# Looking under the Cover

In the previous chapters, we've treated the rule engine as a 'black box'. We've described what goes in and what comes out, but not what happens 'under the covers'. In this chapter, we open the box and explain how a rule engine works. We demonstrate how it is a faster and easier-to-maintain solution than traditional coding methods. We'll also introduce debugging and other ways of seeing what is going on inside the rule engine in real time, as it happens.

In this chapter, we will cover the following topics:

* Rule engine concepts
* Logging
* Rete
* Debugging rules

## Rule engine concepts

To understand what we're seeing when we look inside the rule engine, we first need to understand a couple of rule engine concepts, including a repetition (in more depth) of something we first covered back in Chapter 1.

### Facts or objects

Facts are pretty straightforward. They're the container that we use to transport information into (and out of) the rule engine. You'll remember that because facts are standard JavaBeans (we compared them to Lego blocks), a lot of the code already in your organisation can be used for this purpose. Or, you can write your own—such as the **CustomerOrder** and **ChocolateShipment**.

An 'object' is just another term for a fact (it's where the term **Object** **Orientated** **Programming** or **OOP** comes from). Earlier, we saw how we can insert, update, and retract (remove) facts or objects from the rule engine, and how the rules would react as a result.

Or rather, we inserted, updated, and retracted the facts from the **Working Memory**, and saw the **Agenda** and **Activations** change as a result. All three of these are parts of the Drools Rule Engine, and are a critical part of what happens 'under the covers'.

### Working memory

To keep things simple, all of the examples that we've given until now have been about a 'single user'. That way, we could treat the rule engine and working memory as almost the same. In real life, the sorts of Enterprise Applications that use rules will have many people calling rules at the same time. If all of these people shared the same memory, then things could get pretty confusing. (Imagine if the rules returned somebody else's bank balance!) The solution is to give everybody their own workspace, or working memory.

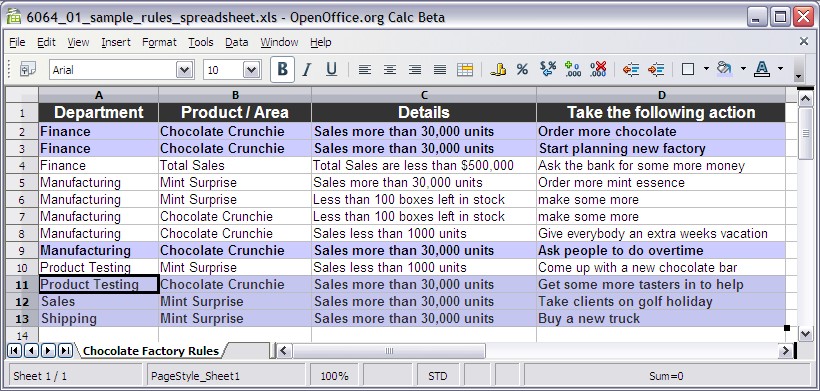
If you're used to working with web servers, you will notice a similar problem and a similar solution—each user who is interacting with the server has his or her own unique session, which is isolated from any other user's session.

We touched upon working memory when we used stateful sessions as a part of the ruleflow, but working memory is also behind the stateless session. Both stateless and stateful sessions are types of working memory. In both cases, working memory acts as a scratchpad, unique to each user, which contains all of the knowledge (facts) that Drools has been told about the case. When we insert, update, or retract a fact, it's the working memory that changes. Likewise, when a rule inserts, updates, or retract a fact, it's the working memory that changes.

Let's repeat it because it's important: Rules change facts in the working memory. Because of the way that Java works, even though our facts are a part of working memory, the code that passed in the values still has a 'handle' to them. Thus, it automatically knows about the changed values.

### Pattern matching, Agenda, and Activations

You'll remember the spreadsheet below from Chapter 1. It is about the business rules for the chocolate factory. It's far simpler than the Excel-based decision tables that we used later in the book. In the following table, the first three columns represent the 'when' part of the rule, and the final, fourth column shows the actions to be taken:



Highlighted are the rules that become available to fire whenever we sell more than 30,000 Chocolate Crunchie bars. This pattern matching (identifying the relevant rules for any given situation) is a core part of Drools, and something that rule engines do very well.

In real life we would insert facts to the rule engine (or rather, to working memory), telling the rule engine that all those candy bars have been sold. As soon as we did so, the rule engine would identify the relevant rules via pattern matching. These rules would become 'active' or ready to fire. So, *Activations are rules that are ready to fire, and Agenda is the list of all of these activations*.

### Conflict resolution

Note that we didn't say Activations 'are rules that fire' but 'rules that are ready to fire'. Just because there are facts in the working memory that match a rule (that is, a rule *could* fire) doesn't mean that the rule will fire. The first question, which might not be obvious from the chocolate factory sample, is which rule amongst the six highlighted ones will fire first?

Because (at a simple level) only one rule fires at a time, Drools needs to decide which of these rules should fire first. To make this decision, let's introduce an idea called **conflict resolution**.

Strictly speaking, more than one rule can fire at a time, especially if you have multiple working memories. Within the working memory, the Rete algorithm optimises the matching and firing process.

As soon as the 'when' part of the rule it satisfied, it is added to the Agenda as an Activation. This creates a list of rules that are available to fire. If there is only one rule Activation on the agenda, there is no need for conflict resolution—it is fired straight away. If there are two or more rules, conflict resolution is used to put an order to the list.

By default, conflict resolution decides the order of priority in the following manner:

1. **Salience**: Back in Chapter 5, we came across salience as a rule attribute that we could add via Guvnor (or the other rule editors). Under this conflict resolution method, a rule with a higher salience value will be closer to the top of the agenda, and thus more likely to fire. Remember that salience only changes the order of rules whose 'if' part matches the current facts. If this doesn't happen, you can increase the salience if you wish.
2. **Recency:** If salience doesn't resolve the rule agenda order, then Drools looks at Recency, or how many times a rule has fired previously. The more the rule has fired, the higher it will be on the agenda.
3. **Complexity:** A rule with more conditions in the 'when' part will be more specific, that is, it will be fussier when it fires. On the other hand, when it does fire, it is likely to be more relevant to the current situation. Therefore, the more complex rules tend to move toward the top of the agenda.
4. **LoadOrder:** If none of the first three strategies work, then the fallback situation is to use the LoadOrder of the rules—that is, rules that got loaded first get fired first. Because we can load rules from many different files, we can implicitly state which blocks of rules are more important than others. If you need to refresh your memory, look at **RuleRunner.Java**, from the previous chapter.

Although this is the default conflict resolution strategy, it is possible to change the methodology employed. There can be good reasons for doing this, but in general, a well-written set of rules shouldn't need this level of fine-tuning.

### A more dynamic Agenda

The order in which rules/Activations appear on the Agenda is critical. This is because only the first rule on the Agenda is guaranteed to fire, even though all of the other rules on the Agenda match the current set of facts in the working memory. The other rules may fire in turn, but only if an earlier rule hasn't changed the facts in the working memory. This means that rules can be dropped from the Agenda before they get a chance to fire, because there are no longer valid.

Let me explain how housework and chores get done in my home. My wife gives me a list of tasks to complete—wash the dishes, brush the floor, and put out the bins. By the time I finish washing the dishes, my wife has lost patience and has already brushed the floor—so that's no longer on my 'todo' list. On the other hand, she has now asked me to tidy up the yard and wash the car, so that gets added to the list.

Think of the 'to-do' list as the Drools Agenda and you'll get an idea of the dynamic addition or removal of 'rules to fire' from the list.

### Truth maintenance

We change the working memory by inserting and removing facts. Changes in working memory cause rules to match, get added to the agenda, and fire. Because rules can also insert, remove, or update the facts in the working memory, a rule firing can cascade and cause other rules to be added to the agenda and subsequently fire. These rules, in turn, can cause further rules to become active and fire.

What is going on here is the process of **truth maintenance**. Given a set of facts, the rule engine applies the appropriate business rules and modifies the facts as necessary. If the newly-changed facts mean that other business rules come into play, then they are applied. Eventually, all necessary rules are fired, and the facts stored in the working memory represent the truth, or at least the truth as understood by our business rules.

Any change in facts (for example, if a user submits more information about his or her insurance application) causes new rules to come into play, in an attempt to maintain the truth.

Truth maintenance is a bit like playing pool (or snooker) because of the following reasons:

1. You take a shot, hitting the white cue ball (that is, introduce a new fact to the system).
2. The rules of physics are applied and the ball rolls across the green baize, and bounces off the cushions.
3. It may hit another ball (that is, cause another rule to fire). The rules of physics are applied again, and now two balls are moving.
4. These may hit other balls (more facts getting introduced, more rules getting fired, and more balls getting moved—although slightly more slowly), until eventually the whole system (all the balls on the table) is still again.
5. The truth (according to the laws of physics) has been maintained! Or at least, until another player takes a shot.

This 'rules changing facts causing other rules to fire' process can lead to rules getting caught in a loop. Rule 1 changes a fact, which causes rule 2 to fire, rule 2 changes a fact, which causes rule 1 to fire, and so on. There are a couple of features in Drools that help to avoid this. Marking a rule with an attribute of 'no-loop' means the rule cannot cause itself to fire. In Ruleflow, Agenda groups control the amount of rules available to fire. Writing the conditions (the 'when' part of the rule) as precisely as possible also helps to avoid this scenario.

If you're a programmer, truth maintenance is a fundamental difference to any language you may have used before. With those languages, you (the programmer) have to ensure that your code gives the correct results. With Drools, the rule engine ensures that the result is always consistent with the business rules that you have given it.

## Back to the future (with chocolate shipping)

We've just looked at a couple of make-believe rules from the chocolate factory. Wouldn't it be great to see some real rules and use it to see the internals of the rule engine that we've just talked about? Fortunately, our Chapter 11 example allows us to do just that (this example is available for download from the same site as the previous samples, **http://code.google.com/p/red-piranha/**).

This sample is the same as the chocolate factory that we saw back in Chapter 6, but with some extra code that makes it easier to see the internal happenings in the rule engine. The following are the basic rules found in the **shipping-rules.drl** file:

* A rule to confirm (print out) the OoompaLoompa holidays is passed, as a fact, to the rule engine
* A rule matches against unfilled customer orders, and starts sending out shipments of 210 units at a time
* A rule adds an estimated date to these new shipments of chocolate
* A rule adjusts this shipment date if it falls on a day that the OoompaLoompas take as a holiday
* A rule ensures that we don't overshoot on the last shipment, adjusting it so that only the remaining balance is sent

Running the Chapter 11 example is similar to before. Open EventRulesExample.

java, right-click anywhere in the file, and select **Run As | Java Application** from the context menu that appears.

The sample comes with an Eclipse project, so you should be able to open it with **File | New Project**, and then select the folder to which you unzipped the sample. If this doesn't work (for example, you are using a newer version of Eclipse since the book came out), you can regenerate the project with Maven (assuming that you have it installed). Open a command window (Dos Prompt) at the folder containing the project (the one containing **pom.xml**). After executing the command **mvn** **eclipse: clean** **eclipse:eclipse**, refresh or reopen the project in Eclipse, and everything should be set up correctly.

After running the sample, you should get output similar to the following in the Console window as before, ending the schedule that we have calculated.

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

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

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

**….**

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

**Shipment amount:210 date:31/03/2009 chocolate bars left in order:110**

**Shipment amount:110 date:07/04/2009 chocolate bars left in order:0**

## Logging working memory

We've used logging in our earlier examples to print what is going on to the Console. The output above, showing the shipment amounts, is a good example of this. Mostly, this is our own logging statement. Although we can add logging statements that get output when the rules fire, until now we've treated the decisions by the rule engine about which rule to fire as taking place in a 'black box'. Wouldn't it be great to look inside and see which rules are getting fired, and why?

The main difference between the example in this chapter and the one in Chapter 6 is that the code in **RuleRunner.Java** has been modified to display what is going on internally in the rule engine. The main lines that have been added to this file are:

**// create a new Working Memory Logger, that logs to file.**

**WorkingMemoryFileLogger wmLogger = new**

**WorkingMemoryFileLogger(workingMemory);**

**// Set the log file that we will be using wmLogger.setFileName("event-log");**

**….**

**//Fire using the facts workingMemory.execute(facts);**

**//stop logging wmLogger.writeToDisk();**

It's not a big change. The snippet above has only four lines of code, of which (**workingMemory.execute**), we're familiar with one already. What happens is:

* We create a new **WorkingMemoryFileLogger**, passing in a handle to the working memory that we want to log
* We set the name of the log file in which the information will be written (in this case **event-log**)
* We fire the rules as normal, using **workingMemory.execute**
* When the rules have fired, we write the log to disk using **wmLogger.writeToDisk**

In real life the log file can become very large, so be careful to disable the logging of production systems. At the simplest level, this means commenting out the above code (by placing '//' in front of it).

### Looking at the working memory log

Assuming that you ran the sample in the previous section, you'll already have the **event-log.log** file generated. This file should sit at the root of the project (just under the **chap11-sample** folder in Eclipse, next to the **pom.xml** file).

Eclipse doesn't display changes to files by default, so you may need to refresh the project to see the changes. To refresh your project in Eclipse, select the project name, then press the **F5** key or choose **File | Refresh** from the main menu.

If we open this file, we'll see something similar to the following text:

**<object-stream>**

**<list>**

**<org.drools.audit.event.ActivationLogEvent>**

**<activationId>Chocolate Shipment [1]</activationId>**

**<rule>Chocolate Shipment</rule>**

**<declarations>$CustomerOrder=Initial Chocolate Order:2000 itemsStillToShip:2000 shipments:none-listed(1)</declarations>**

**<type>4</type>**

**</org.drools.audit.event.ActivationLogEvent>**

**<org.drools.audit.event.ObjectLogEvent>**

**<factId>1</factId>**

**<objectToString>Initial Chocolate Order:2000 itemsStillToShip:2000 shipments:none-listed</objectToString>**

**<type>1</type>**

**</org.drools.audit.event.ObjectLogEvent>**

**<org.drools.audit.event.ActivationLogEvent>**

**<activationId>confirm holidays [2]</activationId>**

**<rule>confirm holidays</rule>**

**<declarations>$holiday=17/03/2009(2)</declarations>**

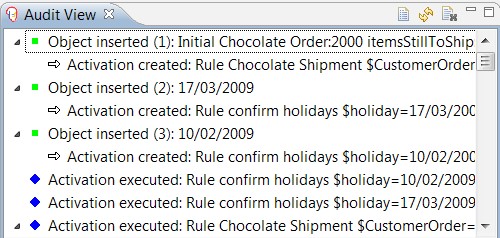
**<type>4</type>**

**</org.drools.audit.event.ActivationLogEvent>**

**<org.drools.audit.event.ObjectLogEvent>**

### Drools Audit Log Viewer

Not very readable, is it? To understand what the working memory log is showing, we need to use the Drools Audit Log View.



To open this view in Eclipse, select **Window | Show View | Other | (Drools Folder) Audit View** from the toolbar. You should see an empty version of the above window. The window is empty as we haven't told the viewer where our working memory log file is.

See those icons at the top right of the **Audit View** window? Click on the file icon (it's the highlighted one on the left, looking like a piece of paper) and choose the audit log that we opened earlier.



Now the window looks a bit more like the **Audit View** in the diagram above. But what does it actually mean? Working through the list, you'll see that the first item in the log is where the initial chocolate order is inserted into memory. The icon is a green square.



Because the rules start firing as soon as the facts are inserted, (before the second fact is inserted) an Activation is immediately created for the **Chocolate** **Shipment** rule. The icon is a right-facing arrow.



The third line in the log is the second object being inserted (one of the OompaLoompa dates passed in as a holiday), causing another activation to be created for the rule **confirm** **holidays**.

A couple of lines later, we see the first rule—the one that confirms holidays—actually being executed. The icon is a blue diamond.



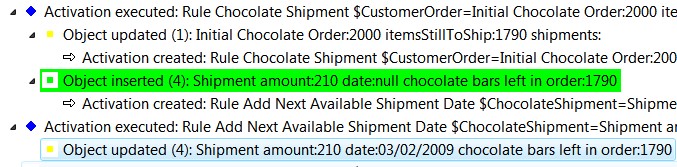
In total, there are three activation executions in this part of the log. Note that the executions are not in the order in which they were created, but in the order that Drools deems fit when it resolves the conflicts. As the third rule is the **chocolate** **shipment** rule, we immediately see an object update in the log. The icon for an object update is a yellow square.



The line immediately following this in the log is a new rule/Activation created (as a result of this object being created). The line immediately after that is interesting—it is another object insertion (we've seen those before). Because the object (a shipment) has been created or inserted by a rule (the **chocolate shipment** rule), with an existing object being changed, the following line has a new Activation created (**shipment** rule).

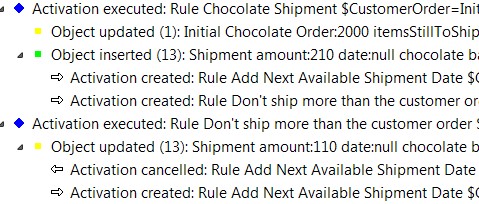


Even in this simple example, there are a couple of hundred lines, so we'll skip them and move on. Before we do this, it's a good time to mention the **show cause** functionality in the **Audit View**. To activate it, right-click on the log and select **Show Cause** from the context menu.



For this example, we clicked on the **Object updated (4)** and the action that caused it is highlighted. As it happens, what caused the update was the action in the previous screenshot (that is, the object that had just been inserted by the rule).

The previous section talked about conflict resolution when there are more than two rules on the Agenda available to fire. Near the very end of the audit log is an example of this happening.



The conflict happens where the two right arrows (that is, rules that are on the Agenda) fire one after the other. The rules are **Add Next Available Shipment Date** and **Don't ship more than the customer order**.

Obviously, this conflict is resolved as one of these rules is activated and fired in the next line: the **Don't ship more than the customer order** rule. This section also shows an Activation cancellation—the second line from the bottom of the screenshot. The icon for a cancellation is a left-facing arrow.

There is another icon, not shown in the above samples, when an object or fact is removed from the working memory. The icon for fact removal is a red square.

Other icons not shown in this diagram, but which might appear in your audit log, are:

* Drools logo icon: Rule or rule package added or removed
* Process icon: Ruleflow started or ended, or Ruleflow group activated or deactivated

## Rete algorithm

The Rete algorithm is at the heart of the events that we just saw in the audit log. The one-line explanation of Rete is:

Rete makes rules run incredibly fast by sorting the rules in such a way that when facts change, the rule engine knows instantly which rules need to fire.

The Rete algorithm is a bit like the algorithm used by Google for searching the entire Web. You don't need to know how it works to use it, but it's nice to have a vague idea. If you want to know more than the overview here, there are hundreds of pages on the Web, including *Dr Charles Forgy 1974 to 1979 original papers*, where he first outlined Rete—just do a Google search for *Charles Forgy Rete Algorithm*.

If I was to write a rule engine, and didn't know about Rete, I would do it incredibly badly. I would probably write a series of loops checking each part of my business rules every single time a fact might have changed, including every time a rule fires.

For example, the shipping rules have 5 rules, with 3 facts, each holding about 4 values.

In my loop (which I would have to run every time something changed) that would be 60 values (5x3x4 = 60) that I need to check after every single change. That's 60 values checked hundreds or thousands of times, over the life of a simple application.

That's not going to scale very well—a medium-sized sample would have 100 rules, with 20 facts in total, holding about 4 values each. This becomes 8000 items to be checked, and checked repeatedly. Even if we check 2000 items a second, it comes to a 4-second delay each and every time something changes, which can be very frequent in an enterprise application.

Don't laugh at this description. If you've written any code, you've probably made the same mistake yourself (I have).

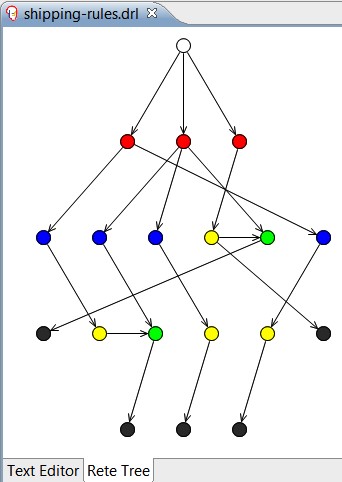
Think of the hundreds of 'if…then' statements that you have scattered throughout your code, and imagine them gathered together in one place, often getting called more than once. Sounds a bit more like the example above? That (along with other optimisations) is why rule engines are often more efficient than the equivalent 'traditional' code.

It's just as well that I don't write rule engines. Let's try again.

Mostly, the checking of facts is slow because there are duplications in the checks. For example, if both rules use a common shipment date, the same fact will get checked twice for changes. What if we spent a bit of time grouping the rules so that one check will do everything for both rules?

### Rete in action

One example of this grouping is shown in the following screenshot, from the Drools IDE. It shows the shipping rules from our example, but displayed as a **Rete Tree**. To view this yourself, open the **Shipping-rules.drl** file as normal. Using the tabs at the bottom of **Rete View**, click on **Rete Tree**.



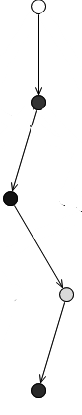
While this looks complicated, the concepts are simple. Drools (the white node at the top) has analyzed the 'when' part of the five shipping rules, and picked out the facts (the three red nodes on the second line), and the fields on those facts (the four blue nodes, most of the third line). These are then connected via the green and yellow nodes on the third and fourth line, (used to cache the previous values to further reduce the amount of checks that we need) to the black nodes (the rules that can fire) at the very bottom of the diagram.

Remember that when we were running our previous examples, there was a noticeable pause before the rules fired. This was the time taken to construct the Rete network from the rules. It's a trade off—a slight pause at the start, compensated for by by a greatly-increased performance later.

To analyze which rule needs to fire, all that Drools has to do is to start at the top node and, depending on what has changed, walk down the tree until it gets to the black node (representing the 'when' part of the rule) to fire. That's the hard bit done. Firing the rule once it's been decided is more or less standard Java.

To find out more information on each node, click on it in the **Rete Tree View**. If you have the property view open in Eclipse (from the toolbar, select **Window | Show view | Properties**), you'll see the name of the fact/field/rule displayed.

To give an example of this 'walking the Rete tree', imagine that a shipment date changes on one of the shipments. The path that Drools will follow is shown below. From these five nodes, one is standard Drools (the one at the top), and the next two are 'real' checks on the fact and field respectively. The fourth node is an internal connector. The final node is the actual rule that will fire.



In this example, when the shipment date changes, instead of the of the 60+ checks previously required, now only two checks need to be done (in the second and third nodes).

Two checks are obviously faster than 60 (or 8000 in the larger sample), especially if they are repeated 100 times. The actual Rete algorithm is much more sophisticated (and faster) than we describe. Drools has further optimizations that make the process faster still—it's easy to optimise business rules when they are written in standard 'when…then' format.

### Debugging rules

Now that we understand how a rule engine works, let's take a peek at its internals, in real time.

This book has shown you how to print logging statements to the console to see what is going on in your rules. It has also shown you how to use the working memory logger to show you what is happening within the rule engine. But these methods are 'after the fact'—the rules have completed and we are trying to reconstruct what has happened, long after the program has finished. What if we want to see rules firing and values changing as they happen?

What we need is debugging. Debugging is like pausing a movie on a DVD player. Normally, we would run our program (movie) from end-to-end using the play (run) button. If we don't like the movie (program) we can halt it by clicking on the stop button (or pressing *Ctrl+C*). If we need to take a break we can pause it, and maybe change a few of the movie options such as to view it in widescreen, or to switch the director's commentary on or off, and so on.

If you've already debugged Java programs, you'll recognise the above description. The tools we're about to describe are very familiar. But how we do start to debug the rules?

Normally, we can only debug rules when running them through the Eclipse IDE. To start debugging, we need to do the following two things:

1. We need to tell the Drools IDE that we want to run the rules in a debug mode, and not just run straight through (which is how we've run our examples up to this point).
2. We need to tell Drools where to 'pause'. Although it is possible to debug right from the start of the program, telling Drools to run until it hits a 'breakpoint' saves us from trawling through hundreds, or thousands, of lines of code to get to the bit that we're interested in.

When the debugger hits the breakpoint in our code, it will pause the program and allow us to see a snapshot of the frozen program. When we're ready, we can continue the program, or step through it one line at a time.

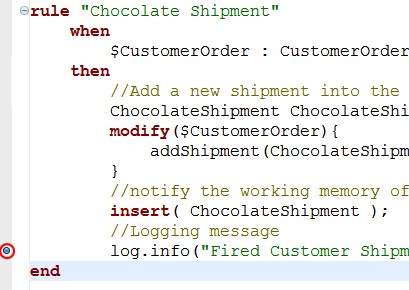
#### Debugging rules in the Eclipse IDE

If you have the rules project for Chapter 11 already open in Eclipse, then we're almost ready to start debugging. We'll first set the breakpoints—where we want the debugger to pause:

1. Make sure that the project has **Drools Nature** enabled. To do this, right-click on the project name, and then select **Convert to Drools Project** from the context menu. This command will only switch on Drools. So if you're not sure, it does no harm to do this again.
2. Open the rules file (**ShippingRules.drl**) as normal and go to where we want the debugger to pause.
3. Select the line where we want to place the breakpoint. (In our sample, scroll down to the second-to-last line of the **Chocolate** **shipment** rule.)
4. In the lefthand margin of the editor, right-click and select the command **Toggle breakpoint** from the context menu.

The screen should now look like the following screenshot (with the new breakpoint circled in red).

Using this technique, we can also set breakpoints in Java code—for example, at more or less any line after the **main[]** method in **EventRulesExample.java**. This is the method of debugging that programmers reading this book will be familiar with.



If you try clicking elsewhere in the rules file, you'll notice that *you can only add breakpoints in the consequence (then) portion of the rules*. This is because of the way that the Rete algorithm works—effectively, the 'when' parts of the rules are shared at run-time. Even if we could set a breakpoint in the 'when' part, the Drools IDE would have no way of knowing which of the shared rules we wished to examine.

Now that we've told the Drools IDE where to pause for debugging, we need to start the application in debug mode. This is almost, but not quite, the same as running the application normally.

1. Open the start point of the application (**EventRulesExample.java**).
2. Start debugging by right-clicking and selecting **Debug as | Drools Application** from the context menu.

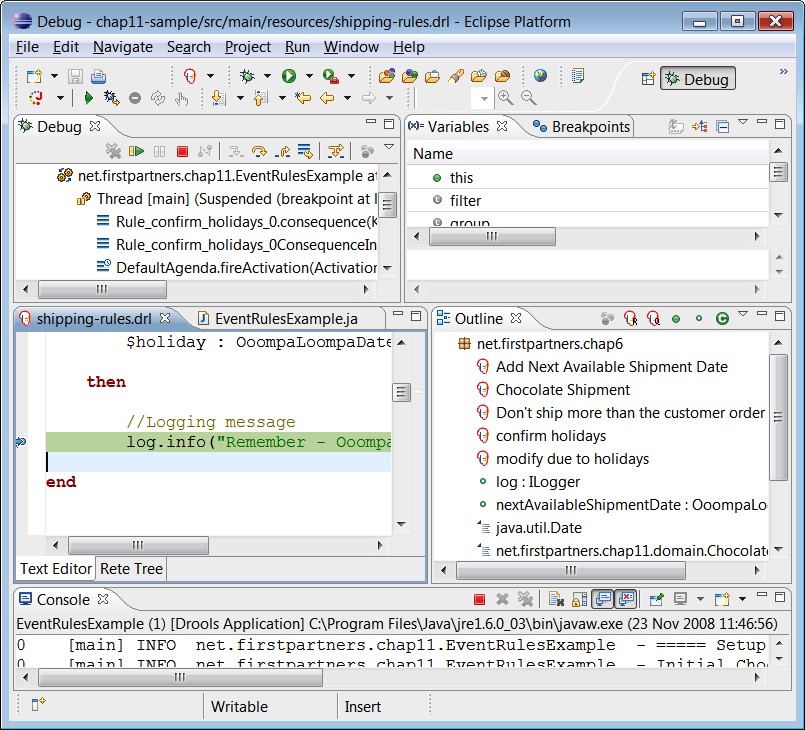
The program will now start running (much more slowly!) in debug mode. It is important to debug the program as a **Drools Application** (and not a standard debug application), otherwise the breakpoints that you set in the rules will not be noticed and the program will continue (slowly!) right until the very end.

A typical program will run five to ten times slower while debugging because of the additional checks (such as 'Do I need to stop now?') that need constantly to be made.

#### Rules debug perspective

When you started debugging, you may have been asked the question **Debug perspective is normally associated with this action—do you wish to switch to it now?** Click on **Yes**, and Eclipse will look like the example shown in the following screenshot. You can always switch later, manually, by selecting the Eclipse Toolbar and going to **Window | Open Perspective | Other | Debug**.

Both actions leave a button on the toolbar (top right of the picture) to make switching perspectives easier the next time.



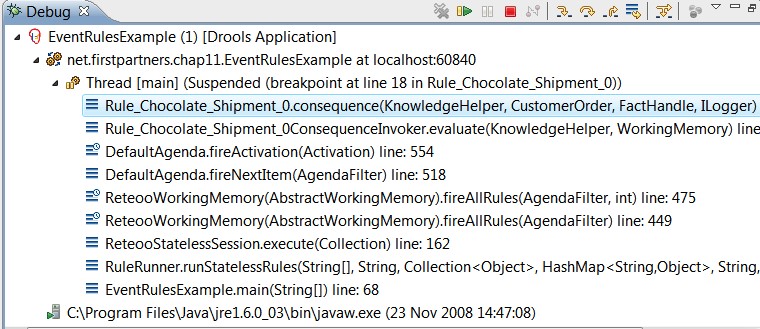
Previously, we opened views (such as the logging console) to give us more information about the project. Perspectives are just bundles of views that are displayed whenever we carry out a specific activity. Previously, (although we didn't know it) we were using the editing perspective. Now that we're debugging, it's useful to see our project from a different angle. The normal perspective is great for editing or writing files. The debug perspective (that we're looking at now) is great for debugging projects.

At first, the debug perspective looks very different from the one that we are used to. However, if we look at the screen from the bottom up, we'll see that two-thirds of the bottom screen looks familiar. It contains the following:

* The **Console** (the bottom panel): Like before, this shows text that has been output from the program
* The **Editor** (the middle left panel): This is for editing the **Shipping-Rules.drl** file, with the current line (where the program is paused) highlighted
* The **Outline** (the middle right panel): This shows an outline of the rules within the current file

The top two panels are new to this view. On the top left is the debug view (as shown in the screenshot below). For the moment, ignore the video type controls at the top of the panel, and look at the main panel instead. This main panel shows the stack trace.

When we start a program, our **main** method calls another method, which in turn calls another method until the breakpoint that we set is reached (and the program pauses). It's a bit like those Russian dolls—one method call fits inside another, and that call that fits inside another, and so on.

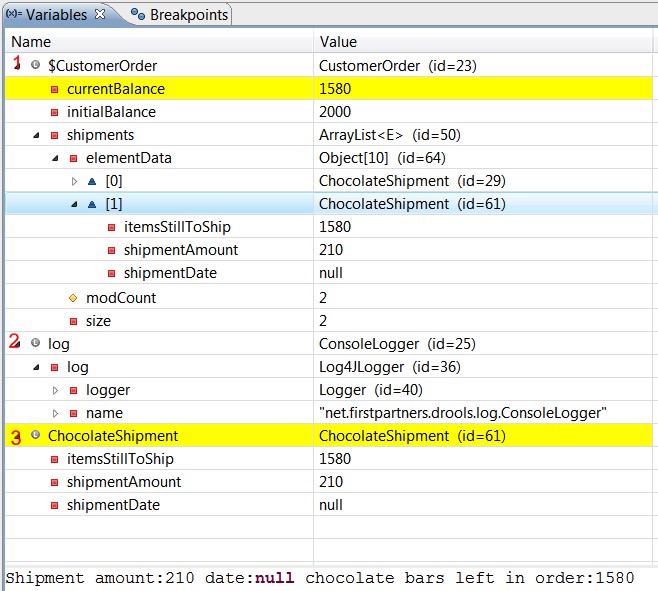


We read the stack trace from the bottom up. In this case it is an **EventRulesExample**, called the **RuleRunner**, which called various Drools files (**Retoo**, **Default Agenda**, and so on), which in turn called our rules (**Rule\_Chocolate\_Shipment\_0.consequence**).



At the top of the debug panel is a DVD-like control panel, which allows you to stop, start, and step through the program one line at a time (a bit like the 'forward by one frame' feature on most DVD players. The main icons or buttons are as follows:

* Resume (the green arrow) — continue until next breakpoint is reached, or the code ends
* Pause (greyed out): Pause the code that is running before it hits the next breakpoint
* Terminate (the red button): Stop the debug session.
* Arrows: Allow you to step through the code one line at a time



The other new panel is the **Variables** view, which is on the top right of the debug perspective (as shown in the screenshot above). This shows the variables (value placeholders) in our rules. More precisely, it shows only the variables that the current rule (highlighted in the editing window) has access to.

The variable view gives us a window into the program in real time. If we step through the rules line by line, we can see the values being updated step by step.

The format of the screen is a 'treeview', that is, there are three top-level variables (as used by this rule)—**$CustomerOrder**, a handle to the logger, and an individual chocolate shipment that is the process of being created by the current rule. The panel at the very bottom shows more detail on the currently-highlighted value (the chocolate shipment).

The three JavaBeans highlighted in the above variable view are as follows:

1. **Customer Order**: The same fact that was inserted by the Java code and updated by the rules. Remember that it contains the current and the initial balance, and a list (array) of all of the shipments as the rules create them. In the current snapshot there are two shipments that are already populated in the list (elements **[0]** and **[1]**). Each of these elements is a **ChocolateShipment** JavaBean (containing the shipment amount). Because this snapshot is taken before the set shipment date has had a chance to fire, the shipment date is null or empty.
2. **log**: This is a global variable, as passed in when we called the rule. It is displayed in this view. Even through it is a global variable, it is passed in to the rule.
3. **ChocolateShipment**: This is the same JavaBean that we are in the process of creating in this rule. (That is, it is the same chocolate shipment that has been added to the list at position number **[1]**.) Given that it's 'work in progress', the **shipmentDate** is **null** (empty).

Behind the variable view is a tab **Breakpoint**. This shows all of the sets in the rules and Java code. It is far easier to find them here than having to search through multiple files.

#### Other Drools views while debugging

There are other views available to help you while debugging. All of these can be added to the current view using the following option from the toolbar: **Window | Show View | Other | Drools**:

* Working memory: A live snapshot of the working memory, similar to the variables view above
* Agenda View: A list of the rules that are available to fire (and the order) in real time, as you step through the rules
* Rules: A list of the rules that have been loaded

### When to log, when to test, and when to debug

Debugging, testing, and logging are all useful tools to help you understand what is going on with your rules. But when it is best to use each of these tools can be explained as follows:

* Logging is used the most. This is especially true when using a toolkit such as Log4j or Apache Commons, where it can be turned on and off via a configuration file with little or no perceptible impact on performance.
* Writing a test is proactive, and will catch errors before the users do. It also has the advantage that once it is written it goes on working, and keeps checking for a recurrence of the problem, and alerts you if required.
* Debug is the most powerful feature. However, once you've solved the problem, most of the effort is lost; and if a similar problem happens in the future, you're likely to have to start again from the beginning.

## Summary

This chapter opened up the internals of the Drools rule engine so that we can understand concepts such as truth maintenance, conflict resolution, pattern matching, and the rules Agenda. We explored the Rete algorithm, and why it makes rules run faster than most comparable business logic. Finally, we saw the working memory audit log and the rules debug capabilities of the Drools IDE. The next chapter uses this power and knowledge to take advantage of some of the more advanced features.