# More rules in the JBoss IDE

In the previous chapter we moved away from the Guvnor editor and started using JBoss IDE to edit our rules. Our Hello World example was a useful start, but we know that rules can do much more. For example, consider the poor Ooompa Loompas that are shipping the chocolate bars out of the factory door. Wouldn't it be nice if we drew up a shipping schedule for them? Before we do that we'll look again at the structure of a rule file. Towards the end of this chapter, we'll look at some more advanced rules that we can write and run in the IDE.

## Rule syntax

We bumped into our first rule (**.drl**) file in the previous chapter. We will see a lot more of it here, so it's worth going over it again. Our rule file can contain the following elements:

1. package: Package-name is the name of the folder in which a rule file lives. This is useful for organising our rules (especially when you build up hundreds of them).
2. import: This pulls in any Java class files (including fact models) that we use in our rules.
3. global: This defines the global variables that we use. Remember variables (boxes that hold values)? Earlier, they were emptied as soon as our rule had fired and only changes to the facts lived on. Compared to 'normal' variables, global variables live longer, and allow us to pass information into and out of our rules.

Passing information into and out of our rules via a global variable is almost the same as passing a fact into the rules. The difference is that the rule engine does not match (or fire) against global variables. This makes the rule engine more suitable for passing in items that change slowly, such as the current date, counter, and so on, and giving rules access to external resources (such as log files).

1. function: Sometimes you may need to perform the same calculation in many rules. Defining a function allows you to perform the same calculation again and again. Note that it's often easier to call a normal Java function to carry out the same task.
2. rule: This is the 'when…then' structure that we've spent most of this book talking about.
3. comments: These are pieces of text ignored by the rule engine, which explain what is going on with us. They can be on a single line (anything after '//' until the end of the line) or split over multiple lines (everything between the /\* and \*/ Comments split over many lines).

In the Hello World example, our rule (the 'when...then') was fairly simple. We matched **Message.STATUS** (when), updated that status and printed a message (then). Let's look at the other options available to us. We'll start with the When part of the rule.

### Patterns for the When part

Remember our rule from Chapter 4 that calculated the discount on the chocolate sales? Using the fact model from that example (**sales.java**), the following are all valid 'when' conditions. The following code extract shows most of the simple conditions that we can use in our rules. Note that it would be impossible for this rule to fire, given that all of the contradictory conditions cannot be met at the same time. Drools would also complain that we use same variable name (**$mySales**) twice in one rule.

Drools recommends the convention $variableName; dollar sign, first letter small, and capital letter for each of the following words in the variable name. But it's only a convention. Note that variable names are also case sensitive, that is, $variableNameis not the same as $VARIABLENAME.

Note the use of the "//" single-line comments to explain what each line does.

**rule "show various conditions" when**

**//Simple match on all sales, no assignment of variable Sales()**

**// matches all sales lines, one by one, assigns to local var MySales $mySales : Sales()**

**//additional filter on customer name. Corrpesponds to getName() on sales JavaBean $mySales : Sales (name=="acme corp")**

**//'or' - both lines do the same thing**

**$mySales : Sales (name=="acme corp" or name=="beta corp")**

**$mySales : Sales (name=="acme corp" name=="beta corp")**

**//and - three lines do the same thing**

**$mySales : Sales (name=="acme corp" and sales>100) $mySales : Sales (name=="acme corp" , sales>100)**

**$mySales : Sales (name=="acme corp" && sales>100)**

**//Number comparison**

**$mySales : Sales (sales==100) //equals 100**

**$mySales : Sales (sales<100) //less than 100**

**$mySales : Sales (sales>100) //greater than 100**

**$mySales : Sales (sales!=100) //not equal to 100**

**$mySales : Sales (sales>=100) //greater than or equals 100 $mySales : Sales (sales<=100) //less than or equals 100**

**//Use of bracket reorder the evaluation of the condition**

**//The 'and' condition is performed first, then the 'or' condition**

**$mySales : Sales (name=="acme corp",(sales>100 && sales<200)) then**

**//do something end**

In the above example, we could also test for customers with empty names by using a condition similar to:

**$mySales : Sales (name== null)**

**null** is another of those words with a special meaning; think of it as void or completely empty. Note that this is different from **name== ""**, which means it does have a name, but that name is blank. **""** is like a blank sheet of paper, whereas null means no piece of paper at all. Still confused? Then you can test for both as follows:

**$mySales: Sales (name== null or name=="")**

Sometimes when you're running rules, do you get an error related to **null** called **nullPointerException**? OK, make that you will frequently get a **nullPointerException**.

This error means that we are trying to do something like **sales.**

**getName()**, except that sales are **null**. Drools and Java do not know how to handle this. So it stops, tells you what went wrong, and waits for your next move.

### Patterns for the Then part

In the Then part of the rule we can use just about any Java code, plus the Drools constructs given as follows. The following consequences are all valid, as long as we have a variable called **$mySales** defined in the When part (like we defined in the previous example):

* Any valid Java code, such as the **System.out.println("HelloWorld")** from the previous examples.
* The insert statement tells Drools that we have created a new fact that it should be aware of. This can be done using a variable (for example, **insert ($mySales)**) or creating a new fact on the spot (for example, **insert (new Sales()**).
* The update statement is similar to insert, but is used where the fact existed before the rule started (for example, **update($mySales)**).

In the next example, we'll use a variant of **update()**, called **modify()**. This is a useful shortcut when we need to change several items in a fact at once (for example, Sales number, date, and name).

### Shipping chocolate bars

Armed with the latest business rule information, we can now go about helping our Ooompa Loompas. They work hard at loading all of the chocolate bars that we are making onto trucks, to ship to the customers who are busy sending us orders. In fact, we are getting so many orders that we are limiting each customer to one box of 210 chocolate bars a week.

Now, if you've ever worked in a packing or a shipping department, you know that your hands are always busy. You just want to be told what and when to pack, and you don't have to calculate your next step. So, using Drools, we're going to write a list of what to ship and when to ship to each customer.

Because we are nice people, we give our overworked Ooompa Loompas a holiday now and then. We don't work on holidays, so we ship our chocolate bars the next working day.

How do we write this in a way a computer can understand?

### The problem (and remind me why I need business rules)

If we were writing this in a normal computer language, we'd have something like the following:

**Start Loop Have we shipped all the chocolate bars yet?**

**Yes – go to end of loop**

**Is today a holiday**

**No – Ship 210 bars and update totals Is tomorrow a holiday?**

**No – Ship 210 bars and update totals . . . .**

**Have we shipped all the chocolate bars yet?**

**Yes – go to end of loop**

**End Loop**

This code will (almost) work, but we've got the following problems:

* Even though it's written in plain English, can you identify the six business rules that are embedded in it? Take a peek down to our business rule solution to find them all.
* Most code is not written in English, but in Java or C#. Would you be able to find the business rules hidden in a technical language like Java?
* This example will break if we have more than two holidays in a row, but it's not immediately obvious from the text.
* What if we want to change our business rules (for example, to have time off on the weekends)? If we do this on a separate line (to keep our sample clear) and have two days off per weekend, the code becomes 40% (four lines) longer.

#### Why rules scale better—a reminder

Imagine that each additional business rule in the above example adds 40% to our sample function's length. Adding 10 business rules would take about 500 lines to write (and the size increases exponentially). For the mathematically inclined, that is 10 lines times our 40% extra complexity per rule times our 10 extra business rules or 10\*1.4^10. A fault in any of these lines could break the entire function.

By contrast, our Drools business rules are independent. Adding another means that we add only 10 extra lines each time (not including whitespaces or comments). See the business rules example below and count the lines yourself! That's 10 lines extra for the first rule and 10 lines extra for the 16th rule, as each rule does not make the previous rules more complex to write.

Rules may give a slightly longer solution at the beginning, but increase in efficiency as the project grows for all but the most trivial solutions.

### Getting and building the sample

You could type in the entire example that we will describe in the next couple of pages. Or you could download the Chapter 6 example from **http://code.google. com/p/red-piranha/**, unzip it into a directory of your choice, and then review it at your leisure. We recommend downloading, as it's a much more pleasant option.

Once you've downloaded and unzipped the Chapter 6 example, you're ready to open the JBoss IDE. The sample includes the necessary Eclipse settings. Create a new Eclipse project (as we've done for the previous samples) and in the wizard that appears, open the folder where you have unzipped the Chapter 6 example. Eclipse should automatically pick up the project settings.

When you've created the Eclipse project, you will notice that the supporting libraries (such as the Drools core) are missing. This is deliberate, as it reduces the size of the example file that you have to download. Fortunately, the sample also contains a Maven project file (**pom.xml**). We set up Maven in Chapter 2, so you can build the project (and download the required libraries) as follows:

1. Open a command window (DOS prompt).
2. Go to the folder containing the project, for example, **cd \some-project-folder**.
3. Type **mvn eclipse:eclipse**. This will generate the Eclipse project. Then, download all of the dependencies.

At this point (once you see the **build successful** message) when you refresh the project (right-click on the project name in Eclipse and select **Refresh**), all of the required libraries will be available.

Downloading the libraries using Maven is a lot faster than downloading them as part of the ZIP file. After the first download, Maven keeps a local copy, and as long as the version number is up to date, it will use that file for future examples. You'll notice the speed difference in future chapters.

For information, if you wanted to go one step further and build the project from the command line you can also run the command **mvn clean package**. This will give you a deployable JAR file. For the rest of this chapter, we'll concentrate on running the examples through Eclipse.

#### Rules

Let's take a quick look at our rule file, **shipping-rules.drl**. The first part of the rule file contains the usual package and import information. At the bottom of the extract, we can see the declarations for two global variables.

**package net.firstpartners.chap6; import java.util.Date; import org.apache.commons.logging.Log; import net.firstpartners.chap6.domain.CustomerOrder; import net.firstpartners.chap6.domain.ChocolateShipment; import net.firstpartners.chap6.domain.OoompaLoompaDate; global OoompaLoompaDate nextAvailableShipmentDate; global Log log;**

In this case we import the following: handles to Date; where to get information on the Apache log (a smarter way of printing to the console); three facts that we use to organize our data; and two global variables.

The three other facts that we import to organize our data are **OoompaLoompaDate**, **CustomerOrder**, and **ChocolateShipment**. A customer order is the total amount of chocolate that the customer wants, and will contain many **chocolateShipments** as we send them one box a week to meet their order.

Our global variables (the **nextAvailableShipmentDate** and the handle to the external log) are placeholders for items that we pass in when we call our rules.

Our first rule confirms the holidays when the Ooompa Loompas will not work. Whenever we find a date, we print out a message to the console. If we don't want to see this message, we could safely remove this rule.

**rule "confirm holidays"**

**when**

**$holiday : OoompaLoompaDate()**

**then**

**//Logging message**

**log.info("Remember - Ooompa Loompas don't work on:"+$holiday); end**

This rule actually matches against all **OoompaLoompaDates** in working memory. This works in this particular example, as the only OoompaLoompa dates directly in working memory will be holidays. In real life, our rule would need to be a bit more particular, that is, it should check whether the date is a holiday using something similar to the following:

**When**

**$holiday : OoompaLoompaDate(holiday==true)**

The next rule is one of the key ones in the example. When we find a customer order for which we haven't shipped all of the chocolate, then we add a new chocolate shipment to the order.

**rule "Chocolate Shipment" when $CustomerOrder : CustomerOrder(currentBalance>0)**

**then**

**//Add a new shipment into the CustomerOrder**

**ChocolateShipment ChocolateShipment = new**

**ChocolateShipment(210); modify($CustomerOrder){ addShipment(ChocolateShipment) }**

**//notify the working memory of the new shipment insert( ChocolateShipment );**

**//Logging message**

**log.info("Fired Customer Shipment rule - customer is still waiting for "+$CustomerOrder.getCurrentBalance()+" chocolate bars");**

**end**

The next rule looks for any shipments (like the ones created in the previous rule) that have no shipment date set yet. When it matches, it sets the shipment date to the next available date (as retrieved from the **nextAvailableShipmentDate** global variable), and then rolls forward the **next** **available** **shipment** **date** by a week.

**rule "Add Next Available Shipment Date" when $ChocolateShipment : ChocolateShipment(shipmentDate ==null)**

**then**

**modify($ChocolateShipment){**

**setShipmentDate(nextAvailableShipmentDate.getCopy()) }**

**nextAvailableShipmentDate.rollForward(7);**

**//Logging message**

**log.info("Add Next Available Shipment Date:"+$ChocolateShipment**

**.getShipmentDate()); end**

Our holidays are passed in as facts, so rules can match against them. Whenever a shipment date lands on a holiday, our next rule will fire and move the shipment forward one day. If that date also happens to be a holiday, this rule will automatically fire again. We don't need to write any specific code to handle the 'holidays in a row' situation. We just state what we know to be true, and let the rule engine manage the complexity.

**rule "modify due to holidays"**

**when**

**$holiday : OoompaLoompaDate()**

**$ChocolateShipment : ChocolateShipment(shipmentDate==$holiday)**

**then**

**modify($ChocolateShipment){ getShipmentDate().rollForward(1) }**

**//Logging message log.info("Reschedule Shipment Date to:"+$ChocolateShipment. getShipmentDate()+" due to holiday on:"+$holiday); end**

When we are shipping our boxes of chocolate, it is likely that the last box won't be full. For example, an order for 500 fudge chocolate bars will be made up of two boxes of 210, and one smaller box of 80. Our next rule covers this situation by ensuring that the last box we ship does not leave us with **items** **still** **to** **ship** of less than 0. It does this by matching any customer order with a negative **items** **still** **to** **ship** and adjusting the numbers accordingly.

**rule "Don't ship more than the customer order"**

**when**

**$CustomerOrder : CustomerOrder(currentBalance<0)**

**$ChocolateShipment : ChocolateShipment(itemsStillToShip<0) then**

**long $newShipment = $ChocolateShipment.getShipmentAmount()+$ ChocolateShipment.getItemsStillToShip();**

**modify($ChocolateShipment){ setShipmentAmount($newShipment), setItemsStillToShip(0) }**

**modify($CustomerOrder){ setCurrentBalance(0) }**

**//Logging message**

**log.info("Removed CustomerOrder Overshipping - new shipment:"+**

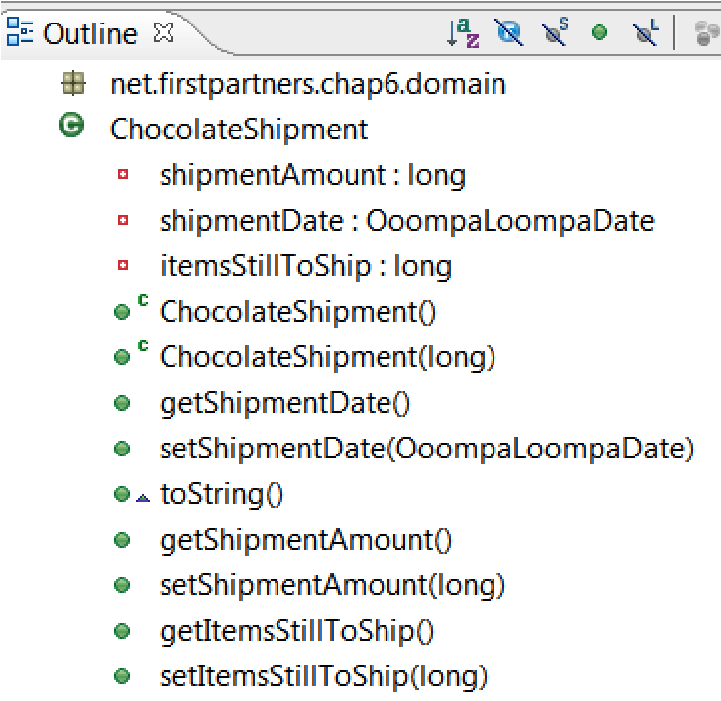
**$newShipment); end**

#### ChocolateShipment.java

Our rules depend on three Java-based facts that we saw earlier when we imported them into our business rules. The first of these is the **ChocolateShipment**—a note to the Ooompa Loompas in the shipping department to put a box onto a truck. You can inspect the code if you want, but given that it is a simple JavaBean (that is, a placeholder for carrying information into and out of our rules) we can see a better overview using the outline view in Eclipse / the JBoss IDE.

Reading from the bottom up, we can see that our bean allows us to get (and set) the following values: the number of items still to ship; the amount in this shipment; and the date that we are shipping on. The **toString** method is a Java convention that makes it easier to print the information that we hold in this Java class to the log. The two **ChocolateShipment** methods give us information on how we can create new shipments.

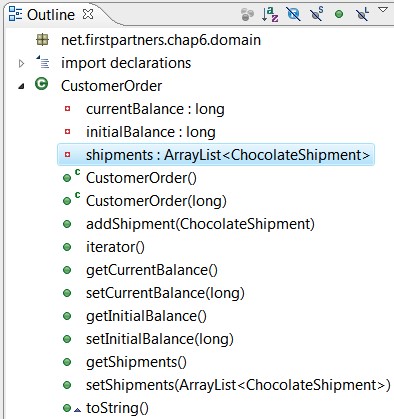
All of these are referenced in the rules. In fact, if the spelling used in the rules doesn't match the one we have here, we get an error.



#### CustomerOrder.java

We also saw the **CustomerOrder** JavaBean imported into our **ChocolateShipment** JavaBean. In this sample, we can have one overall customer order containing many customer shipments to fulfill the order. The outline of the **CustomerOrder.java** file (image below) reflects this.

The **toString** (logging) method, and the get or set methods for **shipments**, **InitialBalance**, and **currentBalance** follow the normal JavaBean style (remember that JavaBeans are still just a means of passing information around). The two **CustomerOrder** methods give us different options for how we create this bean. (For information, **CustomerOrder()** means that we can create the bean with no parameters, which is perhaps the easiest way of all.)



Note the subtle difference in three similar methods that we use to deal with shipments:

* The **addShipment** method allows us to add another customer shipment to the existing list.
* The **getShipments** method returns the current list of customerShipments.
* The **setShipments** method allows us to pass in an entirely new list of customerShipments.

There is also an iterator method to make it easier for us to loop over the list of shipments.

More information on the full power of Java collections (a more powerful form of the lists we're dealing with here) is available on the Java web site, **http://java.sun.com**. We are able to write rules using this power.

For example, the following condition would match **customerOrders** with a first shipment to a customer of less than **100**.

**CustomerOrder** **(shipments[0].shipmentAmount** **<100)**

Note the use of **[0]** to refer to the first shipment in the **CustomerOrder**. For those interested in how this works, the **shipmentAmount** matches to the **getShipmentAmount()** method on our **CustomerShipment** class.

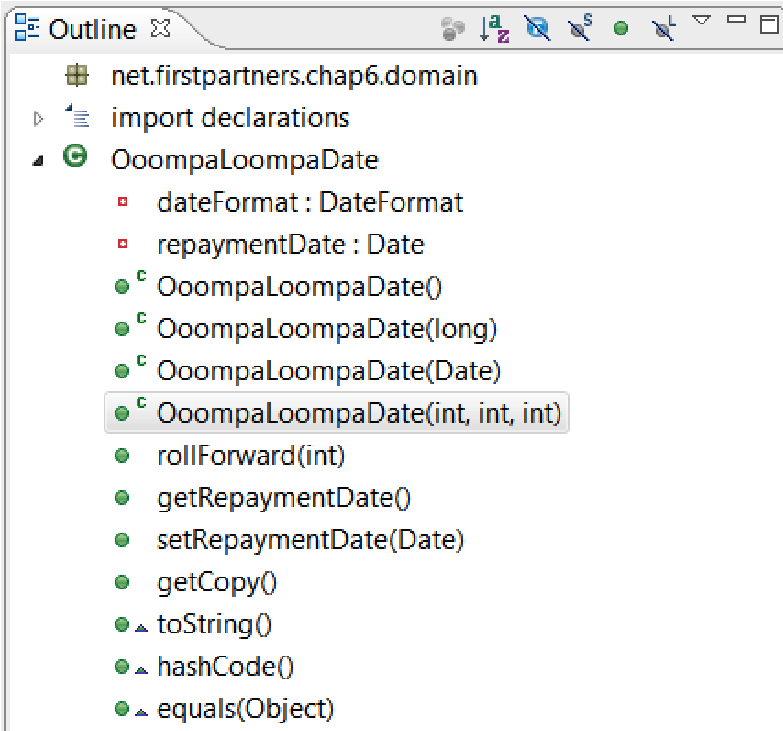
#### OompaLoompaDate

Dealing with dates in Java can sometimes be nasty. For example, Java counts the days in the month from 1 (as you'd expect), but months in the year start from 0 (January). For this reason we have wrapped the Java Date in **OoompaLoopaDate** to hide some of the ugliness and make dates behave as you'd expect.

There's a second, even better, reason for the having **OoompaLoompaDate.java** in this sample. Remember that we talked about functions in our rule (**.drl**) file and said you'd be better off using normal Java code? The **rollForward()** method is an example of this, as it allows us to move to a date 'x' days in the future. In our rules, we don't care how this is done, just that it works!

Although business logic works great in rules, calculating dates isn't business logic. Calculating dates in Java makes it easier to unit test. There is more on unit tests in the next chapter. If you can't wait until then, it's a method of quality assurance—when we ask to move 10 days on, we know that it will do exactly that.

The outline of the **OompaLoompaDate.java** file can be seen in the following screenshot:



The other new methods to note are **hashCode()** and **equals()**. These are standard JavaBean conventions that allow us to match one date against another more easily.

#### The RuleRunner file

We saw some code in the HelloWorld example in the previous chapters that loaded our rule (**.drl** file) and called the rule engine. For convenience, we have put it into one file called **RuleRunner.java**.

The main starting point in this file is the second **runRules()** method. The steps to follow for this method are:

1. Load the rule file(s) from disk. We pass in the names of the files when calling the methods.
2. Create a RuleBase and a session (our scratchpad) using these rules.
3. Pass any global variables (name, value) into our rule session.
4. Pass any facts.

This code is fairly generic and could easily be reused in your own rule projects. We will look at the content of the **RuleRunner** file in more detail in Chapter 11. For the moment, we'll concentrate on the actual business rules.

#### MultipleRulesExample

By now you must be itching to run this sample; don't worry, we are almost there. We will run the **MultipleRulesExample.java** file soon. It's the starting point for the example. Reading through the file, the main points are:

* The package and import statements tell Java what directory this file lives in and the files from other directories that we need
* **NEXT\_AVAILABLE\_SHIPMENT\_DATE** is a constant value that matches the global variable of the same name in our rules file
* **RULES\_FILES** is a constant value of the name where we find our rules file
* The main method, like our previous sample, is where we start when we click the **Run** button

**public static void main(String[] args) throws Exception**

The step-by-step procedure for running the **MultipleRulesExample.java** class is given as follows:

1. We create our customer order.

**//Initial order**

**CustomerOrder candyBarOrder = new CustomerOrder(2000);**

1. We create a placeholder for the starting date for a first shipment. We add it under the **NEXT\_AVAILABLE\_SHIPMENT\_DATE**, so that we can pass it into the rules as a global variable.

**HashMap<String,Object> startDate = new**

**HashMap<String,Object>(); startDate.put(NEXT\_AVAILABLE\_SHIPMENT\_DATE, new OoompaLoompaDate(2009,02,03));**

1. We create the two Oompa Loompa holidays that we will pass to the rules later.

**//Holidays**

**OoompaLoompaDate holiday2= new OoompaLoompaDate(2009,2,10);**

**OoompaLoompaDate holiday1= new OoompaLoompaDate(2009,3,17);**

1. We print out our starting point before we fire the rules.

**log.info("===== Setup ====="); log.info(candyBarOrder);**

1. We put our facts into an array (a type of collection) so that we can pass them all at once.

**//Call the rule engine**

**log.info("========= Calling Rule Runner ==========");**

**Object[] facts = new Object[3]; facts[0]=candyBarOrder; facts[1]=holiday1; facts[2]=holiday2;**

1. We call the rules via the **RuleRunner** class that we looked at earlier.

**// A lot of the running rules uses the same code. The**

**// RuleRunner (code in this project)**

**// keeps this code in one place. It needs to know**

**// - the name(s) of the files containing our rules**

**// - the fact object(s) containing the**

**// information to be passed in and out of our rules**

**// - a list of global values**

**new RuleRunner().runRules( RULES\_FILES, facts,startDate );**

1. Finally, we print out our results.

**//Look at the results**

**log.info("======= Results - shipping schedule ======="); log.info(candyBarOrder);**

### Running the sample

Let's run the sample. Right-click on the **MultipleRulesExample.java** file in either the navigator view or the package view (just as we did for **DroolsTest** in the previous chapter). From the pop-up menu that appears, select **Run As** | **Java Application**. After a second or two the output should appear on the console.

#### Console

1. In the output we can see the starting customer order (date and time removed for clarity).

**INFO: ===== Setup =====**

**INFO: Initial Chocolate Order:2000 itemsStillToShip:2000 shipments:nonelisted**

1. We can see the rules file being loaded.

**INFO: =========== Calling Rule Runner ==============**

**INFO: Loading file: src/main/java/net/firstpartners/chap6/shipping-rules.drl**

**INFO: found file:src/main/java/net/firstpartners/chap6/shipping-rules.drl**

1. We can see the globals being passed into the rules session.

**INFO: Inserting global name: nextAvailableShipmentDate value:03/02/2009 INFO: Inserting handle to logger (via global)**

1. The next step we see is that the rule engine is called (we pass the facts as we call the rule engine).

**INFO: ========= Calling Rule Engine ===========**

1. The first rule fires twice, confirming the facts/dates that Ooompa Loompas take a holiday.

**INFO: Remember - Ooompa Loompas don't work on:10/02/2009**

**INFO: Remember - Ooompa Loompas don't work on:17/03/2009**

1. The shipment and the shipment date rules fire (several times) in turns.

**INFO: Fired Customer Shipment rule - customer is still waiting for 1790 chocolate bars**

**INFO: Add Next Available Shipment Date:03/02/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 1580 chocolate bars**

**INFO: Add Next Available Shipment Date:10/02/2009**

1. Because **10/02/2009** is a holiday, our holiday rule fires.

**INFO: Reschedule Shipment Date to:11/02/2009 due to holiday on:10/02/2009**

1. The shipment and the shipment date rules fire in turns again, with our holiday rule firing again as we don't work on March 17.

**INFO: Fired Customer Shipment rule - customer is still waiting for 1790 chocolate bars**

**INFO: Add Next Available Shipment Date:03/02/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 1370 chocolate bars**

**INFO: Add Next Available Shipment Date:17/02/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 1160 chocolate bars**

**INFO: Add Next Available Shipment Date:24/02/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 950 chocolate bars**

**INFO: Add Next Available Shipment Date:03/03/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 740 chocolate bars**

**INFO: Add Next Available Shipment Date:10/03/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for 530 chocolate bars**

**INFO: Add Next Available Shipment Date:17/03/2009**

**INFO: Reschedule Shipment Date to:18/03/2009 due to holiday on:17/03/2009**

1. The rules continue firing in this manner until we reach the final shipment, which should contain only 110 bars. However, our normal shipping rules put too many in the box, 210, leaving us with a negative number. At this point, our **Overshipping** rule steps in and corrects the number.

**INFO: Fired Customer Shipment rule - customer is still waiting for 110 chocolate bars**

**INFO: Add Next Available Shipment Date:31/03/2009**

**INFO: Fired Customer Shipment rule - customer is still waiting for -100 chocolate bars**

**INFO: Removed CustomerOrder Overshipping - new shipment:110**

**INFO: Add Next Available Shipment Date:07/04/2009**

1. At this point, our **CustomerOrder** object has been completely populated, and there are no more rules waiting to fire. Control returns to our Java code, which prints out our completed shipping schedule.

**INFO: ======= Results - shipping schedule =======**

**INFO: Initial Chocolate Order:2000 itemsStillToShip:0 shipments:**

**Shipment:210 date:03/02/2009 chocolate bars left in order:1790**

**Shipment:210 date:11/02/2009 chocolate bars left in order:1580**

**Shipment:210 date:17/02/2009 chocolate bars left in order:1370**

**Shipment:210 date:24/02/2009 chocolate bars left in order:1160**

**Shipment:210 date:03/03/2009 chocolate bars left in order:950**

**Shipment:210 date:10/03/2009 chocolate bars left in order:740**

**Shipment:210 date:18/03/2009 chocolate bars left in order:530**

**Shipment:210 date:24/03/2009 chocolate bars left in order:320**

**Shipment:210 date:31/03/2009 chocolate bars left in order:110 Shipment:110 date:07/04/2009 chocolate bars left in order:0**

That's it. The example has now finished running.

### More powerful rule syntax

Some of the rules in the previous section hinted at more powerful key words than we have used in our rules to date. Let us go through the remaining key words in our rules and explain what they mean. You're unlikely to use all of these, although it is good to know that they are there for some of the weird and wonderful rules that you'll be asked to write (or review). More information on the rules syntax can be found in the Drools documentation, if you're into that kind of thing.

#### Lefthand side

##### not

We came across **not** at the start of the chapter, where we used it within an expression to select lines where **Sales** were not equal (!=) to 100. We can also use it at the top level of a rule. For example, if we add **not** to our last shipping rule we completely change its meaning. In this case, it fires only if we do not have any customer orders left to fulfil.

**rule when not (**

**$CustomerOrder : CustomerOrder(currentBalance<0)**

**$ChocolateShipment : ChocolateShipment(itemsStillToShip<0)**

**) then**

Note the use of brackets after **not**—they make clear to Drools the order in which we want to read the conditions.

##### Contains and memberOf

If we wanted to check whether a customer order had shipments, we could use the **contains** keyword. **contains** allows us to examine a list or collection (in this case, our shipments) and act accordingly. The following rule will fire when a particular shipment (if we previously assigned a value to **$mySpecialShipment**) is actually a part of a customer order.

**rule when**

**CustomerOrder(shipments contains $mySpecialShipment)**

**) then**

The **not** **contains** keyword would have the opposite effect.

**memberOf** is a similar test, but in the reverse order. We have a handle to a fact, and we want to check if it is contained within a wider collection (for example, **$shipmentsToAcmeCorp**).

**rule when**

**Shipment(shipment memberOf $shipmentsToAcmeCorp)**

**) then**

Similarly, **not** **memberOf** is the reverse of this.

##### matches and Soundslike

Both **matches** and **soundslike** allow powerful text comparisons. Previously, we compared **text1 == text2** using the standard equals operator. **Soundslike** uses the well-documented Soundex algorithm (**http://en.wikipedia.org/wiki/Soundex**) to gauge if two pieces of text sound similar when spoken—for example, to match all **Sales** where the customer name sounds like **acme**.

**Sales (name soundslike "acme")**

**matches** is even more powerful, but requires an understanding of the industry standard regexp (regular expressions). For example, the following condition will match all customer names that begin with **a**:

**Sales (name matches "a\*")**

We can also use **not** **matches** and **not** **soundsLike**, if required.

##### in

**in** allows Drools to check if a single value is a part of a collection. For example, if we want our rule to fire only for our three favourite customers (**Acme** **Corp**, **Breakfast** **Roll** **Inc** and **Chocolate** **Creams** **Co**), we could add the following condition:

**Sales (name in ("Acme Corp","Breakfast Roll Inc","Chocolate Creams Co")) exists**

Another way of saying **exists** is 'at least one'. So the following rule will fire when we have pushed at least one customer shipment out the door:

**rule when**

**exists (ChocolateShipment()) then**

**log.info("Hurray! We made a shipment!");**

**end**

##### forall

Normally, when we write a rule, it will match against any of the facts in memory and fire against the matched facts. But what if we want our rule to fire only if all of the facts match our condition? Something like 'fire only if all our customer shipments have dates' could be written as follows:

**rule when**

**forall( Shipment( shipmentDate !=null ) ) then**

**forall()** can also take multiple conditions, all of which have to be true for all facts matched, before the rule fires. **collect/accumulate**

**collect** allows us to check how many facts in memory match a given rule, and then test the results. For example, if we had more than one customer order, we'd have no easy way of counting the total number of shipments that we made on a particular date. (We can get shipments for single customer using **getShipments()**, but this approach won't scale.)

The following rule will display the total number of shipments (for all customers) on 1st January, 2009:

**rule when $numberofShipmentsOnDay : collect (**

**ChocolateShipment(**

**new OoomplaLoompaDate(1,1,2009)**

**)**

**) then**

**log.info("Boxes shipped today:"+$numberofShipmentsOnDay); end**

**accumulate** is a more powerful form of **collect**. It can do everything that **collect** can do, along with additional things such as firing custom code as it works it's way through the collection. (The Drools documentation has more details on how to use this power.)

##### eval

Sometimes, after looking at all of those expressions, you still can't find one that fits. **eval** allows any expression or formula that returns a Boolean value (that is, a true or false answer) to be used in evaluating a rule. For example, if we want to get the names of customers that are longer than 10 letters, we could use the following:

**rule when**

**$s : Sales()**

**eval ($s.getName().length >10) then**

**eval** is also useful when calling functions. For example, if we had a function, 'name too long', which returned a Boolean, we could call it using **eval(nameTooLongFunction())**.

**eval** is very convenient as it allows us to include pretty much any condition in a rule. However, it's considerably slower. With other conditions, Drools can cache (remember) the results because it can figure out when these results need to change. With **eval**, Drools doesn't have this visibility. Therefore, all **eval** expressions need to be rechecked every time the rule is true.

If you have to use **eval**, it's best to add it as the last condition in a rule—meaning, it will be called less often. If any of the previous conditions return 'false', then Drools shortcuts, because there is no need to check any of the remaining conditions.

##### from

For the examples we've given so far, we've expressed (or pushed) our information (the Java facts) into our rules. The **from** keyword allows the rule engine rules to pull information as required, for example, from a database using the Hibernate framework. Assuming that we have correctly set up Hibernate to talk to the database (a topic so vast that a whole book about it can be written), our rule can use the following syntax:

**rule when**

**Order() from $myHibernateObjectPassedAsGlobal.subquery then …**

#### Righthand side—Then

Just like the When part of the rule, there are also more powerful options available on the righthand side (Then clause) of the rule. At the start of the chapter we came across simple uses of **update()** and **insert()**, but the following keywords are also valid in our rules:

* **$someHandle** **=** **insert(something)**: This form of insert works the same as **insert** **(something)**, that is, putting 'something' into the working memory. The difference is that we keep a hold of it (in the variable **$someHandle**) so that we can do things with it later on.
* **retract($someHandle)**: This removes a fact from the working memory. For example, we may decide to insert a fact representing a discount, only to retract it later if the customer cancels an order.
* **insertLogical(new** **someFact())**: At the start, **insertLogical()** works the same as insert—it puts an object into memory. The only difference is that as soon as the rule stops being true, the fact (someFact) is automatically retracted from the working memory.
* **update(object,$somehandle)**: This is similar to **update** (object), although we can specify to the object the handle that we wish to update. If the handle is not passed (that is, the update we did previously), Drools can use a 'best guess'.

The advanced 'When...Then' options are available both in the text editor (like the previous examples) and the guided editor.

### Guided editor in the JBoss IDE

Until now we've focussed on using the advanced text editor in the JBoss IDE. But it's also possible to use a guided editor, similar to the one we used in Guvnor. To use this editor, you can take the following steps:

1. Open any Drools-enabled project in the JBoss IDE, and right-click on the folder where you want to create the file.
2. From the context menu that appears, select **New | Other | Guided rule**. (You may need to filter, or open, the **drools** folder in the dialog box to see this last option.) Click on **Next**.
3. Give the new rule file (for example, **myNewRule**) a name, and click on **Finish**.
4. That's it. You should see a new tab open in the editor, as shown in the following screenshot:



The guided rule editor is very similar to the guided editor in Guvnor, which we covered in the previous chapters. It has the same 'when...then' options layout. The guided editor also has tabs to allow us to view the text that was built behind the scenes using this graphical builder.

## Summary

This chapter extended the limits of what can be done with business rules. We went through almost every keyword available to us in writing rules and used them to generate a sample to help the Ooompa Loompas pack chocolate bars onto trucks. Now that we understand how the rules work, in the next chapter we will look at testing to ensure that our rules keep on working the way we intend them to.