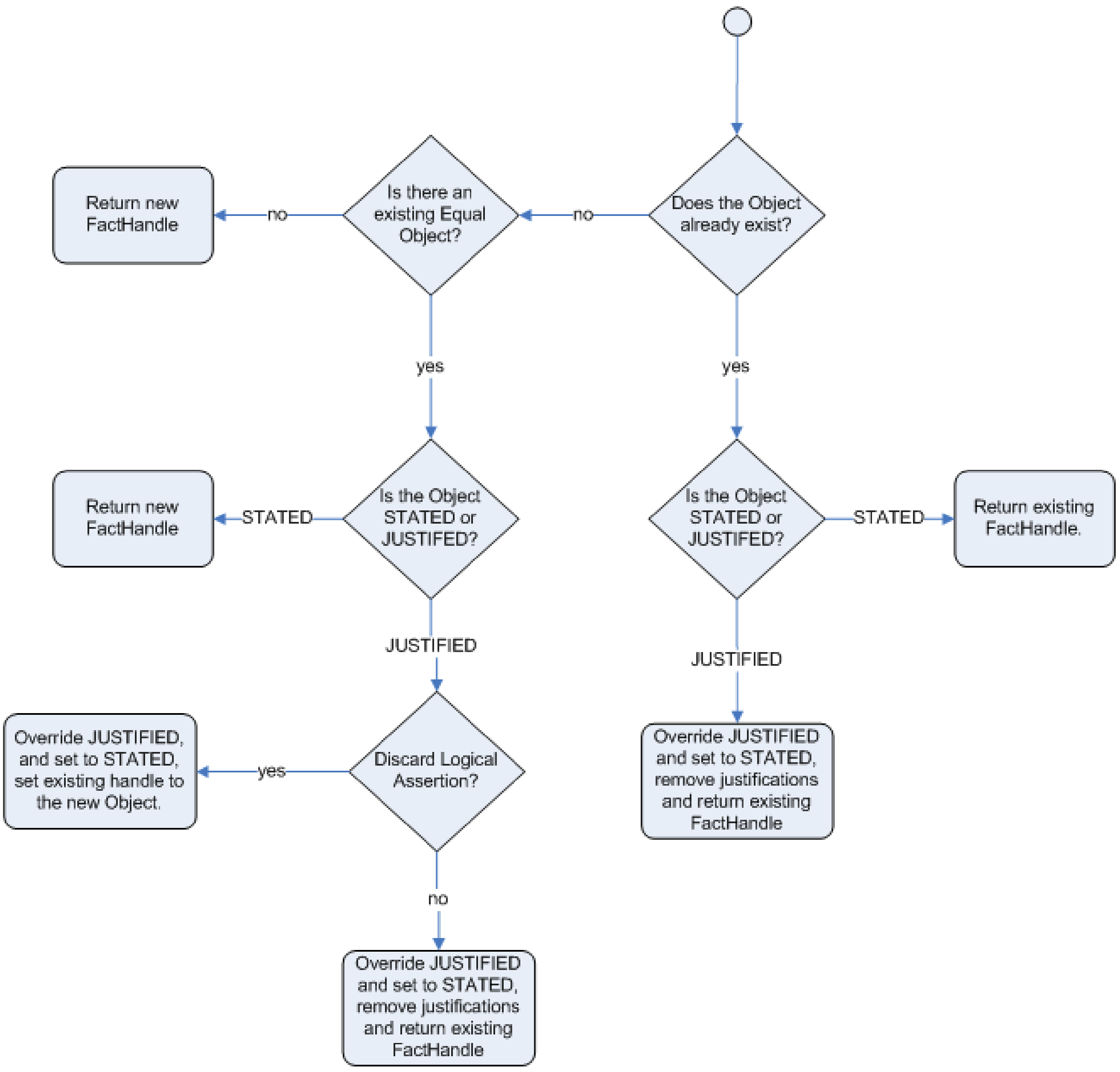
Report a bug

4.20. THE WM_BEHAVIOR_PRESERVE SETTING

If you try to *logically* insert an object when there is an equal *stated* object, this will fail and return null. If you *state* an object that has an existing equal object that is *justified*, you will override the Fact. How this override works depends on the configuration setting **`WM_BEHAVIOR_PRESERVE`**. When the property is set to **`discard`**, you can use the existing handle and replace the existing instance with the new Object, which is the default behavior. Otherwise you should override it to *stated* but create an new **`FactHandle`**.

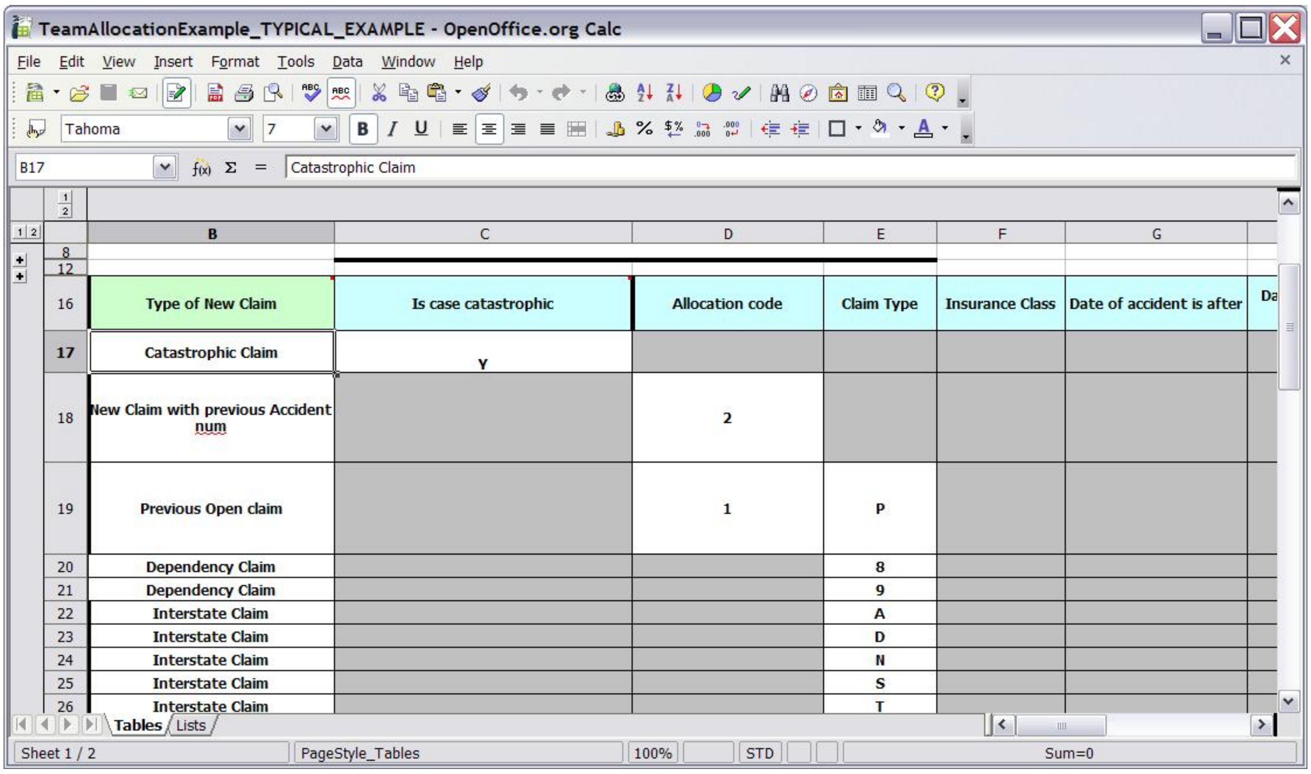
Report a bug

4.21. STATED INSERTION FLOWCHART


Does the Object already exist?
no
Is there an existing Equal Object?
no
Return new FactHandle
yes
yes
Is the Object STATED or JUSTIFED?
STATED
Return new FactHandle
Is the Object STATED or JUSTIFED?
STATED
Return existing FactHandle.
JUSTIFIED
Discard Logical Assertion?
yes
Override JUSTIFIED, and set to STATED, set existing handle to the new Object.
no
Override JUSTIFIED and set to STATED, remove justifications and return existing FactHandle
JUSTIFIED
Override JUSTIFIED and set to STATED, remove justifications and return existing FactHandle

Report a bug

4.22. LOGICAL INSERTION FLOWCHART


TeamAllocationExample_TYPICAL_EXAMPLE - OpenOffice.org Calc
File Edit View Insert Format Tools Data Window Help
Tahoma
7
B17
Tables
Lists
Sheet 1 / 2
PageStyle_Tables
100%
STD
Sum=0

Catastrophic Claim

	B	C	D	E	F	G	
16	Type of New Claim	Is case catastrophic	Allocation code	Claim Type	Insurance Class	Date of accident is after	Da
17	Catastrophic Claim	Y					
18	New Claim with previous Accident num		2				
19	Previous Open claim		1	P			
20	Dependency Claim			8			
21	Dependency Claim			9			
22	Interstate Claim			A			
23	Interstate Claim			D			
24	Interstate Claim			N			
25	Interstate Claim			S			
26	Interstate Claim			T			

Report a bug

4.23. THE TRUTH MAINTENANCE SYSTEM

The JBoss Rules *Truth Maintenance System* (TMS) is a method of representing beliefs and their related dependencies/justifications in a knowledge base.

IMPORTANT

For Truth Maintenance (and logical assertions) to work, the Fact objects must override equals and hashCode methods correctly. As the truth maintenance system needs to know when two different physical objects are equal in value, *both* equals and hashCode must be overridden correctly, as per the Java standard.

Two objects are equal only if their equals methods return true for each other and if their hashCode methods return the same values. You must override both equals and hashCode.

Report a bug

4.24. THE INSERTLOGICAL FACT

The **`insertLogical`** fact is part of the JBoss Rules TMS. It "inserts logic" so that rules behave and are modified according to the situation. For example, the **`insertLogical`** fact can be added to a set of rules so that when a rule becomes false, the fact is automatically retracted.

Report a bug

4.25. USING INFERENCE AND TMS

Procedure 4.7. Task

1. In this example we will use a bus pass issuing system. See the code snippet below:

```
rule "Issue Child Bus Pass" when
  $p : Person( age < 16 )
then
  insert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
  $p : Person( age >= 16 )
then
  insert(new AdultBusPass( $p ) );
end
```

2. Insert the **`insertLogical`** property to provide inference:

```
rule "Infer Child" when
    $p : Person( age < 16 )
then
    insertLogical( new IsChild( $p ) )
end
rule "Infer Adult" when
    $p : Person( age >= 16 )
then
    insertLogical( new IsAdult( $p ) )
end
```

3. Re-enter the code to issue the passes. These two configurations can also be logically inserted, as the TMS supports chaining of logical insertions for a cascading set of retracts:

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

rule "Issue Adult Bus Pass" when
  $p : Person( age >= 16 )
       IsAdult( person =$p )
then
    insertLogical(new AdultBusPass( $p ) );
end
```

   Now when the person changes from being 15 to 16, both the **`IsChild`** fact and the person's **`ChildBusPass`** fact are automatically retracted.

4. Optionally, insert the **`not`** conditional element to handle notifications. (In this example, a request for the returning of the pass.) When the TMS automatically retracts the **`ChildBusPass`** object, this rule triggers and sends a request to the person:

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

Report a bug

CHAPTER 5. DECISION TABLES

5.1. DECISION TABLES

Decision tables are a way of representing conditional logic. They are well suited to *business* level rules.

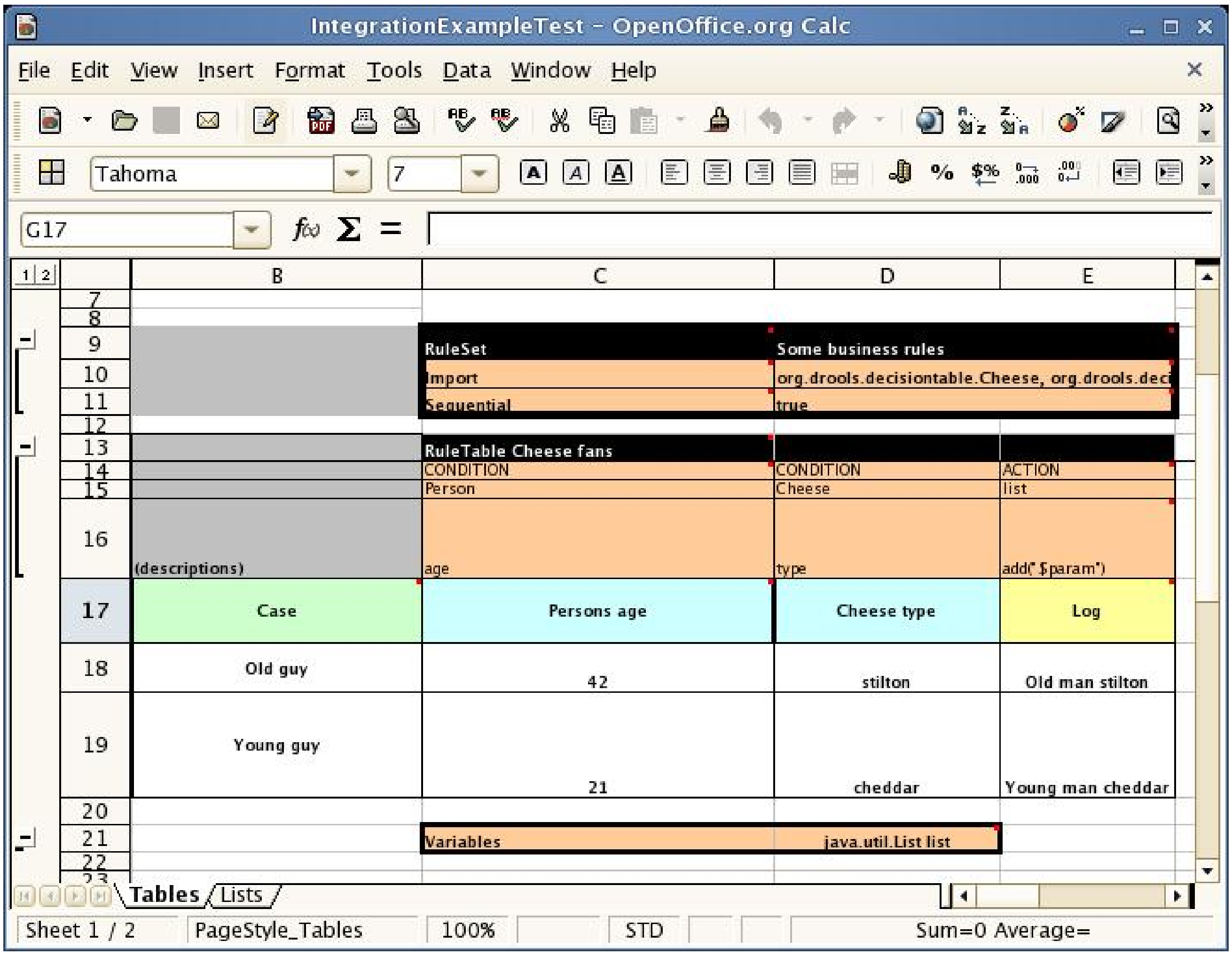
Report a bug

5.2. DECISION TABLES IN SPREADSHEETS

JBoss Rules supports managing rules in a spreadsheet format. Supported formats are Excel (XLS) and CSV. This means that a variety of spreadsheet programs (such as Microsoft Excel, OpenOffice.org Calc, and others) can be utilized.

Report a bug

5.3. OPEN OFFICE EXAMPLE


IntegrationExampleTest - OpenOffice.org Calc
File Edit View Insert Format Tools Data Window Help
Tahoma
7
G17
Tables
Lists
Sheet 1 / 2
PageStyle_Tables
100%
STD
Sum=0 Average=

	B	C	D	E
7				
8				
9		RuleSet	Some business rules	
10		Import	org.drools.decisiontable.Cheese, org.drools.deci	
11		Sequential	true	
12				
13		RuleTable Cheese fans		
14		CONDITION	CONDITION	ACTION
15		Person	Cheese	list
16	(descriptions)	age	type	add("$param")
17	Case	Persons age	Cheese type	Log
18	Old guy	42	stilton	Old man stilton
19	Young guy	21	cheddar	Young man cheddar
20				
21		Variables	java.util.List list	
22				

Figure 5.1. Open Office Screenshot

In the above examples, the technical aspects of the decision table have been collapsed away (using a standard spreadsheet feature).

The rules start from row 17, with each row resulting in a rule. The conditions are in columns C, D, E, etc., and the actions are off-screen. The values' meanings are indicated by the headers in Row 16. (Column B is just a description.)

NOTE

Although the decision tables look like they process top down, this is not necessarily the case. Ideally, rules are authored without regard for the order of rows. This makes maintenance easier, as rows will not need to be shifted around all the time.

Report a bug

5.4. RULES AND SPREADSHEETS

Rules inserted into rows

As each row is a rule, the same principles apply as with written code. As the rule engine processes the facts, any rules that match may fire.

Agendas

It is possible to clear the agenda when a rule fires and simulate a very simple decision table where only the first match effects an action.

Multiple tables

You can have multiple tables on one spreadsheet. This way, rules can be grouped where they share common templates, but are still all combined into one rule package.

Report a bug

5.5. THE RULETABLE KEYWORD

When using decision tables, the spreadsheet searches for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column).

IMPORTANT

Keywords should all be in the same column.

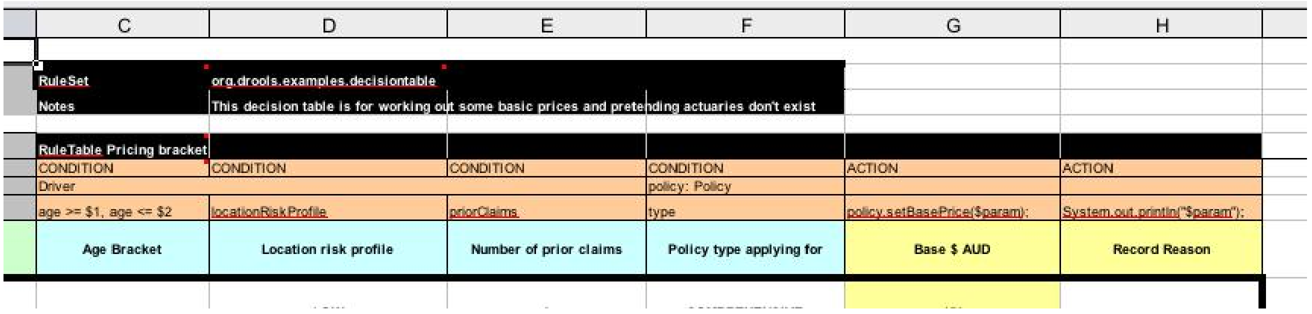
Report a bug

5.6. THE RULESET KEYWORD

The *RuleSet* keyword indicates the name to be used in the *rule package* that will encompass all the rules. This name is optional, using a default, but it *must* have the *RuleSet* keyword in the cell immediately to the right.

Report a bug

5.7. RULE TEMPLATE EXAMPLE



C	D	E	F	G	H
RuleSet	org.drools.examples.decisiontable				
Notes	This decision table is for working out some basic prices and pretending actuaries don't exist				
RuleTable Pricing bracket					
CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION
Driver			policy: Policy		
age >= $1, age <= $2	locationRiskProfile	priorClaims	type	policy.setBasePrice($param);	System.out.println("$param");
Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason

Figure 5.2. Rule Template

- The RuleSet keyword indicates the name to be used in the *rule package* that will encompass all the rules. This name is optional, using a default, but it *must* have the *RuleSet* keyword in the cell immediately to the right. The other keywords visible in Column C are Import and Sequential.
- After the RuleTable keyword there is a name, used to prefix the names of the generated rules. The row numbers are appended to guarantee unique rule names.
- The column of RuleTable indicates the column in which the rules start; columns to the left are ignored.
- Referring to row 14 (the row immediately after RuleTable), the keywords CONDITION and ACTION indicate that the data in the columns below are for either the LHS or the RHS parts of a rule. There are other attributes on the rule which can also be optionally set this way.
- Row 15 contains declarations of *ObjectTypes*. The content in this row is optional, but if this option is not in use, the row must be left blank. When using this row, the values in the cells below (row 16) become constraints on that object type. In the above case, it generates **`Person(age=="42")`** and **`Cheese(type=="stilton")`**, where 42 and "stilton" come from row 18. In the above example, the "==" is implicit. If just a field name is given, the translator assumes that it is to generate an exact match.
- Row 16 contains the rule templates themselves. They can use the "$param" placeholder to indicate where data from the cells below should be interpolated. (For multiple insertions, use "$1", "$2", etc., indicating parameters from a comma-separated list in a cell below.) Row 17 is ignored. It may contain textual descriptions of the column's purpose.
- Rows 18 and 19 show data, which will be combined (interpolated) with the templates in row 15, to generate rules. If a cell contains no data, then its template is ignored. (This would mean that some condition or action does not apply for that rule row.) Rule rows are read until there is a blank row. Multiple RuleTables can exist in a sheet.
- Row 20 contains another keyword, and a value. The row positions of keywords like this do not matter (most people put them at the top) but their column should be the same one where the RuleTable or RuleSet keywords should appear. In our case column C has been chosen to be significant, but any other column could be used instead.

NOTE

An ObjectType declaration can span columns (via merged cells), meaning that all columns below the merged range are to be combined into one set of constraints within a single pattern matching a single fact at a time, as opposed to non-merged cells containing the same ObjectType, but resulting in different patterns, potentially matching different or identical facts.

In the above example, rules would be rendered like the following (as it uses the "ObjectType" row):

```
//row 18
rule "Cheese_fans_18"
when
    Person(age=="42")
    Cheese(type=="stilton")
then
    list.add("Old man stilton");
end
```

NOTE

The constraints **`age=="42"`** and **`type=="stilton"`** are interpreted as single constraints, to be added to the respective ObjectType in the cell above. If the cells above were spanned, then there could be multiple constraints on one "column".

WARNING

Very large decision tables may have very large memory requirements.

Report a bug

5.8. DATA-DEFINING CELLS

There are two types of rectangular areas *defining data* that is used for generating a DRL file. One, marked by a cell labelled **`RuleSet`**, defines all DRL items except rules. The other one may occur repeatedly and is to the right and below a cell whose contents begin with **`RuleTable`**. These areas represent the actual decision tables, each area resulting in a set of rules of similar structure.

A Rule Set area may contain cell pairs, one below the **`RuleSet`** cell and containing a keyword designating the kind of value contained in the other one that follows in the same row.

Report a bug

5.9. RULE TABLE COLUMNS

The columns of a Rule Table area define patterns and constraints for the left hand sides of the rules derived from it, actions for the consequences of the rules, and the values of individual rule attributes. A Rule Table area should contain one or more columns, both for conditions and actions, and an arbitrary selection of columns for rule attributes, at most one column for each of these. The first four rows following the row with the cell marked with `RuleTable` are earmarked as header area, mostly used for the definition of code to construct the rules. It is any additional row below these four header rows that spawns another rule, with its data providing for variations in the code defined in the Rule Table header.

NOTE

All keywords are case insensitive.

Only the first worksheet is examined for decision tables.

Report a bug

5.10. RULE SET ENTRIES

Entries in a Rule Set area may define DRL constructs (except rules), and specify rule attributes. While entries for constructs may be used repeatedly, each rule attribute may be given at most once, and it applies to all rules unless it is overruled by the same attribute being defined within the Rule Table area.

Entries must be given in a vertically stacked sequence of cell pairs. The first one contains a keyword and the one to its right the value. This sequence of cell pairs may be interrupted by blank rows or even a Rule Table, as long as the column marked by `RuleSet` is upheld as the one containing the keyword.

Report a bug

5.11. ENTRIES IN THE RULE SET AREA

Table 5.1. Entries in the Rule Set area

Keyword	Value	Usage
RuleSet	The package name for the generated DRL file. Optional, the default is `rule_table`.	Must be First entry.
Sequential	"true" or "false". If "true", then salience is used to ensure that rules fire from the top down.	Optional, at most once. If omitted, no firing order is imposed.
EscapeQuotes	"true" or "false". If "true", then quotation marks are escaped so that they appear literally in the DRL.	Optional, at most once. If omitted, quotation marks are escaped.
Import	A comma-separated list of Java classes to import.	Optional, may be used repeatedly.

Keyword	Value	Usage
Variables	Declarations of DRL globals, i.e., a type followed by a variable name. Multiple global definitions must be separated with a comma.	Optional, may be used repeatedly.
Functions	One or more function definitions, according to DRL syntax.	Optional, may be used repeatedly.
Queries	One or more query definitions, according to DRL syntax.	Optional, may be used repeatedly.
Declare	One or more declarative types, according to DRL syntax.	Optional, may be used repeatedly.

Report a bug

5.12. RULE ATTRIBUTE ENTRIES IN THE RULE SET AREA

Table 5.2. Rule Attribute Entries in the Rule Set Area

Keyword	Initial	Value
PRIORITY	P	An integer defining the "salience" value for the rule. Overridden by the "Sequential" flag.
DURATION	D	A long integer value defining the "duration" value for the rule.
TIMER	T	A timer definition. See "Timers and Calendars".
CALENDARS	E	A calendars definition. See "Timers and Calendars".
NO-LOOP	U	A Boolean value. "true" inhibits looping of rules due to changes made by its consequence.
LOCK-ON-ACTIVE	L	A Boolean value. "true" inhibits additional activations of all rules with this flag set within the same ruleflow or agenda group.

Keyword	Initial	Value
AUTO-FOCUS	F	A Boolean value. "true" for a rule within an agenda group causes activations of the rule to automatically give the focus to the group.
ACTIVATION-GROUP	X	A string identifying an activation (or XOR) group. Only one rule within an activation group will fire, i.e., the first one to fire cancels any existing activations of other rules within the same group.
AGENDA-GROUP	G	A string identifying an agenda group, which has to be activated by giving it the "focus", which is one way of controlling the flow between groups of rules.
RULEFLOW-GROUP	R	A string identifying a rule-flow group.

Report a bug

5.13. THE RULETABLE CELL

All Rule Tables begin with a cell containing "RuleTable", optionally followed by a string within the same cell. The string is used as the initial part of the name for all rules derived from this Rule Table, with the row number appended for distinction. (This automatic naming can be overridden by using a NAME column.) All other cells defining rules of this Rule Table are below and to the right of this cell.

Report a bug

5.14. COLUMN TYPES

The next row after the RuleTable cell defines the column type. Each column results in a part of the condition or the consequence, or provides some rule attribute, the rule name or a comment. Each attribute column may be used at most once.

Report a bug

5.15. COLUMN HEADERS IN THE RULE TABLE

Table 5.3. Column Headers in the Rule Table

Keyword	Initial	Value	Usage
NAME	N	Provides the name for the rule generated from that row. The default is constructed from the text following the RuleTable tag and the row number.	At most one column
DESCRIPTION	I	A text, resulting in a comment within the generated rule.	At most one column
CONDITION	C	Code snippet and interpolated values for constructing a constraint within a pattern in a condition.	At least one per rule table
ACTION	A	Code snippet and interpolated values for constructing an action for the consequence of the rule.	At least one per rule table
METADATA	@	Code snippet and interpolated values for constructing a metadata entry for the rule.	Optional, any number of columns

Report a bug

5.16. CONDITIONAL ELEMENTS

Given a column headed CONDITION, the cells in successive lines result in a conditional element.

- Text in the first cell below CONDITION develops into a pattern for the rule condition, with the snippet in the next line becoming a constraint. If the cell is merged with one or more neighbours, a single pattern with multiple constraints is formed: all constraints are combined into a parenthesized list and appended to the text in this cell. The cell may be left blank, which means that the code snippet in the next row must result in a valid conditional element on its own.

  To include a pattern without constraints, you can write the pattern in front of the text for another pattern.

  The pattern may be written with or without an empty pair of parentheses. A "from" clause may be appended to the pattern.

  If the pattern ends with "eval", code snippets are supposed to produce boolean expressions for inclusion into a pair of parentheses after "eval".

- Text in the second cell below CONDITION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. If you want to create a constraint consisting of a comparison using "==" with the value from the cells below, the field selector alone is sufficient. Any other comparison operator must be specified as the last item within the snippet, and the value from the cells below is appended. For all other constraint forms, you must mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A text according to the pattern **`forall(`***delimiter***`){`***snippet***`}`** is expanded by repeating the *snippet* once for each of the values of the comma-separated list of values in each of the cells below, inserting the value in place of the symbol **`$`** and by joining these expansions by the given *delimiter*. Note that the forall construct may be surrounded by other text.

  2. If the cell in the preceding row is not empty, the completed code snippet is added to the conditional element from that cell. A pair of parentheses is provided automatically, as well as a separating comma if multiple constraints are added to a pattern in a merged cell.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below CONDITION is for documentation only. It should be used to indicate the column's purpose to a human reader.

- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the conditional element or constraint for this rule.

Report a bug

5.17. ACTION STATEMENTS

Given a column headed ACTION, the cells in successive lines result in an action statement:

- Text in the first cell below ACTION is optional. If present, it is interpreted as an object reference.

- Text in the second cell below ACTION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. For a singular insertion, mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A method call without interpolation can be achieved by a text without any marker symbols. In this case, use any non-blank entry in a row below to include the statement.

     The forall construct is available here, too.

  2. If the first cell is not empty, its text, followed by a period, the text in the second cell and a terminating semicolon are stringed together, resulting in a method call which is added as an action statement for the consequence.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below ACTION is for documentation only. It should be used to indicate the column's purpose to a human reader.

- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the action statement for this rule.

NOTE

Using **`$1`** instead of **`$param`** will fail if the replacement text contains a comma.

Report a bug

5.18. METADATA STATEMENTS

Given a column headed METADATA, the cells in successive lines result in a metadata annotation for the generated rules:

- Text in the first cell below METADATA is ignored.
- Text in the second cell below METADATA is subject to interpolation, as described above, using values from the cells in the rule rows. The metadata marker character `@` is prefixed automatically, and should not be included in the text for this cell.
- Text in the third cell below METADATA is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the metadata annotation for this rule.

Report a bug

5.19. INTERPOLATING CELL DATA EXAMPLE

- If the template is **`Foo(bar == $param)`** and the cell is **`42`**, then the result is **`Foo(bar == 42)`**.
- If the template is **`Foo(bar < $1, baz == $2)`** and the cell contains **`42,43`**, the result will be **`Foo(bar < 42, baz ==43)`**.
- The template **`forall(&&){bar != $}`** with a cell containing **`42,43`** results in **`bar != 42 && bar != 43`**.

Report a bug

5.20. TIPS FOR WORKING WITHIN CELLS

- Multiple package names within the same cell must be comma-separated.
- Pairs of type and variable names must be comma-separated.
- Functions must be written as they appear in a DRL file. This should appear in the same column as the "RuleSet" keyword. It can be above, between or below all the rule rows.

- You can use Import, Variables, Functions and Queries repeatedly instead of packing several definitions into a single cell.
- Trailing insertion markers can be omitted.
- You can provide the definition of a binding variable.
- Anything can be placed in the object type row. Apart from the definition of a binding variable, it could also be an additional pattern that is to be inserted literally.
- The cell below the ACTION header can be left blank. Using this style, anything can be placed in the consequence, not just a single method call. (The same technique is applicable within a CONDITION column.)

Report a bug

5.21. THE SPREADSHEETCOMPILER CLASS

The **`SpreadsheetCompiler`** class is the main class used with API spreadsheet-based decision tables in the drools-decisiontables module. This class takes spreadsheets in various formats and generates rules in DRL.

The **`SpreadsheetCompiler`** can be used to generate partial rule files and assemble them into a complete rule package after the fact. This allows the separation of technical and non-technical aspects of the rules if needed.

Report a bug

5.22. USING SPREADSHEET-BASED DECISION TABLES

Procedure 5.1. Task

1. Generate a sample spreadsheet can to use as the base.
2. If the Rule Workbench IDE plug-in is being used, use the wizard to generate a spreadsheet from a template.
3. Use an XSL-compatible spreadsheet editor to modify the XSL.

Report a bug

5.23. LISTS

In Excel, you can create **`lists`** of values. These can be stored in other worksheets to provide valid lists of values for cells.

Report a bug

5.24. REVISION CONTROL

When changes are being made to rules over time, older versions are archived. Some applications in JBoss Rules provide a limited ability to keep a history of changes, but it is recommended to use an alternative means of revision control.

Report a bug

5.25. TABULAR DATA SOURCES

A tabular data source can be used as a source of rule data. It can populate a template to generate many rules. This can allow both for more flexible spreadsheets, but also rules in existing databases for instance (at the cost of developing the template up front to generate the rules).

Report a bug

5.26. RULE TEMPLATE CAPABILITIES

With Rule Templates the data is separated from the rule and there are no restrictions on which part of the rule is data-driven. So while you can do everything you could do in decision tables you can also do the following:

- store your data in a database (or any other format)
- conditionally generate rules based on the values in the data
- use data for any part of your rules (e.g. condition operator, class name, property name)
- run different templates over the same data

Report a bug

5.27. RULE TEMPLATE EXAMPLE

This is what a rule template looks like:

```
template header
age
type
log

package org.drools.examples.templates;

global java.util.List list;

template "employees"

rule "Current employee_@{row.rowNumber}"
when
   Name(location == @{location})
   role(type == "@{type}")
then
   list.add("@{log}");
```

```
18 end
19
20 end template
```

This is what the above example comprises:

- Line 1: All rule templates start with **`template header`**.
- Lines 2-4: Following the header is the list of columns in the order they appear in the data. In this case we are calling the first column **`location`**, the second **`type`** and the third **`log`**.
- Line 5: An empty line signifies the end of the column definitions.
- Lines 6-9: Standard rule header text. This is standard rule DRL and will appear at the top of the generated DRL. Put the package statement and any imports and global and function definitions into this section.
- Line 10: The keyword **`template`** signals the start of a rule template. There can be more than one template in a template file, but each template should have a unique name.
- Lines 11-18: The rule template.
- Line 20: The keywords **`end template`** signify the end of the template.

This example template would generate the following rule:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Current employee_1"
when
  Person(location == Melbourne)
  role(type == "receptionist")
then
  list.add("melbourne admin");
end

rule "Current employee_2"
when
  Person(location == Sydney)
  Cheese(type == "recruiter")
then
  list.add("sydney HR");
end
```

Report a bug

5.28. EXECUTING RULE TEMPLATES

Procedure 5.2. Task

- Run this code to execute your rule template:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
dtableconfiguration.setInputType( DecisionTableInputType.XLS );

KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource(
getSpreadsheetName(),
                                                    getClass() ),
              ResourceType.DTABLE,
              dtableconfiguration );
```

Report a bug

5.29. EXTENDED CHANGESET EXAMPLE

The example below is expanded to load the rules from a http URL location, and an Excel decision table from the classpath:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
             xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
             xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd http://anonsvn.jboss.org/repos/labs/labs/jbossrules/trunk/drools-
api/src/main/resources/change-set-1.0.0.xsd' >
   <add>
       <resource source='http:org/domain/myrules.drl' type='DRL' />
       <resource source='classpath:data/IntegrationExampleTest.xls'
type="DTABLE">
           <decisiontable-conf input-type="XLS" worksheet-name="Tables_2"
/>
       </resource>
   </add>
 </change-set>
```

Report a bug

5.30. CHANGESETS AND DIRECTORIES EXAMPLE

You can specify a directory to put content into. It is expected that all the files are of the specified type, since type is not yet inferred from the file name extensions:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
             xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
             xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd http://anonsvn.jboss.org/repos/labs/labs/jbossrules/trunk/drools-
api/src/main/resources/change-set-1.0.0.xsd' >
   <add>
       <resource source='file://myfolder/' type='DRL' />
   </add>
 </change-set>
```

Report a bug

5.31. KNOWLEDGE AGENT

The *Knowledge Agent* provides automatic loading, caching and re-loading of resources and is configured from a Knowledge Base properties files. The Knowledge Agent can update or rebuild a Knowledge Base as the resources it uses are changed. The strategy for this is determined by the configuration given to the factory, but it is typically pull-based using regular polling.

Report a bug

5.32. KNOWLEDGE AGENT EXAMPLE

This is what the Knowledge Agent looks like:

```
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent"
);
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

Report a bug

5.33. KNOWLEDGEAGENT OBJECTS

A **`KnowledgeAgent`** object will continuously scan all resources using a default polling interval of 60 seconds. When a modification date is updated, it will applied the changes into the cached Knowledge Base using the new resources. The previous **`KnowledgeBase`** reference will still exist and you'll have to call **`getKnowledgeBase()`** to access the newly built **`KnowledgeBase`**. If a directory is specified as part of the change set, the entire contents of that directory will be scanned for changes. The way modifications are applied depends on **`drools.agent.newInstance`** property present in the KnowledgeAgentConfiguration object passed to the agent.

Report a bug

5.34. STARTING POLLING SERVICES

Procedure 5.3. Task

- For polling to occur, the polling and notifier services must be started. Use this code:

  ```
  ResourceFactory.getResourceChangeNotifierService().start();
  ResourceFactory.getResourceChangeScannerService().start();
  ```

Report a bug

5.35. CUSTOM CLASSLOADERS FOR KNOWLEDGEBUILDER

Procedure 5.4. Task

1. Open a KnowledgeBuilderConfiguration and specify a custom classloader.
2. If you need to pass custom configuration to these compilers, sends a KnowledgeBuilderConfiguration object to KnowledgeAgentFactory.newKnowledgeAgent().

Report a bug

5.36. REUSING THE KNOWLEDGEBASE CLASSLOADER

Most of the time, the classloader used in the compilation process of remote resources is the same needed in the agent's kbase so the rules could be executed.

If you want to use this approach, you will need to setup the desired ClassLoader to the agent kbase and use the **`drools.agent.useKBaseClassLoaderForCompiling`** property of KnowledgeAgentConfiguration object.

This approach lets you modify agent's kbuilder classloader in runtime by modifying the classloader the agent's kbase uses. This will serve also when not using incremental change set processing. When the kbase is recreated its configuration is reused, so the classloader is maintained.

Report a bug

5.37. KNOWLEDGEAGENTCONFIGURATION EXAMPLE

The following is an example of the KnowledgeAgentConfiguration property:

```
KnowledgeBaseConfiguration kbaseConfig =
    KnowledgeBaseFactory.newKnowledgeBaseConfiguration(null,
customClassLoader);
KnowledgeBase kbase =
    KnowledgeBaseFactory.newKnowledgeBase(kbaseConfig); //kbase with
custom classloader
KnowledgeAgentConfiguration aconf =
    KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
aconf.setProperty("drools.agent.newInstance", "false"); //incremental
change set processing enabled
aconf.setProperty("drools.agent.useKBaseClassLoaderForCompiling", "true");
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent(
                "test agent", kbase, aconf);
```

Report a bug

5.38. THE NEWINSTANCE PROPERTY

The *newInstance* property assists in processing change sets.

A Knowledge Agent can process change sets in two different ways: recreating the knowledge base every time a new change set is processed or applying the change set in the cached knowledge base without destroying it. This behavior is controlled by the KnowledgeAgentConfiguration object's newInstance property when it is passed to the Agent's constructor.

Report a bug

5.39. USING THE NEWINSTANCE PROPERTY

When newInstance is set to true (the default value), the agent will destroy the cached Knowledge Base it contains and populate a new one containing the change set modifications. When newInstance is set to false, change sets are applied directly to the cached Knowledge Base. The rules that were not modified in the change sets' resources are not replaced in the Knowledge Base, the modified or deleted rules are modified or deleted from the cached Knowledge Base. Functions, Queries and Definition Types are always replaced in the cached Knowledge Base whether they are modified or not.

Report a bug

5.40. NEWINSTANCE EXAMPLE

The following code snippet creates a new Knowledge Agent with its newInstance property set to false:

```
KnowledgeAgentConfiguration aconf =
KnowledgeAgentFactory.newKnowledgeAgentConfiguration();
aconf.setProperty("drools.agent.newInstance", "false");
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent("test
agent", null, aconf);
```

Report a bug

5.41. REMOTE HTTP RESOURCE CACHING

The knowledge agent is able to remotely "pull" resources from a http(s) URL.

Report a bug

5.42. RESTORING RESOURCE CACHING AFTER A RESTART

Procedure 5.5. Task

- To survive a restart when a resource is no longer available remotely (for example, the remote server is being restarted), set a System Property: **`drools.resource.urlcache`**. (Make sure it is set to a directory that has write permissions for the application.) The Knowledge Agent will cache copies of the remote resources in that directory.

  For example, using the java command line **`-Ddrools.resource.urlcache=/users/someone/KnowledgeCache`** - will keep local copies of the resources (rules, packages etc) in that directory, for the agent to use should it be restarted. (When a remote resource becomes available, and is updated, it will automatically update the local cache copy.)

Report a bug

CHAPTER 6. PROCESSING

6.1. AGENDA

The Agenda is a *Rete* feature. During actions on the **`WorkingMemory`**, rules may become fully matched and eligible for execution. A single Working Memory Action can result in multiple eligible rules. When a rule is fully matched an Activation is created, referencing the rule and the matched facts, and placed onto the Agenda. The Agenda controls the execution order of these Activations using a Conflict Resolution strategy.

Report a bug

6.2. AGENDA PROCESSING

The engine cycles repeatedly through two phases:

1. Working Memory Actions. This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls **`fireAllRules()`** the engine switches to the Agenda Evaluation phase.

2. Agenda Evaluation. This attempts to select a rule to fire. If no rule is found it exits, otherwise it fires the found rule, switching the phase back to Working Memory Actions.

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

Report a bug

6.3. DEFAULT CONFLICT RESOLUTION STRATEGIES

Salience (Priority)

A user can specify that a certain rule has a higher priority (by giving it a higher number) than other rules. In that case, the rule with higher salience will be preferred.

LIFO (last in, first out)

LIFO priorities are based on the assigned Working Memory Action counter value, with all rules created during the same action receiving the same value. The execution order of a set of firings with the same priority value is arbitrary.

NOTE

As a general rule, it is a good idea not to count on rules firing in any particular order, and to author the rules without worrying about a "flow". However when a flow is needed a number of possibilities exist, including but not limited to: agenda groups, rule flow groups, activation groups, control/semaphore facts. These are discussed in later sections.

Report a bug

6.4. AGENDAGROUP

Agenda groups are a way to partition rules on the agenda. At any one time, only one group has "focus" which means that activations for rules in that group only will take effect. You can also have rules with "auto focus" which means that the focus is taken for its agenda group when that rule's conditions are true.

Agenda groups are known as "modules" in CLIPS terminology. Agenda groups provide a way to create a "flow" between grouped rules. You can switch the group which has focus either from within the rule engine, or via the API. If your rules have a clear need for multiple "phases" or "sequences" of processing, consider using agenda-groups for this purpose.

Report a bug

6.5. SETFOCUS()

Each time **`setFocus()`** is called it pushes the specified Agenda Group onto a stack. When the focus group is empty it is popped from the stack and the focus group that is now on top evaluates. An Agenda Group can appear in multiple locations on the stack. The default Agenda Group is "MAIN", with all rules which do not specify an Agenda Group being in this group. It is also always the first group on the stack, given focus initially, by default.

Report a bug

6.6. SETFOCUS() EXAMPLE

This is what the setFocus() element looks like:

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

Report a bug

6.7. ACTIVATIONGROUP

An activation group is a set of rules bound together by the same "activation-group" rule attribute. In this group only one rule can fire, and after that rule has fired all the other rules are cancelled from the agenda. The **`clear()`** method can be called at any time, which cancels all of the activations before one has had a chance to fire.

Report a bug

6.8. ACTIVATIONGROUP EXAMPLE

This is what an ActivationGroup looks like:

```
ksession.getAgenda().getActivationGroup( "Group B" ).clear();
```

Report a bug

6.9. RULEFLOWGROUP

A rule flow group is a group of rules associated by the "ruleflow-group" rule attribute. These rules can only fire when the group is activate. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group. Here too, the **`clear()`** method can be called at any time to cancels all activations still remaining on the Agenda.

Report a bug

6.10. RULEFLOWGROUP EXAMPLE

This is what the RuleFlowGroup property looks like:

```
ksession.getAgenda().getRuleFlowGroup( "Group C" ).clear();
```

Report a bug

6.11. THE DIFFERENCE BETWEEN RULES AND METHODS

- Methods are called directly.
- Specific instances are passed.
- One call results in a single execution.
- Rules execute by matching against any data as long it is inserted into the engine.
- Rules can never be called directly.
- Specific instances cannot be passed to a rule.
- Depending on the matches, a rule may fire once or several times, or not at all.

Report a bug

6.12. CROSS PRODUCT EXAMPLE

Below, a rule consisting of an unconstrained fire alarm situation is shown:

```
rule
when
    $room : Room()
    $sprinkler : Sprinkler()
then
    System.out.println( "room:" + $room.getName() +
                        " sprinkler:" + $sprinkler.getRoom().getName() );
end
```

In SQL terms this would be like doing **`select * from Room, Sprinkler`** and every row in the Room table would be joined with every row in the Sprinkler table resulting in the following output:

```
room:office sprinkler:office
room:office sprinkler:kitchen
room:office sprinkler:livingroom
room:office sprinkler:bedroom
room:kitchen sprinkler:office
room:kitchen sprinkler:kitchen
room:kitchen sprinkler:livingroom
room:kitchen sprinkler:bedroom
room:livingroom sprinkler:office
room:livingroom sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:livingroom sprinkler:bedroom
room:bedroom sprinkler:office
room:bedroom sprinkler:kitchen
room:bedroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

These cross products can become huge and can contain spurious data. This can be averted by constraining the cross products, which is done with the variable constraint:

```
rule
when
    $room : Room()
    $sprinkler : Sprinkler( room == $room )
then
    System.out.println( "room:" + $room.getName() +
                        " sprinkler:" + $sprinkler.getRoom().getName() );
end
```

This results in just four rows of data, with the correct Sprinkler for each Room. In SQL (actually HQL) the corresponding query would be **`select * from Room, Sprinkler where Room == Sprinkler.room`**.

```
room:office sprinkler:office
room:kitchen sprinkler:kitchen
room:livingroom sprinkler:livingroom
room:bedroom sprinkler:bedroom
```

Report a bug

6.13. ACTIVATIONS, AGENDA AND CONFLICT SETS EXAMPLE

In this example, a cashflow calculation system is featured. These are the three classes implemented:

```
public class CashFlow {
    private Date   date;
    private double amount;
    private int    type;
    long           accountNo;
    // getter and setter methods here
}

public class Account {
    private long   accountNo;
    private double balance;
    // getter and setter methods here
}

public AccountPeriod {
    private Date start;
    private Date end;
    // getter and setter methods here
}
```

Two rules can be used to determine the debit and credit for that quarter and update the Account balance. The two rules below constrain the cashflows for an account for a given time period. Notice the "&&" which use short cut syntax to avoid repeating the field name twice.

```
rule "increase balance for
credits"
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 "decrease balance for
debits"
when
  ap : AccountPeriod()
  acc : Account( $accountNo :
accountNo )
  CashFlow( type == DEBIT,
            accountNo ==
$accountNo,
            date >= ap.start &&
<= ap.end,
            $amount : amount )
then
  acc.balance -= $amount;
end
```

If the **`AccountPeriod`** is set to the first quarter we constrain the rule "increase balance for credits" to fire on two rows of data and "decrease balance for debits" to act on one row of data.

The data is matched during the insertion stage and only fires after **`fireAllRules()`** is called. Meanwhile, the rule plus its matched data is placed on the Agenda and referred to as an Activation. The Agenda is a table of Activations that are able to fire and have their consequences executed, as soon as fireAllRules() is called. Activations on the Agenda are executed in turn. Notice that the order of execution so far is considered arbitrary.

After all of the above activations are fired, the account has a balance of -25.

If the **`AccountPeriod`** is updated to the second quarter, we have just a single matched row of data, and thus just a single Activation on the Agenda.

The firing of that Activation results in a balance of 25.

Report a bug

6.14. CONFLICT RESOLVER STRATEGY

When there is one or more Activations on the Agenda they are said to be in conflict, and a conflict resolver strategy is used to determine the order of execution. At the simplest level the default strategy uses salience to determine rule priority.

Report a bug

6.15. CONFLICT RESOLVER STRATEGY EXAMPLE

Each rule has a default value of 0, the higher the value the higher the priority. To illustrate this, a rule is added to print the account balance. The goal is for the rule to be executed after all the debits and credits have been applied for all accounts. This is done by assigning a negative salience to this rule so that it fires after all rules with the default salience 0.

```
rule "Print balance for AccountPeriod"
        salience -50
    when
        ap : AccountPeriod()
        acc : Account()
    then
        System.out.println( acc.accountNo + " : " + acc.balance );
end
```

Report a bug

6.16. TRIGGER EXAMPLE

Table 6.1. Trigger Example

Rule View	View Trigger

```
select * from Account acc,
              Cashflow cf,
              AccountPeriod ap
where acc.accountNo ==
cf.accountNo and
      cf.type == CREDIT and
      cf.date >= ap.start and
      cf.date <= ap.end
```
	```
	select * from Account acc,
	              Cashflow cf,
	              AccountPeriod ap
	where acc.accountNo ==
	cf.accountNo and
	      cf.type == DEBIT and
	      cf.date >= ap.start and
	      cf.date <= ap.end
	```

```
trigger : acc.balance +=
cf.amount
```
	```
	trigger : acc.balance -=
	cf.amount
	```

Report a bug

6.17. RULEFLOW-GROUP EXAMPLE

The use of the ruleflow-group attribute in a rule is shown below:

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

Report a bug

6.18. INFERENCE EXAMPLE

In the example below, the IsAdult property is used to infer a person's age.

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

This inferred relation can be used in any rule:

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

Further, de-coupling the knowledge process decreases the chance of data leakage and third party modifications to the information.

Report a bug

6.19. IMPLEMENTING INFERENCE AND TRUTHMAINTENANCE

Procedure 6.1. Task

1. Open a set of rules. In this example, a buss pass issuing system will be used:

```
rule "Issue Child Bus Pass" when
  $p : Person( age < 16 )
then
  insert(new ChildBusPass( $p ) );
end

rule "Issue Adult Bus Pass" when
  $p : Person( age >= 16 )
then
  insert(new AdultBusPass( $p ) );
end
```

2. Insert the fact **`insertLogical`** and add the terms you wish to be inferred.

```
rule "Infer Child" when
  $p : Person( age < 16 )
then
    insertLogical( new IsChild( $p ) )
end
rule "Infer Adult" when
    $p : Person( age >= 16 )
then
    insertLogical( new IsAdult( $p ) )
end
```

   The fact has been logically inserted. This fact is dependent on the truth of the "when" clause. It means that when the rule becomes false the fact is automatically retracted. This works particularly well as the two rules are mutually exclusive. In the above rules, the IsChild fact is inserted if the child is under 16. It is then automatically retracted if the person is over 16 and the IsAdult fact is inserted.

3. Insert the code to issue the passes. These can also be logically inserted as the TMS supports chaining of logical insertions for a cascading set of retracts.

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

rule "Issue Adult Bus Pass" when
    $p : Person( age >= 16 )
    IsAdult( person =$p )
then
    insertLogical(new AdultBusPass( $p ) );
end
```

Now when the person changes from being 15 to 16, not only is the IsChild fact automatically retracted, so is the person's ChildBusPass fact.

4. Insert the 'not' conditional element to handle notifications. (In this situation, a request for the returning of the pass.) When the TMS automatically retracts the ChildBusPass object, this rule triggers and sends a request to the person:

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

Report a bug

CHAPTER 7. THE RULE LANGUAGE

7.1. THE KNOWLEDGEBUILDER

The KnowledgeBuilder is responsible for taking source files, such as a DRL file or an Excel file, and turning them into a Knowledge Package of rule and process definitions which a Knowledge Base can consume. An object of the class **`ResourceType`** indicates the type of resource the builder is being asked to process.

Report a bug

7.2. THE RESOURCEFACTORY

The **`ResourceFactory`** provides capabilities to load resources from a number of sources, such as a java.io.Reader, the classpath, a URL, a java.io.File, or a byte array. Binary files, such as decision tables (Excel's .xls files), should not be passed in with Reader, which is only suitable for text based resources.

Report a bug

7.3. CREATING A NEW KNOWLEDGEBUILDER

Procedure 7.1. Task

1. Open the KnowledgeBuilderFactory.
2. Create a new default configuration.
3. Enter this into the configuration:

   ```
   KnowledgeBuilder kbuilder =
   KnowledgeBuilderFactory.newKnowledgeBuilder();
   ```

   The first parameter is for properties and is optional. If left blank, the default options will be used. The options parameter can be used for things like changing the dialect or registering new accumulator functions.

4. To add a KnowledgeBuilder with a custom ClassLoader, use this code:

   ```
   KnowledgeBuilderConfiguration kbuilderConf =
   KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration(null,
   classLoader );
   KnowledgeBuilder kbuilder =
   KnowledgeBuilderFactory.newKnowledgeBuilder(kbuilderConf);
   ```

Report a bug

7.4. ADDING DRL RESOURCES

Procedure 7.2. Task

1. Resources of any type can be added iteratively. Below, a DRL file is added. The Knowledge Builder can handle multiple namespaces, so you can combine resources regardless of their namespace:

   ```
   kbuilder.add( ResourceFactory.newFileResource(
   "/project/myrules.drl" ),
                 ResourceType.DRL);
   ```

2. Check the compilation results after each resource addition. The KnowledgeBuilder can report compilation results of 3 different severities: ERROR, WARNING and INFO.

   - An **`ERROR`** indicates that the compilation of the resource failed. You should not add more resources or retrieve the Knowledge Packages if there are errors. **`getKnowledgePackages()`** returns an empty list if there are errors.
   - **`WARNING`** and **`INFO`** results can be ignored, but are available for inspection nonetheless.

Report a bug

7.5. KNOWLEDGEBUILDER RESULT INSPECTION METHODS

The KnowledgeBuilder API offers several methods to check and retrieve the build results for a list of severities:

```
    /**
     * Return the knowledge builder results for the listed severities.
     * @param severities
     * @return
     */
    KnowledgeBuilderResults getResults(ResultSeverity... severities);

    /**
     * Checks if the builder generated any results of the listed
severities
     * @param severities
     * @return
     */
    boolean hasResults(ResultSeverity... severities ;
```

The KnowledgeBuilder API also has two helper methods to inspect for errors only: **`hasErrors()`** and **`getErrors()`**:

```
if( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
    return;
}
```

Report a bug

7.6. GETTING THE KNOWLEDGEPACKAGES

When all the resources have been added and there are no errors, the collection of Knowledge Packages can be retrieved. It is a java.util.Collection because there is one KnowledgePackage per package namespace. These Knowledge Packages are serializable and often used as a unit of deployment.

This is what it looks like:

```
Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
```

Report a bug

7.7. EXTENDED KNOWLEDGEBUILDER EXAMPLE

This is what a complete KnowledgeBuilder package looks like:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
if( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
    return;
}

KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules1.drl" ),
              ResourceType.DRL);
kbuilder.add( ResourceFactory.newFileResource( "/project/myrules2.drl" ),
              ResourceType.DRL);

if( kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
    return;
}

Collection<KnowledgePackage> kpkgs = kbuilder.getKnowledgePackages();
```

Report a bug

7.8. USING KNOWLEDGEBUILDER IN BATCH MODE

The KnowledgeBuilder has a batch mode with a fluent interface. It allows you to build multiple DRLs at once as in the following example:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.batch()
        .add( ResourceFactory.newFileResource( "/project/myrules1.drl" ),
ResourceType.DRL )
        .add( ResourceFactory.newFileResource( "/project/myrules2.drl" ),
ResourceType.DRL )
        .add( ResourceFactory.newFileResource( "/project/mytypes1.drl" ),
ResourceType.DRL )
        .build();
```

Report a bug

7.9. DISCARD THE BUILD OF THE LAST ADDED DRL

the KnowledgeBuilder (regardless if you are using the batch mode or not) also allows to discard what has been added with the last DRL(s) build. This can be useful to recover from having added an erroneous DRL to the KnowledgeBuilder, as shown below:

```
kbuilder.add( ResourceFactory.newFileResource( "/project/wrong.drl" ),
ResourceType.DRL );
if ( kbuilder.hasErrors() ) {
    kbuilder.undo();
}
```

Report a bug

7.10. BUILDING USING CONFIGURATION AND THE CHANGESET XML

You can create definitions using a configuration within the ChangeSet XML. The simple XML file supports three elements: add, remove, and modify. Each of these have a sequence of <resource> subelements defining a configuration entity.

Report a bug

7.11. XML SCHEMA FOR CHANGESET XML (NOT NORMATIVE)

This is the schema for a non-normative ChangeSet:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://drools.org/drools-5.0/change-set"
           targetNamespace="http://drools.org/drools-5.0/change-set">

  <xs:element name="change-set" type="ChangeSet"/>

  <xs:complexType name="ChangeSet">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="add"    type="Operation"/>
      <xs:element name="remove" type="Operation"/>
      <xs:element name="modify" type="Operation"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="Operation">
    <xs:sequence>
      <xs:element name="resource" type="Resource"
                  maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Resource">
    <xs:sequence>
```

```
      <!-- To be used with <resource type="DTABLE"...> -->
      <xs:element name="decisiontable-conf" type="DecTabConf"
                  minOccurs="0"/>
    </xs:sequence>
    <!-- java.net.URL, plus "classpath" protocol -->
    <xs:attribute name="source" type="xs:string"/>
    <xs:attribute name="type"   type="ResourceType"/>
  </xs:complexType>

  <xs:complexType name="DecTabConf">
    <xs:attribute name="input-type"     type="DecTabInpType"/>
    <xs:attribute name="worksheet-name" type="xs:string"
                  use="optional"/>
  </xs:complexType>

  <!-- according to org.drools.builder.ResourceType -->
  <xs:simpleType name="ResourceType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="DRL"/>
      <xs:enumeration value="XDRL"/>
      <xs:enumeration value="DSL"/>
      <xs:enumeration value="DSLR"/>
      <xs:enumeration value="DRF"/>
      <xs:enumeration value="DTABLE"/>
      <xs:enumeration value="PKG"/>
      <xs:enumeration value="BRL"/>
      <xs:enumeration value="CHANGE_SET"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- according to org.drools.builder.DecisionTableInputType -->
  <xs:simpleType name="DecTabInpType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="XLS"/>
      <xs:enumeration value="CSV"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Report a bug

CHAPTER 8. CHANGESETS

8.1. CHANGESETS

The *changeset* feature facilitates the building of knowledge bases without using the API. Changesets can include any number of resources. They can also support additional configuration information, which currently is only needed for decision tables.

The file **`changeset.xml`** contains a list of resources for this. It can also point recursively to another changeset XML file.

Report a bug

8.2. CHANGESET EXAMPLE

A resource approach is employed that uses a prefix to indicate the protocol. All the protocols provided by **`java.net.URL`**, such as "file" and "http", are supported, as well as an additional "classpath". Currently the type attribute must always be specified for a resource, as it is not inferred from the file name extension. This is demonstrated in the example below:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
             xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
             xs:schemaLocation='http://drools.org/drools-5.0/change-set
http://anonsvn.jboss.org/repos/labs/labs/jbossrules/trunk/drools-
api/src/main/resources/change-set-1.0.0.xsd' >
   <add>
       <resource source='http:org/domain/myrules.drl' type='DRL' />
   </add>
 </change-set>
```

The above example can be used by changing the resource type to **`CHANGE_SET`**:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClasspathResource( "myChangeSet.xml",
getClass() ),
              ResourceType.CHANGE_SET );
if ( kbuilder.hasErrors() ) {
    System.err.println( builder.getErrors().toString() );
}
```

Report a bug

8.3. CHANGESET XML EXAMPLE

This is an example of a basic ChangeSet XML schema:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
             xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
             xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd http://anonsvn.jboss.org/repos/labs/labs/jbossrules/trunk/drools-
```

```
api/src/main/resources/change-set-1.0.0.xsd' >
   <add>
      <resource source='file:/project/myrules.drl' type='DRL' />
   </add>
</change-set>
```

Report a bug

8.4. CHANGESET PROTOCOLS

The ChangeSet supports all the protocols provided by java.net.URL, such as "file" and "http", as well as an additional "classpath". The type attribute must always be specified for a resource, as it is not inferred from the file name extension.

Use the **`file:`** prefix to signify the protocol for the resource.

NOTE

When using XML schema, the Class Loader will default to the one used by the Knowledge Builder unless the ChangeSet XML is itself loaded by the ClassPath resource, in which case it will use the Class Loader specified for that resource.

Report a bug

8.5. LOADING THE CHANGESET XML

Procedure 8.1. Task

1. Use the API to load your ChangeSet.

2. Use this code to access the ChangeSet XML:

   ```
   kbuilder.add( ResourceFactory.newUrlResource( url ),
   ResourceType.CHANGE_SET );
   ```

Report a bug

8.6. CHANGESET XML WITH RESOURCE CONFIGURATION EXAMPLE

This example shows you how to incorporate resource configuration in your ChangeSet:

```
 <change-set xmlns='http://drools.org/drools-5.0/change-set'
             xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
             xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd http://anonsvn.jboss.org/repos/labs/labs/drools/trunk/drools-
api/src/main/resources/change-set-1.0.0.xsd' >
  <add>
       <resource source='http:org/domain/myrules.drl' type='DRL' />
       <resource source='classpath:data/IntegrationExampleTest.xls'
```

```
type="DTABLE">
           <decisiontable-conf input-type="XLS" worksheet-name="Tables_2"
/>
       </resource>
   </add>
 </change-set>
```

Report a bug

8.7. CHANGESET XML AND DIRECTORIES

The following code allows you to add a directory's contents to the ChangeSet:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
            xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
            xs:schemaLocation='http://drools.org/drools-5.0/change-set.xsd
http://anonsvn.jboss.org/repos/labs/labs/drools/trunk/drools-
api/src/main/resources/change-set-1.0.0.xsd' >
   <add>
      <resource source='file:/projects/myproject/myrules' type='DRL' />
   </add>
</change-set>
```

NOTE

Currently it is expected that all resources in that folder are of the same type. If you use the Knowledge Agent it will provide a continuous scanning for added, modified or removed resources and rebuild the cached Knowledge Base.

Report a bug

CHAPTER 9. BUILDING

9.1. BUILD RESULT SEVERITY

You can change the default severity of a type of build result. This can be useful if, for example, a new rule with a duplicate name of an existing rule is added to a package. (In this case, the default behavior is to replace the old rule with the new rule and report it as an INFO.) In some deployments the user might want to prevent the rule update and report it as an error.

Report a bug

9.2. SETTING THE DEFAULT BUILD RESULT SEVERITY

Procedure 9.1. Task

1. To configure it using system properties or configuration files, insert the following properties:

```
// sets the severity of rule updates
drools.kbuilder.severity.duplicateRule = <INFO|WARNING|ERROR>
// sets the severity of function updates
drools.kbuilder.severity.duplicateFunction = <INFO|WARNING|ERROR>
```

2. To use the API to change the severities, use this code:

```
KnowledgeBuilderConfiguration kbconf = ...

// sets the severity of rule updates to error
kbconf.setOption( KBuilderSeverityOption.get(
"drools.kbuilder.severity.duplicateRule", ResultSeverity.ERROR ) );
// sets the severity of function updates to error
kbconf.setOption( KBuilderSeverityOption.get(
"drools.kbuilder.severity.duplicateFunction", ResultSeverity.ERROR )
);
```

Report a bug

9.3. KNOWLEDGEPACKAGE

A *Knowledge Package* is a collection of Knowledge Definitions, such as rules and processes. It is created by the Knowledge Builder. Knowledge Packages are self-contained and serializable, and they currently form the basic deployment unit.

NOTE

A Knowledge Package instance cannot be reused once it's added to the Knowledge Base. If you need to add it to another Knowledge Base, serialize it and use the "cloned" result.

Report a bug

9.4. CREATING A NEW KNOWLEDGEBASE

Procedure 9.2. Task

1. Use this default configuration to create a new KnowledgeBase:

   ```
   KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
   ```

2. If a custom class loader was used with the **`KnowledgeBuilder`** to resolve types not in the default class loader, then that must also be set on the **`KnowledgeBase`**. The technique for this is the same as with the **`KnowledgeBuilder`** and is shown below:

   ```
   KnowledgeBaseConfiguration kbaseConf =
       KnowledgeBaseFactory.createKnowledgeBaseConfiguration( null, cl
   );
   KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase(
   kbaseConf );
   ```

Report a bug

9.5. IN-PROCESS BUILDING AND DEPLOYMENT

In-process building is the simplest form of deployment. It compiles the knowledge definitions and adds them to the Knowledge Base in the same JVM. This approach requires **`drools-core.jar`** and **`drools-compiler.jar`** to be on the classpath.

Report a bug

9.6. ADD KNOWLEDGEPACKAGES TO A KNOWLEDGEBASE

Procedure 9.3. Task

- To add KnowledgePackages to a KnowledgeBase, use this code:

  ```
  Collection<KnowledgePackage> kpkgs =
  kbuilder.getKnowledgePackages();

  KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
  kbase.addKnowledgePackages( kpkgs );
  ```

  The **`addKnowledgePackages(kpkgs)`** method can be called iteratively to add additional knowledge.

Report a bug

9.7. BUILDING AND DEPLOYMENT IN SEPARATE PROCESSES

Both the **KnowledgeBase** and the **KnowledgePackage** are units of deployment and serializable. This means you can have one machine do any necessary building, requiring **drools-compiler.jar**, and have another machine deploy and execute everything, needing only **drools-core.jar**.

Report a bug

9.8. WRITING THE KNOWLEDGEPACKAGE TO AN OUTPUTSTREAM

This is the code for writing the KnowledgePackage to an OutputStream:

```
 ObjectOutputStream out = new ObjectOutputStream( new FileOutputStream(
fileName ) );
out.writeObject( kpkgs );
out.close();
```

Report a bug

9.9. READING THE KNOWLEDGEPACKAGE FROM AN INPUTSTREAM

Use this code for reading the KnowledgePackage from an InputStream:

```
ObjectInputStream in = new ObjectInputStream( new FileInputStream( fileName
) );
// The input stream might contain an individual
// package or a collection.
@SuppressWarnings( "unchecked" )
Collection<KnowledgePackage> kpkgs =
    ()in.readObject( Collection<KnowledgePackage> );
in.close();

KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kpkgs );
```

Report a bug

9.10. THE JBOSS RULES MANAGEMENT SYSTEM

The JBoss Rules Management system is a repository for KnowledgeBases. It helps to maintain large sets of rules.

Additionally, the system compiles and publishes serialized Knowledge Packages to a URL which is then used to load the packages.

Report a bug

9.11. STATEFULKNOWLEDGESESSIONS AND KNOWLEDGEBASE MODIFICATIONS

The **KnowledgeBase** creates and returns **StatefulKnowledgeSession** objects and can optionally keep references to them.

When **KnowledgeBase** modifications occur, they are applied against the data in the sessions. This reference is a weak reference and it is also optional. It is controlled by a boolean flag.

Report a bug

9.12. NEW KNOWLEDGEAGENTS

This is the code for making a new KnowledgeAgent:

```
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent"
);
```

Report a bug

9.13. WRITING THE KNOWLEDGEPACKAGE TO AN OUTPUTSTREAM

This is how to write the KnowledgePackage to an OutputStream:

```
KnowledgeAgent kagent = KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent"
);
kagent.applyChangeSet( ResourceFactory.newUrlResource( url ) );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

NOTE

Resource scanning is not on by default. It must be started. This also applies to notification. Both can be done via the ResourceFactory.

Report a bug

9.14. STARTING THE SCANNING AND NOTIFICATION SERVICES

This is the code for starting scanning and notification services:

```
ResourceFactory.getResourceChangeNotifierService().start();
ResourceFactory.getResourceChangeScannerService().start();
```

Report a bug

9.15. THE RESOURCECHANGESCANNER

The ResourceChangeScanner is used to scan for services. The default resource scanning period may be changed via the **ResourceChangeScannerService**. A suitably updated

ResourceChangeScannerConfiguration object is passed to the service's **configure()** method, which allows for the service to be reconfigured on demand.

Report a bug

9.16. CHANGING THE SCANNING INTERVALS

This is the code to use to change scanning intervals:

```
ResourceChangeScannerConfiguration sconf =

ResourceFactory.getResourceChangeScannerService().newResourceChangeScanner
Configuration();
// Set the disk scanning interval to 30s, default is 60s.
sconf.setProperty( "drools.resource.scanner.interval", "30" );
ResourceFactory.getResourceChangeScannerService().configure( sconf );
```

Report a bug

9.17. INTERACTIONS BETWEEN KNOWLEDGE AGENTS AND KNOWLEDGE BASES

Knowledge Agents can process both an empty Knowledge Base or a populated one. If a populated Knowledge Base is provided, the Knowledge Agent will run an iterator from Knowledge Base and subscribe to the resources that it finds. While it is possible for the Knowledge Builder to build all resources found in a directory, that information is lost by the Knowledge Builder so that those directories will not be continuously scanned. Only directories specified as part of the **applyChangeSet(Resource)** method are monitored.

One of the advantages of providing **KnowledgeBase** as the starting point is that you can provide it with a **KnowledgeBaseConfiguration**. When resource changes are detected and a new **KnowledgeBase** object is instantiated, it will use the **KnowledgeBaseConfiguration** of the previous **KnowledgeBase** object.

Report a bug

9.18. USING AN EXISTING KNOWLEDGEBASE

This is the code for utilizing an existing KnowledgeBase:

```
KnowledgeBaseConfiguration kbaseConf =
    KnowledgeBaseFactory.createKnowledgeBaseConfiguration( null, cl );
KnowledgeBase kbase KnowledgeBaseFactory.newKnowledgeBase( kbaseConf );
// Populate kbase with resources here.

KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "MyAgent", kbase );
KnowledgeBase kbase = kagent.getKnowledgeBase();
```

In the above example **`getKnowledgeBase()`** will return the same provided kbase instance until resource changes are detected and a new Knowledge Base is built. When the new Knowledge Base is built, it will be done with the **`KnowledgeBaseConfiguration`** that was provided to the previous **`KnowledgeBase`**.

Report a bug

9.19. THE APPLYCHANGESET() METHOD

If a ChangeSet XML is used with the **`applyChangeSet()`** method it will add any directories to the scanning process. When the directory scan detects an additional file, it will be added to the Knowledge Base. Any removed file is removed from the Knowledge Base, and modified files will be removed from the Knowledge Base.

Report a bug

9.20. CHANGESET XML TO ADD DIRECTORY CONTENTS

Use this XML to add the contents of a directory to a ChangeSet:

```
<change-set xmlns='http://drools.org/drools-5.0/change-set'
            xmlns:xs='http://www.w3.org/2001/XMLSchema-instance'
            xs:schemaLocation='http://drools.org/drools-5.0/change-
set.xsd' >
   <add>
      <resource source='file:/projects/myproject/myrules' type='PKG' />
   </add>
</change-set>
```

NOTE

Note that for the resource type PKG, the drools-compiler dependency is not needed. The Knowledge Agent is able to handle those with just drools-core.

Report a bug

9.21. THE KNOWLEDGEAGENTCONFIGURATION PROPERTY

The **`KnowledgeAgentConfiguration`** can be used to modify a Knowledge Agent's default behavior. You can use this to load the resources from a directory while inhibiting the continuous scan for changes of that directory.

Report a bug

9.22. CHANGE THE SCANNING BEHAVIOR

Use this code to change the scanning behavior:

–

```
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();

KnowledgeAgentConfiguration kaconf =
    KnowledgeAgentFactory.newKnowledgeAgentConfiguation();
// Do not scan directories, just files.
kaconf.setProperty( "drools.agent.scanDirectories", "false" );
KnowledgeAgent kagent =
    KnowledgeAgentFactory.newKnowledgeAgent( "test agent", kaconf );
```

Report a bug

CHAPTER 10. SESSIONS

10.1. SESSIONS IN JBOSS RULES

Sessions are created from the **`KnowledgeBase`** into which data can be inserted and from which process instances may be started. Creating the **`KnowledgeBase`** can be resource-intensive, whereas session creation is not. For this reason, it is recommended that KnowledgeBases be cached where possible to allow for repeated session creation.

Report a bug

10.2. CREATE A STATEFULKNOWLEDGESESSION FROM A KNOWLEDGEBASE

This is the code for creating a new StatefulKnowledgeSession from a KnowledgeBase:

```
StatefulKnowledgeSession ksession = kbase.newStatefulKnowledgeSession();
```

Report a bug

10.3. THE WORKINGMEMORYENTRYPOINT METHOD

The **`WorkingMemoryEntryPoint`** provides the methods around inserting, updating and retrieving facts. The term "entry point" is related to the fact that there are multiple partitions in a Working Memory and you can choose which one you are inserting into. Most rule based applications will work with the default entry point alone.

Report a bug

10.4. THE KNOWLEDGERUNTIME INTERFACE

The **`KnowledgeRuntime`** interface provides the main interaction with the engine. It is available in rule consequences and process actions. The **`KnowledgeRuntime`** inherits methods from both the **`WorkingMemory`** and the **`ProcessRuntime`**, thereby providing a unified API to work with processes and rules. When working with rules, three interfaces form the **`KnowledgeRuntime`**: **`WorkingMemoryEntryPoint`**, **`WorkingMemory`** and the **`KnowledgeRuntime`** itself.

Report a bug

10.5. FACT INSERTION

Insertion is the act of telling the **`WorkingMemory`** about a fact. You can do this by using **`ksession.insert(yourObject)`**, for example. When you insert a fact, it is examined for matches against the rules. This means *all* of the work for deciding about firing or not firing a rule is done during insertion. However, no rule is executed until you call **`fireAllRules()`**, which you call after you have finished inserting your facts.

Report a bug

10.6. THE FACTHANDLE TOKEN

When an Object is inserted, it returns a **`FactHandle`**. This **`FactHandle`** is the token used to represent your inserted object within the **`WorkingMemory`**. It is also used for interactions with the **`WorkingMemory`** when you wish to retract or modify an object.

Report a bug

10.7. FACTHANDLE EXAMPLE

```
Job accountant = new Job("accountant");
FactHandle accountantHandle = ksession.insert( accountant );
```

Report a bug

10.8. IDENTITY AND EQUALITY

These are the two assertion nodes used by the Working Memory:

Identity

This means that the Working Memory uses an **`IdentityHashMap`** to store all asserted objects. New instance assertions always result in the return of new **`FactHandle`**, but if an instance is asserted again then it returns the original fact handle (that is, it ignores repeated insertions for the same object).

Equality

This means that the Working Memory uses a **`HashMap`** to store all asserted objects. An object instance assertion will only return a new **`FactHandle`** if the inserted object is not equal (according to its **`equal`** method) to an already existing fact.

NOTE

New instance assertions always result in the return of new **`FactHandle`**, but if an instance is asserted again then it returns the original fact handle (that is, it ignores repeated insertions for the same object).

Report a bug

10.9. RETRACTION

Retraction is the removal of a fact from Working Memory. This means that it will no longer track and match that fact, and any rules that are activated and dependent on that fact will be canceled. It is possible to have rules that depend on the nonexistence of a fact, in which case retracting a fact may cause a rule to activate. Retraction may be done using the **FactHandle** that was returned by the insert call. On the right hand side of a rule the **retract** statement is used, which works with a simple object reference.

Report a bug

10.10. RETRACTION EXAMPLE

```
Job accountant = new Job("accountant");
FactHandle accountantHandle = ksession.insert( accountant );
....
ksession.retract( accountantHandle );
```

Report a bug

10.11. THE UPDATE() METHOD

The Rule Engine must be notified of modified facts so they can be reprocessed. The **update()** method can be used to notify the **WorkingMemory** of changed objects for those objects that are not able to notify the **WorkingMemory** themselves. The **update()** method always takes the modified object as a second parameter, which allows you to specify new instances for immutable objects.

NOTE

On the right hand side of a rule the **modify** statement is recommended, as it makes the changes and notifies the engine in a single statement. Alternatively, after changing a fact object's field values through calls of setter methods you must invoke **update** immediately, event before changing another fact, or you will cause problems with the indexing within the rule engine. The modify statement avoids this problem.

Report a bug

10.12. UPDATE() EXAMPLE

```
Job accountant = new Job("accountant");
FactHandle accountantHandle = workingMemory.insert( accountant );
...
accountant.setSalary( 45000 );
workingMemory.update( accountantHandle, accountant );
```

Report a bug

10.13. QUERIES

Queries are used to retrieve fact sets based on patterns, as they are used in rules. Patterns may make use of optional parameters. Queries can be defined in the Knowledge Base, from where they are called up to return the matching results. While iterating over the result collection, any identifier bound in the query can be used to access the corresponding fact or fact field by calling the **get** method with the binding variable's name as its argument. If the binding refers to a fact object, its FactHandle can be retrieved by calling **getFactHandle**, again with the variable's name as the parameter.

Report a bug

10.14. QUERY EXAMPLE

```
QueryResults results =
    ksession.getQueryResults( "my query", new Object[] { "string" } );
for ( QueryResultsRow row : results ) {
    System.out.println( row.get( "varName" ) );
}
```

Report a bug

10.15. LIVE QUERIES

Invoking queries and processing the results by iterating over the returned set is not a good way to monitor changes over time.

To alleviate this, JBoss Rules provides Live Queries, which have a listener attached instead of returning an iterable result set. These live queries stay open by creating a view and publishing change events for the contents of this view. To activate, start your query with parameters and listen to changes in the resulting view. The **dispose** method terminates the query and discontinues this reactive scenario.

Report a bug

10.16. VIEWCHANGEDEVENTLISTENER IMPLEMENTATION EXAMPLE

```
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 LiveQuery
LiveQuery query = ksession.openLiveQuery( "cars",
                                                   new Object[] { "sedan",
"hatchback" },
                                                   listener );
...
...
query.dispose() // calling dispose to terminate the live query
```

NOTE

For an example of Glazed Lists integration for live queries, visit http://blog.athico.com/2010/07/glazed-lists-examples-for-drools-live.html

Report a bug

10.17. KNOWLEDGERUNTIME

The **`KnowledgeRuntime`** provides further methods that are applicable to both rules and processes, such as setting globals and registering channels. ("Exit point" is an obsolete synonym for "channel".)

Report a bug

CHAPTER 11. OBJECTS AND INTERFACES

11.1. GLOBALS

Globals are named objects that are made visible to the rule engine, but unlike facts, changes in the object backing a global do not trigger reevaluation of rules. Globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or else your changes will not have any effect on the behavior of your rules.

Report a bug

11.2. WORKING WITH GLOBALS

Procedure 11.1. Task

1. To start implementing globals into the Working Memory, declare a global in a rules file and back it up with a Java object:

   ```
   global java.util.List list
   ```

2. With the Knowledge Base now aware of the global identifier and its type, you can call **`ksession.setGlobal()`** with the global's name and an object (for any session) to associate the object with the global:

   ```
   List list = new ArrayList();
   ksession.setGlobal("list", list);
   ```

   IMPORTANT

   Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

3. Set the global before it is used in the evaluation of a rule. Failure to do so results in a **`NullPointerException`**.

Report a bug

11.3. RESOLVING GLOBALS

Globals can be resolved in three ways:

getGlobals()

The Stateless Knowledge Session method **`getGlobals()`** returns a Globals instance which provides access to the session's globals. These are shared for all execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

Delegates

Using a delegate is another way of providing global resolution. Assigning a value to a global (with **`setGlobal(String, Object)`**) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) will be used.

Execution

Execution scoped globals use a **`Command`** to set a global which is then passed to the **`CommandExecutor`**.

Report a bug

11.4. SESSION SCOPED GLOBAL EXAMPLE

This is what a session scoped Global looks like:

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
// Set a global hbnSession, that can be used for DB interactions in the
rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// Execute while being able to resolve the "hbnSession" identifier.
ksession.execute( collection );
```

Report a bug

11.5. STATEFULRULESESSIONS

The **`StatefulRuleSession`** property is inherited by the **`StatefulKnowledgeSession`** and provides the rule-related methods that are relevant from outside of the engine.

Report a bug

11.6. AGENDAFILTER OBJECTS

`AgendaFilter` objects are optional implementations of the filter interface which are used to allow or deny the firing of an activation. What is filtered depends on the implementation.

Report a bug

11.7. USING THE AGENDAFILTER

Procedure 11.2. Task

- To use a filter specify it while calling **`fireAllRules()`**. The following example permits only rules ending in the string **`"Test"`**. All others will be filtered out:

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

Report a bug

11.8. RULE ENGINE PHASES

The engine cycles repeatedly through two phases:

Working Memory Actions

This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls **`fireAllRules()`** the engine switches to the Agenda Evaluation phase.

Agenda Evaluation

This attempts to select a rule to fire. If no rule is found it exits. Otherwise it fires the found rule, switching the phase back to Working Memory Actions.

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

Report a bug

11.9. THE EVENT MODEL

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you, for instance, to separate logging and auditing activities from the main part of your application (and the rules).

Report a bug

11.10. THE KNOWLEGERUNTIMEEVENTMANAGER

The **`KnowlegeRuntimeEventManager`** interface is implemented by the **`KnowledgeRuntime`** which provides two interfaces, **`WorkingMemoryEventManager`** and **`ProcessEventManager`**.

Report a bug

11.11. THE WORKINGMEMORYEVENTMANAGER

The **`WorkingMemoryEventManager`** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

Report a bug

11.12. ADDING AN AGENDAEVENTLISTENER

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print activations after they have fired:

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

Report a bug

11.13. PRINTING WORKING MEMORY EVENTS

This code lets you print all Working Memory events by adding a listener:

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

Report a bug

11.14. KNOWLEGERUNTIMEEVENTS

All emitted events implement the **`KnowlegeRuntimeEvent`** interface which can be used to retrieve the actual **`KnowlegeRuntime`** the event originated from.

Report a bug

11.15. SUPPORTED EVENTS FOR THE KNOWLEDGERUNTIMEEVENT INTERFACE

The events currently supported are:

- ActivationCreatedEvent
- ActivationCancelledEvent
- BeforeActivationFiredEvent
- AfterActivationFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectRetractedEvent

- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

Report a bug

11.16. THE KNOWLEDGERUNTIMELOGGER

The KnowledgeRuntimeLogger uses the comprehensive event system in JBoss Rules to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

Report a bug

11.17. ENABLING A FILELOGGER

To enable a FileLogger to track your files, use this code:

```
KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
"logdir/mylogfile");
...
logger.close();
```

Report a bug

11.18. USING STATELESSKNOWLEDGESESSION IN JBOSS RULES

The **StatelessKnowledgeSession** wraps the **StatefulKnowledgeSession**, instead of extending it. Its main focus is on decision service type scenarios. It avoids the need to call **dispose()**. Stateless sessions do not support iterative insertions and the method call **fireAllRules()** from Java code. The act of calling **execute()** is a single-shot method that will internally instantiate a **StatefulKnowledgeSession**, add all the user data and execute user commands, call **fireAllRules()**, and then call **dispose()**. While the main way to work with this class is via the **BatchExecution** (a subinterface of **Command**) as supported by the **CommandExecutor** interface, two convenience methods are provided for when simple object insertion is all that's required. The **CommandExecutor** and **BatchExecution** are talked about in detail in their own section.

Report a bug

11.19. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH A COLLECTION

This the code for performing a StatelessKnowledgeSession execution with a collection:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileSystemResource( fileName ),
ResourceType.DRL );
if (kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
} else {
    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
    StatelessKnowledgeSession ksession =
kbase.newStatelessKnowledgeSession();
    ksession.execute( collection );
}
```

Report a bug

11.20. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH THE INSERTELEMENTS COMMAND

This is the code for performing a StatelessKnowledgeSession execution with the InsertElements Command:

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

NOTE

To insert the collection and its individual elements, use **`CommandFactory.newInsert(collection)`**.

Report a bug

11.21. THE BATCHEXECUTIONHELPER

Methods of the **`CommandFactory`** create the supported commands, all of which can be marshaled using XStream and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the XML format as well as how to use JBoss Rules Pipeline to automate the marshaling of **`BatchExecution`** and **`ExecutionResults`**.

Report a bug

11.22. THE COMMANDEXECUTOR INTERFACE

The **`CommandExecutor`** interface allows users to export data using "out" parameters. This means that inserted facts, globals and query results can all be returned using this interface.

Report a bug

11.23. OUT IDENTIFIERS

This is an example of what out identifiers look like:

```
// 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" );
```

Report a bug

CHAPTER 12. MODES AND METHODS

12.1. SEQUENTIAL MODE

Using *sequential mode* in JBoss Rules allows you to utilize the engine in a more simplified way. It allows for rules to be used without needing to be re-evaluated if a user is working with a stateless session and no more data can be asserted or modified after the initial data set.

Report a bug

12.2. SEQUENTIAL MODE OPTIONS

These are some of the options you can choose to utilize when using the sequential mode:

1. Order the Rules by salience and position in the ruleset (by setting a sequence attribute on the rule terminal node).
2. Create an array (one element for each possible rule activation). Element position indicates firing order.
3. Turn off all node memories, except the right-input Object memory.
4. Disconnect the Left Input Adapter Node propagation and let the Object and the Node be referenced in a Command object. This is added to a list in the Working Memory for later execution.
5. Assert all objects. When all assertions are finished and the right-input node memories are populated, you can check the Command list and execute each in turn.
6. All resulting Activations should be placed in the array, based upon the determined sequence number of the Rule. Record the first and last populated elements, to reduce the iteration range.
7. Iterate the array of Activations, executing populated element in turn.
8. If there is a maximum number of allowed rule executions, exit the network evaluations early to fire all the rules in the array.

Report a bug

12.3. ACTIVATING SEQUENTIAL MODE

Procedure 12.1. Task

1. Start a stateless session.
2. The sequential mode will be turned off by default. To turn it on, call **`RuleBaseConfiguration.setSequential(true)`**. Alternatively, set the rulebase configuration property **`drools.sequential`** to true.
3. To allow sequential mode to fall back to a dynamic agenda, call **`setSequentialAgenda`** with **`SequentialAgenda.DYNAMIC`**.

4. Optionally, set the **`JBossRules.sequential.agenda`** property to **`sequential`** or **`dynamic`**.

Report a bug

12.4. THE COMMANDFACTORY

The **`CommandFactory`** object allows for commands to be executed on stateless sessions. Upon its conclusion, the factory will execute **`fireAllRules()`** before disposing the session.

Report a bug

12.5. SUPPORTED COMMANDFACTORY OPTIONS

All of these options are compatible with the CommandFactory:

- FireAllRules
- GetGlobal
- SetGlobal
- InsertObject
- InsertElements
- Query
- StartProcess
- BatchExecution

Report a bug

12.6. THE INSERT COMMAND

`InsertObject` will insert a single object with an optional "out" identifier. **`InsertElements`** will iterate an Iterable, inserting each of the elements. This allows a Stateless Knowledge Session to process or execute queries in any order.

Report a bug

12.7. INSERT COMMAND EXAMPLE

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
ExecutionResults bresults =
  ksession.execute( CommandFactory.newInsert( new Car( "sedan" ),
"sedan_id" ) );
Sedan sedan = bresults.getValue( "sedan_id" );
```

Report a bug

12.8. THE EXECUTE METHOD

The execute method is used to execute commands one at a time. It always returns an **`ExecutionResults`** instance, which allows access to any command results if they specify an out identifier such as **`stilton_id`**.

Report a bug

12.9. EXECUTE METHOD EXAMPLE

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();
Command cmd = CommandFactory.newInsertElements( Arrays.asList( Object[] {
                  new Car( "sedan" ),
                  new Car( "hatchback" ),
                  new Car( "convertible" ),
              });
ExecutionResults bresults = ksession.execute( cmd );
```

Report a bug

12.10. THE BATCHEXECUTION COMMAND

The **`BatchExecution`** command allows you to execute multiple commands at once. It represents a composite command that is created from a list of commands. Execute will iterate over the list and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query, all in a single **`execute(...)`** call.

Report a bug

12.11. THE FIREALLRULES COMMAND

The **`FireAllRules`** command disables the automatic execution of rules at the end. It is a type of manual override function.

Report a bug

12.12. OUT IDENTIFIERS

Commands support out identifiers. Any command that has an out identifier set on it will add its results to the returned ExecutionResults instance.

Report a bug

12.13. OUT IDENTIFIER EXAMPLE

This example will use the BatchExecution command to show how out identifiers work:

```
StatelessKnowledgeSession ksession = kbase.newStatelessKnowledgeSession();

List cmds = new ArrayList();
cmds.add( CommandFactory.newInsertObject( new Car( "red", 1), "red") );
cmds.add( CommandFactory.newStartProcess( "process cars" ) );
cmds.add( CommandFactory.newQuery( "cars" ) );
ExecutionResults bresults = ksession.execute(
CommandFactory.newBatchExecution( cmds ) );
Car red = ( Car ) bresults.getValue( "red" );
QueryResults qresults = ( QueryResults ) bresults.getValue( "cars" );
```

In the above example multiple commands are executed, two of which populate the **`ExecutionResults`**. The query command defaults to use the same identifier as the query name, but it can also be mapped to a different identifier.

Report a bug

12.14. EXECUTION XML EXAMPLES

A custom XStream marshaler can be used with the JBoss Rules Pipeline to achieve XML scripting, which is perfect for services. Here are two examples of this:

BatchExecution XML:

```
<batch-execution>
   <insert out-identifier='outRed'>
      <org.drools.Car>
         <type>red</type>
         <price>25000</price>
         <oldPrice>0</oldPrice>
      </org.drools.Car>
   </insert>
</batch-execution>
```

ExecutionResults XML:

```
<execution-results>
   <result identifier='outBlue'>
      <org.drools.Car>
         <type>Blue</type>
         <oldPrice>25</oldPrice>
         <price>30000</price>
      </org.drools.Car>
   </result>
</execution-results>
```

Report a bug

12.15. EXECUTION MARSHALLING EXAMPLES

This is an example of BatchExecution marshalled to XML

```
<batch-execution>
  <insert out-identifier="sedan">
    <org.drools.Car>
      <type>sedan</type>
      <price>1</price>
      <oldPrice>0</oldPrice>
    </org.drools.Car>
  </insert>
  <query out-identifier='cars2' name='carsWithParams'>
    <string>hatchback</string>
    <string>sedan</string>
  </query>
</batch-execution>
```

The **`CommandExecutor`** returns an **`ExecutionResults`**, and this is handled by the pipeline code snippet as well. A similar output for the <batch-execution> XML sample above would be:

```
<execution-results>
  <result identifier="sedan">
    <org.drools.Car>
      <type>sedan</type>
      <price>2</price>
    </org.drools.Car>
  </result>
  <result identifier='cars2'>
    <query-results>
      <identifiers>
        <identifier>car</identifier>
      </identifiers>
      <row>
        <org.drools.Car>
          <type>hatchback</type>
          <price>2</price>
          <oldPrice>0</oldPrice>
        </org.drools.Car>
      </row>
      <row>
        <org.drools.Car>
          <type>hatchback</type>
          <price>1</price>
          <oldPrice>0</oldPrice>
        </org.drools.Car>
      </row>
    </query-results>
  </result>
</execution-results>
```

Report a bug

12.16. BATCH-EXECUTION AND COMMAND EXAMPLES

1. There is currently no XML schema to support schema validation. This is the basic format. The root element is <batch-execution> and it can contain zero or more commands elements:

   ```
   <batch-execution>
   ...
   </batch-execution>
   ```

2. The insert element features an "out-identifier" attribute so the inserted object will be returned as part of the result payload:

   ```
   <batch-execution>
      <insert out-identifier='userVar'>
         ...
      </insert>
   </batch-execution>
   ```

3. It's also possible to insert a collection of objects using the <insert-elements> element. This command does not support an out-identifier. The **`org.domain.UserClass`** is just an illustrative user object that XStream would serialize:

   ```
   <batch-execution>
      <insert-elements>
         <org.domain.UserClass>
            ...
         </org.domain.UserClass>
         <org.domain.UserClass>
            ...
         </org.domain.UserClass>
         <org.domain.UserClass>
            ...
         </org.domain.UserClass>
      </insert-elements>
   </batch-execution>
   ```

4. The **`<set-global>`** element sets a global for the session:

   ```
   <batch-execution>
      <set-global identifier='userVar'>
         <org.domain.UserClass>
            ...
         </org.domain.UserClass>
      </set-global>
   </batch-execution>
   ```

5. **`<set-global>`** also supports two other optional attributes: **`out`** and **`out-identifier`**. A true value for the boolean **`out`** will add the global to the **`<batch-execution-results>`** payload, using the name from the **`identifier`** attribute. **`out-identifier`** works like **`out`** but additionally allows you to override the identifier used in the **`<batch-execution-results>`** payload:

   ```
   <batch-execution>
   ```

```
   <set-global identifier='userVar1' out='true'>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </set-global>
   <set-global identifier='userVar2' out-
identifier='alternativeUserVar2'>
      <org.domain.UserClass>
         ...
      </org.domain.UserClass>
   </set-global>
</batch-execution>
```

6. There is a **`<get-global>`** element without contents. It only has an **`out-identifier`** attribute. There is no need for an **`out`** attribute because retrieving the value is the sole purpose of a **`<get-global>`** element:

```
<batch-execution>
   <get-global identifier='userVar1' />
   <get-global identifier='userVar2' out-
identifier='alternativeUserVar2'/>
</batch-execution>
```

7. The query command supports both parameter and parameterless queries. The **`name`** attribute is the name of the query to be called, and the **`out-identifier`** is the identifier to be used for the query results in the **`<execution-results>`** payload:

```
<batch-execution>
   <query out-identifier='cars' name='cars'/>
   <query out-identifier='cars2' name='carsWithParams'>
      <string>red</string>
      <string>blue</string>
   </query>
</batch-execution>
```

8. The **`<start-process>`** command accepts optional parameters:

```
<batch-execution>
   <startProcess processId='org.drools.actions'>
      <parameter identifier='person'>
         <org.drools.TestVariable>
            <name>John Doe</name>
         </org.drools.TestVariable>
      </parameter>
   </startProcess>
</batch-execution
```

9. The signal event command allows you to identify processes:

```
<signal-event process-instance-id='1' event-type='MyEvent'>
   <string>MyValue</string>
</signal-event>
```

10. The complete work item command notifies users when a process is completed:

```
<complete-work-item id='" + workItem.getId() + "' >
   <result identifier='Result'>
      <string>SomeOtherString</string>
   </result>
</complete-work-item>
```

11. The abort work item command lets you cancel a process while it is running:

```
<abort-work-item id='21' />
```

Report a bug

12.17. THE MARSHALLERFACTORY

The **`MarshallerFactory`** is used to marshal and unmarshal Stateful Knowledge Sessions.

Report a bug

12.18. MARSHALLER EXAMPLE

This is what a marsheller looks like in practice:

```
// ksession is the StatefulKnowledgeSession
// kbase is the KnowledgeBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller = MarshallerFactory.newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

Report a bug

12.19. MARSHALLING OPTIONS

Table 12.1. ** table title **

Option	Description
ObjectMarshallingStrategy	This interface provides implementations for marshalling and allows for greater flexibility.
SerializeMarshallingStrategy	This is the default strategy for calling the **`Serializable`** or **`Externalizable`** methods on a user instance.

Option	Description
IdentityMarshallingStrategy	This strategy creates an integer id for each user object and stores them in a Map, while the id is written to the stream. When unmarshalling it accesses the **`IdentityMarshallingStrategy`** map to retrieve the instance. This means that if you use the **`IdentityMarshallingStrategy`**, it is stateful for the life of the Marshaller instance and will create ids and keep references to all objects that it attempts to marshal.

Report a bug

12.20. IDENTITYMARSHALLINGSTRATEGY EXAMPLE

This is the code for using the IdentityMarshallingStrategy:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectMarshallingStrategy oms =
MarshallerFactory.newIdentityMarshallingStrategy()
Marshaller marshaller =
  MarshallerFactory.newMarshaller( kbase, new ObjectMarshallingStrategy[]{
oms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Report a bug

12.21. THE OBJECTMARSHALLINGSTRATEGYACCEPTOR

The **`ObjectMarshallingStrategyAcceptor`** is the interface that each Object Marshalling Strategy contains. The Marshaller has a chain of strategies. When it attempts to read or write a user object, it uses the ObjectMarshallingStrategyAcceptor to determine if they are to be used for marshalling the user object.

Report a bug

12.22. THE CLASSFILTERACCEPTOR IMPLEMENTATION

The **`ClassFilterAcceptor`** implementation allows strings and wild cards to be used to match class names. The default is "*.*".

Report a bug

12.23. IDENTITYMARSHALLINGSTRATEGY WITH ACCEPTOR EXAMPLE

This is an example of using the IdentityMarshallingStrategy with the acceptor. Note that the acceptance checking order is in the natural order of the supplied array:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
ObjectMarshallingStrategyAcceptor identityAcceptor =
  MarshallerFactory.newClassFilterAcceptor( new String[] {
"org.domain.pkg1.*" } );
ObjectMarshallingStrategy identityStrategy =
  MarshallerFactory.newIdentityMarshallingStrategy( identityAcceptor );
ObjectMarshallingStrategy sms =
MarshallerFactory.newSerializeMarshallingStrategy();
Marshaller marshaller =
  MarshallerFactory.newMarshaller( kbase,
                                   new ObjectMarshallingStrategy[]{
identityStrategy, sms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Report a bug

12.24. PERSISTENCE AND TRANSACTIONS IN JBOSS RULES

Long-term out of the box persistence with Java Persistence API (JPA) is possible with JBoss Rules. You will need to have some implementation of the Java Transaction API (JTA) installed. For development purposes you can use the Bitronix Transaction Manager as it's simple to set up and works embedded. For production use, JBoss Transactions is recommended.

Report a bug

12.25. TRANSACTION EXAMPLE

This is what performing a transaction looks like:

```
Environment env = KnowledgeBaseFactory.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
         Persistence.createEntityManagerFactory( "emf-name" ) );
env.set( EnvironmentName.TRANSACTION_MANAGER,
         TransactionManagerServices.getTransactionManager() );

// KnowledgeSessionConfiguration may be null, and a default will be used
StatefulKnowledgeSession ksession =
  JPAKnowledgeService.newStatefulKnowledgeSession( kbase, null, env );
int sessionId = ksession.getId();

UserTransaction ut =
  (UserTransaction) new InitialContext().lookup(
"java:comp/UserTransaction" );
ut.begin();
```

```
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```

Report a bug

12.26. USING A JPA

Procedure 12.2. Task

1. Make sure the environment is set with both the **`EntityManagerFactory`** and the **`TransactionManager`**.
2. Launch the JPA from your GUI or command line.
3. Use the id to load a previously persisted Stateful Knowledge Session. If rollback occurs the ksession state is also rolled back, you can continue to use it after a rollback.

Report a bug

12.27. LOADING A STATEFULKNOWLEDGESESSION WITH JPA

This is the code for loading a StatefulKnowledgeSession implementing the JPA:

```
StatefulKnowledgeSession ksession =
  JPAKnowledgeService.loadStatefulKnowledgeSession( sessionId, kbase,
null, env );
```

Report a bug

12.28. CONFIGURING JPA

To enable persistence several classes must be added to your persistence.xml, as in the example below:

```
<persistence-unit name="org.drools.persistence.jpa" transaction-
type="JTA">
   <provider>org.hibernate.ejb.HibernatePersistence</provider>
   <jta-data-source>jdbc/BitronixJTADataSource</jta-data-source>
   <class>org.drools.persistence.session.SessionInfo</class>

<class>org.drools.persistence.processinstance.ProcessInstanceInfo</class>

<class>org.drools.persistence.processinstance.ProcessInstanceEventInfo</cl
ass>
   <class>org.drools.persistence.processinstance.WorkItemInfo</class>
   <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
```

```
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.transaction.manager_lookup_class"

value="org.hibernate.transaction.BTMTransactionManagerLookup" />
   </properties>
</persistence-unit>
```

Report a bug

12.29. CONFIGURING JTA DATASOURCE

This is the code for configuring the JTA DataSource:

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADataSource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions( true );
ds.getDriverProperties().put( "user", "sa" );
ds.getDriverProperties().put( "password", "sasa" );
ds.getDriverProperties().put( "URL", "jdbc:h2:mem:mydb" );
ds.init();
```

Report a bug

12.30. JNDI PROPERTIES

Bitronix also provides a simple embedded JNDI service, ideal for testing. To use it add a jndi.properties file to your META-INF and add the following line to it:

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFactory
```

Report a bug

12.31. KNOWLEDGEBASE NAMESPACES

This is a list of namespaces you can attach to the KnowledgeBase for building purposes:

- deftemplate
- defrule
- deffunction
- and/or/not/exists/test conditional elements
- Literal, Variable, Return Value and Predicate field constraints

Report a bug

CHAPTER 13. USING SPREADSHEET DECISION TABLES

13.1. HARD KEYWORDS

Hard keywords are words which you cannot use when naming your domain objects, properties, methods, functions and other elements that are used in the rule text. These are words such as **`true`**, **`false`** and other words which could easily be mistaken for a command.

Report a bug

13.2. SOFT KEYWORDS

Soft keywords can be used for naming domain objects, properties, methods, functions and other elements. The rules engine recognizes their context and processes them accordingly.

Report a bug

13.3. LIST OF SOFT KEYWORDS

- **`lock-on-active`**
- **`date-effective`**
- **`date-expires`**
- **`no-loop`**
- **`auto-focus`**
- **`activation-group`**
- **`agenda-group`**
- **`ruleflow-group`**
- **`entry-point`**
- **`duration`**
- **`package`**
- **`import`**
- **`dialect`**
- **`salience`**
- **`enabled`**
- **`attributes`**

- **`rule`**
- **`extend`**
- when
- then
- **`template`**
- **`query`**
- **`declare`**
- **`function`**
- **`global`**
- **`eval`**
- **`not`**
- **`in`**
- **`or`**
- **`and`**
- **`exists`**
- **`forall`**
- accumulate
- collect
- from
- **`action`**
- **`reverse`**
- **`result`**
- **`end`**
- over
- **`init`**

Report a bug

13.4. COMMENTS

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks (like a rule's RHS).

Report a bug

13.5. SINGLE LINE COMMENT EXAMPLE

This is what a single line comment looks like. To create single line comments, you can use '//'. The parser will ignore anything in the line after the comment symbol:

```
rule "Testing Comments"
when
    // this is a single line comment
    eval( true ) // this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
end
```

Report a bug

13.6. MULTI-LINE COMMENT EXAMPLE

This is what a multi-line comment looks like. This configuration comments out blocks of text, both in and outside semantic code blocks:

```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

Report a bug

13.7. ERROR MESSAGES

101: No viable alternative

Indicates when the parser came to a decision point but couldn't identify an alternative.

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

This message means the parser has encountered the token **`WHEN`** (a hard keyword) which is in the wrong place, since the rule name is missing.

[ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern [name]

Indicates an open quote, apostrophe or parentheses.

102: Mismatched input

Indicates that the parser was looking for a particular symbol that it didn't end at the current input position.

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern [name]

This error is the result of an incomplete rule statement. Usually when you get a 0:-1 position, it means that parser reached the end of source.

103: Failed predicate

A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords.

104: Trailing semi-colon not allowed

This error is associated with the **`eval`** clause, where its expression may not be terminated with a semicolon.

105: Early Exit

The recognizer came to a subrule in the grammar that must match an alternative at least once, but the subrule did not match anything.

Report a bug

13.8. PACKAGE

A *package* is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other, such as HR rules. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). It is not possible to merge into the same package resources declared under different names. A single Rulebase may, however, contain multiple packages built on it. A common structure is to have all the rules for a package in the same file as the package declaration (so that is it entirely self-contained).

Report a bug

13.9. IMPORT STATEMENTS

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. JBoss Rules automatically imports classes from the Java package of the same name, and also from the package **`java.lang`**.

Report a bug

13.10. USING GLOBALS

In order to use globals you must:

1. Declare the global variable in the rules file and use it in rules. Example:

   ```
   global java.util.List myGlobalList;

   rule "Using a global"
   when
       eval( true )
   then
       myGlobalList.add( "Hello World" );
   end
   ```

2. Set the global value on the working memory. It is best practice to set all global values before asserting any fact to the working memory. Example:

   ```
   List list = new ArrayList();
   WorkingMemory wm = rulebase.newStatefulSession();
   wm.setGlobal( "myGlobalList", list );
   ```

Report a bug

13.11. THE FROM ELEMENT

The *from* element allows you to pass a Hibernate session as a global. It also lets you pull data from a named Hibernate query.

Report a bug

13.12. USING GLOBALS WITH AN E-MAIL SERVICE

Procedure 13.1. Task

1. Open the integration code that is calling the rule engine.
2. Obtain your emailService object and then set it in the working memory.
3. In the DRL, declare that you have a global of type emailService and give it the name "email".
4. In your rule consequences, you can use things like email.sendSMS(number, message).

WARNING

Globals are not designed to share data between rules and they should never be used for that purpose. 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.

IMPORTANT

Do not set or change a global value from inside the rules. We recommend to you always set the value from your application using the working memory interface.

Report a bug

CHAPTER 14. FUNCTIONS

14.1. FUNCTIONS

Functions are a way to put semantic code in a rule source file, as opposed to in normal Java classes. The main advantage of using functions in a rule is that you can keep the logic all in one place. You can change the functions as needed.

Functions are most useful for invoking actions on the consequence (**`then`**) part of a rule, especially if that particular action is used repeatedly.

Report a bug

14.2. FUNCTION DECLARATION EXAMPLE

A typical function declaration looks like this:

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

NOTE

Note that the **`function`** keyword is used, even though it's not technically part of Java. Parameters to the function are defined as for a method. You don't have to have parameters if they are not needed. The return type is defined just like in a regular method.

Report a bug

14.3. FUNCTION DECLARATION WITH STATIC METHOD EXAMPLE

This example of a function declaration shows the static method in a helper class (**`Foo.hello()`**. JBoss Rules supports the use of function imports, so the following code is all you would need to enter the following:

```
import function my.package.Foo.hello
```

Report a bug

14.4. CALLING A FUNCTION DECLARATION EXAMPLE

Irrespective of the way the function is defined or imported, you use a function by calling it by its name, in the consequence or inside a semantic code block. This is shown below:

```
rule "using a static function"
when
    eval( true )
```

```
then
    System.out.println( hello( "Bob" ) );
end
```

Report a bug

14.5. TYPE DECLARATIONS

Type declarations have two main goals in the rules engine: to allow the declaration of new types, and to allow the declaration of metadata for types.

Report a bug

14.6. TYPE DECLARATION ROLES

Table 14.1. ** table title **

Role	Description
Declaring new types	JBoss Rules out of the box with plain Java objects as facts. However, should a user wish to define the model directly to the rules engine, they can do so by declaring a new type. This can also be used when there is a domain model already built, but the user wants to complement this model with additional entities that are used mainly during the reasoning process.
Declaring metadata	Facts may have meta information associated to them. Examples of meta information include any kind of data that is not represented by the fact attributes and is consistent among all instances of that fact type. This meta information may be queried at runtime by the engine and used in the reasoning process.

Report a bug

14.7. DECLARING NEW TYPES

To declare a new type, the keyword **`declare`** is used, followed by the list of fields and the keyword **end**. A new fact must have a list of fields, otherwise the engine will look for an existing fact class in the classpath and raise an error if not found.

Report a bug

14.8. DECLARING A NEW FACT TYPE EXAMPLE

In this example, a new fact type called **`Address`** is used. This fact type will have three attributes: **`number`**, **`streetName`** and **`city`**. Each attribute has a type that can be any valid Java type, including any other class created by the user or other fact types previously declared:

```
declare Address
   number : int
   streetName : String
   city : String
end
```

Report a bug

14.9. DECLARING A NEW FACT TYPE ADDITIONAL EXAMPLE

This fact type declaration uses a **`Person`** example. **`dateOfBirth`** is of the type **`java.util.Date`** (from the Java API) and **`address`** is of the fact type Address.

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

Report a bug

14.10. USING IMPORT EXAMPLE

This example illustrates how to use the **`import`** feature to avoid he need to use fully qualified class names:

```
import java.util.Date

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

Report a bug

14.11. GENERATED JAVA CLASSES

When you declare a new fact type, JBoss Rules generate bytecode that implements a Java class representing the fact type. The generated Java class will be a one-to-one Java Bean mapping of the type definition.

Report a bug

14.12. GENERATED JAVA CLASS EXAMPLE

This is an example of a generated Java class using the **`Person`** fact type:

```
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/hashCode
    // toString
}
```

Report a bug

14.13. USING THE DECLARED TYPES IN RULES EXAMPLE

Since the generated class is a simple Java class, it can be used transparently in the rules like any other fact:

```
rule "Using a declared Type"
when
    $p : Person( name == "Bob" )
then
    // Insert Mark, who is Bob's manager.
    Person mark = new Person();
    mark.setName("Mark");
    insert( mark );
end
```

Report a bug

14.14. DECLARING METADATA

Metadata may be assigned to several different constructions in JBoss Rules: fact types, fact attributes and rules. JBoss Rules uses the at sign ('@') to introduce metadata and it always uses the form:

```
@metadata_key( metadata_value )
```

The parenthesized *metadata_value* is optional.

Report a bug

14.15. WORKING WITH METADATA ATTRIBUTES

JBoss Rules allows the declaration of any arbitrary metadata attribute, but some will have special meaning to the engine, while others are simply available for querying at runtime. JBoss Rules allows the declaration of metadata both for fact types and for fact attributes. Any metadata that is declared before the attributes of a fact type are assigned to the fact type, while metadata declared after an attribute are assigned to that particular attribute.

Report a bug

14.16. DECLARING A METADATA ATTRIBUTE WITH FACT TYPES EXAMPLE

This is an example of declaring metadata attributes for fact types and attributes. There are two metadata items declared for the fact type (**`@author`** and **`@dateOfCreation`**) and two more defined for the name attribute (**`@key`** and **`@maxLength`**). The **`@key`** metadata has no required value, and so the parentheses and the value were omitted:

```
import java.util.Date

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

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

Report a bug

14.17. THE @POSITION ATTRIBUTE

The **`@position`** attribute can be used to declare the position of a field, overriding the default declared order. This is used for positional constraints in patterns.

Report a bug

14.18. @POSITION EXAMPLE

This is what the @position attribute looks like in use:

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

–

Report a bug

14.19. PREDEFINED CLASS LEVEL ANNOTATIONS

Table 14.2. Predefined Class Level Annotations

Annotation	Description
@role(<fact | event>)	This attribute can be used to assign roles to facts and events.
@typesafe(<boolean>)	By default, all type declarations are compiled with type safety enabled. `@typesafe( false )` provides a means to override this behavior by permitting a fall-back, 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.
@timestamp(<attribute name>)	Creates a timestamp.
@duration(<attribute name>)	Sets a duration for the implementation of an attribute.
@expires(<time interval>)	Allows you to define when the attribute should expire.
@propertyChangeSupport	Facts that implement support for property changes as defined in the Javabean spec can now be annotated so that the engine register itself to listen for changes on fact properties. .
@propertyReactive	Makes the type property reactive.

Report a bug

14.20. @KEY ATTRIBUTE FUNCTIONS

Declaring an attribute as a key attribute has 2 major effects on generated types:

1. The attribute will be used as a key identifier for the type, and as so, the generated class will implement the equals() and hashCode() methods taking the attribute into account when comparing instances of this type.

2. JBoss Rules will generate a constructor using all the key attributes as parameters.

Report a bug

14.21. @KEY DECLARATION EXAMPLE

This is an example of @key declarations for a type. JBoss Rules will generate equals() and hashCode() methods that will check the firstName and lastName attributes to determine if two instances of Person are equal to each other. It will not check the age attribute. It will also generate a constructor taking firstName and lastName as parameters:

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

Report a bug

14.22. CREATING AN INSTANCE WITH THE KEY INSTRUCTOR EXAMPLE

This is what creating an instance using the key constructor looks like:

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

Report a bug

14.23. POSITIONAL ARGUMENTS

Patterns support positional arguments on type declarations and are defined by the **`@position`** attribute.

Positional arguments are when you don't need to specify the field name, as the position maps to a known named field. (That is, Person(name == "mark") can be rewritten as Person("mark";).) The semicolon ';' is important so that the engine knows 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 used in a positional that have not yet been bound will be bound to the field that maps to that position.

Report a bug

14.24. POSITIONAL ARGUMENT EXAMPLE

Observe the example below:

```
declare Cheese
    name : String
    shop : String
    price : int
end
```

The default order is the declared order, but this can be overridden using @position

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

Report a bug

14.25. THE @POSITION ANNOTATION

The @Position annotation 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 of methods.

Report a bug

14.26. EXAMPLE PATTERNS

These example patterns have two constraints and a binding. The semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in positional arguments, but not variables:

```
Cheese( "stilton", "Cheese Shop", p; )
Cheese( "stilton", "Cheese Shop"; p : price )
Cheese( "stilton"; shop == "Cheese Shop", p : price )
Cheese( name == "stilton"; shop == "Cheese Shop", p : price )
```

Report a bug

CHAPTER 15. ADDITIONAL DECLARATIONS

15.1. DECLARING METADATA FOR EXISTING TYPES

JBoss Rules allows the declaration of metadata attributes for existing types in the same way as when declaring metadata attributes for new fact types. The only difference is that there are no fields in that declaration.

Report a bug

15.2. DECLARING METADATA FOR EXISTING TYPES EXAMPLE

This example shows how to declare metadata for an existing type:

```
import org.drools.examples.Person

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

Report a bug

15.3. DECLARING METADATA USING A FULLY QUALIFIED CLASS NAME EXAMPLE

This example shows how you can declare metadata using the fully qualified class name instead of using the import annotation:

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

Report a bug

15.4. PARAMETRIZED CONSTRUCTORS FOR DECLARED TYPES EXAMPLE

For a declared type like the following:

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

The compiler will implicitly generate 3 constructors: one without parameters, one with the @key fields and one with all fields.

```
Person() // parameterless constructor
Person( String firstName, String lastName )
Person( String firstName, String lastName, int age )
```

Report a bug

15.5. NON-TYPESAFE CLASSES

The @typesafe(<boolean>) annotation has been added to type declarations. By default all type declarations are compiled with type safety enabled. @typesafe(false) provides a means to override this behaviour by permitting a fall-back, 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.

Report a bug

15.6. ACCESSING DECLARED TYPES FROM THE APPLICATION CODE

Sometimes applications need to access and handle facts from the declared types. In such cases, JBoss Rules provides a simplified API for the most common fact handling the application wishes to do. A declared fact will belong to the package where it was declared.

Report a bug

15.7. DECLARING A TYPE

This illustrates the process of declaring a type:

```
package org.drools.examples

import java.util.Date

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

Report a bug

15.8. HANDLING DECLARED FACT TYPES THROUGH THE API EXAMPLE

This example illustrates the handling of declared fact types through the API:

```
// get a reference to a knowledge base with a declared type:
KnowledgeBase kbase = ...

// get the declared FactType
FactType personType = kbase.getFactType( "org.drools.examples",
                                         "Person" );

// handle the type as necessary:
// create instances:
Object bob = personType.newInstance();

// set attributes values
personType.set( bob,
                "name",
                "Bob" );
personType.set( bob,
                "age",
                42 );

// insert fact into a session
StatefulKnowledgeSession ksession = ...
ksession.insert( bob );
ksession.fireAllRules();

// read attributes
String name = personType.get( bob, "name" );
int age = personType.get( bob, "age" );
```

Report a bug

15.9. TYPE DECLARATION EXTENDS

Type declarations support the 'extends' keyword for inheritance. To extend a type declared in Java by a DRL declared subtype, repeat the supertype in a declare statement without any fields.

Report a bug

15.10. TYPE DECLARATION EXTENDS EXAMPLE

This illustrates the use of the **`extends`** annotation:

```
import org.people.Person

declare Person
end

declare Student extends Person
    school : String
end

declare LongTermStudent extends Student
```

```
    years : int
    course : String
end
```

Report a bug

15.11. TRAITS

Traits allow you to model multiple dynamic types which do not fit naturally in a class hierarchy. A trait is an interface that can be applied (and eventually removed) to an individual object at runtime. To create a trait out of an interface, a **`@format(trait)`** annotation is added to its declaration in DRL.

Report a bug

15.12. TRAITS EXAMPLE

```
declare GoldenCustomer
    @format(trait)
    // fields will map to getters/setters
    code     : String
    balance  : long
    discount : int
    maxExpense : long
end
```

In order to apply a trait to an object, the new don keyword is added:

```
when
    $c : Customer()
then
    GoldenCustomer gc = don( $c, Customer.class );
end
```

Report a bug

15.13. CORE OBJECTS AND TRAITS

When a core object dons a trait, a proxy class is created on the fly (one such class will be generated lazily for each core/trait class combination). The proxy instance, which wraps the core object and implements the trait interface, is inserted automatically and will possibly activate other rules. An immediate advantage of declaring and using interfaces, getting the implementation proxy for free from the engine, is that multiple inheritance hierarchies can be exploited when writing rules. The core classes, however, need not implement any of those interfaces statically, also facilitating the use of legacy classes as cores. Any object can don a trait. For efficiency reasons, however, you can add the @Traitable annotation to a declared bean class to reduce the amount of glue code that the compiler will have to generate. This is optional and will not change the behavior of the engine.

Report a bug

15.14. @TRAITABLE EXAMPLE

This illustrates the use of the @traitable annotation:

```
declare Customer
    @Traitable
    code    : String
    balance : long
end
```

Report a bug

15.15. WRITING RULES WITH TRAITS

The only connection between core classes and trait interfaces is at the proxy level. (That is, a trait is not specifically tied to a core class.) This means that the same trait can be applied to totally different objects. For this reason, the trait does not transparently expose the fields of its core object. When writing a rule using a trait interface, only the fields of the interface will be available, as usual. However, any field in the interface that corresponds to a core object field, will be mapped by the proxy class.

Report a bug

15.16. RULES WITH TRAITS EXAMPLE

This example illustrates the trait interface being mapped to a field:

```
when
    $o: OrderItem( $p : price, $code : custCode )
    $c: GoldenCustomer( code == $code, $a : balance, $d: discount )
then
    $c.setBalance( $a - $p*$d );
end
```

Report a bug

15.17. HIDDEN FIELDS

Hidden fields are fields in the core class not exposed by the interface.

Report a bug

15.18. THE TWO-PART PROXY

The *two-part proxy* has been developed to deal with soft and hidden fields which are not processed intuitively. Internally, proxies are formed by a proper proxy and a wrapper. The former implements the interface, while the latter manages the core object fields, implementing a name/value map to supports soft fields. The proxy uses both the core object and the map wrapper to implement the interface, as needed.

Report a bug

15.19. WRAPPERS

The *wrapper* provides a looser form of typing when writing rules. However, it has also other uses. The wrapper is specific to the object it wraps, regardless of how many traits have been attached to an object. All the proxies on the same object will share the same wrapper. Additionally, the wrapper contains a back-reference to all proxies attached to the wrapped object, effectively allowing traits to see each other.

Report a bug

15.20. WRAPPER EXAMPLE

This is an example of using the wrapper:

```
when
    $sc : GoldenCustomer( $c : code, // hard getter
                          $maxExpense : maxExpense > 1000 // soft getter
    )
then
    $sc.setDiscount( ... ); // soft setter
end
```

Report a bug

15.21. WRAPPER WITH ISA ANNOTATION EXAMPLE

This illustrates a wrapper in use with the isA annotation:

```
$sc : GoldenCustomer( $maxExpense : maxExpense > 1000,
                      this isA "SeniorCustomer"
)
```

Report a bug

15.22. REMOVING TRAITS

The business logic may require that a trait is removed from a wrapped object. There are two ways to do so:

Logical don

Results in a logical insertion of the proxy resulting from the traiting operation.

```
then
    don( $x, // core object
         Customer.class, // trait class
         true // optional flag for logical insertion
    )
```

The shed keyword

The shed keyword causes the retraction of the proxy corresponding to the given argument type

```
then
    Thing t = shed( $x, GoldenCustomer.class )
```

This operation returns another proxy implementing the org.drools.factmodel.traits.Thing interface, where the getFields() and getCore() methods are defined. Internally, all declared traits are generated to extend this interface (in addition to any others specified). This allows to preserve the wrapper with the soft fields which would otherwise be lost.

Report a bug

15.23. RULE SYNTAX EXAMPLE

This is an example of the syntax you should use when creating a rule:

```
rule "<name>"
    <attribute>*
when
    <conditional element>*
then
    <action>*
end
```

Report a bug

15.24. TIMER ATTRIBUTE EXAMPLE

This is what the **`timer`** attribute looks like:

```
timer ( int: <initial delay> <repeat interval>? )
timer ( int: 30s )
timer ( int: 30s 5m )

timer ( cron: <cron expression> )
timer ( cron:* 0/15 * * * ? )
```

Report a bug

15.25. TIMERS

The following timers are available in JBoss Rules:

Interval

Interval (indicated by "int:") timers follow the semantics of java.util.Timer objects, with an initial delay and an optional repeat interval.

Cron

Cron (indicated by "cron:") timers follow standard Unix cron expressions.

Report a bug

15.26. CRON TIMER EXAMPLE

This is what the Cron timer looks like:

```
rule "Send SMS 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
```

Report a bug

15.27. CALENDARS

Calendars are used to control when rules can fire. JBoss Rules uses the Quartz calendar.

Report a bug

15.28. QUARTZ CALENDAR EXAMPLE

This is what the Quartz calendar looks like:

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

Report a bug

15.29. REGISTERING A CALENDAR

Procedure 15.1. Task

1. Start a StatefulKnowledgeSession.

2. Use the following code to register the calendar:

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

3. If you wish to utilize the calendar and a timer together, use the following code:

```
rule "weekdays are high priority"
   calendars "weekday"
   timer (int:0 1h)
when
    Alarm()
then
    send( "priority high - we have an alarm” );
end

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

Report a bug

15.30. LEFT HAND SIDE

The Left Hand Side (LHS) is a common name for the conditional part of the rule. It consists of zero or more Conditional Elements. If the LHS is empty, it will be considered as a condition element that is always true and it will be activated once, when a new WorkingMemory session is created.

Report a bug

15.31. CONDITIONAL ELEMENTS

Conditional elements work on one or more *patterns*. The most common conditional element is **and**. It is implicit when you have multiple patterns in the LHS of a rule that is not connected in any way.

Report a bug

15.32. RULE WITHOUT A CONDITIONAL ELEMENT EXAMPLE

This is what a rule without a conditional element looks like:

```
rule "no CEs"
when
    // empty
then
    ... // actions (executed once)
end

// The above rule is internally rewritten as:

rule "eval(true)"
when
```

```
    eval( true )
then
    ... // actions (executed once)
end
```

Report a bug

CHAPTER 16. PATTERNS

16.1. PATTERNS

A pattern element is the most important Conditional Element. It can potentially match on each fact that is inserted in the working memory. A pattern contains constraints and has an optional pattern binding.

Report a bug

16.2. PATTERN EXAMPLE

This is what a pattern looks like:

```
rule "2 unconnected patterns"
when
    Pattern1()
    Pattern2()
then
    ... // actions
end

// The above rule is internally rewritten as:

rule "2 and connected patterns"
when
    Pattern1()
    and Pattern2()
then
    ... // actions
end
```

NOTE

An **and** cannot have a leading declaration binding. This is because a declaration can only reference a single fact at a time, and when the **and** is satisfied it matches both facts.

Report a bug

16.3. PATTERN MATCHING

A pattern matches against a fact of the given type. The type need not be the actual class of some fact object. Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes. The constraints are defined inside parentheses.

Report a bug

16.4. PATTERN BINDING

Patterns can be bound to a matching object. This is accomplished using a pattern binding variable such as **$p**.

Report a bug

16.5. PATTERN BINDING WITH VARIABLE EXAMPLE

This is what pattern binding using a variable looks like:

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

NOTE

The prefixed dollar symbol (**$**) is not mandatory.

Report a bug

16.6. CONSTRAINTS

A constraint is an expression that returns **`true`** or **`false`**. For example, you can have a constraint that states five is smaller than six.

Report a bug

CHAPTER 17. ELEMENTS AND VARIABLES

17.1. PROPERTY ACCESS ON JAVA BEANS (POJOS)

Any bean property can be used directly. A bean property is exposed using a standard Java bean getter: a method **`getMyProperty()`** (or **`isMyProperty()`** for a primitive boolean) which takes no arguments and return something.

JBoss Rules uses the standard JDK **`Introspector`** class to do this mapping, so it follows the standard Java bean specification.

WARNING

Property accessors must not change the state of the object in a way that may effect the rules. The rule engine effectively caches the results of its matching in between invocations to make it faster.

Report a bug

17.2. POJO EXAMPLE

This is what the bean property looks like:

```
Person( age == 50 )

// this is the same as:
Person( getAge() == 50 )
```

The age property

The age property is written as **`age`** in DRL instead of the getter **`getAge()`**

Property accessors

You can use property access (**`age`**) instead of getters explicitly (**`getAge()`**) because of performance enhancements through field indexing.

Report a bug

17.3. WORKING WITH POJOS

Procedure 17.1. Task

1. Observe the example below:

```
public int getAge() {
    Date now = DateUtil.now(); // Do NOT do this
    return DateUtil.differenceInYears(now, birthday);
}
```

2. To solve this, insert a fact that wraps the current date into working memory and update that fact between **`fireAllRules`** as needed.

Report a bug

17.4. POJO FALLBACKS

When working with POJOs, a fallback method is applied. If the getter of a property cannot be found, the compiler will resort to using the property name as a method name and without arguments. Nested properties are also indexed.

Report a bug

17.5. FALLBACK EXAMPLE

This is what happens when a fallback is implemented:

```
Person( age == 50 )

// If Person.getAge() does not exists, this falls back to:
Person( age() == 50 )
```

This is what it looks like as a nested property:

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

// this is the same as:
Person( getAddress().getHouseNumber() == 50 )
```

WARNING

In a stateful session, care should be taken when using nested accessors as the Working Memory is not aware of any of the nested values and does not know when they change. Consider them immutable while any of their parent references are inserted into the Working Memory. If you wish to modify a nested value you should mark all of the outer facts as updated. In the above example, when the **`houseNumber`** changes, any **`Person`** with that **`Address`** must be marked as updated.

Report a bug

17.6. JAVA EXPRESSIONS

Table 17.1. Java Expressions

Capability	Example
You can use any Java expression that returns a `boolean` as a constraint inside the parentheses of a pattern. Java expressions can be mixed with other expression enhancements, such as property access.	`Person( age == 50 )`
You can change the evaluation priority by using parentheses, as in any logic or mathematical expression.	`Person( age > 100 && ( age % 10 == 0 ) )`
You can reuse Java methods.	`Person( Math.round( weight / ( height * height ) ) < 25.0 )`
Type coercion is always attempted if the field and the value are of different types; exceptions will be thrown if a bad coercion is attempted.	`Person( age == "10" ) // "10" is coerced to 10`

WARNING

Methods must not change the state of the object in a way that may affect the rules. Any method executed on a fact in the LHS should be a *read only* method.

WARNING

The state of a fact should not change between rule invocations (unless those facts are marked as updated to the working memory on every change):

```
Person( System.currentTimeMillis() % 1000 == 0 ) // Do NOT
do this
```

IMPORTANT

All operators have normal Java semantics except for **==** and **!=**.

The **==** operator has null-safe **equals()** semantics:

```
// Similar to: java.util.Objects.equals(person.getFirstName(),
"John")
// so (because "John" is not null) similar to:
// "John".equals(person.getFirstName())
Person( firstName == "John" )
```

The **!=** operator has null-safe **!equals()** semantics:

```
// Similar to: !java.util.Objects.equals(person.getFirstName(),
"John")
Person( firstName != "John" )
```

Report a bug

17.7. COMMA-SEPARATED OPERATORS

The comma character ('**,**') is used to separate constraint groups. It has implicit and connective semantics.

The comma operator is used at the top level constraint as it makes them easier to read and the engine will be able to optimize them.

Report a bug

17.8. COMMA-SEPARATED OPERATOR EXAMPLE

The following illustrates comma-separated scenarios in implicit and connective semantics:

```
// Person is at least 50 and weighs at least 80 kg
Person( age > 50, weight > 80 )
```

```
// Person is at least 50, weighs at least 80 kg and is taller than 2
meter.
Person( age > 50, weight > 80, height > 2 )
```

NOTE

The comma (**,**) operator cannot be embedded in a composite constraint expression, such as parentheses.

Report a bug

17.9. BINDING VARIABLES

You can bind properties to variables in JBoss Rules. It allows for faster execution and performance.

Report a bug

17.10. BINDING VARIABLE EXAMPLES

This is an example of a property bound to a variable:

```
// 2 persons of the same age
Person( $firstAge : age ) // binding
Person( age == $firstAge ) // constraint expression
```

NOTE

For backwards compatibility reasons, it's allowed (but not recommended) to mix a constraint binding and constraint expressions as such:

```
// Not recommended
Person( $age : age * 2 < 100 )
```

```
// Recommended (separates bindings and constraint expressions)
Person( age * 2 < 100, $age : age )
```

Report a bug

17.11. UNIFICATION

You can *unify* arguments across several properties. While positional arguments are always processed with unification, the unification symbol, ':=', exists for named arguments.

Report a bug

17.12. UNIFICATION EXAMPLE

This is what unifying two arguments looks like:

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

Report a bug

17.13. OPTIONS AND OPERATORS IN JBOSS RULES

Table 17.2. Options and Operators in JBoss Rules

Option	Description	Example
Date literal	The date format **`dd-mmm-yyyy`** is supported by default. You can customize this by providing an alternative date format mask as the System property named **`drools.dateformat`**. If more control is required, use a restriction.	`Cheese( bestBefore < "27-Oct-2009" )`

List and Map access	You can directly access a **`List`** value by index.	`// Same as childList(0).getAge() == 18`
		`Person( childList[0].age == 18 )`

Value key	You can directly access a **`Map`** value by key.	`// Same as credentialMap.get("jsmith").isValid()`
		`Person( credentialMap["jsmith"].valid )`

Abbreviated combined relation condition	This allows you to place more than one restriction on a field using the restriction connectives **&&** or **`||`**. Grouping via parentheses is permitted, resulting in a recursive syntax pattern.	`// 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" )`

Option	Description	Example

Operators	Operators can be used on properties with natural ordering. For example, for Date fields, **<** means *before*, for **`String`** fields, it means alphabetically lower.	`Person( firstName < $otherFirstName )`
		
		`Person( birthDate < $otherBirthDate )`

Operator matches	Matches a field against any valid Java **`regular expression`**. Typically that regexp is a string literal, but variables that resolve to a valid regexp are also allowed. It only applies on **`String`** properties. Using **`matches`** against a **`null`** value always evaluates to false.	`Cheese( type matches "(Buffalo)? \\S*Mozarella" )`
Operator not matches	The operator returns true if the String does not match the regular expression. The same rules apply as for the **`matches`** operator. It only applies on **`String`** properties.	`Cheese( type not matches "(Buffulo)? \\S*Mozarella" )`
The operator contains	The operator **`contains`** is used to check whether a field that is a Collection or array contains the specified value. It only applies on **`Collection`** properties.	`CheeseCounter( cheeses contains "stilton" ) // contains with a String literal CheeseCounter( cheeses contains $var ) // contains with a variable`
The operator not contains	The operator **`not contains`** is used to check whether a field that is a **`Collection`** or array does *not* contain the specified value. It only applies on **`Collection`** properties.	`CheeseCounter( cheeses not contains "cheddar" ) // not contains with a String literal CheeseCounter( cheeses not contains $var ) // not contains with a variable`
The operator memberOf	The operator **`memberOf`** is used to check whether a field is a member of a collection or array; that collection must be a variable.	`CheeseCounter( cheese memberOf $matureCheeses )`

Option	Description	Example
The operator not memberOf	The operator **`not memberOf`** is used to check whether a field is not a member of a collection or array. That collection must be a variable.	`CheeseCounter( cheese not memberOf $matureCheeses )`
The operator soundslike	This operator is similar to **`matches`**, but it checks whether a word has almost the same sound (using English pronunciation) as the given value.	`// match cheese "fubar" or "foobar"` `Cheese( name soundslike 'foobar' )`

The operator str	The operator **`str`** is used to check whether a field that is a **`String`** starts with or ends with a certain value. It can also be used to check the length of the String.	`Message( routingValue str[startsWith] "R1" )`
		 `Message( routingValue str[endsWith] "R2" )`
		 `Message( routingValue str[length] 17 )`

Compound Value Restriction	Compound value restriction is used where there is more than one possible value to match. Currently only the **`in`** and **`not in`** evaluators support this. The second operand of this operator must be a comma-separated list of values, enclosed in parentheses. Values may be given as variables, literals, return values or qualified identifiers. Both evaluators are actually *syntactic sugar*, internally rewritten as a list of multiple restrictions using the operators **`!=`** and **`==`**.	`Person( $cheese : favouriteCheese )` `Cheese( type in ( "stilton", "cheddar", $cheese ) )`

Report a bug

17.14. OPERATOR PRECEDENCE

Table 17.3. Operator precedence

Operator type	Operators	Notes
(nested) property access	`.`	Not normal Java semantics
List/Map access	`[ ]`	Not normal Java semantics
constraint binding	`:`	Not normal Java semantics
multiplicative	`* /%`	
additive	`+ -`	
shift	`<< >>>>>`	
relational	`< ><= >=instanceof`	
equality	`== !=`	Does not use normal Java (*not*) *same* semantics: uses (*not*) *equals* semantics instead.
non-short circuiting AND	`&`	
non-short circuiting exclusive OR	`^`	
non-short circuiting inclusive OR	`|`	
logical AND	`&&`	
logical OR	`||`	
ternary	`? :`	
Comma separated AND	`,`	Not normal Java semantics

Report a bug

17.15. FINE GRAINED PROPERTY CHANGE LISTENERS

This feature allows the pattern matching to only react to modification of properties actually constrained or bound inside of a given pattern. This helps with performance and recursion and avoid artificial object splitting.

NOTE

By default this feature is off in order to make the behavior of the rule engine backward compatible with the former releases. When you want to activate it on a specific bean you have to annotate it with @propertyReactive.

Report a bug

17.16. FINE GRAINED PROPERTY CHANGE LISTENER EXAMPLE

DRL example

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

Java class example

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

Report a bug

17.17. WORKING WITH FINE GRAINED PROPERTY CHANGE LISTENERS

Using these listeners means you do not need to implement the no-loop attribute to avoid an infinite recursion. The engine recognizes that the pattern matching is done on the property while the RHS of the rule modifies other the properties. On Java classes, you can also annotate any method to say that its invocation actually modifies other properties.

Report a bug

17.18. USING PATTERNS WITH @WATCH

Annotating a pattern with @watch allows you to modify the inferred set of properties for which that pattern will react. The properties named in the @watch annotation are added to the ones automatically inferred. You can explicitly exclude one or more of them by beginning their name with a `!` and to make the pattern to listen for all or none of the properties of the type used in the pattern respectively with the wildcards `*` and `!*`.

Report a bug

17.19. @WATCH EXAMPLE

This is the @watch annotation in a rule's LHS:

```
// listens for changes on both firstName (inferred) and lastName
          Person( firstName == $expectedFirstName ) @watch( lastName )
          // listens for all the properties of the Person bean
          Person( firstName == $expectedFirstName ) @watch( * )

          // listens for changes on lastName and explicitly exclude
firstName
          Person( firstName == $expectedFirstName ) @watch( lastName,
!firstName )

          // listens for changes on all the properties except the age one
          Person( firstName == $expectedFirstName ) @watch( *, !age )
```

NOTE

Since doesn't make sense to use this annotation on a pattern using a type not annotated with @PropertyReactive the rule compiler will raise a compilation error if you try to do so. Also the duplicated usage of the same property in @watch (for example like in: @watch(firstName, ! firstName)) will end up in a compilation error.

Report a bug

17.20. USING @PROPERTYSPECIFICOPTION

You can enable @watch by default or completely disallow it using the **on** option of the KnowledgeBuilderConfiguration. This new PropertySpecificOption can have one of the following 3 values:

```
- DISABLED => the feature is turned off and all the other related
annotations are just ignored
          - ALLOWED => this is the default behavior: types are not
property reactive unless they are not annotated with @PropertySpecific
          - ALWAYS => all types are property reactive by default
```

Report a bug

17.21. BASIC CONDITIONAL ELEMENTS

Table 17.4. Basic Conditional Elements

Name	Description	Example	Additional options

Name	Description	Example	Additional options

and	The Conditional Element **and** is used to group other Conditional Elements into a logical conjunction. JBoss Rules supports both prefix **and** and infix **and**. It supports explicit grouping with parentheses. You can also use traditional infix and prefix **and**.	`//infixAnd`
		`Cheese( cheeseType : type ) and Person( favouriteCheese == cheeseType )`
		
		`//infixAnd with grouping`
		`( Cheese( cheeseType : type ) and`
		`  ( Person( favouriteCheese == cheeseType ) or`
		`    Person( favouriteCheese == cheeseType ) )`
			Prefix **and** is also supported:
			
			`(and Cheese( cheeseType : type )`
			`     Person( favouriteCheese == cheeseType ) )`
			
			The root element of the LHS is an implicit prefix **and** and doesn't need to be specified:
			
			`when`
			`    Cheese( cheeseType : type )`
			`    Person( favouriteCheese == cheeseType )`
			`then`
			`    ...`

Name	Description	Example	Additional options

or	This is a shortcut for generating two or more similar rules. JBoss Rules supports both prefix **`or`** and infix **`or`**. You can use traditional infix, prefix and explicit grouping parentheses.	`//infixOr`
		`Cheese( cheeseType : type ) or Person( favouriteCheese == cheeseType )`
		
		`//infixOr with grouping`
		`( Cheese( cheeseType : type ) or`
		`  ( Person( favouriteCheese == cheeseType ) and`
		`    Person( favouriteCheese == cheeseType ) )`
		
		`(or Person( sex == "f", age > 60 )`
		`    Person( sex == "m", age > 65 )`
			Allows for optional pattern binding. Each pattern must be bound separately, using eponymous variables:
			
			`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 ) )`

Name	Description	Example	Additional options

not	This checks to ensure an object specified as absent is not included in the Working Memory. It may be followed by parentheses around the condition elements it applies to. (In a single pattern you can omit the parentheses.)	`// Brackets are optional:`
		`not Bus(color == "red")`
		`// Brackets are optional:`
		`not ( Bus(color == "red", number == 42) )`
		`// "not" with nested infix` **`and`** `- two patterns,`
		`// brackets are requires:`
		`not ( Bus(color == "red") and`
		
		`Bus(color == "blue") )`	

exists	This checks the working memory to see if a specified item exists. The keyword **`exists`** must be followed by parentheses around the CEs that it applies to. (In a single pattern you can omit the parentheses.)	`exists Bus(color == "red")`
		`// brackets are optional:`
		`exists ( Bus(color == "red", number == 42) )`
		`// "exists" with nested infix` **`and`**`,`
		`// brackets are required:`
		`exists ( Bus(color == "red") and`
		
		`Bus(color == "blue") )`	

NOTE

The behavior of the Conditional Element **`or`** is different from the connective **`||`** for constraints and restrictions in field constraints. The engine cannot interpret the Conditional Element **`or`**. Instead, a rule with **`or`** is rewritten as a number of subrules. This process ultimately results in a rule that has a single **`or`** as the root node and one subrule for each of its CEs. Each subrule can activate and fire like any normal rule; there is no special behavior or interaction between these subrules.

Report a bug

17.22. THE CONDITIONAL ELEMENT FORALL

This element evaluates to true when all facts that match the first pattern match all the remaining patterns. It is a *scope delimiter*. Therefore, it can use any previously bound variable, but no variable bound inside it will be available for use outside of it.

`Forall` can be nested inside other CEs. For instance, **`forall`** can be used inside a **`not`** CE. Only single patterns have optional parentheses, so with a nested **`forall`** parentheses must be used.

Report a bug

17.23. FORALL EXAMPLES

Evaluating to true

```
rule "All English buses are red"
when
    forall( $bus : Bus( type == 'english')
                   Bus( this == $bus, color = 'red' ) )
then
    // all English buses are red
end
```

Single pattern forall

```
rule "All Buses are Red"
when
    forall( Bus( color == 'red' ) )
then
    // all Bus facts are red
end
```

Multi-pattern forall

```
rule "all employees have health and dental care programs"
when
    forall( $emp : Employee()
            HealthCare( employee == $emp )
            DentalCare( employee == $emp )
          )
```

```
then
    // all employees have health and dental care
end
```

Nested forall

```
rule "not all employees have health and dental care"
when
    not ( forall( $emp : Employee()
                  HealthCare( employee == $emp )
                  DentalCare( employee == $emp ) )
        )
then
    // not all employees have health and dental care
end
```

Report a bug

17.24. THE CONDITIONAL ELEMENT FROM

The Conditional Element **`from`** enables users to specify an arbitrary source for data to be matched by LHS patterns. This allows the engine to reason over data not in the Working Memory. The data source could be a sub-field on a bound variable or the results of a method call. It is a powerful construction that allows out of the box integration with other application components and frameworks. One common example is the integration with data retrieved on-demand from databases using hibernate named queries.

The expression used to define the object source is any expression that follows regular MVEL syntax. Therefore, it allows you to easily use object property navigation, execute method calls and access maps and collections elements.

IMPORTANT

Using **`from`** with **`lock-on-active`** rule attribute can result in rules not being fired.

There are several ways to address this issue:

- Avoid the use of **`from`** when you can assert all facts into working memory or use nested object references in your constraint expressions (shown below).
- Place the variable assigned used in the modify block as the last sentence in your condition (LHS).
- Avoid the use of **`lock-on-active`** when you can explicitly manage how rules within the same rule-flow group place activations on one another.

Report a bug

17.25. FROM EXAMPLES

Reasoning and binding on patterns

```
rule "validate zipcode"
when
    Person( $personAddress : address )
    Address( zipcode == "23920W") from $personAddress
then
    // zip code is ok
end
```

Using a graph notation

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

Iterating over all objects

```
rule "apply 10% discount to all items over US$ 100,00 in an order"
when
    $order : Order()
    $item  : OrderItem( value > 100 ) from $order.items
then
    // apply discount to $item
end
```

Use with lock-on-active

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

rule "Apply a discount to people in the city of Raleigh"
ruleflow-group "test"
lock-on-active true
when
    $p : Person(address.city == "Raleigh" )
then
    modify ($p) {} //Apply discount to person in a modify block
end
```

Report a bug

17.26. THE CONDITIONAL ELEMENT COLLECT

The Conditional Element **`collect`** allows rules to reason over a collection of objects obtained from the given source or from the working memory. In First Oder Logic terms this is the cardinality quantifier.

The result pattern of **`collect`** 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 ArrayList, LinkedList and HashSet or your own class, as long as it implements the **`java.util.Collection`** interface and provide a default no-arg public constructor.

Variables bound before the **`collect`** CE are in the scope of both source and result patterns and therefore you can use them to constrain both your source and result patterns. Any binding made inside **`collect`** is not available for use outside of it.

Report a bug

17.27. THE CONDITIONAL ELEMENT ACCUMULATE

The Conditional Element **`accumulate`** is a more flexible and powerful form of **`collect`**, in the sense that it can be used to do what **`collect`** does and also achieve results that the CE **`collect`** is not capable of doing. It allows a rule to iterate over a collection of objects, executing custom actions for each of the elements. At the end it returns a result object.

Accumulate supports both the use of pre-defined accumulate functions, or the use of inline custom code. Inline custom code should be avoided though, as it is harder for rule authors to maintain, and frequently leads to code duplication. Accumulate functions are easier to test and reuse.

The Accumulate CE also supports multiple different syntaxes. The preferred syntax is the top level accumulate, as noted bellow, but all other syntaxes are supported for backward compatibility.

Report a bug

17.28. SYNTAX FOR THE CONDITIONAL ELEMENT ACCUMULATE

Top level accumulate syntax

```
accumulate( <source pattern>; <functions> [;<constraints>] )
```

Syntax example

```
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
```

In the above example, min, max and average are Accumulate Functions and will calculate the minimum, maximum and average temperature values over all the readings for each sensor.

Report a bug

17.29. FUNCTIONS OF THE CONDITIONAL ELEMENT ACCUMULATE

- average
- min
- max
- count
- sum
- collectList
- collectSet

These common functions accept any expression as input. For instance, if someone wants to calculate the average profit on all items of an order, a rule could be written using the average function:

```
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
```

Report a bug

17.30. THE CONDITIONAL ELEMENT ACCUMULATE AND PLUGGABILITY

Accumulate functions are all pluggable. That means that if needed, custom, domain specific functions can easily be added to the engine and rules can start to use them without any restrictions. To implement a new Accumulate Function all one needs to do is to create a Java class that implements the **`org.drools.runtime.rule.TypedAccumulateFunction`** interface and add a line to the configuration file or set a system property to let the engine know about the new function.

Report a bug

17.31. THE CONDITIONAL ELEMENT ACCUMULATE AND PLUGGABILITY EXAMPLE

As an example of an Accumulate Function implementation, the following is the implementation of the **average** function:

```
/**
 * An implementation of an accumulator capable of calculating average
values
 */
public class AverageAccumulateFunction implements
org.drools.runtime.rule.TypedAccumulateFunction {

    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.drools.base.accumulators.AccumulateFunction#createContext()
     */
    public Serializable createContext() {
        return new AverageData();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#init(java.lang.Object)
     */
    public void init(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        data.count = 0;
        data.total = 0;
    }
    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#accumulate(java.lang.Objec
t, java.lang.Object)
     */
    public void accumulate(Serializable context,
                           Object value) {
        AverageData data = (AverageData) context;
        data.count++;
        data.total += ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#reverse(java.lang.Object,
java.lang.Object)
     */
    public void reverse(Serializable context,
                        Object value) throws Exception {
        AverageData data = (AverageData) context;
        data.count--;
        data.total -= ((Number) value).doubleValue();
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#getResult(java.lang.Object
)
     */
    public Object getResult(Serializable context) throws Exception {
        AverageData data = (AverageData) context;
        return new Double( data.count == 0 ? 0 : data.total / data.count
);
    }

    /* (non-Javadoc)
     * @see
org.drools.base.accumulators.AccumulateFunction#supportsReverse()
     */
    public boolean supportsReverse() {
        return true;
    }

    /**
     * {@inheritDoc}
     */
    public Class< ? > getResultType() {
        return Number.class;
    }

}
```

Report a bug

17.32. CODE FOR THE CONDITIONAL ELEMENT ACCUMULATE'S FUNCTIONS

Code for plugging in functions (to be entered into the config file)

```
jbossrules.accumulate.function.average =
   org.jbossrules.base.accumulators.AverageAccumulateFunction
```

Alternate Syntax: single function with return type

```
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
```

**** item name ****

** item description **

Report a bug

17.33. ACCUMULATE WITH INLINE CUSTOM CODE

WARNING

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 allows for simpler testing. This form of accumulate is supported for backward compatibility only.

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 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 engine can do decremental calculation when a source object is modified or retracted, 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 engine tries to match against the object returned from the *<result expression>*. If it matches, the **`accumulate`** conditional element evaluates to *true* and the 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 engine stops evaluating CEs for that rule.

Report a bug

17.34. ACCUMULATE WITH INLINE CUSTOM CODE EXAMPLES

Inline custom code

```
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 above example, for each **`Order`** in the Working Memory, the 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 above example, the value of variable **`total`**). Finally, the engine will try to match the result with the **`Number`** pattern, and if the double value is greater than 100, the rule will fire.

Instantiating and populating 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
```

Report a bug

17.35. CONDITIONAL ELEMENT 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 LHS 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 engine. While **`eval`** can be used anywhere in the patterns, the best practice is to add it as the last conditional element in the LHS of a rule.

Evals 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.

Report a bug

17.36. CONDITIONAL ELEMENT EVAL EXAMPLES

This is what **`eval`** looks like in use:

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

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

Report a bug

17.37. THE RIGHT HAND SIDE

The Right Hand Side (RHS) is a common name for the consequence or action part of the rule. The main purpose of the RHS is to insert, retractor modify working memory data. It should contain a list of actions to be executed. The RHS part of a rule should also be kept small, thus keeping it declarative and readable.

NOTE

If you find you need imperative and/or conditional code in the RHS, break the rule down into multiple rules.

Report a bug

17.38. RHS CONVENIENCE METHODS

Table 17.5. RHS Convenience Methods

Name	Description
`update(`*object, handle***`);`**	Tells the engine that an object has changed (one that has been bound to something on the LHS) and rules that need to be reconsidered.
`update(`*object***`);`**	Using **`update()`**, the Knowledge Helper will look up the facthandle via an identity check for the passed object. (If you provide Property Change Listeners to your Java beans that you are inserting into the engine, you can avoid the need to call **`update()`** when the object changes.). After a fact's field values have changed you must call update before changing another fact, or you will cause problems with the indexing within the rule engine. The modify keyword avoids this problem.
`insert(new`*object***`());`**	Places a new object of your creation into the Working Memory.
`insertLogical(new`*object***`());`**	Similar to insert, but the object will be automatically retracted when there are no more facts to support the truth of the currently firing rule.
`retract(`*handle***`);`**	Removes an object from Working Memory.

Report a bug

17.39. CONVENIENCE METHODS USING THE DROOLS VARIABLE

- The call **`drools.halt()`** terminates rule execution immediately. This is required for returning control to the point whence the current session was put to work with **`fireUntilHalt()`**.
- Methods **`insert(Object o)`**, **`update(Object o)`** and **`retract(Object o)`** can be called on **`drools`** as well, but due to their frequent use they can be called without the object reference.

- **`drools.getWorkingMemory()`** returns the **`WorkingMemory`** object.
- **`drools.setFocus( String s)`** sets the focus to the specified agenda group.
- **`drools.getRule().getName()`**, called from a rule's RHS, returns the name of the rule.
- **`drools.getTuple()`** returns the Tuple that matches the currently executing rule, and **`drools.getActivation()`** delivers the corresponding Activation. (These calls are useful for logging and debugging purposes.)

Report a bug

17.40. CONVENIENCE METHODS USING THE KCONTEXT VARIABLE

- The call **`kcontext.getKnowledgeRuntime().halt()`** terminates rule execution immediately.
- The accessor **`getAgenda()`** returns a reference to the session's **`Agenda`**, which in turn provides access to the various rule groups: activation groups, agenda groups, and rule flow groups. A fairly common paradigm is the activation of some agenda group, which could be done with the lengthy call:

  ```
  // give focus to the agenda group CleanUp
  kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "CleanUp"
  ).setFocus();
  ```

  (You can achieve the same using **`drools.setFocus( "CleanUp" )`**.)
- To run a query, you call **`getQueryResults(String query)`**, whereupon you may process the results.
- A set of methods dealing with event management lets you add and remove event listeners for the Working Memory and the Agenda.
- Method**`getKnowledgeBase()`** returns the **`KnowledgeBase`** object, the backbone of all the Knowledge in your system, and the originator of the current session.
- You can manage globals with **`setGlobal(...)`**, **`getGlobal(...)`** and **`getGlobals()`**.
- Method **`getEnvironment()`** returns the runtime's **`Environment`**.

Report a bug

17.41. THE MODIFY STATEMENT

Table 17.6. The Modify Statement

Name	Description	Syntax	Example

Name	Description	Syntax	Example
modify	This provides a structured approach to fact updates. It combines the update operation with a number of setter calls to change the object's fields.	`modify ( <fact-expression> ) { <expression> [ , <expression> ]* }` The parenthesized *<fact-expression>* must yield a fact object reference. The expression list in the block should consist of setter calls for the given object, to be written without the usual object reference, which is automatically prepended by the compiler.	`rule "modify stilton" when $stilton : Cheese(type == "stilton") then modify( $stilton ){ setPrice( 20 ), setAge( "overripe" ) } end`

Report a bug

17.42. QUERY EXAMPLES

NOTE

To return the results use **`ksession.getQueryResults("name")`**, where "name" is the query's name. This returns a list of query results, which allow you to retrieve the objects that matched the query.

Query for people over the age of 30

```
query "people over the age of 30"
    person : Person( age > 30 )
end
```

Query for people over the age of X, and who live in Y

```
query "people over the age of x"  (int x, String y)
    person : Person( age > x, location == y )
end
```

Report a bug

17.43. QUERYRESULTS EXAMPLE

We iterate over the returned QueryResults using a standard "for" loop. Each element is a QueryResultsRow which we can use to access each of the columns in the tuple. These columns can be accessed by bound declaration name or index position:

```
QueryResults results = ksession.getQueryResults( "people over the age of
30" );
System.out.println( "we have " + results.size() + " people over the age  of
30" );

System.out.println( "These people are are over 30:" );

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

Report a bug

17.44. QUERIES CALLING OTHER QUERIES

Queries can 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 you cannot mix expressions with variables.

NOTE

Using the '?' symbol in this process means the query is pull only and once the results are returned you will not receive further results as the underlying data changes.

Report a bug

17.45. QUERIES CALLING OTHER QUERIES EXAMPLE

Query calling another query

```
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
```

Using live queries to reactively receive changes over time from query results

```
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
```

Report a bug

17.46. UNIFICATION FOR DERIVATION QUERIES

JBoss Rules supports unification for derivation queries. This means that arguments are optional. It is possible to call queries from Java leaving arguments unspecified using the static field org.drools.runtime.rule.Variable.v. (You must use 'v' and not an alternative instance of Variable.) These are referred to as 'out' arguments.

NOTE

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.

Report a bug

CHAPTER 18. DOMAIN SPECIFIC LANGUAGES (DSLS)

18.1. 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. You can 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. There are also DSL and DSLR editors you can use, both in the IDE as well as in the web based BRMS, although they may not provide you with the full DSL functionality.

Report a bug

18.2. USING DSLS

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modeling of domain object and the rule engine's native language and methods. A DSL 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.

Report a bug

18.3. DSL EXAMPLE

Table 18.1. DSL Example

Example	Description
`[when]Something is {colour}=Something(colour=="{colour}")`	**`[when]`** indicates the scope of the expression (that is, 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. 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.

Report a bug

18.4. HOW THE DSL PARSER WORKS

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:

- The DSL extracts the string values appearing where the expression contains variable names in brackets.
- The values obtained from these captures are interpolated wherever that name occurs on the right hand side of the mapping.
- The interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.

NOTE

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 means that all wildcard characters in Java's pattern syntax have to be escaped with a preceding backslash ('\').

Report a bug

18.5. THE DSL COMPILER

The DSL compiler transforms DSL rule files line by line. If you do not wish for this to occur, ensure that the captures are surrounded by characteristic text (words or 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. (The characters that surround the capture are not included during interpolation, just the contents between them.)

Report a bug

18.6. DSL SYNTAX EXAMPLES

Table 18.2. DSL Syntax Examples

Name	Description	Example

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

Name	Description	Example
Braces	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(); \}`

Mapping with correct syntax example	n/a	`# 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`

Name	Description	Example

Expanded DSL example	n/a	`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")`

NOTE

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.

Report a bug

18.7. CHAINING DSL EXPRESSIONS

DSL expressions can be chained together one one line to be used at once. It must be clear 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.

Report a bug

18.8. ADDING CONSTRAINTS TO FACTS

Table 18.3. Adding Constraints to Facts

Name	Description	Example
Expressing LHS conditions	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. In the example, the class **`Cheese`**, has these fields: type, price, age and country. You can express some LHS condition in normal DRL.	`Cheese(age < 5, price == 20, type=="stilton", country=="ch")`

DSL definitions	The DSL definitions given in this example 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}'`

"-"	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.	`There is a Cheese with`
		`        - age is less than 42`
		`        - type is 'stilton'`
		
		`Cheese(age<42, type=='stilton')`

Name	Description	Example

Defining DSL phrases	Defining DSL phrases for various operators and even a generic expression that handles any field constraint reduces the amount of DSL entries.	`[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}

DSL definition rule	n/a	`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')`

NOTE

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.

Report a bug

18.9. TIPS FOR DEVELOPING DSLS

- Write representative samples of the rules your application requires and test them as you develop.
- 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.
- Writing rules is easier if most of the data model's types are facts.
- Mark variable parts as parameters. This provides reliable leads for useful DSL entries.
- 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.)
- New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.
- Keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints.

Report a bug

18.10. 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.

Report a bug

18.11. THE MAKE UP OF A DSL ENTRY

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, that is, 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.

Report a bug

18.12. DEBUG OPTIONS FOR DSL EXPANSION

Table 18.4. 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.

Report a bug

18.13. DSL DEFINITION EXAMPLE

This is what a DSL definition looks like:

```
# 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} )\{ \}
```

Report a bug

18.14. TRANSFORMATION OF A DSLR FILE

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. 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, that is, 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.

NOTE

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").

Report a bug

18.15. STRING TRANSFORMATION FUNCTIONS

Table 18.5. 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.

Report a bug

18.16. STRINGING DSL TRANSFORMATION FUNCTIONS

Table 18.6. Stringing DSL Transformation Functions

Name	Description	Example

Name	Description	Example
.dsl	A file containing a DSL definition is customarily given the extension **`.dsl`**. It is passed to the Knowledge Builder with **`ResourceType.DSL`**. For a file using DSL definition, the extension **`.dslr`** should be used. The Knowledge Builder expects **`ResourceType.DSLR`**. The IDE, however, relies on file extensions to correctly recognize and work with your rules file.	`# 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}`
DSL passing	The DSL must be passed to the Knowledge Builder ahead of any rules file using the DSL. 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.	`KnowledgeBuilder kBuilder = new KnowledgeBuilder(); Resource dsl = ResourceFactory.newClassPathResource( dslPath, getClass() ); kBuilder.add( dsl, ResourceType.DSL ); Resource dslr = ResourceFactory.newClassPathResource( dslrPath, getClass() ); kBuilder.add( dslr, ResourceType.DSLR );`

Report a bug

CHAPTER 19. XML

19.1. THE XML FORMAT

WARNING

The XML rule language has not been updated to support functionality introduced in Drools 5.x and is consider a deprecated feature.

As an option, JBoss Rules supports a "native" rule language as an alternative to DRL. This allows you to capture and manage your rules as XML data. Just like the non-XML DRL format, the XML format is parsed into the internal "AST" representation as fast as possible (using a SAX parser). There is no external transformation step required.

Report a bug

19.2. XML RULE EXAMPLE

This is what a rule looks like in XML:

```
<?xml version="1.0" encoding="UTF-8"?>

<package name="com.sample"
         xmlns="http://drools.org/drools-5.0"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0 drools-5.0.xsd">

<import name="java.util.HashMap" />
<import name="org.drools.*" />

<global identifier="x" type="com.sample.X" />
<global identifier="yada" type="com.sample.Yada" />

<function return-type="void" name="myFunc">
    <parameter identifier="foo" type="Bar" />
    <parameter identifier="bada" type="Bing" />

    <body>
     System.out.println("hello world");
    </body>
</function>

<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />
```

```
<lhs>
    <pattern identifier="foo2" object-type="Bar" >
            <or-constraint-connective>
                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <or-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator=">"
value="60" />
                                <literal-restriction evaluator="<"
value="70" />
                            </and-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator="<"
value="50" />
                                <literal-restriction evaluator=">"
value="55" />
                            </and-restriction-connective>
                        </or-restriction-connective>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="black"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="40" />
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="pink"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="12"/>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <or-restriction-connective>
                            <literal-restriction evaluator="=="
value="yellow"/>
                            <literal-restriction evaluator="=="
value="blue" />
                        </or-restriction-connective>
                    </field-constraint>
                </and-constraint-connective>
            </or-constraint-connective>
        </pattern>
        <not>
            <pattern object-type="Person">
                <field-constraint field-name="likes">
                    <variable-restriction evaluator="=="
identifier="type"/>
                </field-constraint>
            </pattern>

            <exists>
                <pattern object-type="Person">
                    <field-constraint field-name="likes">
                        <variable-restriction evaluator="=="
identifier="type"/>
                    </field-constraint>
                </pattern>
            </exists>
        </not>

        <or-conditional-element>
            <pattern identifier="foo3" object-type="Bar" >
                <field-constraint field-name="a">
                    <or-restriction-connective>
                        <literal-restriction evaluator="==" value="3" />
                        <literal-restriction evaluator="==" value="4" />
                    </or-restriction-connective>
                </field-constraint>
                <field-constraint field-name="a3">
                    <literal-restriction evaluator="==" value="hello" />
                </field-constraint>
                <field-constraint field-name="a4">
                    <literal-restriction evaluator="==" value="null" />
                </field-constraint>
            </pattern>

            <pattern identifier="foo4" object-type="Bar" >
                <field-binding field-name="a" identifier="a4" />
                <field-constraint field-name="a">
                    <literal-restriction evaluator="!=" value="4" />
                    <literal-restriction evaluator="!=" value="5" />
                </field-constraint>
            </pattern>
        </or-conditional-element>

        <pattern identifier="foo5" object-type="Bar" >
            <field-constraint field-name="b">
                <or-restriction-connective>
                    <return-value-restriction evaluator="==" >a4 +
1</return-value-restriction>
                    <variable-restriction evaluator=">" identifier="a4"
/>
                    <qualified-identifier-restriction evaluator="==">
                        org.drools.Bar.BAR_ENUM_VALUE
                    </qualified-identifier-restriction>
                </or-restriction-connective>
            </field-constraint>
        </pattern>
        <pattern identifier="foo6" object-type="Bar" >
            <field-binding field-name="a" identifier="a4" />
            <field-constraint field-name="b">
                <literal-restriction evaluator="==" value="6" />
            </field-constraint>
        </pattern>
  </lhs>
 <rhs>
    if ( a == b ) {
      assert( foo3 );
    } else {
      retract( foo4 );
    }
    System.out.println( a4 );
   </rhs>
</rule>

</package>
```

Report a bug

19.3. XML ELEMENTS

Table 19.1. XML Elements

Name	Description
global	Defines global objects that can be referred to in the rules.
function	Contains a function declaration for a function to be used in the rules. You have to specify a return type, a unique name and parameters, in the body goes a snippet of code.
import	Imports the types you wish to use in the rule.

Report a bug

19.4. DETAIL OF A RULE ELEMENT

This example rule has LHS and RHS (conditions and consequence) sections. The RHS is a block of semantic code that will be executed when the rule is activated. The LHS is slightly more complicated as it contains nested elements for conditional elements, constraints and restrictions:

```
<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
```

```
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
    <pattern identifier="cheese" object-type="Cheese">
        <from>
            <accumulate>
                <pattern object-type="Person"></pattern>
                <init>
                    int total = 0;
                </init>
                <action>
                    total += $cheese.getPrice();
                </action>
                <result>
                    new Integer( total ) );
                </result>
            </accumulate>
        </from>
    </pattern>

    <pattern identifier="max" object-type="Number">
        <from>
            <accumulate>
                <pattern identifier="cheese" object-type="Cheese">
</pattern>
                <external-function evaluator="max" expression="$price"/>
            </accumulate>
        </from>
    </pattern>
</lhs>
<rhs>
    list1.add( $cheese );
</rhs>
</rule>
```

Report a bug

19.5. XML RULE ELEMENTS

Table 19.2. XML Rule Elements

Element	Description
Pattern	This allows you to specify a type (class) and perhaps bind a variable to an instance of that class. Nested under the pattern object are constraints and restrictions that have to be met. The Predicate and Return Value constraints allow Java expressions to be embedded.

Element	Description
Conditional elements (not, exists, and, or)	These work like their DRL counterparts. Elements that are nested under and an "and" element are logically "anded" together. Likewise with "or" (and you can nest things further). "Exists" and "Not" work around patterns, to check for the existence or nonexistence of a fact meeting the pattern's constraints.
Eval	Allows the execution of a valid snippet of Java code as long as it evaluates to a boolean (do not end it with a semi-colon, as it is just a fragment). This can include calling a function. The Eval is less efficient than the columns, as the rule engine has to evaluate it each time, but it is a "catch all" feature for when you can express what you need to do with Column constraints.

Report a bug

19.6. AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

JBoss Rules comes with some utility classes to transform between formats. This works by parsing the rules from the source format into the AST and then "dumping" out to the appropriate target format. This allows you to, for example, write rules in DRL and export them to XML.

Report a bug

19.7. CLASSES FOR AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

These are the classes to use when transforming between XML and DRL files. Using combinations of these, you can convert between any format (including round trip):

- DrlDumper - for exporting DRL
- DrlParser - for reading DRL
- XmlPackageReader - for reading XML

NOTE

DSLs will not be preserved (from DRLs that are using a DSL) - but they will be able to be converted.

Report a bug

CHAPTER 20. THE JAVA RULE ENGINE APPLICATION PROGRAMMING INTERFACE

20.1. JSR94

JSR94 is an API used in JBoss Rules. It allows for support of multiple rule engines from a single API. JSR94 does not deal in any way with the rule language itself.

The JSR94 standard represents the "least common denominator" in features across rule engines. This means that there is less functionality in the JSR94 API than in the standard Knowledge API. By using JSR94, you cannot use all the capabilities of JBoss Rules. You should use globals and support for DRL, DSL and XML, via property maps in conjunction with JSR94. This introduces non-portable functionality.

NOTE

As JSR94 does not provide a rule language, you are only solving a small fraction of the complexity of switching rule engines with very little gain. Red Hat recommends you program against the Knowledge (JBoss Rules and jBPM) API.

Report a bug

20.2. JAVAX.RULES INTERFACES

- **`Handle`**

  The Handle is used to retrieve an Object back from the **`WorkingMemory`** which was added in a **`StatefulRuleSession`** . With the **`Handle`** you can modify or remove an **`Object`** from the **`WorkingMemory`**. To modify an Object call **`updateObject()`** from the **`StatefulRuleSession`**. To remove it, call **`removeObject()`** with the **`Handle`** as the Parameter. Inside of the implementation of the Java Rule Engine API will be called the **`modifyObject()`** and **`retractObject()`** methods of the encapsulated Knowledge (Drools and jBPM) API.

- **`ObjectFilter`**

  This interface is used to filter objects for RuleSession.

- **`RuleExecutionSetMetadata`**

  The RuleExecutionSetMetadata is used to store name, description and URI for a RuleExecutionSet.

- **`RuleRuntime`**

  The RuleRuntime is the key to a RuleSession. The RuleRuntime obtained from the RuleServiceProvider.

  If you retrieve a RuleRuntime call createRuleSession() to open a RuleSession.

  Through the RuleRuntime you can retrieve a list of URIs of all RuleExecutionSets, which were registered by a RuleAdministrator. You need the URI as a String to open a RuleSession to the rule engine. The rule engine will use the rules of the RuleExecutionSet inside of the

RuleSession.

The Map is used for Globals. Globals were formerly called ApplicationData (in Drools 2.x). The key needs to be the identifier of the Global and the Value the object you want to use as a Global.

- **`RuleSession`**

  The RuleSession is the object you are working with if you want to contact the rule engine.

  If you are getting a RuleSession from the RuleRuntime, then it will be either a StatefulRuleSession or a StatelessRuleSession.

  Call the release()-method so that all resources will be freed.

- **`StatefulRuleSession`**

  If you need to run the rule engine more than once, run a StatefulRuleSession. You can assert objects, execute rules and so on.

  You will get a Handle for every object which you are asserting to the Rule Session. Do not lose it, you will need it, to retract or modify objects in the Working Memory. You are having no direct contact to Drools´ Working Memory which is used inside the implementation, for this you got the RuleSession.

- **`StatelessRuleSession`**

  A StatelessRuleSession means that you are having only one contact to the rule engine. You are giving a list of objects to the rule engine and the rule engine asserts them all and starts execution immediately. The result is a list of objects. The content of the result list depends on your rules. If your rules are not modifying or retracting any objects from the Working Memory, you should get all objects you re-added.

  You can use the ObjectFilter which will filter the resulting list of objects before you get it.

Report a bug

20.3. JAVAX.RULES CLASSES

- **`RuleServiceProvider`**

  The RuleServiceProvider gives you access to the RuleAdministrator or a RuleRuntime, which you need to open a new Rule Session. To get the RuleServiceProvider call RuleServiceProviderManager.getRuleServiceProvider().

  In a J2EE environment you can bind the RuleServiceProvider to the JNDI and create a lookup to place it in all your applications.

- **`RuleServiceProviderManager`**

  The RuleServiceProvider is often compared with the DriverManager, which you use in JDBC. It works like setting up the Driver for a DataBase.

Report a bug

20.4. JAVAX.RULES EXCEPTIONS

- **`ConfigurationException`**

  This exception is thrown when a user configuration error has been made.

- **`InvalidHandleException`**

  This exception is thrown when a client passes an invalid Handle to the rule engine.

- **`InvalidRuleSessionException`**

  The InvalidRuleSessionException should be thrown when a method is invoked on a RuleSession and the internal state of the RuleSession is invalid. This may have occurred because a StatefulRuleSession has been serialized and external resources can no longer be accessed. This exception is also used to signal that a RuleSession is in an invalid state (such as an attempt to use it after the release method has been called) (Taken from JCP API Documentation).

- **`RuleException`**

  Base class for all Exception classes in the javax.rules package.

- **`RuleExecutionException`**

  This exception is not thrown in the Drools 3 JSR 94 implementation

- **`RuleExecutionSetNotFoundException`**

  This exception is thrown if a client requests a RuleExecutionSet from the RuleRuntime and the URI or RuleExecutionSet cannot be found (Taken from JCP API Documentation).

- **`RuleSessionCreateException`**

  This exception is thrown when a client requests a RuleSession from the RuleRuntime and an error occurs that prevents a RuleSession from being returned (Taken from JCP API Documentation).

- **`RuleSessionTypeUnsupportedException`**

  This exception is thrown when a client requests a RuleSession and the vendor does not support the given type (defined in the RuleRuntime) or the RuleExecutionSet itself does not support the requested mode (Taken from JCP API Documentation).

Report a bug

20.5. USING A RULE SERVICE PROVIDER

Procedure 20.1. Task

1. Use the following code to load the JBoss Rules rule service provider:

   ```
   Class ruleServiceProviderClass =
   Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");
   ```

2. Use the following code to register it:

   ```
   RuleServiceProviderManager.registerRuleServiceProvider(
   "http://jboss.com/products/rules", ruleServiceProviderClass);
   ```

3. Call to the RuleServiceProvider using the following code:

   ```
   RuleServiceProviderManager.getRuleServiceProvider("http://jboss.com/pro
   ducts/rules");
   ```

4. To stop the rule service, deregister it with this code:

   ```
   RuleServiceProviderManager.deregisterRuleServiceProvider(
   "http://jboss.com/products/rules");
   ```

Report a bug

20.6. JAVAX.RULES.ADMIN INTERFACES

- `LocalRuleExecutionSetProvider`
- `Rule`
- `RuleAdministrator`
- `RuleExecutionSet`
- `RuleExecutionSetProvider`

Report a bug

20.7. JAVAX.RULES.ADMIN EXCEPTIONS

- `RuleAdministrationException`

  Base class for all administration RuleException classes in the javax.rules.admin package (Taken from JCP API Documentation).

- `RuleExecutionSetCreateException`

  Occurs when there is an error in creating a rule execution set.

- `RuleExecutionSetDeregistrationException`

  Occurs if there is an error upon attempting to unregister a rule execution set from a URI.

- `RuleExecutionSetRegisterException`

  Occurs if there is an error upon attempting to register a rule execution set to a URI.

Report a bug

20.8. THE RULESERVICEPROVIDER

The RuleServiceProvider provides access to the RuleRuntime and RuleAdministrator APIs. The RuleAdministrator provides an administration API for the management of RuleExecutionSet objects, making it possible to register a RuleExecutionSet that can then be retrieved via the RuleRuntime.

Report a bug

20.9. THE RULESERVICEPROVIDERMANAGER

The *RuleServiceProviderManager* manages the registration and retrieval of RuleServiceProviders. The JBossRules RuleServiceProvider implementation is automatically registered via a static block when the class is loaded using Class.forName, in much the same way as JDBC drivers.

Report a bug

20.10. AUTOMATIC RULESERVICEPROVIDER REGISTRATION EXAMPLE

This is an example of registering the automatic RuleServiceProvider:

```
// RuleServiceProviderImpl is registered to "http://drools.org/"
// via a static initialization block
Class.forName("org.drools.jsr94.rules.RuleServiceProviderImpl");

// Get the rule service provider from the provider manager.
RuleServiceProvider ruleServiceProvider =
  RuleServiceProviderManager.getRuleServiceProvider("http://drools.org/");
```

Report a bug

20.11. REGISTERING A LOCALRULEEXECUTIONSET WITH THE RULEADMINISTRATOR API

Procedure 20.2. Task

1. Create a RuleExecutionSet. You can do so by using the RuleAdministrator which provides factory methods to return an empty LocalRuleExecutionSetProvider or RuleExecutionSetProvider.

2. Specify the name for the RuleExecutionSet.

3. Register the RuleExecutionSet as shown below:

   ```
   // Register the RuleExecutionSet with the RuleAdministrator
   String uri = ruleExecutionSet.getName();
   ruleAdministrator.registerRuleExecutionSet(uri, ruleExecutionSet,
   null);
   ```

4. Use the LocalRuleExecutionSetProvider to load a RuleExecutionSets from local sources that are not serializable, like Streams.

5. Use the RuleExecutionSetProvider to load RuleExecutionSets from serializable sources, like DOM Elements or Packages. Both the "ruleAdministrator.getLocalRuleExecutionSetProvider(null);" and the "ruleAdministrator.getRuleExecutionSetProvider(null);" (use null as a parameter).

6. The example below shoes you how to register the LocalRuleExecutionSet:

```
// Get the RuleAdministration
RuleAdministrator ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
  ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(  drlUrl.openStream()  );

// Create the RuleExecutionSet for the drl
RuleExecutionSet ruleExecutionSet =
  ruleExecutionSetProvider.createRuleExecutionSet( drlReader, null
);
```

7. You can use the "ruleExecutionSetProvider.createRuleExecutionSet(reader, null)" property to provide configuration for the incoming source. When null is passed the default is used to load the input as a drl. Allowed keys for a map are "source" and "dsl". The key "source" takes "drl" or "xml" as its value.

8. Set "source" to "drl" to load a DRL, or to "xml" to load an XML source. "xml" will ignore any "dsl" key/value settings. The "dsl" key can take a Reader or a String (the contents of the dsl) as a value. See the following dsl example:

```
// Get the RuleAdministration
RuleAdministration ruleAdministrator =
ruleServiceProvider.getRuleAdministrator();
LocalRuleExecutionSetProvider ruleExecutionSetProvider =
  ruleAdministrator.getLocalRuleExecutionSetProvider( null );

// Create a Reader for the drl
URL drlUrl = new URL("http://mydomain.org/sources/myrules.drl");
Reader drlReader = new InputStreamReader(  drlUrl.openStream()  );

// Create a Reader for the dsl and a put in the properties map
URL dslUrl = new URL("http://mydomain.org/sources/myrules.dsl");
Reader dslReader = new InputStreamReader( dslUrl.openStream()  );
Map properties = new HashMap();
properties.put( "source", "drl" );
properties.put( "dsl", dslReader );

// Create the RuleExecutionSet for the drl and dsl
RuleExecutionSet ruleExecutionSet =
  ruleExecutionSetProvider.createRuleExecutionSet( reader,
properties );
```

Report a bug

20.12. USING STATEFUL AND STATELESS RULESESSIONS

Procedure 20.3. Task

1. Get the runtime by accessing the RuleServiceProvider as shown:

   ```
   RuleRuntime ruleRuntime = ruleServiceProvider.getRuleRuntime();
   ```

2. To create a rule session, use one of the two RuleRuntime public constants. These are "RuleRuntime.STATEFUL_SESSION_TYPE" and "RuleRuntime.STATELESS_SESSION_TYPE", accompanying the URI to the RuleExecutionSet you wish to instantiate a RuleSession for.

3. Optionally, access the properties to specify globals.

4. The createRuleSession(...) method will return a RuleSession instance. You should cast it to StatefulRuleSession or StatelessRuleSession:

   ```
   (StatefulRuleSession) session =
     ruleRuntime.createRuleSession( uri,
                                    null,
                                    RuleRuntime.STATEFUL_SESSION_TYPE
   );
   session.addObject( new PurchaseOrder( "cheese" ) );
   session.executeRules();
   ```

5. When using a StatelessRuleSession, you can only call executeRules(List list) passing a list of objects, and an optional filter, the resulting objects are then returned:

   ```
   (StatelessRuleSession) session =
     ruleRuntime.createRuleSession( uri,
                                    null,

   RuleRuntime.STATELESS_SESSION_TYPE );
   List list = new ArrayList();
   list.add( new PurchaseOrder( "even more cheese" ) );

   List results = new ArrayList();
   results = session.executeRules( list );
   ```

Report a bug

20.13. USING GLOBALS WITH JSR94

JSR94 supports globals (in a manner that is not portable) by using the properties map passed to the RuleSession factory method. Globals must be defined in the DRL or XML file first, otherwise an exception will be thrown. The key represents the identifier declared in the DRL or XML, and the value is the instance you wish to be used in the execution.

Report a bug

20.14. USING GLOBALS WITH JSR94 EXAMPLE

Here is an example of implementing a global in JSR94:

```
java.util.List globalList = new java.util.ArrayList( );
java.util.Map map = new java.util.HashMap( );
map.put( "list", globalList );
//Open a stateless Session
StatelessRuleSession srs =
  (StatelessRuleSession) runtime.createRuleSession( "SistersRules",
                                                     map,

RuleRuntime.STATELESS_SESSION_TYPE );
...
// Persons added to List
// call executeRules( ) giving a List of Objects as parameter
// There are rules which will put Objects in the List
// fetch the list from the map
List list = (java.util.List) map.get("list");
```

Do not forget to declare the global "list" in your DRL:

```
package SistersRules;
import org.drools.jsr94.rules.Person;
global java.util.List list
rule FindSisters
when
    $person1 : Person ( $name1:name )
    $person2 : Person ( $name2:name )
    eval( $person1.hasSister($person2) )
then
    list.add($person1.getName() + " and " + $person2.getName() +" are
sisters");
    assert( $person1.getName() + " and " + $person2.getName() +" are
sisters");
end
```

Report a bug

20.15. FURTHER READING ABOUT JSR94

If you need more information on JSR94, please refer to the following links:

1. Official JCP Specification for Java Rule Engine API (JSR 94)
   - http://www.jcp.org/en/jsr/detail?id=94
2. The Java Rule Engine API documentation
   - http://www.javarules.org/api_doc/api/index.html

3. The Logic From The Bottom Line: An Introduction to The Drools Project. By N. Alex Rupp, published on TheServiceSide.com in 2004
    - http://www.theserverside.com/articles/article.tss?l=Drools
4. Getting Started With the Java Rule Engine API (JSR 94): Toward Rule-Based Applications. By Dr. Qusay H. Mahmoud, published on Sun Developer Network in 2005
    - http://java.sun.com/developer/technicalArticles/J2SE/JavaRule.html
5. Jess and the javax.rules API. By Ernest Friedman-Hill, published on TheServerSide.com in 2003
    - http://www.theserverside.com/articles/article.tss?l=Jess

Report a bug

CHAPTER 21. JBOSS DEVELOPER STUDIO

21.1. THE RULES INTEGRATED DEVELOPMENT ENVIRONMENT (IDE)

The Integrated Development Environment (IDE) provides an environment to edit and test rules in various formats, then integrate them with applications.

The JBoss Rules IDE is delivered as an Eclipse plug-in, which allows you to author and manage rules from within Eclipse, as well as integrate rules with your application. This is an optional tool. The JBoss Rules IDE is also a part of the Red Hat Developer Studio (formerly known as JBoss IDE).

Report a bug

21.2. RULES IDE FEATURES

The rules IDE has the following features:

1. Textual/graphical rule editor
    1. An editor that is aware of DRL syntax, and provides content assistance (including an outline view)
    2. An editor that is aware of DSL (domain specific language) extensions, and provides content assistance.
2. RuleFlow graphical editor

    You can edit visual graphs which represent a process (a rule flow). The RuleFlow can then be applied to your rule package to have imperative control.
3. Wizards for fast creation of
    1. a "rules" project
    2. a rule resource, (a DRL file)
    3. a Domain Specific language
    4. a decision table
    5. a ruleflow
4. A domain specific language editor
    1. Create and manage mappings from your user's language to the rule language
5. Rule validation
    1. As rules are entered, the rule is "built" in the background and errors reported via the problem view where possible

Report a bug

21.3. JBOSS RULES RUNTIMES

A JBoss Rules runtime is a collection of jar files that represents one specific release of the JBoss Rules project jars. To create a runtime, you must point the IDE to the release of your choice. You can also create a new runtime based on the latest JBoss Rules project jars included in the plugin itself. You are required to specify a default JBoss Rules runtime for your Eclipse workspace, but each individual project can override the default and select the appropriate runtime for that project specifically.

Report a bug

21.4. DEFINING A JBOSS RULES RUNTIME

Procedure 21.1. Task

1. Create a new session in JBoss Rules.

2. Select **Window Preferences**. A "Preferences" dialog will appear.

3. On the left side of this dialog, under the JBoss Rules category, select "Installed JBoss Rules runtimes". The panel on the right should then show the currently defined JBoss Rules runtimes.

4. Click on the **add** button. A dialog will pop up asking for the name of your runtime and the location on your file system where it can be found.

5. To use the default jar files from the JBoss Rules Eclipse plugin, create a new JBoss Rules runtime automatically by clicking the **Create a new Drools 5 runtime ...** button. A file browser will appear, asking you to select the folder on your file system where you want this runtime to be created. The plugin will then automatically copy all required dependencies to the specified folder.

6. To use one specific release of the JBoss Rules project, create a folder on your file system that contains all the necessary JBoss Rules libraries and dependencies. Give your runtime a name and select the location of this folder containing all the required jars. Click OK.

7. The runtime will appear in your table of installed JBoss Rules runtimes. Click on checkbox in front of the newly created runtime to make it the default JBoss Rules runtime. The default JBoss Rules runtime will be used as the runtime of all your JBoss Rules projects that have not selected a project-specific runtime.

8. Restart Eclipse if you changed the default runtime and you want to make sure that all the projects that are using the default runtime update their classpath accordingly.

Report a bug

21.5. SELECTING A RUNTIME FOR JBOSS RULES PROJECTS

Procedure 21.2. Task

1. Use the **`New JBoss Rules Project`** wizard to create a new JBoss Rules Project.

2. Alternatively, convert an existing Java project to a JBoss Rules project using the action by right-clicking on a Java object then clicking the **`Convert to JBoss Rules Project`** dialogue. The plugin will automatically add all the required jars to the classpath of your project.

3. Optionally, on the last page of the **`New JBoss Rules Project`** wizard you can choose to have a project-specific runtime. Uncheck the **`Use default JBoss Rules runtime`** checkbox and select the appropriate runtime in the drop-down box.

4. To access the preferences and add runtimes, go to the workspace preferences and click **`Configure workspace settings ...`**.

5. You can change the runtime of a JBoss Rules project at any time by opening the project properties and selecting the JBoss Rules category. Mark the **`Enable project specific settings`** checkbox and select the appropriate runtime from the drop-down box.

6. Click the **`Configure workspace settings ...`** link. This opens the workspace preferences showing the currently installed runtimes. Use the menu to add new runtimes in this space. If you deselect the **`Enable project specific settings`** checkbox, it will use the default runtime as defined in your global preferences.

Report a bug

21.6. EXAMPLE RULE FILES

A newly created project contains an example rule file (Sample.drl) in the src/rules directory and an example Java file (DroolsTest.java) that can be used to execute the rules in a JBoss Rules engine. This is in the folder src/java, in the com.sample package. All the other jars that are necessary during execution are added to the classpath in a custom classpath container called JBoss Rules Library.

Report a bug

21.7. THE JBOSS RULES BUILDER

The JBoss Rules plug-in adds a *JBoss Rules Builder* capability to your Eclipse instance. This means you can enable a builder on any project that will build and validate your rules when resources change. This happens automatically with the Rule Project Wizard, but you can also enable it manually on any project. To fully validate the rules you will need to run them in a unit test of course.

NOTE

If you have rule files with more than 500 rules per file, it may result in a slower performance. To counter this, turn off the builder or put the large rules into .rule files where you can still use the rule editor, but it won't build them in the background.

Report a bug

21.8. CREATING A NEW RULE

Procedure 21.3. Task

1. Create an empty text .drl file.

2. Copy and paste your rule into it.

3. Save and exit.

4. Alternatively, use the Rules Wizard to create a rule but pressing **`Ctrl+N`** or by choosing it from the toolbar.

5. The wizard will ask for some basic options for generating a rule resource. For storing rule files you would typically create a directory src/rules and create suitably named subdirectories. The package name is mandatory, and is similar to a package name in Java. (That is, it establishes a namespace for grouping related rules.)

6. Select the options that suit you and click **`Finish`**.

Result

You now have a rule skeleton which you can expand upon.

Report a bug

21.9. THE RULE EDITOR

The *rule editor* is where rule managers and developers are modified. The rule editor follows the pattern of a normal text editor in Eclipse. It provides pop-up content assistance. You can invoke pop-up content assistance by pressing **`Ctrl+Space`**.

The rule editor works on files that have a .drl (or .rule) extension. Usually these contain related rules, but it would also be possible to have rules in individual files, grouped by being in the same package namespace. These DRL files are plain text files.

Report a bug

21.10. JBOSS RULES VIEWS

You can alternate between these views when modifying rules:

Working Memory View

Shows all elements in the JBoss Rules working memory.

Agenda View

Shows all elements on the agenda. For each rule on the agenda, the rule name and bound variables are shown.

Global Data View

Shows all global data currently defined in the JBoss Rules working memory.

Audit View

Can be used to display audit logs containing events that were logged during the execution of a rules engine, in tree form.

Rete View

This shows you the current Rete Network for your DRL file. You display it by clicking on the tab "Rete

Tree" at the bottom of the DRL Editor window. With the Rete Network visualization being open, you can use drag-and-drop on individual nodes to arrange optimal network overview. You may also select multiple nodes by dragging a rectangle over them so the entire group can be moved around.

NOTE

The Rete view works only in projects where the JBoss Rules Builder is set in the project´s properties. For other projects, you can use a workaround. Set up a JBoss Rule Project next to your current project and transfer the libraries and the DRLs you want to inspect with the Rete view. Click on the right tab below in the DRL Editor, then click "Generate Rete View".

Report a bug

21.11. USING JBOSS RULES VIEWS

Procedure 21.4. Task

1. To be able to use JBoss Rules views, create breakpoints in your code by invoking the working memory. For example, the line where you call **`workingMemory.fireAllRules()`** is an ideal place to place a break.

2. If the debugger halts at a joinpoint, select the working memory variable in the debug variables view. The available views can then be used to show the details of the selected working memory.

Report a bug

21.12. THE SHOW LOGICAL STRUCTURE

The Show Logical Structure is used with JBoss Rules views. It can toggle showing the logical structure of elements in the working memory or agenda items. Logical structures allow for example visualizing sets of elements in a more obvious way.

Report a bug

21.13. CREATING AUDIT LOGS

Procedure 21.5. Task

1. To create an audit log, execute the rules engine. You will be given the option of creating a new audit log.

2. Enter the following code:

```
StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();
// Create a new Knowledge Runtime Logger, that logs to file.
// An event.log file is created in the subdirectory log dir (which
must exist) of the working directory
KnowledgeRuntimeLogger logger =
KnowledgeRuntimeLoggerFactory.newFileLogger( ksession, "log/event");

ksession.insert(...);
ksession.fireAllRules();

// stop logging
logger.close();
```

3. Open the log by clicking the Open Log action, the first icon in the Audit View, and select the file. The Audit View now shows all events that where logged during the executing of the rules.

Report a bug

21.14. EVENT ICONS IN AUDIT VIEW

Table 21.1. Event Icons in Audit View

Icon	Description
Green square	Object has been inserted.
Yellow square	Object has been updated.
Red square	Object has been removed.
Right arrow	Activation has been created.
Left arrow	Activation has been canceled.
Blue diamond	Activation has been executed.
Process icon	Ruleflow has started or ended.
Activity icon	Ruleflow-group activation or deactivation.
JBoss Rules icon	Rule or rule package has been added or removed.

Report a bug

21.15. METHODS FOR RETRIEVING CAUSES

If an event occurs when executing an activation, it is shown as a child of the activation's execution event. You can retrieve the cause in the following events::

1. The cause of an object modified or retracted event is the last object event for that object. This is either the object asserted event, or the last object modified event for that object.

2. The cause of an activation canceled or executed event is the corresponding activation created event.

NOTE

When selecting an event, the cause is shown in green in the audit view. You can right-click the action and select the "Show Cause" menu item. This will scroll you to the cause of the selected event.

Report a bug

21.16. THE DSL EDITOR

The DSL editor provides a tabular view of the mapping of Language to Rule Expressions. The Language Expression feeds the content assistance for the rule editor so that it can suggest Language Expressions from the DSL configuration. (The rule editor loads the DSL configuration when the rule resource is loaded for editing.

Report a bug

21.17. RULE LANGUAGE MAPPING

Rule language mapping defines the way which a language expression will be compiled by the rule engine compiler. The form of this rule language expression depends on whether it is intended for the condition or the action part of a rule. (For instance, in the RHS it may be a snippet of Java.) The **scope** item indicates where the expression belongs, the **when** item indicates the LHS, then the RHS, and the ***** item means it can go anywhere. It's also possible to create aliases for keywords.

Report a bug

21.18. WORKING WITH RULE LANGUAGE MAPPING

Procedure 21.6. Task

1. Open the DSL editor and select the mapping tab.
2. Select a mapping item (a row in the table) to see the expression and mapping in the text fields below the table.
3. Double click or press the edit button to open the edit dialog.
4. Other buttons let you remove and add mappings. Don't remove mappings while they are still in use.

Report a bug

21.19. DSL TRANSLATION COMPONENTS

Table 21.2. DSL Translation Components

Name	Duty
Parser	The parser reads the rule text in a DSL line by line and tries to match some of the Language Expression. After a match, the values that correspond to a placeholder between curly braces (for example, {age}) are extracted from the rule source.
Placeholders	The placeholders in the corresponding rule expression are replaced by their corresponding value. For example, a natural language expression maps to two constraints on a fact of type Person based on the fields age and location. The {age} and {location} values are then extracted from the original rule text.

NOTE

If you do not wish to use a language mapping for a particular rule in a drl, prefix the expression with > and the compiler will not try to translate it according to the language definition. Also note that Domain Specific Languages are optional. When the rule is compiled, the .dsl file will also need to be available.

Report a bug

21.20. TIPS FOR WORKING WITH LARGE DRL FILES

1. Depending on the JDK you use,you can increase the maximum size of the permanent generation. Do this by starting Eclipse with `-XX:MaxPermSize=###m`

2. Rulesets of 4000 rules or greater should have the permanent generation set to at least 128Mb.

3. You can put rules in a file with the `.rule` extension. The background builder will not try to compile them every time there is a change which will help the IDE run faster.

Report a bug

21.21. CREATING BREAKPOINTS

Procedure 21.7. Task

1. To create breakpoints for easier debugging of rules, open the DRL editor and load the DRL file you wish to use.

2. Double-click the ruler of the DRL editor at the line where you want to add a breakpoint. Note that rule breakpoints can only be created in the consequence of a rule. Double-clicking on a line where no breakpoint is allowed will do nothing. A breakpoint can be removed by double-clicking the ruler once more.

3. Right-click the ruler. A popup menu will show up, containing the **`Toggle breakpoint`** action. Note that rule breakpoints can only be created in the consequence of a rule. The action is automatically disabled if no rule breakpoint is allowed at that line.

4. Click the action to add a breakpoint at the selected line, or remove it if there was one already.

NOTE

The Debug Perspective contains a Breakpoints view which can be used to see all defined breakpoints, get their properties, enable/disable or remove them, and so on.

Report a bug

21.22. DEBUGGING AS A JBOSS RULES APPLICATION

Procedure 21.8. Task

1. Open the DRL Editor.

2. Select the main class of your application.

3. Right-click on it and select the **`Debug As >`** sub-menu and select **`JBoss Rules Application`**.

   Alternatively, you can also select the **`Debug ...`** menu item to open a new dialog for creating, managing and running debug configurations.

4. Select the **`Drools Application`** item in the left tree and click the **`New launch configuration`** button (leftmost icon in the toolbar above the tree). This will create a new configuration with some of the properties already filled in based on the main class you selected in the beginning.

5. Change the name of your debug configuration to something meaningful. You can accept the defaults for all other properties.

6. Click the **`Debug`** button on the bottom to start debugging your application. You only have to define your debug configuration once. The next time you run your JBoss Rules application, you can select the previously defined configuration in the tree as a sub-element of the JBoss Rules tree node, and then click the JBoss Rules button. The Eclipse toolbar also contains shortcut buttons to quickly re-execute one of your previous configurations.

Result

After clicking the **`Debug`** button, the application starts executing and will halt if any breakpoint is encountered. Whenever a JBoss Rules breakpoint is encountered, the corresponding DRL file is opened and the active line is highlighted. The Variables view also contains all rule parameters and their value. You can then use the default Java debug actions to decide what to do next: resume, terminate, step over, and so on. The debug view can also be used to inspect the contents of the Working Memory and the Agenda at that time as well. You don't have to select a Working Memory as the current executing working memory is automatically shown.

Report a bug

21.23. RULES IDE PREFERENCES

The rules IDE comes with a set of customizable preferences that allow you to configure the following options:

Automatically reparse all rules if a Java resource is changed

Triggers a rebuilding of all the rules when a Java class is modified.

Allow cross reference in DRL files

Makes it possible to have a resource in a DRL file reference another resource defined in a different file. For example you could have a rule in a file using a type declared in another file. By enabling this option it will no longer possible to declare the same resource (that is, two rule with the same name in the same package) in two different DRL files.

Internal Drools classes use

Allows, disallows or discourages (generating warning) the use of JBoss Rules classes not exposed in the public API.

Report a bug

CHAPTER 22. HELLO WORLD EXAMPLE

22.1. HELLOWORLD EXAMPLE: CREATING THE KNOWLEDGEBASE AND SESSION

```
final KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

// this will parse and compile in one step
kbuilder.add(ResourceFactory.newClassPathResource("HelloWorld.drl",
        HelloWorldExample.class), ResourceType.DRL);

// Check the builder for errors
if (kbuilder.hasErrors()) {
    System.out.println(kbuilder.getErrors().toString());
    throw new RuntimeException("Unable to compile \"HelloWorld.drl\".");
}

// get the compiled packages (which are serializable)
final Collection<KnowledgePackage> pkgs = kbuilder.getKnowledgePackages();

// add the packages to a KnowledgeBase (deploy the knowledge packages).
final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages(pkgs);

final StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();
```

- A **`KnowledgeBuilder`** is used to turn a DRL source file into **`Package`** objects which the Knowledge Base can consume.
- The add method takes a **`Resource`** interface and a Resource Type as parameters. The **`Resource`** can be used to retrieve a DRL source file from various locations; in this case the DRL file is being retrieved from the classpath using a **`ResourceFactory`**, but it could come from a disk file or a URL.
- Multiple packages of different namespaces can be added to the same Knowledge Base.
- While the Knowledge Base will validate the package, it will only have access to the error information as a String, so if you wish to debug the error information you should do it on the **`KnowledgeBuilder`** instance.
- Once the builder is error free, get the **`Package`** collection, instantiate a **`KnowledgeBase`** from the **`KnowledgeBaseFactory`** and add the package collection.

Report a bug

22.2. HELLOWORLD EXAMPLE: EVENT LOGGING AND AUDITING

```
// setup the debug listeners
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugWorkingMemoryEventListener() );
```

```
// setup the audit logging
KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession, "log/helloworld");
```

- Two default debug listeners are supplied: **`DebugAgendaEventListener`** and **`DebugWorkingMemoryEventListener`**. These print out debug event information to the **`System.err`** stream displayed in the Console window.
- The **`KnowledgeRuntimeLogger`** provides execution auditing which can be viewed in a graphical viewer. This logger is a specialised implementation built on the Agenda and Working Memory listeners.
- When the engine has finished executing, **`logger.close()`** must be called.

Report a bug

22.3. HELLOWORLD EXAMPLE: MESSAGE CLASS

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
    private int             status;
    ...
}
```

- The single class used in this example has two fields: the message, which is a String, and the status which can be one of the two integers **`HELLO`** or **`GOODBYE`**.

Report a bug

22.4. HELLOWORLD EXAMPLE: EXECUTION

```
final Message message = new Message();
message.setMessage("Hello World");
message.setStatus(Message.HELLO);
ksession.insert(message);

ksession.fireAllRules();

logger.close();

ksession.dispose();
```

- A single **`Message`** object is created with the message text "Hello World" and the status **`HELLO`** and then inserted into the engine, at which point **`fireAllRules()`** is executed.

- All network evaluation is done during the insert time. By the time the program execution reaches the **`fireAllRules()`** method call the engine already knows which rules are fully matches and able to fire.

NOTE

To execute the example as a Java application:

1. Open the class **`org.drools.examples.helloworld.HelloWorldExample`** in your Eclipse IDE.
2. Right-click the class and select **`Run as...`** and then **`Java application`**

Report a bug

22.5. HELLOWORLD EXAMPLE: SYSTEM.OUT IN THE CONSOLE WINDOW

```
Hello
Goodbye
```

```
==>[ActivationCreated(0): rule=Hello World;
                   tuple=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[ObjectInserted: handle=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
;

object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
                   tuple=
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
==>[ActivationCreated(4): rule=Good Bye;
                   tuple=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
;

old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec9
6;

new_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec9
6]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
                   tuple=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
]
[AfterActivationFired(4): rule=Good Bye]
```

▪

- By putting a breakpoint on the **`fireAllRules()`** method and select the **`ksession`** variable, you can see that the "Hello World" rule is already activated and on the Agenda, confirming that all the pattern matching work was already done during the insert.
- The application print outs go to **`System.out`** while the debug listener print outs go to **`System.err`**.

Report a bug

22.6. HELLOWORLD EXAMPLE: RULE "HELLO WORLD"

```
rule "Hello World"
      dialect "mvel"
  when
      m : Message( status == Message.HELLO, message : message )
  then
      System.out.println( message );
      modify ( m ) { message = "Goodbye cruel world",
                     status = Message.GOODBYE };
end
```

- The LHS (after **`when`**) section of the rule states that it will be activated for each **`Message`** object inserted into the Working Memory whose status is **`Message.HELLO`**.
- Two variable bindings are created: the variable **`message`** is bound to the **`message`** attribute and the variable **`m`** is bound to the matched **`Message`** object itself.
- The RHS (after **`then`**) or consequence part of the rule is written using the MVEL expression language, as declared by the rule's attribute **`dialect`**.
- After printing the content of the bound variable **`message`** to **`System.out`**, the rule changes the values of the **`message`** and **`status`** attributes of the **`Message`** object bound to **`m`**.
- MVEL's **`modify`** statement allows you to apply a block of assignments in one statement, with the engine being automatically notified of the changes at the end of the block.

Report a bug

22.7. HELLOWORLD EXAMPLE: USING THE "DEBUG AS..." OPTION

Procedure 22.1. Task

1. To access this debugging option, open the class **`org.drools.examples.HelloWorld`** in your Eclipse IDE.
2. Right-click the class and select "Debug as..." and then "Drools application". The rule will be shown along with information about where it is.

Report a bug

22.8. HELLOWORLD EXAMPLE: RULE "GOOD BYE"

```
rule "Good Bye"
      dialect "java"
  when
      Message( status == Message.GOODBYE, message : message )
  then
      System.out.println( message );
end
```

- The "Good Bye" rule, which specifies the "java" dialect, is similar to the "Hello World" rule except that it matches **`Message`** objects whose status is **`Message.GOODBYE`**

Report a bug

CHAPTER 23. SALIENCE STATE EXAMPLE

23.1. SALIENCE STATE EXAMPLE: STATE CLASS EXAMPLE

```
public class State {
    public static final int NOTRUN   = 0;
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

- Each **`State`** class has fields for its name and its current state (see the class **`org.drools.examples.state.State`**). The two possible states for each objects are **`NOTRUN`** and **`FINISHED`**.

Report a bug

23.2. SALIENCE STATE EXAMPLE: EXECUTION

```
State a = new State( "A" );
State b = new State( "B" );
State c = new State( "C" );
final State d = new State( "D" );

// By setting dynamic to TRUE, Drools will use JavaBean
// PropertyChangeListeners so you don't have to call modify or update().
boolean dynamic = true;

session.insert( a, dynamic );
session.insert( b, dynamic );
session.insert( c, dynamic );
session.insert( d, dynamic );

session.fireAllRules();
session.dispose(); // Stateful rule session must always be disposed when
finished
```

- Each instance is asserted in turn into the Session and then **`fireAllRules()`** is called.

Report a bug

23.3. SALIENCE STATE EXAMPLE: EXECUTING APPLICATIONS

Procedure 23.1. Task

1. Open the class **`org.drools.examples.state.StateExampleUsingSalience`** in the Eclipse IDE.

2. Right-click the class and select **`Run as...`** and then **`Java application`**. The following output will appear:

   ```
   A finished
   B finished
   C finished
   D finished
   ```

Report a bug

23.4. SALIENCE STATE EXAMPLE: USING AUDIT LOGGING WITH OPERATIONS

Procedure 23.2. Task

1. To view the Audit log generated by an operation, open the IDE and click on **`Window`** and then select **`Show View`**, then **`Other...`**, **`Drools`** and **`Audit View`**.

2. In the "Audit View" click the **`Open Log`** button and select the file **`<drools-examples-dir>/log/state.log`**.

Report a bug

23.5. SALIENCE STATE EXAMPLE: RULE "BOOTSTRAP"

```
rule Bootstrap
    when
        a : State(name == "A", state == State.NOTRUN )
    then
        System.out.println(a.getName() + " finished" );
        a.setState( State.FINISHED );
end
```

Result:

```
rule "A to B"
    when
        State(name == "A", state == State.FINISHED )
        b : State(name == "B", state == State.NOTRUN )
    then
        System.out.println(b.getName() + " finished" );
        b.setState( State.FINISHED );
end
```

- Every action and the corresponding changes appear in the Working Memory.

- The assertion of the State object A in the state **`NOTRUN`** activates the **`Bootstrap`** rule, while the assertions of the other **`State`** objects have no immediate effect.

- The execution of rule Bootstrap changes the state of A to **`FINISHED`**, which, in turn, activates rule "A to B".

Report a bug

23.6. SALIENCE STATE EXAMPLE: RULE "B TO C"

```
rule "B to C"
        salience 10
    when
        State(name == "B", state == State.FINISHED )
        c : State(name == "C", state == State.NOTRUN )
    then
        System.out.println(c.getName() + " finished" );
        c.setState( State.FINISHED );
end
```

- The conflict resolution strategy allows the engine's Agenda to decide which rule to fire.

- As rule "B to C" has the *higher salience value* (10 versus the default salience value of 0), it fires first, modifying object C to state **`FINISHED`**.

- The Agenda view can also be used to investigate the state of the Agenda, with debug points being placed in the rules themselves and the Agenda view opened.

Report a bug

23.7. SALIENCE STATE EXAMPLE: RULE "B TO D"

```
rule "B to D"
    when
        State(name == "B", state == State.FINISHED )
        d : State(name == "D", state == State.NOTRUN )
    then
        System.out.println(d.getName() + " finished" );
        d.setState( State.FINISHED );
end
```

- Rule "B to D" fires last, modifying object D to state **`FINISHED`**.

- There are no more rules to execute and so the engine stops.

Report a bug

23.8. SALIENCE STATE EXAMPLE: INSERTING A DYNAMIC FACT

```
// By setting dynamic to TRUE, JBoss Rules will use JavaBean
// PropertyChangeListeners so you don't have to call modify or update().
final boolean dynamic = true;

session.insert( fact, dynamic );
```

- For the engine to see and react to changes of fact properties, the application must tell the engine that changes occurred. This can be done explicitly in the rules by using the **`modify`** statement, or implicitly by letting the engine know that the facts implement **`PropertyChangeSupport`** as defined by the *JavaBeans specification*.
- The above example demonstrates how to use **`PropertyChangeSupport`** to avoid the need for explicit **`modify`** statements in the rules.
- Ensure that your facts implement **`PropertyChangeSupport`**, the same way the class **`org.drools.example.State`** does.

Report a bug

23.9. SALIENCE STATE EXAMPLE: SETTER WITH PROPERTYCHANGESUPPORT

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                     oldState,
                                     newState );
}
```

- The setter for **`state`** in the class **`org.drools.examples`**.
- When using **`PropertyChangeListener`** objects, each setter must implement a little extra code for the notification.

Report a bug

23.10. SALIENCE STATE EXAMPLE: AGENDA GROUP RULES "B TO C"

```
rule "B to C"
      agenda-group "B to C"
      auto-focus true
  when
      State(name == "B", state == State.FINISHED )
      c : State(name == "C", state == State.NOTRUN )
  then
      System.out.println(c.getName() + " finished" );
      c.setState( State.FINISHED );
      kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D"
).setFocus();
end
```

Result:

```
rule "B to D"
      agenda-group "B to D"
  when
      State(name == "B", state == State.FINISHED )
      d : State(name == "D", state == State.NOTRUN )
  then
      System.out.println(d.getName() + " finished" );
      d.setState( State.FINISHED );
end
```

- By default, all rules are in the agenda group "MAIN".
- The "agenda-group" attribute lets you specify a different agenda group for the rule. Initially, a Working Memory has its focus on the Agenda group "MAIN".
- A group's rules will only fire when the group receives the focus. This can be achieved either by using the method **`setFocus()`** or the rule attribute **`auto-focus`**.
- **`auto-focus`** means that the rule automatically sets the focus to its agenda group when the rule is matched and activated. It is this "auto-focus" that enables rule "B to C" to fire before "B to D".
- The rule "B to C" calls **`setFocus()`** on the agenda group "B to D", allowing its active rules to fire, which allows the rule "B to D" to fire.

Report a bug

23.11. SALIENCE STATE EXAMPLE: AGENDA GROUP RULES "B TO D"

```
rule "B to D"
      agenda-group "B to D"
  when
      State(name == "B", state == State.FINISHED )
      d : State(name == "D", state == State.NOTRUN )
  then
      System.out.println(d.getName() + " finished" );
      d.setState( State.FINISHED );
end
```

Report a bug

23.12. SALIENCE STATE EXAMPLE: AGENDA GROUP RULES "D TO E"

```
rule "D to E"
  when
      State(name == "D", state == State.FINISHED )
      e : State(name == "E", state == State.NOTRUN )
  then
```

```
        System.out.println(e.getName() + " finished" );
        e.setState( State.FINISHED );
end
```

This produces the following expected output:

```
A finished
B finished
C finished
D finished
E finished
```

- **`StateExampleWithDynamicRules`** adds another rule to the Rule Base after **`fireAllRules()`**.
-
-

Report a bug

CHAPTER 24. FIBONACCI EXAMPLE

24.1. FIBONACCI EXAMPLE: THE CLASS

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

- The sequence field is used to indicate the position of the object in the Fibonacci number sequence.
- The value field shows the value of that Fibonacci object for that sequence position, using -1 to indicate a value that still needs to be computed.

Report a bug

24.2. FIBONACCI EXAMPLE: EXECUTION

Procedure 24.1. Task

1. Launch the Eclipse IED.
2. Open the class **`org.drools.examples.fibonacci.FibonacciExample`**.
3. Right-click the class and select **`Run as...`** and then **`Java application`**.

Result

Eclipse shows the following output in its console window (with "...snip..." indicating lines that were removed to save space):

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...snip...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
```

```
6 == 8
...snip...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

Report a bug

24.3. FIBONACCI EXAMPLE: EXECUTION DETAILS

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

- To use this with Java, a single Fibonacci object is inserted with a sequence field of 50.
- A recursive rule is used to insert the other 49 **`Fibonacci`** objects.
- This example uses the MVEL dialect. This means you can use the **`modify`** keyword, which allows a block setter action which also notifies the engine of changes.

Report a bug

24.4. FIBONACCI EXAMPLE: RECURSE RULE

```
rule Recurse
    salience 10
    when
        f : Fibonacci ( value == -1 )
        not ( Fibonacci ( sequence == 1 ) )
    then
        insert( new Fibonacci( f.sequence - 1 ) );
        System.out.println( "recurse for " + f.sequence );
end
```

- The Recurse rule matches each asserted **`Fibonacci`** object with a value of -1, creating and asserting a new **`Fibonacci`** object with a sequence of one less than the currently matched object.
- Each time a Fibonacci object is added while the one with a sequence field equal to 1 does not exist, the rule re-matches and fires again.
- The **`not`** conditional element is used to stop the rule's matching once we have all 50 Fibonacci objects in memory.
- The Recurse rule has a salience value so all 50 **`Fibonacci`** objects are asserted before the Bootstrap rule is executed.
- You can switch to the Audit view to show the original assertion of the **`Fibonacci`** object with a sequence field of 50, done with Java code. From there on, the Audit view shows the continual recursion of the rule, where each asserted **`Fibonacci`** object causes the Recurse rule to

become activated and to fire again.

Report a bug

24.5. FIBONACCI EXAMPLE: BOOTSTRAP RULE

```
rule Bootstrap
    when
        f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-
restriction
    then
        modify ( f ){ value = 1 };
        System.out.println( f.sequence + " == " + f.value );
end
```

- When a **`Fibonacci`** object with a sequence field of 2 is asserted the Bootstrap rule is matched and activated along with the Recurse rule.
- Note the multi-restriction on field **`sequence`**, testing for equality with 1 or 2.
- When a **`Fibonacci`** object with a sequence of 1 is asserted the Bootstrap rule is matched again, causing two activations for this rule. The Recurse rule does not match and activate because the **`not`** conditional element stops the rule's matching as soon as a **`Fibonacci`** object with a sequence of 1 exists.

Report a bug

24.6. FIBONACCI EXAMPLE: CALCULATE RULE

```
rule Calculate
    when
        // Bind f1 and s1
        f1 : Fibonacci( s1 : sequence, value != -1 )
        // Bind f2 and v2; refer to bound variable s1
        f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
        // Bind f3 and s3; alternative reference of f2.sequence
        f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
    then
        // Note the various referencing techniques.
        modify ( f3 ) { value = f1.value + v2 };
        System.out.println( s3 + " == " + f3.value );
end
```

- When there are two **`Fibonacci`** objects with values not equal to -1, the Calculate rule is able to match them.
- There are 50 Fibonacci objects in the Working Memory. A suitable triple should be selected to calculate each of value in turn.
- Using three Fibonacci patterns in a rule without field constraints to confine the possible cross products would result in many incorrect rule firings. The Calculate rule uses field constraints to correctly constraint the Fibonacci patterns in the correct order. This technique is called *cross*

product matching.

- The first pattern finds any Fibonacci with a value != -1 and binds both the pattern and the field. The second Fibonacci does this too, but it adds an additional field constraint to ensure that its sequence is greater by one than the Fibonacci bound to **`f1`**. When this rule fires for the first time, the two constraints ensure that **`f1`** references sequence 1 and **`f2`** references sequence 2. The final pattern finds the Fibonacci with a value equal to -1 and with a sequence one greater than **`f2`**.

- There are three **`Fibonacci`** objects correctly selected from the available cross products. You can calculate the value for the third **`Fibonacci`** object that's bound to **`f3`**.

- The **`modify`** statement updates the value of the **`Fibonacci`** object bound to **`f3`**. This means there is now another new Fibonacci object with a value not equal to -1, which allows the Calculate rule to rematch and calculate the next Fibonacci number.

- Switching to the Audit view will show how the firing of the last Bootstrap modifies the **`Fibonacci`** object, enabling the "Calculate" rule to match. This then modifies another Fibonacci object allowing the Calculate rule to match again. This continues till the value is set for all **`Fibonacci`** objects.

-

Report a bug

CHAPTER 25. BANKING EXAMPLE

25.1. BANKING EXAMPLE: RULERUNNER

```
public class RuleRunner {

    public RuleRunner() {
    }

    public void runRules(String[] rules,
                         Object[] facts) throws Exception {

        KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
        KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

        for ( int i = 0; i < rules.length; i++ ) {
            String ruleFile = rules[i];
            System.out.println( "Loading file: " + ruleFile );
            kbuilder.add( ResourceFactory.newClassPathResource( ruleFile,

RuleRunner.class ),
                          ResourceType.DRL );
        }

        Collection<KnowledgePackage> pkgs =
kbuilder.getKnowledgePackages();
        kbase.addKnowledgePackages( pkgs );
        StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();

        for ( int i = 0; i < facts.length; i++ ) {
            Object fact = facts[i];
            System.out.println( "Inserting fact: " + fact );
            ksession.insert( fact );
        }

        ksession.fireAllRules();
    }
```

- The class **`RuleRunner`** is used to execute one or more DRL files against a set of data. It compiles the Packages and creates the Knowledge Base for each execution, allowing us to easily execute each scenario and inspect the outputs.

Report a bug

25.2. BANKING EXAMPLE: RULE IN EXAMPLE1.DRL

```
rule "Rule 01"
    when
        eval( 1==1 )
    then
        System.out.println( "Rule 01 Works" );
end
```

Output:

```
Loading file: Example1.drl
Rule 01 Works
```

- This rule has a single **eval** condition that will always be true, so that this rule will match and fire after it has been started.
- The output shows the rule matches and executes the single print statement.

Report a bug

25.3. BANKING EXAMPLE: JAVA EXAMPLE 2

```
public class Example2 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
wrap(1), wrap(5)};
        new RuleRunner().runRules( new String[] { "Example2.drl" },
                                   numbers );
    }

    private static Integer wrap( int i ) {
        return new Integer(i);
    }
}
```

- This example asserts basic facts and prints them out.

Report a bug

25.4. BANKING EXAMPLE: RULE IN EXAMPLE2.DRL

```
rule "Rule 02"
    when
        Number( $intValue : intValue )
    then
        System.out.println( "Number found with value: " + $intValue );
end
```

Output:

```
Loading file: Example2.drl
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 5
Number found with value: 1
Number found with value: 4
Number found with value: 1
Number found with value: 3
```

- This is a basic rule for printing out the specified numbers. It identifies any facts that are **`Number`** objects and prints out the values. Notice the use of the abstract class **`Number`**.
- The pattern matching engine is able to match interfaces and superclasses of asserted objects.
- The output shows the DRL being loaded, the facts inserted and then the matched and fired rules. You can see that each inserted number is matched and fired and thus printed.

Report a bug

25.5. BANKING EXAMPLE: EXAMPLE3.JAVA

```
public class Example3 {
    public static void main(String[] args) throws Exception {
        Number[] numbers = new Number[] {wrap(3), wrap(1), wrap(4),
wrap(1), wrap(5)};
        new RuleRunner().runRules( new String[] { "Example3.drl" },
                                   numbers );
    }

    private static Integer wrap(int i) {
        return new Integer(i);
    }
}
```

- This is a basic rule-based sorting technique.

Report a bug

25.6. BANKING EXAMPLE: RULE IN EXAMPLE3.DRL

```
rule "Rule 03"
    when
        $number : Number( )
        not Number( intValue < $number.intValue )
    then
        System.out.println("Number found with value: " +
$number.intValue() );
        retract( $number );
end
```

Output:

```
Loading file: Example3.drl
```

```
Inserting fact: 3
Inserting fact: 1
Inserting fact: 4
Inserting fact: 1
Inserting fact: 5
Number found with value: 1
Number found with value: 1
Number found with value: 3
Number found with value: 4
Number found with value: 5
```

- The first line of the rule identifies a **`Number`** and extracts the value.
- The second line ensures that there does not exist a smaller number than the one found by the first pattern. The retraction of the number after it has been printed means that the smallest number has been removed, revealing the next smallest number, and so on.

Report a bug

25.7. BANKING EXAMPLE: CLASS CASHFLOW

```
public class Cashflow {
    private Date   date;
    private double amount;

    public Cashflow() {
    }

    public Cashflow(Date date, double amount) {
        this.date = date;
        this.amount = amount;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public String toString() {
        return "Cashflow[date=" + date + ",amount=" + amount + "]";
    }
}
```

- Class **`Cashflow`** has two simple attributes, a date and an amount. (Using the type **`double`** for monetary units is generally bad practice because floating point numbers cannot represent most numbers accurately.)

- There is an overloaded constructor to set the values and a method **`toString`** to print a cashflow.

Report a bug

25.8. BANKING EXAMPLE: EXAMPLE4.JAVA

```
public class Example4 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new Cashflow(new SimpleDate("01/01/2007"), 300.00),
            new Cashflow(new SimpleDate("05/01/2007"), 100.00),
            new Cashflow(new SimpleDate("11/01/2007"), 500.00),
            new Cashflow(new SimpleDate("07/01/2007"), 800.00),
            new Cashflow(new SimpleDate("02/01/2007"), 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example4.drl" },
                                   cashflows );
    }
}
```

- The Java code in this example inserts five Cashflow objects, with varying dates and amounts.

Report a bug

25.9. BANKING EXAMPLE: CLASS SIMPLEDATE

```
public class SimpleDate extends Date {
    private static final SimpleDateFormat format = new
SimpleDateFormat("dd/MM/yyyy");

    public SimpleDate(String datestr) throws Exception {
        setTime(format.parse(datestr).getTime());
    }
}
```

- The convenience class **`SimpleDate`** extends **`java.util.Date`**, providing a constructor taking a String as input and defining a date format.

Report a bug

25.10. BANKING EXAMPLE: RULE IN EXAMPLE4.DRL

```
rule "Rule 04"
```

```
    when
        $cashflow : Cashflow( $date : date, $amount : amount )
        not Cashflow( date < $date)
    then
        System.out.println("Cashflow: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Output:

```
Loading file: Example4.drl
Inserting fact: Cashflow[date=Mon Jan 01 00:00:00 GMT 2007,amount=300.0]
Inserting fact: Cashflow[date=Fri Jan 05 00:00:00 GMT 2007,amount=100.0]
Inserting fact: Cashflow[date=Thu Jan 11 00:00:00 GMT 2007,amount=500.0]
Inserting fact: Cashflow[date=Sun Jan 07 00:00:00 GMT 2007,amount=800.0]
Inserting fact: Cashflow[date=Tue Jan 02 00:00:00 GMT 2007,amount=400.0]
Cashflow: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Cashflow: Tue Jan 02 00:00:00 GMT 2007 :: 400.0
Cashflow: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Cashflow: Sun Jan 07 00:00:00 GMT 2007 :: 800.0
Cashflow: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

- A **`Cashflow`** is identified and the date and amount are extracted.
- In the second line of the rule in it is determined that there is no Cashflow with an earlier date than the one found.
- In the consequence, the **`Cashflow`** is printed. This satisfies the rule and then retracts it, making way for the next earliest **`Cashflow`**.

Report a bug

25.11. BANKING EXAMPLE: CLASS TYPEDCASHFLOW

```
public class TypedCashflow extends Cashflow {
    public static final int CREDIT = 0;
    public static final int DEBIT  = 1;

    private int             type;

    public TypedCashflow() {
    }

    public TypedCashflow(Date date, int type, double amount) {
        super( date, amount );
        this.type = type;
    }

    public int getType() {
        return type;
    }

    public void setType(int type) {
        this.type = type;
    }

    public String toString() {
        return "TypedCashflow[date=" + getDate() +
               ",type=" + (type == CREDIT ? "Credit" : "Debit") +
               ",amount=" + getAmount() + "]";
    }
}
```

- When the **`Cashflow`** is extended it results in a **`TypedCashflow`**, which can be a credit or a debit operation.

Report a bug

25.12. BANKING EXAMPLE: EXAMPLE5.JAVA

Class:

```
public class Example5 {
    public static void main(String[] args) throws Exception {
        Object[] cashflows = {
            new TypedCashflow(new SimpleDate("01/01/2007"),
                              TypedCashflow.CREDIT, 300.00),
            new TypedCashflow(new SimpleDate("05/01/2007"),
                              TypedCashflow.CREDIT, 100.00),
            new TypedCashflow(new SimpleDate("11/01/2007"),
                              TypedCashflow.CREDIT, 500.00),
            new TypedCashflow(new SimpleDate("07/01/2007"),
                              TypedCashflow.DEBIT, 800.00),
            new TypedCashflow(new SimpleDate("02/01/2007"),
                              TypedCashflow.DEBIT, 400.00),
        };

        new RuleRunner().runRules( new String[] { "Example5.drl" },
                                   cashflows );
    }
}
```

Rule:

```
rule "Rule 05"
    when
        $cashflow : TypedCashflow( $date : date,
                                   $amount : amount,
                                   type == TypedCashflow.CREDIT )
        not TypedCashflow( date < $date,
                           type == TypedCashflow.CREDIT )
    then
        System.out.println("Credit: "+$date+" :: "+$amount);
        retract($cashflow);
end
```

Output:

```
Loading file: Example5.drl
Inserting fact: TypedCashflow[date=Mon Jan 01 00:00:00 GMT
2007,type=Credit,amount=300.0]
Inserting fact: TypedCashflow[date=Fri Jan 05 00:00:00 GMT
2007,type=Credit,amount=100.0]
Inserting fact: TypedCashflow[date=Thu Jan 11 00:00:00 GMT
2007,type=Credit,amount=500.0]
Inserting fact: TypedCashflow[date=Sun Jan 07 00:00:00 GMT
2007,type=Debit,amount=800.0]
Inserting fact: TypedCashflow[date=Tue Jan 02 00:00:00 GMT
2007,type=Debit,amount=400.0]
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
Credit: Fri Jan 05 00:00:00 GMT 2007 :: 100.0
Credit: Thu Jan 11 00:00:00 GMT 2007 :: 500.0
```

- Both the class and the .drl are supplied to the rule engine.
- In the class, a set of **`Cashflow`** objects are created which are either credit or debit operations.
- A **`Cashflow`** fact is identified with a type of **`CREDIT`** and extract the date and the amount. In the second line of the rule we ensure that there is no **`Cashflow`** of the same type with an earlier date than the one found. In the consequence, we print the cashflow satisfying the patterns and then retract it, making way for the next earliest cashflow of type

Report a bug

25.13. BANKING EXAMPLE: CLASS ACCOUNT

```
public class Account {
    private long   accountNo;
    private double balance = 0;

    public Account() {
    }

    public Account(long accountNo) {
        this.accountNo = accountNo;
    }

    public long getAccountNo() {
        return accountNo;
    }

    public void setAccountNo(long accountNo) {
        this.accountNo = accountNo;
    }

    public double getBalance() {
        return balance;
    }

    public void setBalance(double balance) {
        this.balance = balance;
    }
    public String toString() {
        return "Account[" + "accountNo=" + accountNo + ",balance=" +
balance + "]";
    }
}
```

- Two separate **`Account`** objects are created and injected into the **`Cashflows`** objects before being passed to the Rule Engine.

Report a bug

25.14. BANKING EXAMPLE: CLASS ALLOCATEDCASHFLOW

```
public class AllocatedCashflow extends TypedCashflow {
    private Account account;

    public AllocatedCashflow() {
    }

    public AllocatedCashflow(Account account, Date date, int type, double
amount) {
        super( date, type, amount );
        this.account = account;
    }

    public Account getAccount() {
        return account;
    }

    public void setAccount(Account account) {
        this.account = account;
    }

    public String toString() {
        return "AllocatedCashflow[" +
                "account=" + account +
                ",date=" + getDate() +
                ",type=" + (getType() == CREDIT ? "Credit" : "Debit") +
                ",amount=" + getAmount() + "]";
    }
}
```

- Extending the **`TypedCashflow`**, results in **`AllocatedCashflow`**, which includes an **`Account`** reference.

Report a bug

25.15. BANKING EXAMPLE: EXTENDING EXAMPLE5.JAVA

```
public class Example6 {
    public static void main(String[] args) throws Exception {
        Account acc1 = new Account(1);
        Account acc2 = new Account(2);

        Object[] cashflows = {
            new AllocatedCashflow(acc1,new SimpleDate("01/01/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/02/2007"),
                                  TypedCashflow.CREDIT, 100.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/03/2007"),
                                  TypedCashflow.CREDIT, 500.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/02/2007"),
                                  TypedCashflow.DEBIT,  800.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/03/2007"),
                                  TypedCashflow.DEBIT,  400.00),
            new AllocatedCashflow(acc1,new SimpleDate("01/04/2007"),
                                  TypedCashflow.CREDIT, 200.00),
            new AllocatedCashflow(acc1,new SimpleDate("05/04/2007"),
                                  TypedCashflow.CREDIT, 300.00),
            new AllocatedCashflow(acc2,new SimpleDate("11/05/2007"),
                                  TypedCashflow.CREDIT, 700.00),
            new AllocatedCashflow(acc1,new SimpleDate("07/05/2007"),
                                  TypedCashflow.DEBIT,  900.00),
            new AllocatedCashflow(acc2,new SimpleDate("02/05/2007"),
                                  TypedCashflow.DEBIT,  100.00)
        };

        new RuleRunner().runRules( new String[] { "Example6.drl" },
                                   cashflows );
    }
}
```

- This Java code creates two **`Account`** objects and passes one of them into each cashflow in the constructor call.

Report a bug

25.16. BANKING EXAMPLE: RULE IN EXAMPLE6.DRL

```
rule "Rule 06 - Credit"
    when
        $cashflow : AllocatedCashflow( $account : account,
                                       $date : date,
                                       $amount : amount,
                                       type == TypedCashflow.CREDIT )
        not AllocatedCashflow( account == $account, date < $date)
    then
        System.out.println("Credit: " + $date + " :: " + $amount);
        $account.setBalance($account.getBalance()+$amount);
        System.out.println("Account: " + $account.getAccountNo() +
                           " - new balance: " + $account.getBalance());
        retract($cashflow);
end

rule "Rule 06 - Debit"
```

```
    when
        $cashflow : AllocatedCashflow( $account : account,
                            $date : date,
                            $amount : amount,
                            type == TypedCashflow.DEBIT )
        not AllocatedCashflow( account == $account, date < $date)
    then
        System.out.println("Debit: " + $date + " :: " + $amount);
        $account.setBalance($account.getBalance() - $amount);
        System.out.println("Account: " + $account.getAccountNo() +
                           " - new balance: " + $account.getBalance());
        retract($cashflow);
end
```

Output:

```
Loading file: Example6.drl
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Jan 01
00:00:00 GMT 2007,type=Credit,amount=300.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon Feb 05
00:00:00 GMT 2007,type=Credit,amount=100.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Sun Mar 11
00:00:00 GMT 2007,type=Credit,amount=500.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Wed Feb 07
00:00:00 GMT 2007,type=Debit,amount=800.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri Mar 02
00:00:00 GMT 2007,type=Debit,amount=400.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Sun Apr 01
00:00:00 BST 2007,type=Credit,amount=200.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Thu Apr 05
00:00:00 BST 2007,type=Credit,amount=300.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Fri May 11
00:00:00 BST 2007,type=Credit,amount=700.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=1,balance=0.0],date=Mon May 07
00:00:00 BST 2007,type=Debit,amount=900.0]
Inserting fact:
AllocatedCashflow[account=Account[accountNo=2,balance=0.0],date=Wed May 02
00:00:00 BST 2007,type=Debit,amount=100.0]
Debit: Fri Mar 02 00:00:00 GMT 2007 :: 400.0
Account: 2 - new balance: -400.0
Credit: Sun Mar 11 00:00:00 GMT 2007 :: 500.0
Account: 2 - new balance: 100.0
Debit: Wed May 02 00:00:00 BST 2007 :: 100.0
Account: 2 - new balance: 0.0
Credit: Fri May 11 00:00:00 BST 2007 :: 700.0
Account: 2 - new balance: 700.0
Credit: Mon Jan 01 00:00:00 GMT 2007 :: 300.0
```

```
Account: 1 - new balance: 300.0
Credit: Mon Feb 05 00:00:00 GMT 2007 :: 100.0
Account: 1 - new balance: 400.0
Debit: Wed Feb 07 00:00:00 GMT 2007 :: 800.0
Account: 1 - new balance: -400.0
Credit: Sun Apr 01 00:00:00 BST 2007 :: 200.0
Account: 1 - new balance: -200.0
Credit: Thu Apr 05 00:00:00 BST 2007 :: 300.0
Account: 1 - new balance: 100.0
Debit: Mon May 07 00:00:00 BST 2007 :: 900.0
Account: 1 - new balance: -800.0
```

- In this example, each cashflow in date order is applied and calculated, resulting in a print of the balance.
- Although there are separate rules for credits and debits, there is no type specified when checking for earlier cashflows. This is so that all cashflows are applied in date order, regardless of the cashflow type.
- In the conditions the account has identified and in the consequences, the cashflow amount is updated.

Report a bug

CHAPTER 26. PRICING RULE EXAMPLE

26.1. PRICING RULE EXAMPLE: EXECUTING THE PRICING RULE EXAMPLE

Procedure 26.1. Task

1. Open your console.

2. Open the file **`PricingRuleDTExample.java`** and execute it as a Java application. It will produce the following output in the console window:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

3. Use the following code to execute the example:

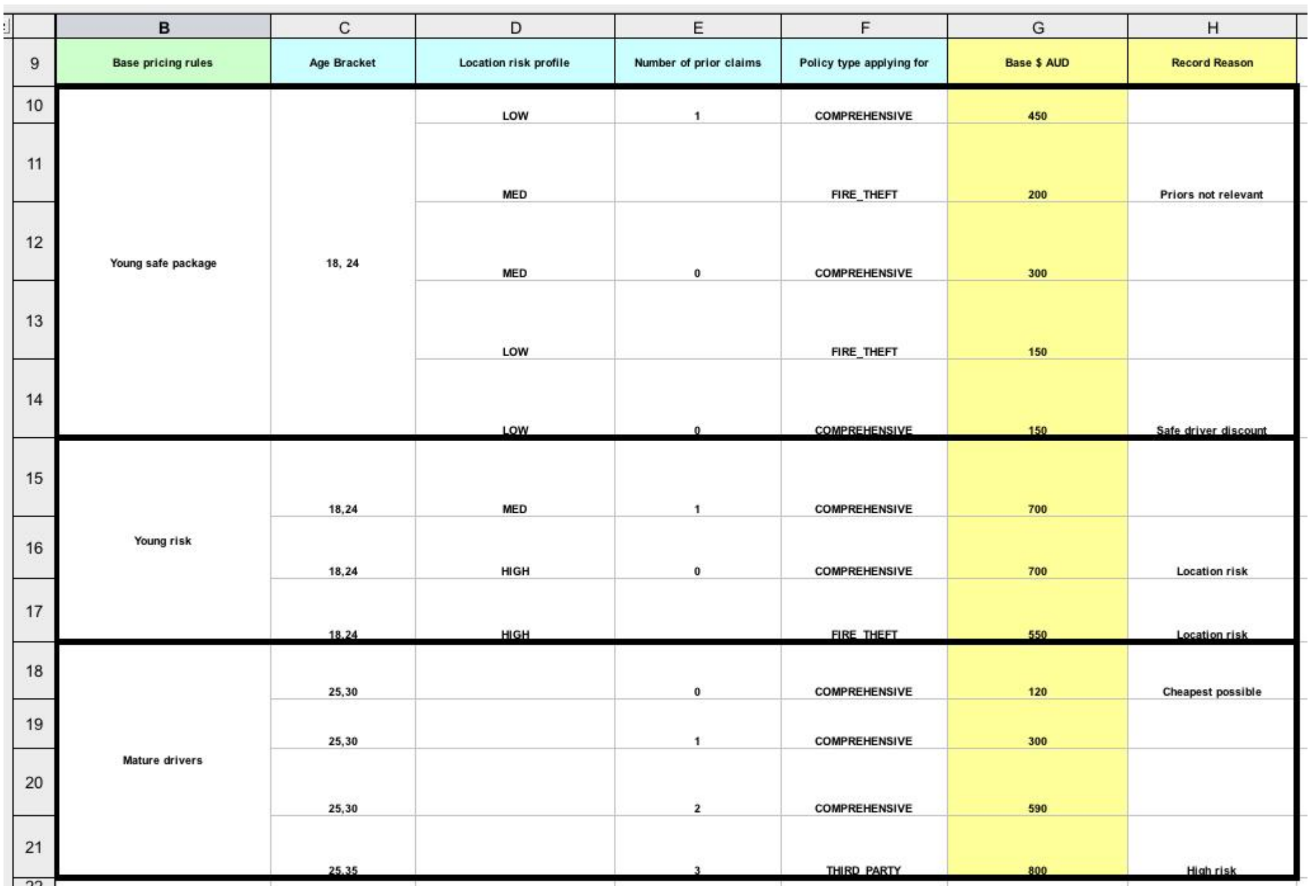
```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
        dtableconfiguration.setInputType( DecisionTableInputType.XLS
);

        KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();

        Resource xlsRes = ResourceFactory.newClassPathResource(
"ExamplePolicyPricing.xls",

getClass() );
        kbuilder.add( xlsRes,
                      ResourceType.DTABLE,
                      dtableconfiguration );
```

The **`DecisionTableConfiguration`** object's type is set to **`DecisionTableInputType.XLS`**.

There are two fact types used in this example, **`Driver`** and **`Policy`**. Both are used with their default values. The **`Driver`** is 30 years old, has had no prior claims and currently has a risk profile of **`LOW`**. The **`Policy`** being applied for is **`COMPREHENSIVE`**, and it has not yet been approved.

Report a bug

26.2. PRICING RULE EXAMPLE: DECISION TABLE CONFIGURATION



	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason
10			LOW	1	COMPREHENSIVE	450	
11			MED		FIRE_THEFT	200	Priors not relevant
12	Young safe package	18, 24	MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	Safe driver discount
15		18,24	MED	1	COMPREHENSIVE	700	
16	Young risk	18,24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18,24	HIGH		FIRE_THEFT	550	Location risk
18		25,30		0	COMPREHENSIVE	120	Cheapest possible
19		25,30		1	COMPREHENSIVE	300	
20	Mature drivers	25,30		2	COMPREHENSIVE	590	
21		25,35		3	THIRD_PARTY	800	High risk

Figure 26.1. Decision Table Configuration

- The **`RuleSet`** declaration provides the package name. There are also other optional items you can put here, such as **`Variables`** for global variables, and **`Imports`** for importing classes. In this case, the namespace of the rules is the same as the fact classes and thus can be omitted.
- The name after the **`RuleTable`** declaration (Pricing bracket) is used as the prefix for all the generated rules.
- "CONDITION or ACTION", indicates the purpose of the column, that is, whether it forms part of the condition or the consequence of the rule that will be generated.
- The driver's data is spread across three cells which means that the template expressions below it are applied. You can observe the driver's age range (which uses **`$1`** and **`$2`** with comma-separated values), **`locationRiskProfile`**, and **`priorClaims`** in the respective columns.
- You can set a policy base price and message log in the Action column.

Report a bug

26.3. PRICING RULE EXAMPLE: BASE PRICE CALCULATION EXAMPLE

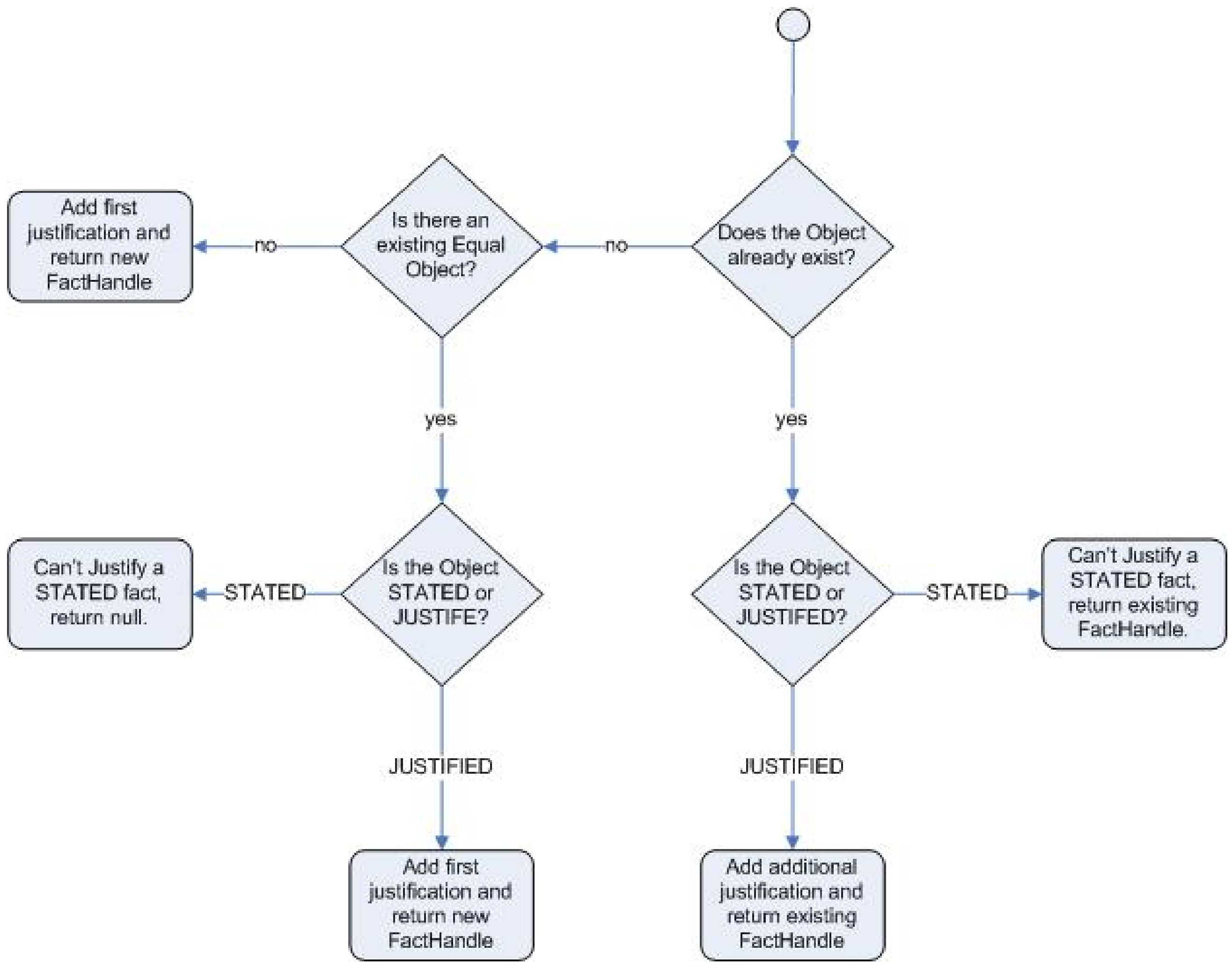
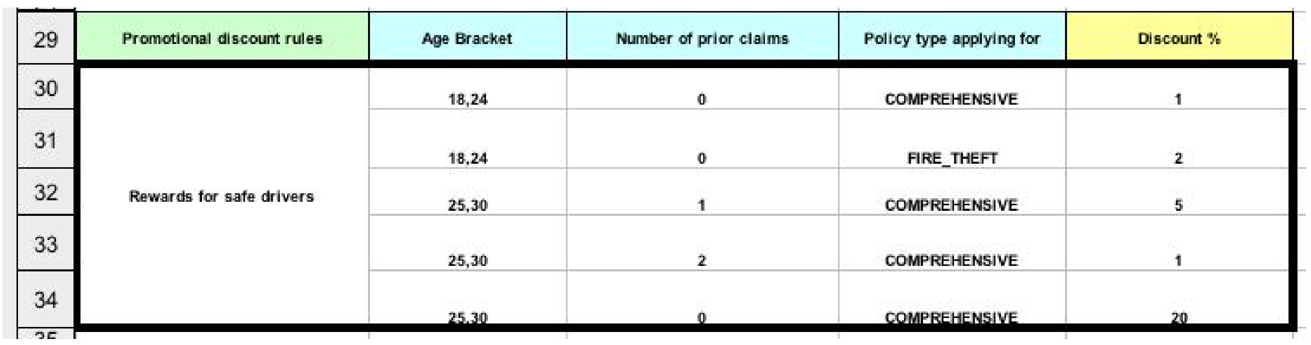

Does the Object already exist?
no
Is there an existing Equal Object?
no
Add first justification and return new FactHandle
yes
yes
Is the Object STATED or JUSTIFE?
STATED
Can't Justify a STATED fact, return null.
Is the Object STATED or JUSTIFED?
STATED
Can't Justify a STATED fact, return existing FactHandle.
JUSTIFIED
JUSTIFIED
Add first justification and return new FactHandle
Add additional justification and return existing FactHandle

Figure 26.2. Base Price Calculation Example

- Broad category brackets are indicated by the comment in the leftmost column.
- The details of the drivers match row number 18 as they have no prior accidents and are 30 years old. This gives us a base price of 120.

Report a bug

26.4. PRICING RULE EXAMPLE: DISCOUNT CALCULATION EXAMPLE



29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32		25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20

Figure 26.3. Discount Calculation Example

- The discount results from the **Age** bracket, the number of prior claims, and the policy type.

- The driver is 30 with no prior claims and is applying for a **`COMPREHENSIVE`** policy. This means a 20% discount can be applied. Note that this is actually a separate table in the same worksheet, so different templates apply.
- The evaluation of the rules is not necessarily in the given order, since all the normal mechanics of the rule engine still apply.

Report a bug

CHAPTER 27. PET STORE EXAMPLE

27.1. PET STORE EXAMPLE

All of the Java code for the Pet Store Example is contained in the file **`PetStore.java`**. It defines the following principal classes (in addition to several classes to handle Swing Events):

- **`Petstore`** contains the **`main()`** method.
- **`PetStoreUI`** is responsible for creating and displaying the Swing based GUI. It contains several smaller classes, mainly for responding to various GUI events such as mouse button clicks.
- **`TableModel`** holds the table data. It is a JavaBean that extends the Swing class **`AbstractTableModel`**.
- **`CheckoutCallback`** allows the GUI to interact with the Rules.
- **`Ordershow`** keeps the items that the customer wishes to buy.
- **`Purchase`** stores details of the order and the products the customer is buying.
- **`Product`** is a JavaBean holding details of the product available for purchase and its price.

Report a bug

27.2. PET STORE EXAMPLE: CREATING THE PETSTORE RULEBASE IN PETSTORE.MAIN

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

kbuilder.add( ResourceFactory.newClassPathResource( "PetStore.drl",
                                                    PetStore.class ),
              ResourceType.DRL );
KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );

// A callback is responsible for populating the
// Working Memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kbase ) );
ui.createAndShowGUI();
```

- The code shown above loads the rules from a DRL file on the classpath. It does so via the second last line where a **`PetStoreUI`** object is created using a constructor. This accepts the **`Vector`** object **`stock`** that collects the products.

- The **`CheckoutCallback`** class contains the Rule Base that has been loaded.

Report a bug

27.3. PET STORE EXAMPLE: FIRING RULES FROM CHECKOUTCALLBACK.CHECKOUT()

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user interaction

    StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();
    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

- The Java code that fires the rules is within the **`CheckoutCallBack.checkout()`** method. This is triggered (eventually) when the Checkout button is pressed by the user.
- Two items get passed into this method. One is the handle to the **`JFrame`** Swing component surrounding the output text frame, at the bottom of the GUI. The second is a list of order items. This comes from the **`TableModel`** storing the information from the "Table" area at the top right section of the GUI.
- The for loop transforms the list of order items coming from the GUI into the **`Order`** JavaBean, also contained in the file **`PetStore.java`**.
- All states in this example are stored in the Swing components. The rules are effectively stateless.
- Each time the "Checkout" button is pressed, the code copies the contents of the Swing **`TableModel`** into the Session's Working Memory.
- There are nine calls to the Working Memory. The first creates a new Working Memory as a

Stateful Knowledge Session from the Knowledge Base. The next two pass in two objects that will be held as global variables in the rules. The Swing text area and the Swing frame used for writing messages.

- More inserts put information on products into the Working Memory and the order list. The final call is the standard **`fireAllRules()`**.

Report a bug

27.4. PET STORE EXAMPLE: PACKAGE, IMPORTS, GLOBALS AND DIALECT FROM PETSTORE.DRL

```
package org.drools.examples

import org.drools.WorkingMemory
import org.drools.examples.petstore.PetStoreExample.Order
import org.drools.examples.petstore.PetStoreExample.Purchase
import org.drools.examples.petstore.PetStoreExample.Product
import java.util.ArrayList
import javax.swing.JOptionPane;

import javax.swing.JFrame

global JFrame frame
global javax.swing.JTextArea textArea
```

- The first part of file **`PetStore.drl`** contains the standard package and import statements to make various Java classes available to the rules.
- The two globals **`frame`** and **`textArea`** hold references to the Swing components **`JFrame`** and **`JTextArea`** components that were previously passed on by the Java code calling the **`setGlobal()`** method. These global variables retain their value for the lifetime of the Session.

Report a bug

27.5. PET STORE EXAMPLE: JAVA FUNCTIONS IN THE RULES EXTRACTED FROM PETSTORE.DRL

```
function void doCheckout(JFrame frame, WorkingMemory workingMemory) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to checkout?",
        "",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        options[0]);
    if (n == 0) {
        workingMemory.setFocus( "checkout" );
    }
}

function boolean requireTank(JFrame frame, WorkingMemory workingMemory,
Order order, Product fishTank, int total) {
    Object[] options = {"Yes",
                        "No"};

    int n = JOptionPane.showOptionDialog(frame,
        "Would you like to buy a tank for your " + total + " fish?",
        "Purchase Suggestion",
        JOptionPane.YES_NO_OPTION,
        JOptionPane.QUESTION_MESSAGE,
        null,
        options,
        options[0]);

    System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                      + total + " fish? - " );

    if (n == 0) {
        Purchase purchase = new Purchase( order, fishTank );
        workingMemory.insert( purchase );
        order.addItem( purchase );
        System.out.println( "Yes" );
    } else {
        System.out.println( "No" );
    }
    return true;
}
```

- Having these functions in the rules file makes the Pet Store example more compact.
- You can have the functions in a file of their own, within the same rules package, or as a static method on a standard Java class, and import them using **`import function my.package.Foo.hello`**.
- **`doCheckout()`** displays a dialog asking users whether they wish to checkout. If they do, focus is set to the **`checkOut`** agenda-group, allowing rules in that group to (potentially) fire.
- **`requireTank()`** displays a dialog asking users whether they wish to buy a tank. If so, a new fish tank **`Product`** is added to the order list in Working Memory.

Report a bug

27.6. PET STORE EXAMPLE: PUTTING ITEMS INTO WORKING MEMORY FROM PETSTORE.DRL

```
// Insert each item in the shopping cart into the Working Memory
rule "Explode Cart"
    agenda-group "init"
    auto-focus true
    salience 10
    dialect "java"
when
    $order : Order( grossTotal == -1 )
    $item : Purchase() from $order.items
then
    insert( $item );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show
items" ).setFocus();
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate"
).setFocus();
end
```

- The first extract fires first because it has the **`auto-focus`** attribute set to **`true`**.
- This rule matches against all orders that do not yet have their **`grossTotal`** calculated . It loops for each purchase item in that order. Some parts of the "Explode Cart" rule should be familiar: the rule name, the salience (suggesting the order for the rules being fired) and the dialect set to **`java`**.
- **`agenda-group init`** defines the name of the agenda group. In this case, there is only one rule in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the next attribute for its chance to fire.
- **`auto-focus true`** ensures that this rule, while being the only rule in the agenda group, can fire when **`fireAllRules()`** is called from the Java code.
- **`kcontext....setFocus()`** sets the focus to the **`show items`** and **`evaluate`** agenda groups in turn, permitting their rules to fire. In practice, you can loop through all items on the order, inserting them into memory, then firing the other rules after each insert.

Report a bug

27.7. PET STORE EXAMPLE: SHOW ITEMS IN THE GUI FROM PETSTORE.DRL

```
rule "Show Items"
    agenda-group "show items"
    dialect "mvel"
when
    $order : Order( )
    $p : Purchase( order == $order )
then
   textArea.append( $p.product + "\n");
end
```

- The **`show items`** agenda-group has only one rule, called "Show Items" (note the difference in case). For each purchase on the order currently in the Working Memory (or Session), it logs details to the text area at the bottom of the GUI. The **`textArea`** variable used for this is a global variables.
- The **`evaluate`** Agenda group also gains focus from the **`Explode Cart`** rule.

Report a bug

27.8. PET STORE EXAMPLE: EVALUATE AGENDA GROUP FROM PETSTORE.DRL

```
// Free Fish Food sample when we buy a Gold Fish if we haven't already
bought
// Fish Food and don't already have a Fish Food Sample
rule "Free Fish Food Sample"
    agenda-group "evaluate"
    dialect "mvel"
when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p )
)
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product
== $p ) )
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p
) )
    $fishFoodSample : Product( name == "Fish Food Sample" );
then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
end

// Suggest a tank if we have bought more than 5 gold fish and don't
already have one
rule "Suggest Tank"
    agenda-group "evaluate"
    dialect "java"
when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p )
)
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name ==
"Gold Fish" ) )
    $fishTank : Product( name == "Fish Tank" )
then
    requireTank(frame, drools.getWorkingMemory(), $order, $fishTank,
$total);
end
```

The rule **`"Free Fish Food Sample"`** will only fire if:

- The store *does not* already have any fish food, *and*
- The store *does not* already have a free fish food sample, *and*
- The store *does* have a Gold Fish in its order.

The rule **`"Suggest Tank"`** will only fire if

- The store *does not*already have a Fish Tank in its order, *and*

- The store *does* have more than five Gold Fish Products in its order.
- If the rule does fire, it calls the **`requireTank()`** function . This shows a Dialog to the user, and adding a Tank to the order and Working Memory if confirmed.
- When calling the *requireTank*() function the rule passes the global *frame* variable so that the function has a handle to the Swing GUI.
- If the rule does fire, it creates a new product (Fish Food Sample), and adds it to the order in Working Memory.

Report a bug

27.9. PET STORE EXAMPLE: DOING THE CHECKOUT EXTRACT FROM PETSTORE.DRL

```
rule "do checkout"
    dialect "java"
    when
    then
        doCheckout(frame, drools.getWorkingMemory());
end
```

- The rule **`"do checkout"`** has no agenda group set and no auto-focus attribute. As such, it is deemed part of the default (MAIN) agenda group. This group gets focus by default when all the rules in agenda-groups that explicitly had focus set to them have run their course.
- There is no LHS to the rule, so the RHS will always call the **`doCheckout()`** function.
- When calling the **`doCheckout()`** function, the rule passes the global **`frame`** variable to give the function a handle to the Swing GUI.
- The **`doCheckout()`** function shows a confirmation dialog to the user. If confirmed, the function sets the focus to the *checkout* agenda-group, allowing the next lot of rules to fire.

Report a bug

27.10. PET STORE EXAMPLE: CHECKOUT RULES FROM PETSTORE.DRL

```
rule "Gross Total"
    agenda-group "checkout"
    dialect "mvel"
when
   $order : Order( grossTotal == -1)
   Number( total : doubleValue )
       from accumulate( Purchase( $price : product.price ), sum( $price )
)
then
    modify( $order ) { grossTotal = total };
    textArea.append( "\ngross total=" + total + "\n" );
```

```
end

rule "Apply 5% Discount"
    agenda-group "checkout"
dialect "mvel"
when
   $order : Order( grossTotal >= 10 && < 20 )
then
   $order.discountedTotal = $order.grossTotal * 0.95;
   textArea.append( "discountedTotal total=" + $order.discountedTotal +
"\n" );
end


rule "Apply 10% Discount"
    agenda-group "checkout"
    dialect "mvel"
when
   $order : Order( grossTotal >= 20 )
then
   $order.discountedTotal = $order.grossTotal * 0.90;
   textArea.append( "discountedTotal total=" + $order.discountedTotal +
"\n" );
end
```

There are three rules in the *checkout* agenda-group:

- **`Gross Total`** accumulates the product prices into a total, puts it into Working Memory, and displays it via the Swing **`JTextArea`** using the **`textArea`** global variable.
- If the gross total is between 10 and 20, **`Apply 5% Discount`** calculates the discounted total and adds it to the Working Memory and displays it in the text area.
- If the gross total is not less than 20, **`Apply 10% Discount`** calculates the discounted total and adds it to the Working Memory and displays it in the text area.

Report a bug

27.11. PET STORE EXAMPLE: RUNNING PETSTORE.JAVA

To use PetStore.java, the following conditions must be met:

1. The **`main()`** method has run and loaded the Rule Base but not yet fired the rules. So far, this is the only code in connection with rules that has been run.
2. A new **`PetStoreUI`** object has been created and given a handle to the Rule Base, for later use.
3. Swing components are deployed and the console waits for user input.

- The file **`PetStore.java`** contains a **`main()`** method, so that it can be run as a standard Java application, either from the command line or via the IDE. This assumes you have your classpath set correctly.

- The first screen that appears is the Pet Store Demo. It has a list of available products, an empty list of selected products, checkout and reset buttons, and an empty system messages area.

Pressing the "Checkout" button fires the business rules:

1. Method **`CheckOutCallBack.checkout()`** is called by the Swing class waiting for the click on the "Checkout" button. This inserts the data from the **`TableModel`** object and inserts it into the Session's Working Memory. It then fires the rules.

2. The first rule to fire will be the one with **`auto-focus`** set to true. It loops through all the products in the cart, ensures that the products are in the Working Memory, and then gives the **`Show Items`** and **`Evaluation`** agenda groups a chance to fire. The rules in these groups add the contents of the cart to the text area (at the bottom of the window), decide whether or not to give the user free fish food, and to ask us whether they want to buy a fish tank.

Report a bug

27.12. PET STORE EXAMPLE: THE DO CHECKOUT RULE

1. The *Do Checkout* rule is part of the default (MAIN) agenda group. It always calls the *doCheckout() function*which displays a 'Would you like to Checkout?' dialog box.

2. The **`doCheckout()`** function sets the focus to the**`checkout`** agenda-group, giving the rules in that group the option to fire.

3. The rules in the **`checkout`** agenda-group display the contents of the cart and apply the appropriate discount.

4. *Swing then waits for user input* to either checkout more products (and to cause the rules to fire again), or to close the GUI.

Report a bug

CHAPTER 28. SUDOKU EXAMPLE

28.1. SUDOKU EXAMPLE: LOADING THE EXAMPLE

Procedure 28.1. Task

1. Open **`sudoku.drl`** in the IDE.

2. Execute **`java org.drools.examples.DroolsExamplesApp`** and click on **`SudokuExample`**. The window contains an empty grid, but the program comes with a number of grids stored internally which can be loaded and solved.

3. Click on **File** → **Samples** → **Simple** to load one of the examples. All buttons are disabled until a grid is loaded. Loading the **`Simple`** example fills the grid according to the puzzle's initial state.

4. Click on the **`Solve`** button and the JBoss Rules engine will fill out the remaining values. The buttons will be inactive again.

5. Alternatively, click on the **`Step`** button to see the next digit found by the rule set. The Console window will display detailed information about the rules which are executing to solve the step in a readable format like the example below:

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```

6. Click on the **`Dump`** button to see the state of the grid. The cells show either the established value or the remaining possible candidates. See the example below:

```
Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col: 5
Col: 6     Col: 7     Col: 8
Row 0:   2 4  7 9   2 456        4567 9   23 56  9  --- 5 ---  ---
1 ---    3  67 9  --- 8 ---     4 67
Row 1:  12    7 9  --- 8 ---  1    67 9   23  6  9  --- 4 ---   23
67    1 3  67 9    3  67 9  --- 5 ---
Row 2:  1  4  7 9  1  456     --- 3 ---      56 89      5 78
5678   --- 2 ---     4 67 9  1  4 67
Row 3:  1234       12345      1  45      12  5  8   --- 6 ---   2
5 78       5 78      45 7    --- 9 ---
Row 4:  --- 6 ---  --- 7 ---      5      --- 4 ---   2  5  8   ---
9 ---      5  8   --- 1 ---  --- 3 ---
Row 5:  --- 8 ---  12 45      1  45   9  12  5      --- 3 ---   2
5 7        567       4567     2 4 67
Row 6:  1 3   7    1 3  6     --- 2 ---    3 56 8       5  8     3
56 8   --- 4 ---    3 567 9  1    678
Row 7:  --- 5 ---  1 34 6     1  4 678     3  6 8   --- 9 ---    34
6 8   1 3  678   --- 2 ---  1    678
Row 8:    34       --- 9 ---     4 6 8   --- 7 ---  --- 1 ---
23456 8     3 56 8     3 56          6 8
```

Report a bug

28.2. SUDOKU EXAMPLE: DEBUGGING A BROKEN EXAMPLE

Procedure 28.2. Task

1. Open **`sudoku.drl`** in your IDE.

2. Click on **File** → **Samples** → **!DELIBERATLEY BROKEN!**. The JBoss Rules engine will inspect the grid and produce the following output:

   ```
   cell [0,8]: 5 has a duplicate in row 0
   cell [0,0]: 5 has a duplicate in row 0
   cell [6,0]: 8 has a duplicate in col 0
   cell [4,0]: 8 has a duplicate in col 0
   Validation complete.
   ```

3. Click on the **`Solve`** button to apply the solving rules to this invalid grid. These rules use the values of the cells for problem solving. The rules detecting these situations insert a Setting fact including the solution value for the specified cell. This fact removes the incorrect value from all cells in the group.

Report a bug

28.3. SUDOKU EXAMPLE: JAVA SOURCE AND RULES

- The Java source code can be found in the **`/src/main/java/org/drools/examples/sudoku`** directory, with the two DRL files defining the rules located in the **`/src/main/rules/org/drools/examples/sudoku`** directory.

- The package **`org.drools.examples.sudoku.swing`** contains a set of classes which implement a framework for Sudoku puzzles. This package does not have any dependencies on the JBoss Rules libraries.

- **`SudokuGridModel`** defines an interface which can be implemented to store a Sudoku puzzle as a 9x9 grid of **`Cell`** objects.

- **`SudokuGridView`** is a Swing component which can visualize any implementation of **`SudokuGridModel`**.

- **`SudokuGridEvent`** and **`SudokuGridListener`** are used to communicate state changes between the model and the view. Events are fired when a cell's value is resolved or changed.

- **`SudokuGridSamples`** provides a number of partially filled Sudoku puzzles for demonstration purposes.

- The package **`org.drools.examples.sudoku.rules`** contains a utility class with a method for compiling DRL files.

- The package **`org.drools.examples.sudoku`** contains a set of classes implementing the elementary **`Cell`** object and its various aggregations. It contains the **`CellFile`** subtypes **`CellRow`**, **`CellCol`** and **`CellSqr`**, all of which are subtypes of **`CellGroup`**.

Report a bug

28.4. SUDOKU EXAMPLE: CELL OBJECTS

- `Cell` and `CellGroup` are subclasses of `SetOfNine`, which provides a property `free` with the type `Set<Integer>`. For a `Cell` it represents the individual candidate set. For a `CellGroup` the set is the union of all candidate sets of its cells, or the set of digits that still need to be allocated.
- You can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set. For example, you can create a list of `Cell` objects with 81 `Cell` and 27 `CellGroup` objects. You can also combine the linkage provided by the `Cell` properties `cellRow`, `cellCol`, `cellSqr` and the `CellGroup` property `cells`.

Report a bug

28.5. SUDOKU EXAMPLE: CLASSES AND OBJECTS

- An object belonging to the `Setting` class is used for triggering the operations that accompany the allocation of a value. The presence of a `Setting` fact is used in all rules that should detect changes in the process. This is to avoid reactions to inconsistent intermediary states.
- An object of class `Stepping` is used in a low priority rule to execute an emergency halt when a "Step" ends unexpectedly. This indicates that the puzzle cannot be solved by the program.
- The class `org.drools.examples.sudoku.SudokuExample` implements a Java application combining the above components.

Report a bug

28.6. SUDOKU EXAMPLE: VALIDATE.DRL

- Sudoku Validator Rules (validate.drl) detect duplicate numbers in cell groups. They are combined in an agenda group which enables them to be activated after loading a puzzle.
- The three rules `duplicate in cell...` are very similar. The first pattern locates a cell with an allocated value. The second pattern pulls in any of the three cell groups the cell belongs to. The final pattern finds a cell with the same value as the first cell and in the same row, column or square, respectively.
- Rule `terminate group` fires last. It prints a message and calls halt.

Report a bug

28.7. SUDOKU EXAMPLE: SUDOKU.DRL

- There are three types of solving rules in Sudoku.drl: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

- The rules **`set a value`**, **`eliminate a value from Cell`** and **`retract setting`** depend on the presence of a **`Setting`** object.

- **`Set a value`** handles the assignment to the cell and the operations for removing the value from the "free" sets of the cell's three groups. Also, it decrements a counter that, when zero, returns control to the Java application that has called **`fireUntilHalt()`**.

- **`Eliminate a value from Cell`** reduces the candidate lists of all cells that are related to the newly assigned cell.

- **`Retract setting`** retracts the triggering **`Setting`** fact when all of the eliminations have been made.

- There are just two rules that detect a situation where an allocation of a number to a cell is possible. Rule **`single`** fires for a **`Cell`** with a candidate set containing a single number. Rule **`hidden single`** fires when there is a cell containing a candidate but this candidate is absent from all other cells in one of the groups the cell belongs to. Both rules create and insert a **`Setting`** fact.

- Rules from the largest group of rules implement, singly or in groups of two or three, various solving techniques, as they are employed when solving Sudoku puzzles manually.

- Rule **`naked pair`** detects two identical candidate sets in two cells of a group. These two values may be removed from all other candidate sets of that group.

- In **`hidden pair in`** rules, the rules look for a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of this group. This means that all other candidates can be eliminated from the two cells harbouring the hidden pair.

- A pair of rules deals with **`X-wings`** in rows and columns. When there are only two possible cells for a value in each of two different rows (or columns) and these candidates are in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. The conditions **`same`** or **`only`** result in patterns with suitable constraints or prefixed with **`not`**.

- The rule pair **`intersection removal...`** is based on the restricted occurrence of a number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column. It can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside the square and within the same cell file.

- To solve very difficult grids, the rule set would need to be extended with more complex rules. (Ultimately, there are puzzles that cannot be solved except by trial and error.)

Report a bug

CHAPTER 29. NUMBER GUESS EXAMPLE

29.1. NUMBER GUESS EXAMPLE: LOADING THE EXAMPLE

```
final KnowledgeBuilder kbuilder =
KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.drl",

ShoppingExample.class ),
              ResourceType.DRL );
kbuilder.add( ResourceFactory.newClassPathResource( "NumberGuess.rf",

ShoppingExample.class ),
              ResourceType.DRF );

final KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
```

- The Number Guess example located in **`NumberGuess.drl`** shows the use of Rule Flow, a way of controlling the order in which rules are fired. It is loaded as shown above.

Report a bug

29.2. NUMBER GUESS EXAMPLE: STARTING THE RULEFLOW

```
final StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();

KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
"log/numberguess");

ksession.insert( new GameRules( 100, 5 ) );
ksession.insert( new RandomNumber() );
ksession.insert( new Game() );

ksession.startProcess( "Number Guess" );
ksession.fireAllRules();

logger.close();

ksession.dispose();
```

- The above code demonstrates the creation of the package and the loading of the rules (using the **`add()`** method).

- There is an additional line to add the Rule Flow (**`NumberGuess.rf`**), which provides the option of specifying different rule flows for the same Knowledge Base.

- Once the Knowledge Base is created it can be used to obtain a Stateful Session. The facts are then inserted.

Report a bug

29.3. NUMBER GUESS EXAMPLE: CLASSES AND METHODS

NOTE

The Number Guess Example classes are all contained within the **`NumberGuessExample.java`** file.

- Class **`GameRules`** provides the maximum range and the number of guesses allowed.
- Class **`RandomNumber`** automatically generates a number between 0 and 100 and makes it available to the rules. It does so by insertion via the **`getValue()`** method.
- Class **`Game`** keeps track of the number of guesses made.
- To start the process, the **`startProcess()`** method is called.
- To execute the rules, the **`fireAllRules()`** method is called.
- To clear the Working Memory session, the **`dispose()`** method is called.

Report a bug

29.4. NUMBER GUESS EXAMPLE: OBSERVING THE RULEFLOW

Procedure 29.1. Task

1. Open the **`NumberGuess.rf`** file in the Drools IDE. A diagram will appear that works much like a standard flowchart.
2. To edit the diagram, use the menu of available components to the left of the diagram in the IDE. This is called a *palette*.
3. Save the diagram in XML. (If installed, you can utilise XStream to do this.)
4. If it is not already open, ensure that the Properties View is visible in the IDE. It can be opened by clicking **Window** → **Show View** → **Other** where you can select the **`Properties`** view. If you do this *before* you select an item on the rule flow (or click on the blank space in the rule flow) you will see the properties. You can use these properties to identify processes and observe changes.

Report a bug

29.5. NUMBER GUESS EXAMPLE: RULEFLOW NODES

In the Number Guess RuleFlow there are several node types:

- The Start node (white arrow in a green circle) and the End node (red box) mark beginning and end of the rule flow.

- A Rule Flow Group box (yellow, without an icon) represents a Rule Flow Groups defined in the rules (DRL) file. For example, when the flow reaches the Rule Flow Group "Too High", only those rules marked with an attribute of **`ruleflow-group "Too High"`** can potentially fire.

- Action nodes (yellow cog-shaped icon) perform standard Java method calls. Most action nodes in this example call **`System.out.println()`**, indicating the program's progress to the user.

- Split and Join Nodes (blue ovals, no icon) such as "Guess Correct?" and "More guesses Join" mark places where the flow of control can split and rejoin.

- Arrows indicate the flow between the various nodes.

Report a bug

29.6. NUMBER GUESS EXAMPLE: FIRING RULES AT A SPECIFIC POINT IN NUMBERGUESS.DRL

```
rule "Get user Guess"
    ruleflow-group "Guess"
    no-loop
    when
        $r : RandomNumber()
        rules : GameRules( allowed : allowedGuesses )
        game : Game( guessCount < allowed )
        not ( Guess() )
    then
        System.out.println( "You have " + ( rules.allowedGuesses -
game.guessCount )
                            + " out of " + rules.allowedGuesses
                            + " guesses left.\nPlease enter your guess
from 0 to "
                            + rules.maxRange );
        br = new BufferedReader( new InputStreamReader( System.in ) );
        i = br.readLine();
        modify ( game ) { guessCount = game.guessCount + 1 }
        insert( new Guess( i ) );
end
```

- The various nodes in combination with the rules make the Number Guess game work. For example, the "Guess" Rule Flow Group allows only the rule "Get user Guess" to fire, because only that rule has a matching attribute of **`ruleflow-group "Guess"`**.

- The LHS section (after **`when`**) of the rule states that it will be activated for each **`RandomNumber`** object inserted into the Working Memory where **`guessCount`** is less than **`allowedGuesses`** from the **`GameRules`** object and where the user has not guessed the correct number.

- The RHS section (or consequence, after **`then`**) prints a message to the user and then awaits user input from **`System.in`**. After obtaining this input (the **`readLine()`** method call blocks until the return key is pressed) it modifies the guess count and inserts the new guess, making both available to the Working Memory.

- The package declares the dialect as MVEL and various Java classes are imported.

In total, there are five rules in this file:

1. Get User Guess, the Rule examined above.
2. A Rule to record the highest guess.
3. A Rule to record the lowest guess.
4. A Rule to inspect the guess and retract it from memory if incorrect.
5. A Rule that notifies the user that all guesses have been used up.

Report a bug

29.7. NUMBER GUESS EXAMPLE: VIEWING RULEFLOW CONSTRAINTS

Procedure 29.2. Task

1. In the IDE, go to the **`Properties`** view and open the Constraints Editor by clicking on the "Constraints" property line.
2. Click on the **`Edit`** button beside **`To node Too High`** to open the dialogue which will present you with various options. The values in the **`Textual Editor`** window follow the standard rule format for the LHS and can refer to objects in Working Memory. The consequence (RHS) is that the flow of control follows this node (that is, **`To node Too High`**) if the LHS expression evaluates to true.

Report a bug

29.8. NUMBER GUESS EXAMPLE: CONSOLE OUTPUT

```
You have 5 out of 5 guesses left.
Please enter your guess from 0 to 100
50
Your guess was too high
You have 4 out of 5 guesses left.
Please enter your guess from 0 to 100
25
Your guess was too low
You have 3 out of 5 guesses left.
Please enter your guess from 0 to 100
37
Your guess was too low
You have 2 out of 5 guesses left.
Please enter your guess from 0 to 100
44
Your guess was too low
You have 1 out of 5 guesses left.
Please enter your guess from 0 to 100
47
Your guess was too low
```

```
You have no more guesses
The correct guess was 48
```

- Since the file **`NumberGuess.java`** contains a **`main()`** method, it can be run as a standard Java application, either from the command line or via the IDE. A typical game might result in the interaction above. The numbers in bold were typed in by the user.
- The **`main()`** method of **`NumberGuessExample.java`** loads a Rule Base, creates a Stateful Session and inserts **`Game`**, **`GameRules`** and **`RandomNumber`** (containing the target number) objects into it. The method also sets the process flow to be used and fires all rules. Control passes to the RuleFlow.
- The RuleFlow file **`NumberGuess.rf`** begins at the "Start" node.
- At the Guess node, the appropriate Rule Flow Group ("Get user Guess") is enabled. In this case the Rule "Guess" (in the **`NumberGuess.drl`** file) is triggered. This rule displays a message to the user, takes the response, and puts it into Working Memory. Flow passes to the next Rule Flow Node.
- At the next node, "Guess Correct", constraints inspect the current session and decide which path to take.

  If the guess in step 4 was too high or too low, flow proceeds along a path which has an action node with normal Java code printing a suitable message and a Rule Flow Group causing a highest guess or lowest guess rule to be triggered. Flow passes from these nodes to step 6.

  If the guess in step 4 was right, we proceed along the path towards the end of the RuleFlow. Before this, an action node with normal Java code prints a statement "you guessed correctly". There is a join node here (just before the Rule Flow end) so the no-more-guesses path (step 7) can also terminate the RuleFlow.
- Control passes as per the RuleFlow via a join node to a "guess incorrect" RuleFlow Group (triggering a rule to retract a guess from Working Memory) and onto the "More guesses" decision node.
- The "More guesses" decision node (on the right hand side of the rule flow) uses constraints, again looking at values that the rules have put into the working memory, to decide if the user has more guesses and. If so, it moves to step 3. If not, the user proceeds to the end of the RuleFlow via a RuleFlow Group that triggers a rule stating "you have no more guesses".
- The loop over steps 3 to 7 continues until the number is guessed correctly or the user runs out of guesses.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 5.3.1-73.400	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.3.1-73	**Tue Feb 05 2013**	**David Le Sage**

Built from Content Specification: 11912, Revision: 371716 by dlesage