



# Drools

business rule management system

## tutorialspoint

SIMPLYEASYLEARNING

[www.tutorialspoint.com](http://www.tutorialspoint.com)



<https://www.facebook.com/tutorialspointindia>



<https://twitter.com/tutorialspoint>

## About the Tutorial

---

Drools is a business logic integration platform written in Java. This is a quick tutorial on how to use Drools in a business environment.

## Audience

---

This tutorial should be useful for all those readers who wish to define rules in their applications to integrate business logic in a standard way.

## Prerequisites

---

Since Drools is written in Java, you should have a working knowledge of Java before proceeding further.

## Copyright & Disclaimer

---

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at [contact@tutorialspoint.com](mailto:contact@tutorialspoint.com)

# Table of Contents

---

About the Tutorial .....	i
Audience .....	i
Prerequisites .....	i
Copyright & Disclaimer .....	i
Table of Contents .....	ii
<b>1. INTRODUCTION.....</b>	<b>1</b>
What is Drools? .....	1
What is a Rule Engine? .....	1
What is a Rule? .....	2
Pattern Matching .....	2
Advantages of a Rule Engine.....	3
<b>2. PLUGIN IN ECLIPSE.....</b>	<b>4</b>
Step 1: Download the Binaries.....	4
Step 2: Install the Software .....	5
<b>3. DROOLS RUNTIME.....</b>	<b>8</b>
<b>4. CREATE A DROOLS PROGRAM.....</b>	<b>10</b>
<b>5. FREQUENTLY USED TERMS.....</b>	<b>13</b>
Rules.....	13
Facts .....	13
Session .....	13
Agenda .....	13
Activations .....	13
<b>6. RULES WRITING .....</b>	<b>14</b>
Knowledge Base .....	15

Knowledge Session .....	15
Stateless Knowledge Session .....	15
Stateful Knowledge Session .....	15
Knowledge Builder .....	16
<b>7. RULE SYNTAX.....</b>	<b>17</b>
Conditions in Rules .....	17
Variables in Rules .....	17
Comments in Rules .....	17
Global Variables .....	17
Functions in Rules.....	18
Dialect .....	18
Saliency .....	18
Rule Consequence Keywords.....	19
<b>8. SAMPLE DROOLS PROGRAM .....</b>	<b>20</b>
DRL Files .....	22
Call an External Function from a DRL File.....	27
<b>9. DEBUGGING .....</b>	<b>29</b>
Using the Debug Perspective in Eclipse.....	32

# 1. INTRODUCTION

Any Java enterprise level application can be split into three parts:

1. UI – User Interface (Frontend)
2. Service layer which is in turn connected to a database
3. Business layer

We have a number of frameworks that handle the UI and service layer together, for example, Spring and Struts. Yet, we did not have a standard way to handle the business logic until Drools came into existence.

## What is Drools?

---

Drools is a **Business Logic integration Platform (BLiP)**. It is written in Java. It is an open source project that is backed by JBoss and Red Hat, Inc. It extends and implements the Rete Pattern matching algorithm.

In layman's terms, Drools is a collection of tools that allow us to separate and reason over logic and data found within business processes. The two important keywords we need to notice are **Logic** and **Data**.

Drools is split into two main parts: **Authoring** and **Runtime**.

- **Authoring:** Authoring process involves the creation of Rules files (.DRL files).
- **Runtime:** It involves the creation of working memory and handling the activation.

## What is a Rule Engine?

---

Drools is Rule Engine or a Production Rule System that uses the rule-based approach to implement an Expert System. Expert Systems are knowledge-based systems that use knowledge representation to process acquired knowledge into a knowledge base that can be used for reasoning.

A Production Rule System is Turing complete with a focus on knowledge representation to express propositional and first-order logic in a concise, non-ambiguous and declarative manner.

The brain of a Production Rules System is an **Inference Engine** that can scale to a large number of rules and facts. The Inference Engine matches facts and data against Production Rules – also called **Productions** or just **Rules** – to infer conclusions which result in actions.

A Production Rule is a two-part structure that uses first-order logic for reasoning over knowledge representation. A business rule engine is a software system that executes one or more business rules in a runtime production environment.

A Rule Engine allows you to define "**What to Do**" and not "**How to do it.**"

## What is a Rule?

Rules are pieces of knowledge often expressed as, "*When* some conditions occur, *then* do some tasks."

```
When
    <Condition is true>
Then
    <Take desired Action>
```

The most important part of a Rule is its **when** part. If the **when** part is satisfied, the **then** part is triggered.

```
rule <rule_name>
    <attribute> <value>
    when
        <conditions>
    then
        <actions>
end
```

## Pattern Matching

The process of matching the new or existing facts against Production Rules is called Pattern Matching, which is performed by the Inference Engine. There are a number of algorithms used for Pattern Matching including:

- Linear
- Rete
- Treat
- Leaps

Drools Implements and extends the Rete Algorithm. The Drools Rete implementation is called ReteOO, signifying that Drools has an enhanced and optimized implementation of the Rete algorithm for object-oriented systems.

## Advantages of a Rule Engine

---

### **Declarative Programming**

Rules make it easy to express solutions to difficult problems and get the solutions verified as well. Unlike codes, Rules are written in less complex language; Business Analysts can easily read and verify a set of rules.

### **Logic and Data Separation**

The data resides in the Domain Objects and the business logic resides in the Rules. Depending upon the kind of project, this kind of separation can be very advantageous.

### **Speed and Scalability**

The Rete OO algorithm on which Drools is written is already a proven algorithm. With the help of Drools, your application becomes very scalable. If there are frequent change requests, one can add new rules without having to modify the existing rules.

### **Centralization of Knowledge**

By using Rules, you create a repository of knowledge (a knowledge base) which is executable. It is a single point of truth for business policy. Ideally, Rules are so readable that they can also serve as documentation.

### **Tool Integration**

Tools such as Eclipse provide ways to edit and manage rules and get immediate feedback, validation, and content assistance. Auditing and debugging tools are also available.

## 2. PLUGIN IN ECLIPSE

Here are the prerequisites to install Drools Plugin:

- Java 1.5 (or higher) SE JDK
- Eclipse 4.2 (or any version) and the Drools plugin








As Drools is a BRMS (Business Rule Management System) written in Java, we will be covering how to add the desired plugins in this section. Considering maximum Java users use Eclipse, let's see how to add the Drools 5.x.0 plugin in Eclipse.

### Step 1: Download the Binaries

Download the binaries from the following link:

<http://download.jboss.org/drools/release/5.3.0.Final/>

#### Index of /drools/release/5.3.0.Final

<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
 <a href="#">Parent Directory</a>		-	
 <a href="#">drools-distribution-5.3.0.Final.zip</a>	16-Nov-2011 07:11	71M	
 <a href="#">drools-osgi-bundles-distribution-5.3.0.Final.zip</a>	21-Oct-2011 08:16	34M	
 <a href="#">drools-planner-distribution-5.3.0.Final.zip</a>	21-Oct-2011 08:18	24M	
 <a href="#">droolsjbpm-integration-distribution-5.3.0.Final.zip</a>	21-Oct-2011 08:23	77M	
 <a href="#">droolsjbpm-tools-distribution-5.3.0.Final.zip</a>	21-Oct-2011 08:26	56M	
 <a href="#">guvnor-distribution-5.3.0.Final.zip</a>	21-Oct-2011 08:48	364M	

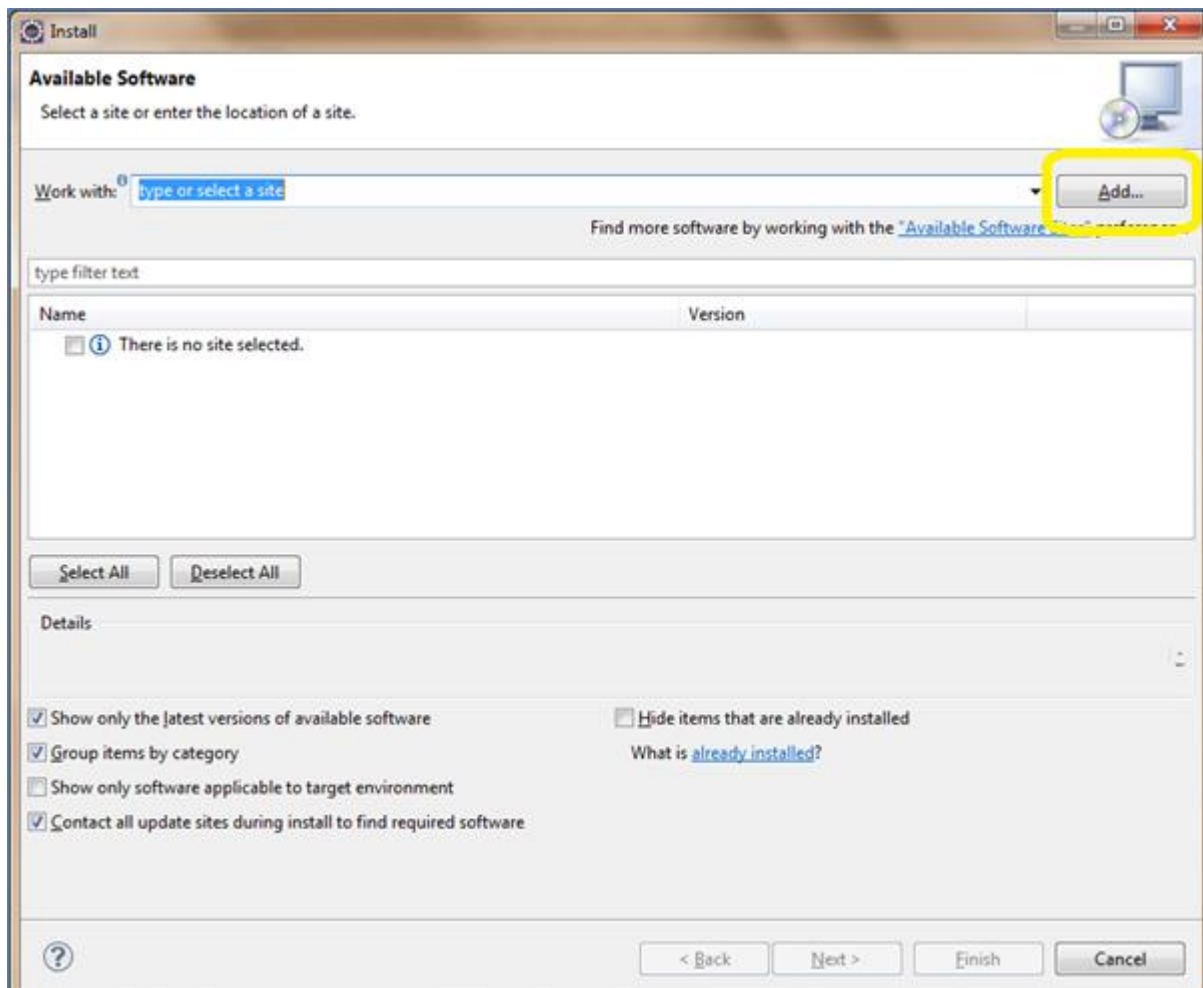
Apache Server at download.jboss.org Port 80

After the download is complete, extract the files to your hard disk.

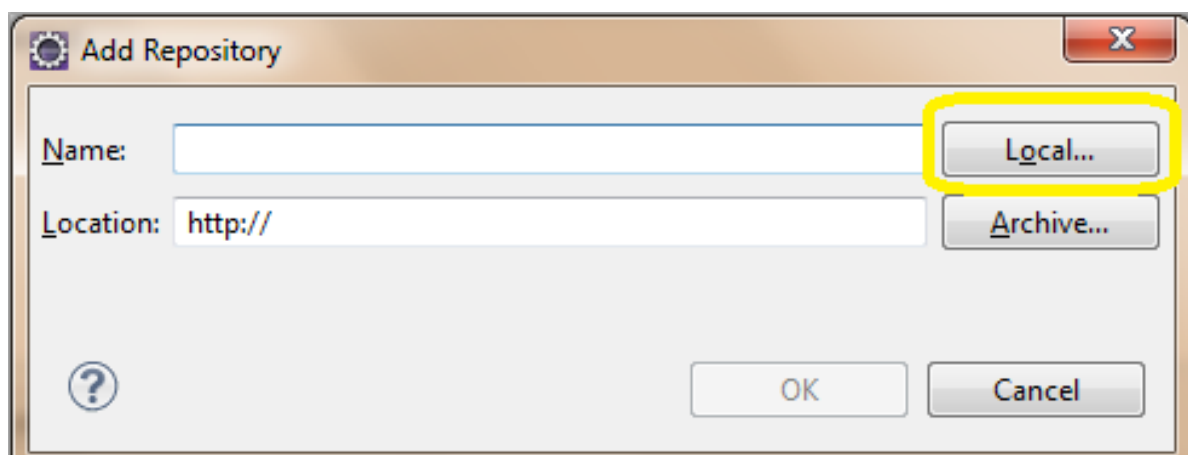


## Step 2: Install the Software

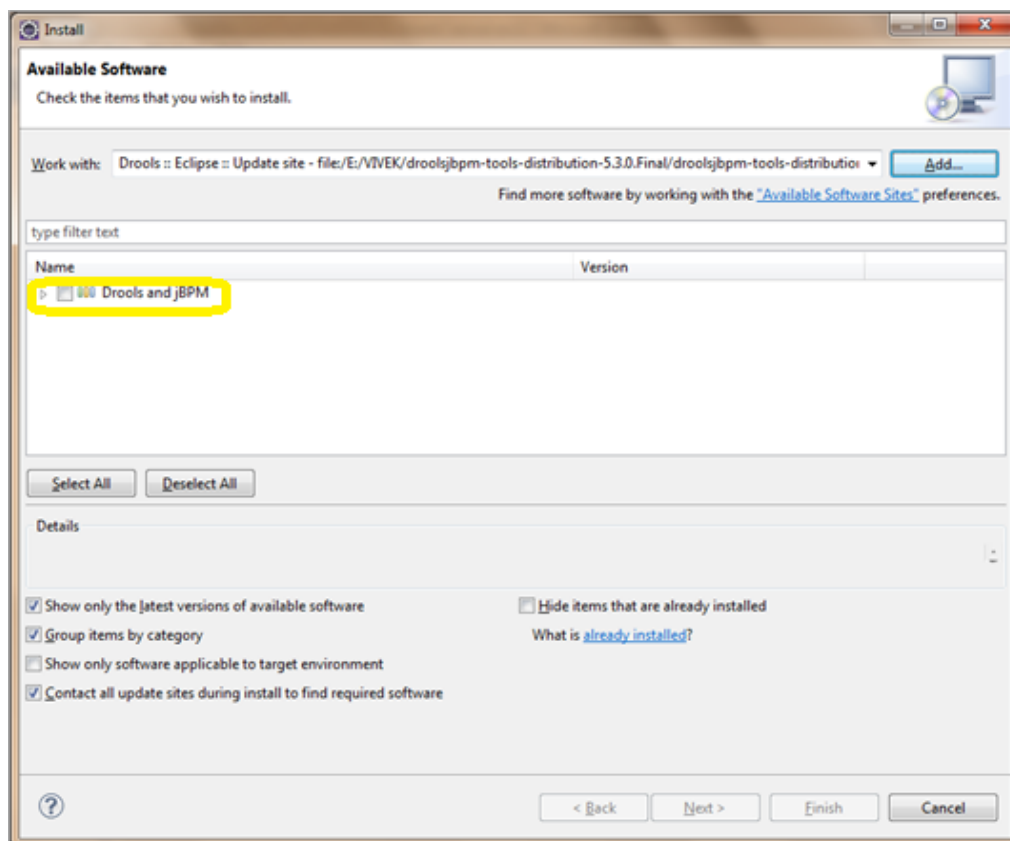
Launch Eclipse and go to help→install new software. Click on Add as shown in the following screenshot.



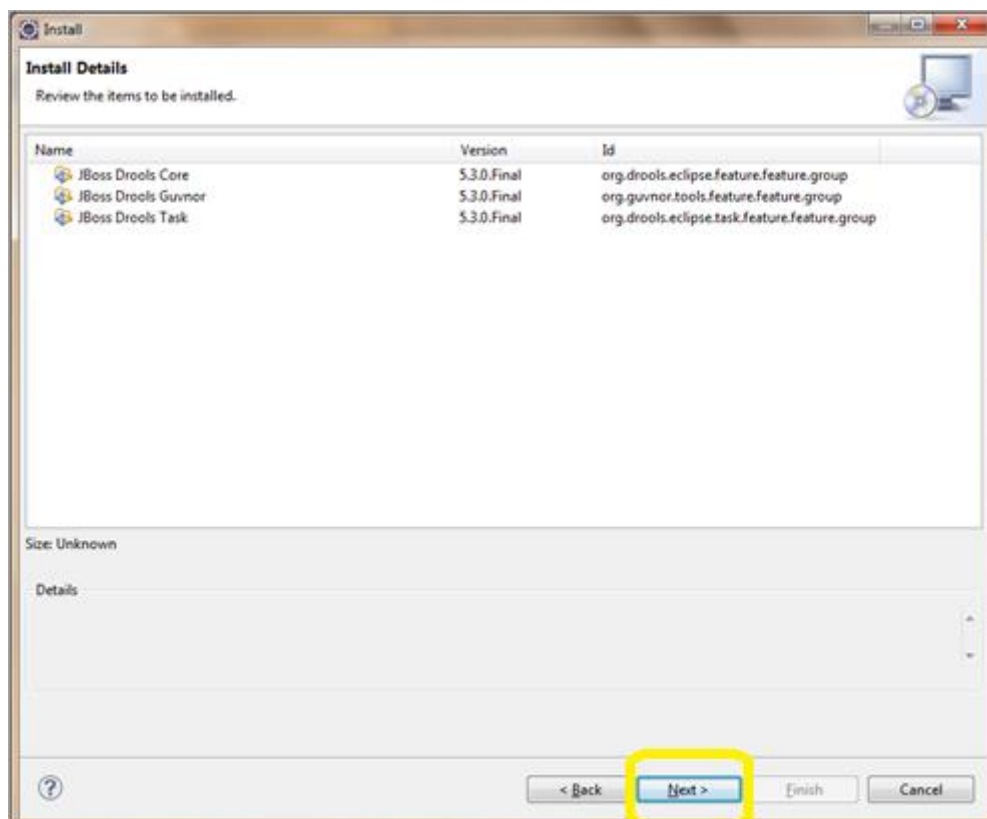
Thereafter, click on Local as shown here and select ".../binaries/org.drools.updatesite".



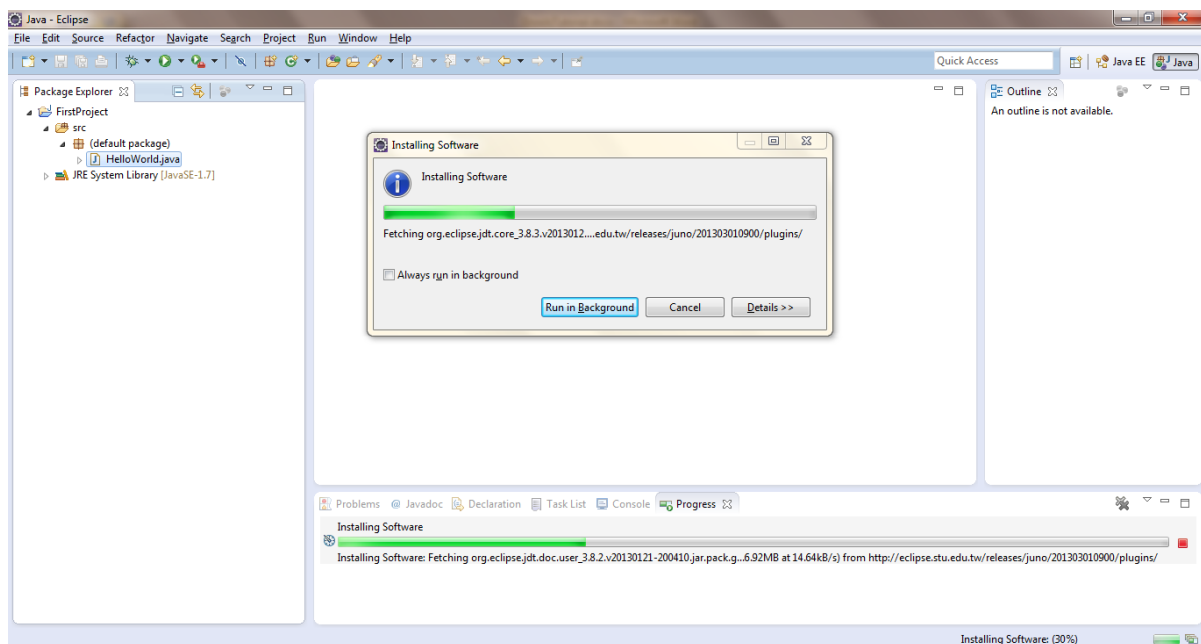
Select Drools and jBPM and click Next.



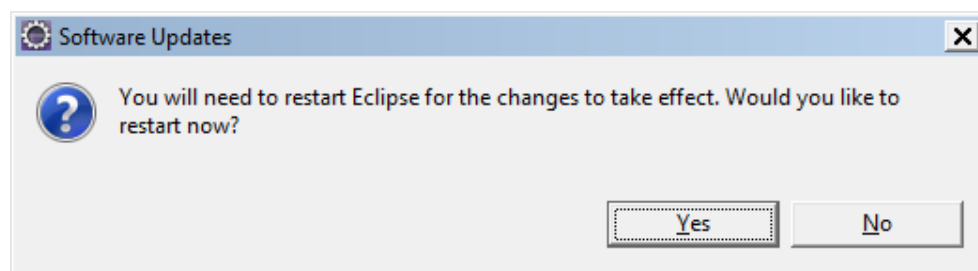
Again click Next. Thereafter, accept the terms and license agreement and click Finish.



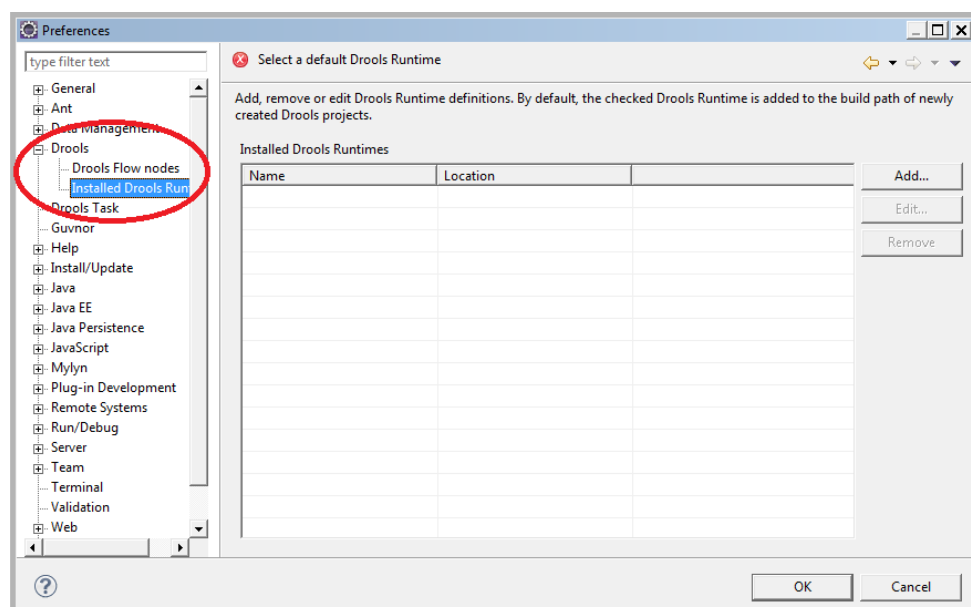
Upon clicking Finish, the software installation starts:



Post successful installation, you will get the following dialog box:



Click on yes. Once Eclipse restarts, go to Windows → Preferences.

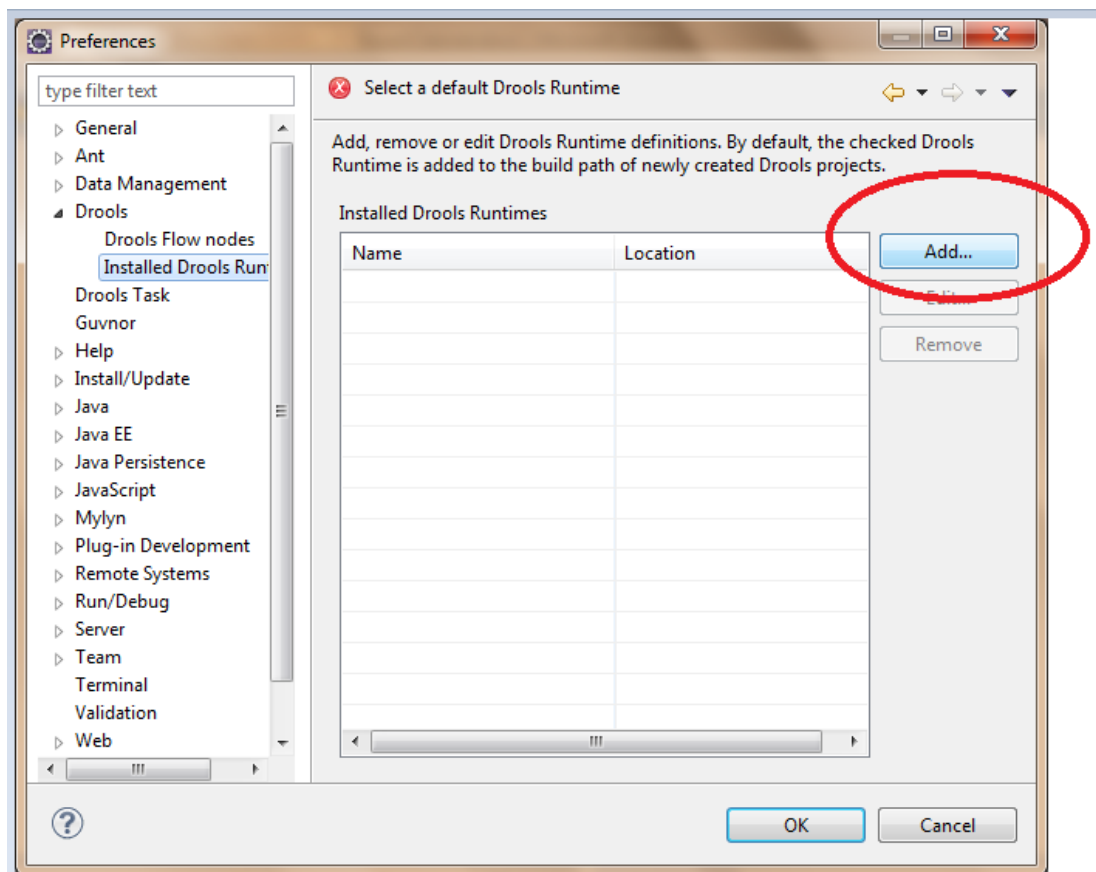


You can see Drools under your preferences. Your Drools plugin installation is complete now.

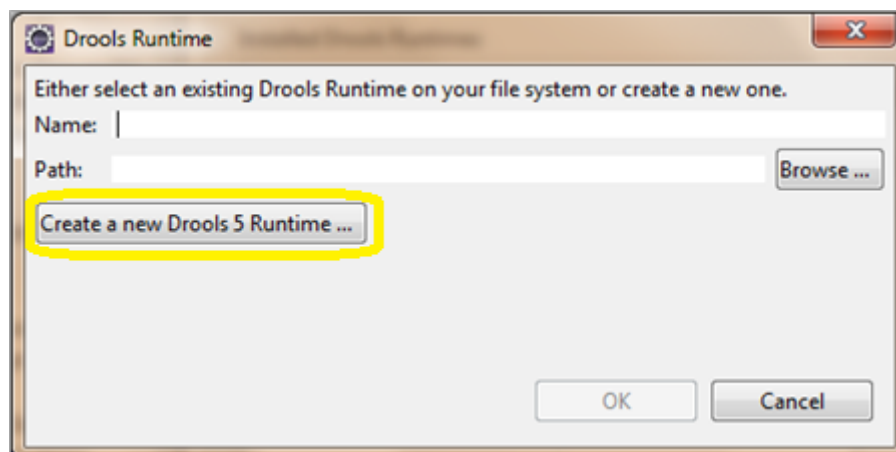
### 3. DROOLS RUNTIME

Drools Runtime is required to instruct the editor to run the program with specific version of Drools jar. You can run your program/application with different Drools Runtime.

Click on Windows → Preference → Drools → Installed Drools Runtime. Then click on Add as shown in the following screenshot.



Thereafter, click on Create a new Drools Runtime as shown here.

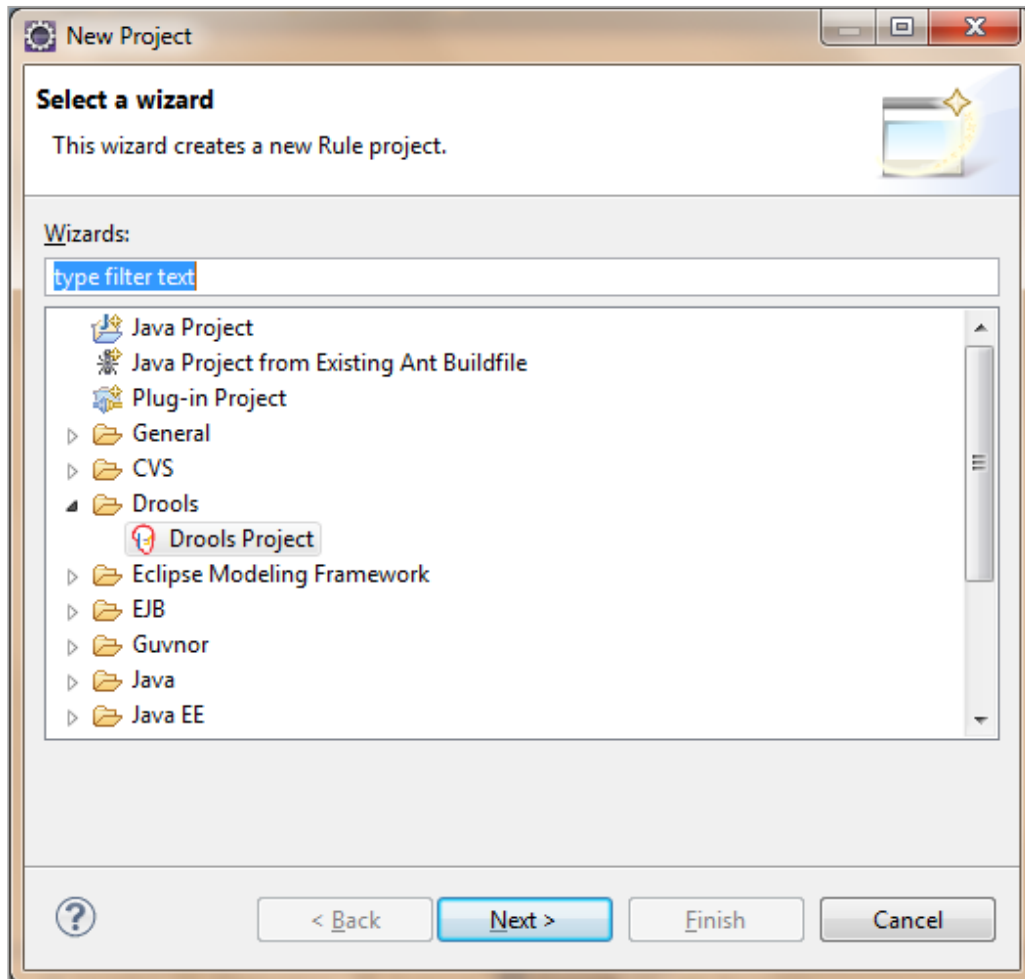


Enter the path till the binaries folder where you have downloaded the [droolsjbpm-tools-distribution-5.3.0.Final.zip](#)

Click on OK and provide a name for the Drools Runtime. The Drools runtime is now created.

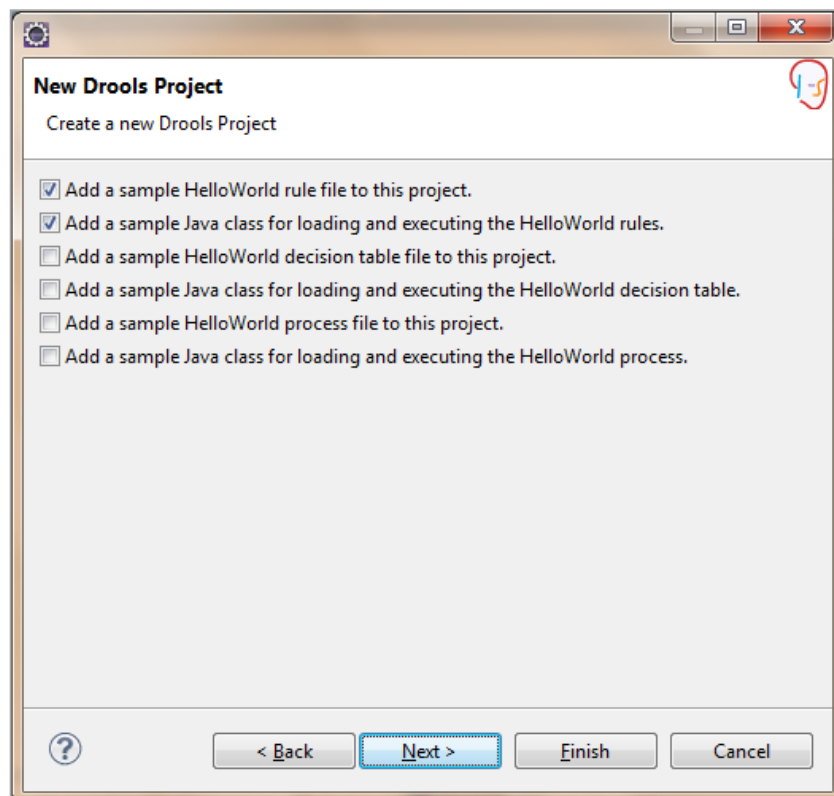
## 4. CREATE A DROOLS PROGRAM

To create a basic Drools program, open Eclipse. Go to Fileb→ New → Project.



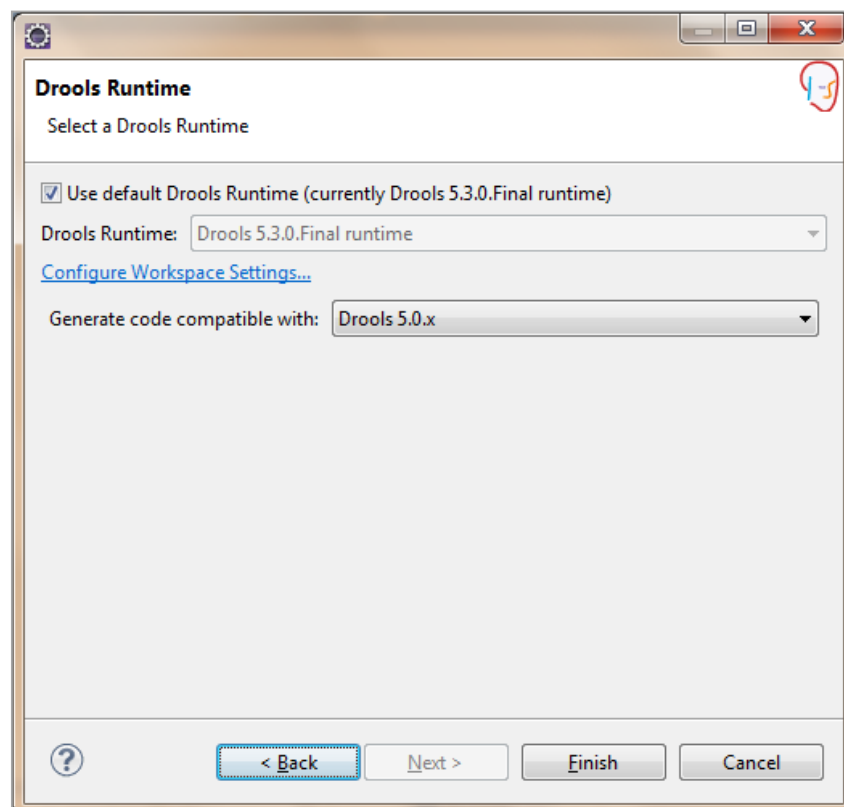
Select Drools Project. Give a suitable name for the project. For example, DroolsTest.

The next screen prompts you to select some files which you want in your first Drools project.

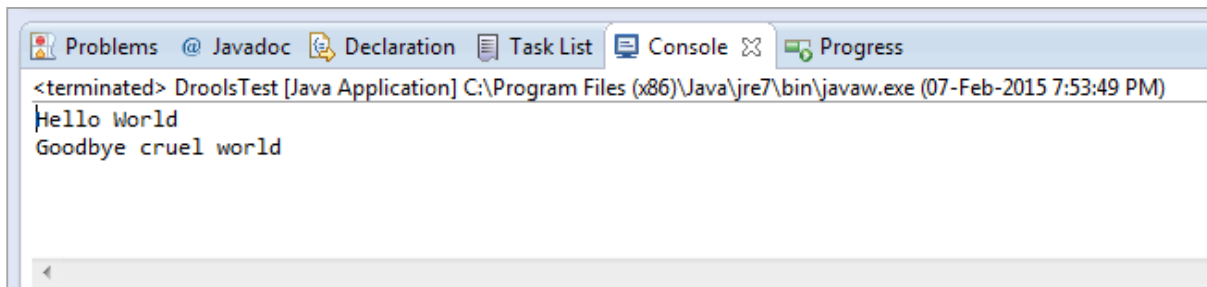


Select the first two files. The first file is a .drl file (Drools Rule File) and the second file is a Java class for loading and executing the HelloWorld rule.

Click on Next → Finish.



Once you click on Finish, a <DroolsTest> project is created in your workspace. Open the Java class and then right-click and run as Java application. You would see the output as shown here:



```
<terminated> DroolsTest [Java Application] C:\Program Files (x86)\Java\jre7\bin\javaw.exe (07-Feb-2015 7:53:49 PM)
Hello World
Goodbye cruel world
```

Next, we will discuss the terms frequently used in a Rule Engine.



# 5. FREQUENTLY USED TERMS

## Rules

---

The heart of the Rules Engine where you specify conditions (if 'a' then 'b').

## Facts

---

Facts are the data on which the rules will act upon. From Java perspective, Facts are the POJO (Plain Old Java Object).

## Session

---

A Knowledge Session in Drools is the core component to fire the rules. It is the knowledge session that holds all the rules and other resources. A Knowledge Session is created from the KnowledgeBase.

For the rules engine to work, Facts are inserted into the session and when a condition is met, the subsequent rule gets fired. A Session is of two types:

- Stateless Knowledge Session
- Stateful Knowledge Session

## Agenda

---

It's a logical concept. The agenda is the logical place where activations are waiting to be fired.

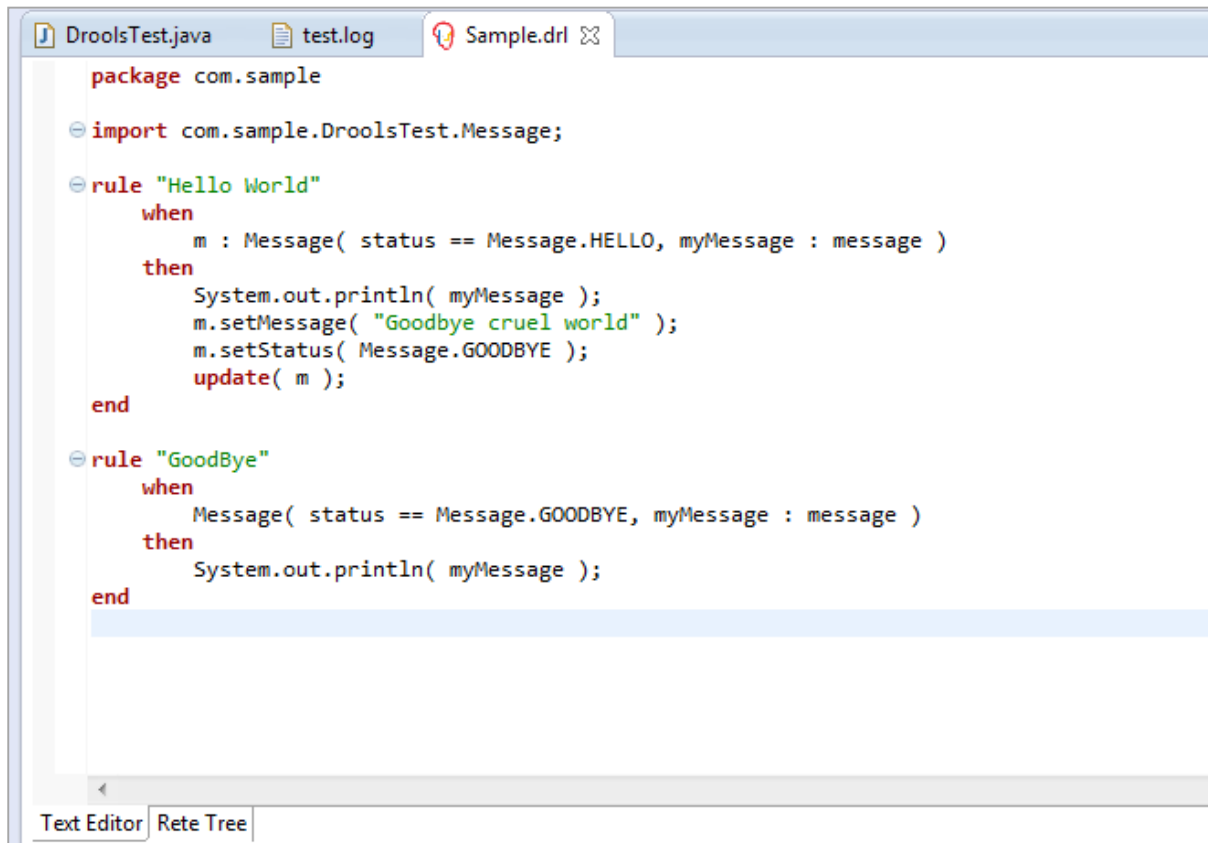
## Activations

---

Activations are the **then** part of the rule. Activations are placed in the agenda where the appropriate rule is fired.

## 6. RULES WRITING

If you see the default rule that is written in the Hello World project (Sample.drl), there are a lot of keywords used which we will be explaining now.



```
package com.sample

import com.sample.DroolsTest.Message;

rule "Hello World"
when
    m : Message( status == Message.HELLO, myMessage : message )
then
    System.out.println( myMessage );
    m.setMessage( "Goodbye cruel world" );
    m.setStatus( Message.GOODBYE );
    update( m );
end

rule "GoodBye"
when
    Message( status == Message.GOODBYE, myMessage : message )
then
    System.out.println( myMessage );
end
```

### Sample.drl

- **Package:** Every Rule starts with a package name. The package acts as a namespace for Rules. Rule names within a package must be unique. Packages in Rules are similar to packages in Java.
- **Import statement:** Whatever facts you want to apply the rule on, those facts needs to be imported. For example, `com.sample.DroolsTest.Message;` in the above example.
- **Rule Definition:** It consists of the Rule Name, the condition, and the Consequence. Drools keywords are **rule**, **when**, **then**, and **end**. In the above example, the rule names are "Hello World" and "GoodBye". The **when** part is the condition in both the rules and the **then** part is the consequence. In rule terminology, the **when** part is also called as LHS (left hand side) and the **then** part as the RHS (right hand side) of the rule.

Now let us walk through the terms used in the Java file used to load the Drools and execute the rules.

## Knowledge Base

---

Knowledge Base is an interface that manages a collection of rules, processes, and internal types. It is contained inside the package **org.drools.KnowledgeBase**. In Drools, these are commonly referred to as **knowledge definitions** or **knowledge**. Knowledge definitions are grouped into **knowledge packages**. Knowledge definitions can be added or removed. The main purpose of Knowledge Base is to store and reuse them because their creation is expensive. Knowledge Base provides methods for creating knowledge sessions.

## Knowledge Session

---

The knowledge session is retrieved from the knowledge base. It is the main interface for interacting with the Drools Engine. The knowledge session can be of two types:

- Stateless Knowledge Session
- Stateful Knowledge Session

## Stateless Knowledge Session

---

Stateless Knowledge Session is a stateless session that forms the simplest use case, not utilizing inference. A stateless session can be called like a function, passing it some data and then receiving some results back. Common examples of a stateless session include:

- **Validation**
  - Is this person eligible for a mortgage?
- **Calculation**
  - Compute a mortgage premium.
- **Routing and Filtering**
  - Filter incoming messages, such as emails, into folders.
  - Send incoming messages to a destination

## Stateful Knowledge Session

---

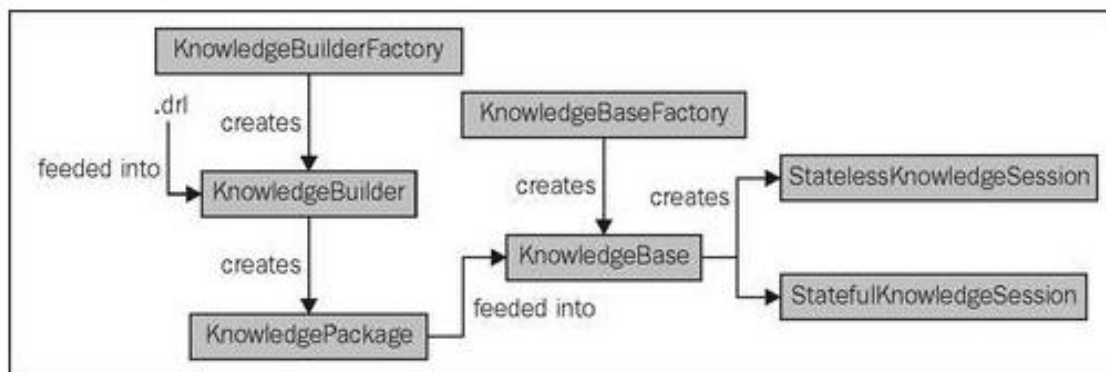
Stateful sessions are longer lived and allow iterative changes over time. Some common use cases for stateful sessions include:

- **Monitoring**
  - Stock market monitoring and analysis for semi-automatic buying.
- **Diagnostics**
  - Fault finding, medical diagnostics
- **Logistics**
  - Parcel tracking and delivery provisioning

## Knowledge Builder

The KnowledgeBuilder interface is responsible for building a KnowledgePackage from knowledge definitions (rules, processes, types). It is contained inside the package **org.drools.builder.KnowledgeBuilder**. The knowledge definitions can be in various formats. If there are any problems with building, the KnowledgeBuilder will report errors through these two methods: **hasErrors** and **getError**.

The following diagram explains the process



In the above example, as we are taking a simple example of stateless knowledge session, we have inserted the fact in the session, and then `fireAllRules()` method is called and you see the output.

In case of a stateful knowledge session, once the rules are fired, the stateful knowledge session object must call the method **dispose()** to release the session and avoid memory leaks.

# 7. RULE SYNTAX

As you saw the .drl (rule file) has its own syntax, let us cover some part of the Rule syntax in this chapter.

## Conditions in Rules

---

A rule can contain many conditions and patterns such as:

- Account (balance == 200)
- Customer (name == "Vivek")

The above conditions check if the Account balance is 200 or the Customer name is "Vivek".

## Variables in Rules

---

A variable name in Drools starts with a Dollar(\$) symbol.

- \$account : Account( )
- \$account is the variable for Account() class

Drools can work with all the native Java types and even Enum.

## Comments in Rules

---

The special characters, # or //, can be used to mark single-line comments.

For multi-line comments, use the following format:

```
/*  
    Another line  
    .....  
    .....  
*/
```

## Global Variables

---

Global variables are variables assigned to a session. They can be used for various reasons as follows:

- For input parameters (for example, constant values that can be customized from session to session).

- For output parameters (for example, reporting—a rule could write some message to a global report variable).
- Entry points for services such as logging, which can be used within rules.

## Functions in Rules

---

Functions are a convenience feature. They can be used in conditions and consequences. Functions represent an alternative to the utility/helper classes. For example,

```
function double calculateSquare (double value) {  
    return value * value;  
}
```

## Dialect

---

A dialect specifies the syntax used in any code expression that is in a condition or in a consequence. It includes return values, evals, inline evals, predicates, salience expressions, consequences, and so on. The default value is **Java**. Drools currently supports one more dialect called **MVEL**. The default dialect can be specified at the package level as follows:

```
package org.mycompany.somePackage  
  
dialect "mvel"
```

## MVEL Dialect

MVEL is an expression language for Java-based applications. It supports field and method/getter access. It is based on Java syntax.

## Salience

---

Salience is a very important feature of Rule Syntax. Salience is used by the conflict resolution strategy to decide which rule to fire first. By default, it is the main criterion.

We can use salience to define the order of firing rules. Salience has one attribute, which takes any expression that returns a number of type int (positive as well as negative numbers are valid). The higher the value, the more likely a rule will be picked up by the conflict resolution strategy to fire.

```
salience ($account.balance * 5)
```

The default salience value is 0. We should keep this in mind when assigning salience values to some rules only.

There are a lot of other features/parameters in the Rule Syntax, but we have covered only the important ones here.

## Rule Consequence Keywords

---

Rule Consequence Keywords are the keywords used in the “**then**” part of the rule.

- **Modify** – The attributes of the fact can be modified in the **then** part of the Rule.
- **Insert** – Based on some condition, if true, one can insert a new fact into the current session of the Rule Engine.
- **Retract** – If a particular condition is true in a Rule and you don't want to act anything else on that fact, you can retract the particular fact from the Rule Engine.

**Note:** It is considered a very bad practice to have a conditional logic (if statements) within a rule consequence. Most of the times, a new rule should be created.

## 8. SAMPLE DROOLS PROGRAM

In this chapter, we will create a Drools project for the following problem statement:

*Depending upon the city and the kind of product (Combination of City and Product), find out the local tax related to that city.*

We will have two DRL files for our Drools project. The two DRL files will signify two cities in consideration (Pune and Nagpur) and four types of products (groceries, medicines, watches, and luxury goods).

- The tax on medicines in both the cities is considered as zero.
- For groceries, we have assumed a tax of Rs 2 in Pune and Rs 1 in Nagpur.

We have used the same selling price to demonstrate different outputs. Note that all the rules are getting fired in the application.

Here is the model to hold each itemType:

```
package com.sample;

import java.math.BigDecimal;

public class ItemCity {

    public enum City {
        PUNE, NAGPUR
    }

    public enum Type {
        GROCERIES, MEDICINES, WATCHES, LUXURYGOODS
    }

    private City purchaseCity;
    private BigDecimal sellPrice;
    private Type typeofItem;
    private BigDecimal localTax;
```



```
public City getPurchaseCity() {  
    return purchaseCity;  
}  
  
public void setPurchaseCity(City purchaseCity) {  
    this.purchaseCity = purchaseCity;  
}  
  
public BigDecimal getSellPrice() {  
    return sellPrice;  
}  
  
public void setSellPrice(BigDecimal sellPrice) {  
    this.sellPrice = sellPrice;  
}  
  
public Type getTypeofItem() {  
    return typeofItem;  
}  
  
public void setTypeofItem(Type typeofItem) {  
    this.typeofItem = typeofItem;  
}  
  
public BigDecimal getLocalTax() {  
    return localTax;  
}  
  
public void setLocalTax(BigDecimal localTax) {  
    this.localTax = localTax;  
}  
  
}
```

## DRL Files

---

As suggested earlier, we have used two DRL files here: Pune.drl and Nagpur.drl.

### Pune.drl

This is the DRL file that executes rules for Pune city.

```
// created on: Dec 24, 2014
package droolsexample

// list any import classes here.
import com.sample.ItemCity;
import java.math.BigDecimal;

// declare any global variables here
dialect "java"
rule "Pune Medicine Item"
    when
        item : ItemCity (purchaseCity == ItemCity.City.PUNE,
            typeofItem == ItemCity.Type.MEDICINES)
    then
        BigDecimal tax = new BigDecimal(0.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
    end

rule "Pune Groceries Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.PUNE,
            typeofItem == ItemCity.Type.GROCERIES)
    then
        BigDecimal tax = new BigDecimal(2.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
    end
```

## Nagpur.drl

This is the DRL file that executes rules for Nagpur city.

```
// created on: Dec 26, 2014
package droolsexample

// list any import classes here.
import com.sample.ItemCity;
import java.math.BigDecimal;

// declare any global variables here
dialect "java"
rule "Nagpur Medicine Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.NAGPUR,
            typeofItem == ItemCity.Type.MEDICINES)
    then
        BigDecimal tax = new BigDecimal(0.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
    end

rule "Nagpur Groceries Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.NAGPUR,
            typeofItem == ItemCity.Type.GROCERIES)
    then
        BigDecimal tax = new BigDecimal(1.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
    end
```

We have written the DRL files based on city, as it gives us extensibility to add any number of rule files later if new cities are being added.

To demonstrate that all the rules are getting triggered from our rule files, we have used two item types (medicines and groceries); and medicine is tax-free and groceries are taxed as per the city.

Our test class loads the rule files, inserts the facts into the session, and produces the output.

## Droolstest.java

```
package com.sample;

import java.math.BigDecimal;

import org.drools.KnowledgeBase;
import org.drools.KnowledgeBaseFactory;
import org.drools.builder.KnowledgeBuilder;
import org.drools.builder.KnowledgeBuilderError;
import org.drools.builder.KnowledgeBuilderErrors;
import org.drools.builder.KnowledgeBuilderFactory;
import org.drools.builder.ResourceType;
import org.drools.io.ResourceFactory;
import org.drools.runtime.StatefulKnowledgeSession;

import com.sample.ItemCity.City;
import com.sample.ItemCity.Type;

/**
 * This is a sample class to launch a rule.
 */
public class DroolsTest {

    public static final void main(String[] args) {
        try {
            // load up the knowledge base
            KnowledgeBase kbase = readKnowledgeBase();
            StatefulKnowledgeSession ksession =
kbase.newStatefulKnowledgeSession();

            ItemCity item1 = new ItemCity();
            item1.setPurchaseCity(City.PUNE);
            item1.setTypeofItem(Type.MEDICINES);
            item1.setSellPrice(new BigDecimal(10));
```

```
ksession.insert(item1);

ItemCity item2 = new ItemCity();
item2.setPurchaseCity(City.PUNE);
item2.setTypeofItem(Type.GROCERIES);
item2.setSellPrice(new BigDecimal(10));
ksession.insert(item2);

ItemCity item3 = new ItemCity();
item3.setPurchaseCity(City.NAGPUR);
item3.setTypeofItem(Type.MEDICINES);
item3.setSellPrice(new BigDecimal(10));
ksession.insert(item3);

ItemCity item4 = new ItemCity();
item4.setPurchaseCity(City.NAGPUR);
item4.setTypeofItem(Type.GROCERIES);
item4.setSellPrice(new BigDecimal(10));
ksession.insert(item4);

ksession.fireAllRules();

System.out.println(item1.getPurchaseCity().toString() + " "
+ item1.getLocalTax().intValue());

System.out.println(item2.getPurchaseCity().toString() + " "
+ item2.getLocalTax().intValue());

System.out.println(item3.getPurchaseCity().toString() + " "
+ item3.getLocalTax().intValue());

System.out.println(item4.getPurchaseCity().toString() + " "
+ item4.getLocalTax().intValue());
} catch (Throwable t) {
```

```

        t.printStackTrace();
    }
}

private static KnowledgeBase readKnowledgeBase() throws Exception {
    KnowledgeBuilder kbuilder =
        KnowledgeBuilderFactory.newKnowledgeBuilder();

    kbuilder.add(ResourceFactory.newClassPathResource("Pune.drl"),
        ResourceType.DRL);

    kbuilder.add(ResourceFactory.newClassPathResource("Nagpur.drl"),
        ResourceType.DRL);

    KnowledgeBuilderErrors errors = kbuilder.getErrors();
    if (errors.size() > 0) {
        for (KnowledgeBuilderError error: errors) {
            System.err.println(error);
        }
        throw new IllegalArgumentException("Could not parse knowledge.");
    }
    KnowledgeBase kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages(kbuilder.getKnowledgePackages());
    return kbase;
}
}

```

If you run this program, its output would be as follows:

```

PUNE 0
PUNE 20
NAGPUR 0
NAGPUR 10

```

For both Pune and Nagpur, when the item is a medicine, the local tax is zero; whereas when the item is a grocery product, the tax is as per the city. More rules can be added in the DRL files for other products. This is just a sample program.

## Call an External Function from a DRL File

Here we will demonstrate how to call a static function from a Java file within your DRL file.

First of all, create a class **HelloCity.java** in the same package **com.sample**.

```
package com.sample;

public class HelloCity {

    public static void writeHello(String name) {
        System.out.println("HELLO " + name + "!!!!!!");
    }

}
```

Thereafter, add the import statement in the DRL file to call the writeHello method from the DRL file. In the following code block, the changes in the DRL file Pune.drl are highlighted in yellow.

```
// created on: Dec 24, 2014
package droolsexample

// list any import classes here.
import com.sample.ItemCity;
import java.math.BigDecimal;
import com.sample.HelloCity;

//declare any global variables here
dialect "java"

rule "Pune Medicine Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.PUNE,
            typeofItem == ItemCity.Type.MEDICINES)
    then
        BigDecimal tax = new BigDecimal(0.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
```

```
        HelloCity.writeHello(item.getPurchaseCity().toString());
    end

    rule "Pune Groceries Item"
        when
            item : ItemCity(purchaseCity == ItemCity.City.PUNE,
                typeofItem == ItemCity.Type.GROCERIES)
        then
            BigDecimal tax = new BigDecimal(2.0);
            item.setLocalTax(tax.multiply(item.getSellPrice()));
        end
    end
```

Run the program again and its output would be as follows:

```
HELLO PUNE!!!!!!
PUNE 0
PUNE 20
NAGPUR 0
NAGPUR 10
```

The difference now in the output is marked in yellow which shows the output of the static method in the Java class.

The advantage to call a Java method is that we can write any utility/helper function in Java and call the same from a DRL file.



## 9. DEBUGGING

There are different ways to debug a Drools project. Here, we will write a Utility class to let you know which rules are being triggered or fired.

With this approach, you can check what all rules are getting triggered in your Drools project. Here is our Utility Class

### Utility.java

```
package com.sample;

import org.drools.spi.KnowledgeHelper;

public class Utility {

    public static void help(final KnowledgeHelper drools,
        final String message){
        System.out.println(message);
        System.out.println("\nrule triggered: " + drools.getRule().getName());
    }

    public static void helper(final KnowledgeHelper drools){
        System.out.println("\nrule triggered: " + drools.getRule().getName());
    }
}
```

The first method **help** prints the rule triggered along with some extra information which you can pass as String via the DRL file.

The second rule **helper** prints whether the particular rule was triggered or not.

We have added one of the Utility methods in each DRL file. We have also added the import function in the DRL file (Pune.drl). In the **then** part of the rule, we have added the utility function call. The modified Pune.drl is given below. Changes are highlighted in blue.

## Modified Pune.drl

```
//created on: Dec 24, 2014
package droolsexample

//list any import classes here.

import com.sample.ItemCity;
import java.math.BigDecimal;
import com.sample.HelloCity;
import function com.sample.Utility.helper;
// declare any global variables here

dialect "java"
rule "Pune Medicine Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.PUNE,
            typeofItem == ItemCity.Type.MEDICINES)
    then
        BigDecimal tax = new BigDecimal(0.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
        HelloCity.writeHello(item.getPurchaseCity().toString());
        helper(drools);
    end

rule "Pune Groceries Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.PUNE,
            typeofItem == ItemCity.Type.GROCERIES)
    then
        BigDecimal tax = new BigDecimal(2.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
        helper(drools);
    end

end
```

Similarly, we have added the other utility function in the second DRL file (Nagpur.drl). Here is the modified code:

## Modified Nagpur.drl

```
// created on: Dec 26, 2014
package droolsexample

// list any import classes here.

import com.sample.ItemCity;
import java.math.BigDecimal;
import function com.sample.Utility.help;

//declare any global variables here
dialect "java"
rule "Nagpur Medicine Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.NAGPUR,
            typeofItem == ItemCity.Type.MEDICINES)
    then
        BigDecimal tax = new BigDecimal(0.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
        help(drools,"added info");
    end

rule "Nagpur Groceries Item"
    when
        item : ItemCity(purchaseCity == ItemCity.City.NAGPUR,
            typeofItem == ItemCity.Type.GROCERIES)
    then
        BigDecimal tax = new BigDecimal(1.0);
        item.setLocalTax(tax.multiply(item.getSellPrice()));
        help(drools,"info");
    end

end
```

Run the program again and it should produce the following output:

```
info

rule triggered: Nagpur Groceries Item
added info

rule triggered: Nagpur Medicine Item

rule triggered: Pune Groceries Item
HELLO PUNE!!!!!!

rule triggered: Pune Medicine Item
PUNE 0
PUNE 20
NAGPUR 0
NAGPUR 10
```

Both the utility functions are called and it shows whether the particular rule was called or not. In the above example, all the rules are being called, but in an enterprise application, this utility function can be really useful to debug and find out whether a particular rule was fired or not.

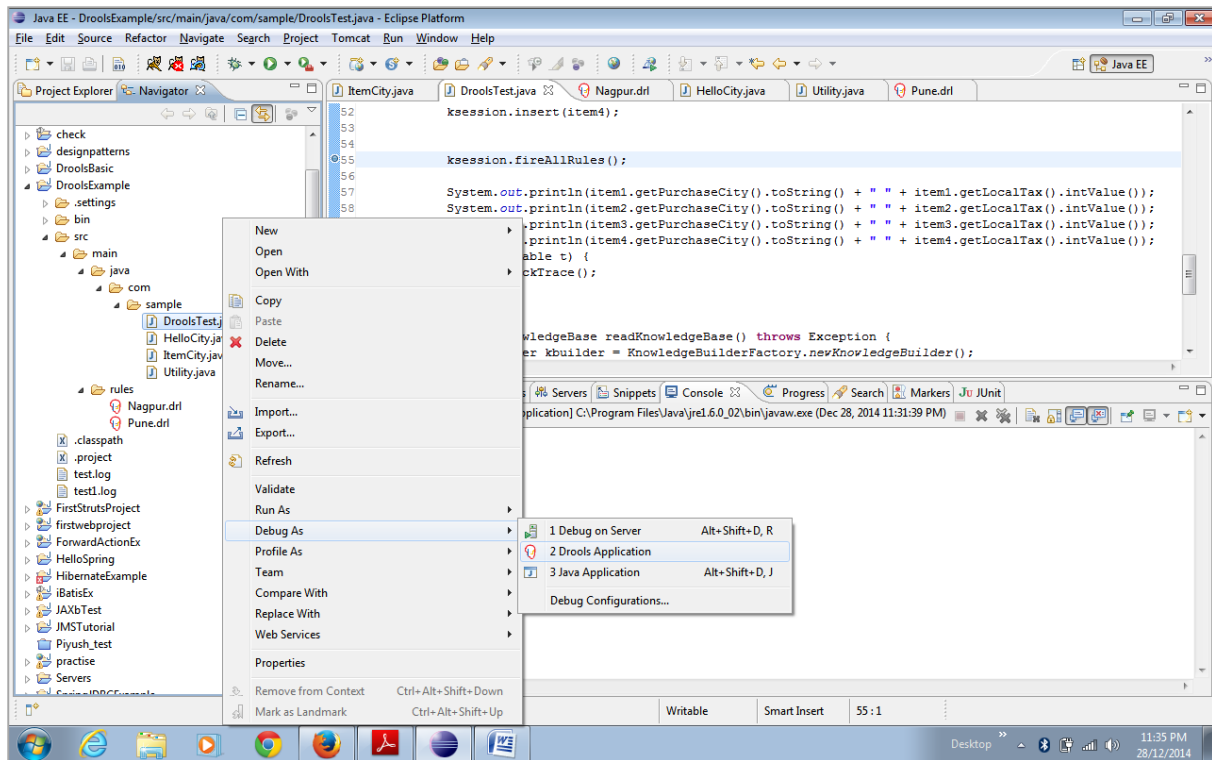
## Using the Debug Perspective in Eclipse

---

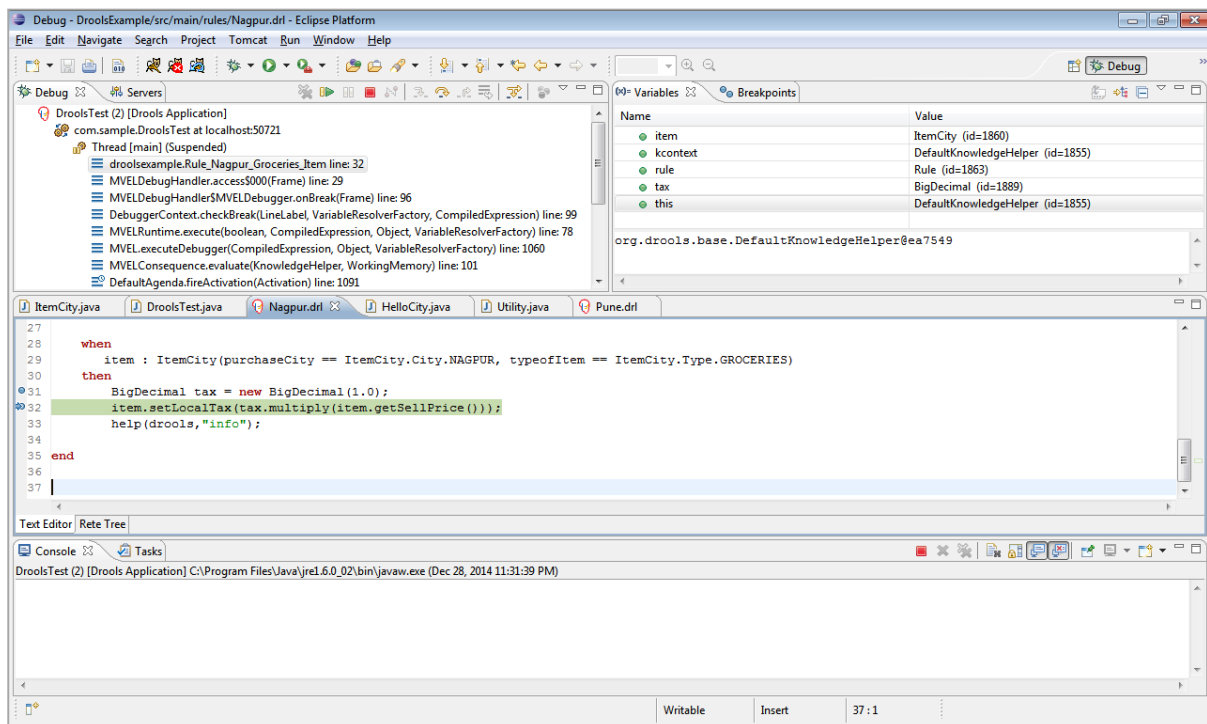
You can debug the rules during the execution of your Drools application. You can add breakpoints in the consequences of your rules, and whenever such a breakpoint is encountered during the execution of the rules, execution is stopped temporarily. You can then inspect the variables known at that point as you do in a Java Application, and use the normal debugging options available in Eclipse.

To create a breakpoint in your DRL file, just double-click at the line where you want to create a breakpoint. Remember, you can only create a breakpoint in the **then** part of a rule. A breakpoint can be removed by double-clicking on the breakpoint in the DRL editor.

After applying the breakpoints, you need to debug your application as a Drools application. Drools breakpoints (breakpoints in DRL file) will only work if your application is being debugged as a Drools application. Here is how you need to do the same:



Once you debug your application as a Drools application, you would see the control on the DRL file as shown in the following screenshot:



You can see the variables and the current values of the object at that debug point. The same control of F6 to move to the next line and F8 to jump to the next debug point are applicable here as well. In this way, you can debug your Drools application.

**Note:** The debug perspective in Drools application works only if the dialect is MVEL until Drools 5.x.