# From Guvnor to JBoss IDE

In the previous chapter, we wrote our first real rule using the guided editor in Guvnor. Although the guided editor is very useful, the rule that we wrote could do with a few enhancements. We also had a glimpse of the behind-the-scenes text rules, but they were for read-only purposes. As a part of our enhancements, it would be good to use the additional power of these text rules. We'll look at using variables in our rules by using rule attributes to provide extra information about our rules, and editing text-based rules using Guvnor and the JBoss IDE.

## A more powerful rule

Our business rule from the last chapter was a bit silly. In real life there is no such thing as 'negative sales'. It would confuse not only our customers, but also our own company. (How would the Oompa Loompas in the shipping department put 10 boxes of chocolate into an empty truck?) Instead, we're more likely to modify the price of our sales and update this price with the discount, to give the actual price to be paid.

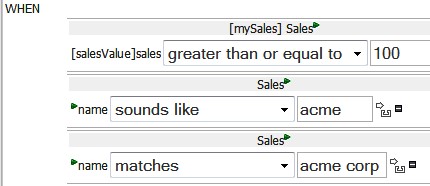
Applying this concept to our business rule, we'd end up with something like this:

**when**

**We have a sale greater than 100 Dollars then**

**Discount the sales cost by 10%**

The following screenshot shows the same rule, expressed in the Guvnor's guided editor:



The screenshot above shows a business rule that applies a discount to the sales price.

We're not going to get very far without understanding variables (such as **mySales** and **salesValue**). You may be familiar with Variables from other programming languages.

Variables are placeholders for things that we want to refer later. They are a bit like cells in Excel, but instead of names such as 'a1' or 'b15', we give them easy-to-remember names. Here, any value put in **mySales** or **salesValue** will be saved until we need it again.

Taking into consideration the knowledge that we have about variables, this rule is more subtle than our plain-English version. What it actually says is:

* When you find a line of sales (in Java or on our spreadsheet) greater than 100, make a note of that line (and store it in a box called **mySales**).

Make a note of the actual sales figure in that line (and store it in a box called **salesValue)**.

* Then modify our line of sales (**mySales**) so that the new sales figure is now the previous figure minus 10.

### Have a go

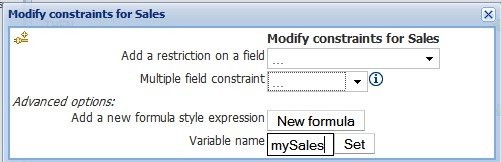
That's the easy bit. Now you're going to build it, in order to try it out! Try it yourself before skipping on to the step-by-step guide below.

### Updating the rule—step by step

Our new or updated rule uses the same Java fact model as the previous chapter, so it's going to be a lot easier to build. In fact, the step process is very similar, as it uses '+' signs to add When (conditions) and Then (consequences). We'll concentrate on the changes.

#### The When part

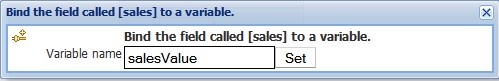
1. Create a new rule and add a condition using a sales fact, exactly as before.
2. When adding the first constraint, enter a variable name (**mySales**), as shown in the following dialog box.



1. This brings you back to the main guided editor, which will now show a rule similar to the following:

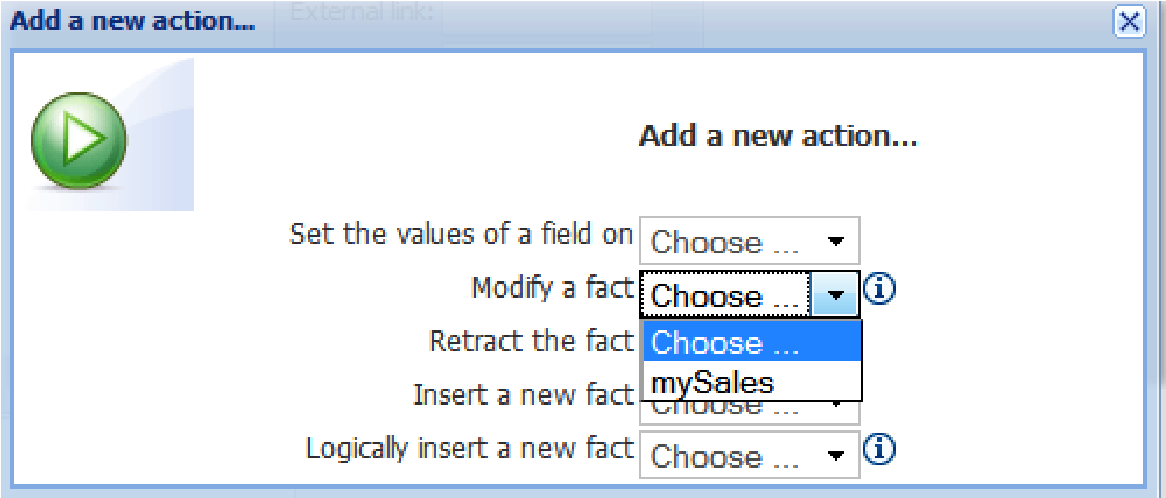


1. As before, select **equal or greater** in the drop-down box (**please choose,** in the previous diagram), and use the 'pencil' icon to enter the value (**100**) that we are filtering on.
2. On the second line of the newly updated rule, click on the green arrow above **sales**. An option to bind the actual sales total to a variable is shown. In this dialog box, assign it to **salesValue**.



Remember that our **mySales** variable is akin to one line in a spreadsheet. The **salesValue** is the sales column from within that line.

1. In the guided editor, the **Add a new action** dialog box now has additional options. Click on the green '**+'** sign, which is next to **Then**, to make these options appear.



1. In this dialog box we choose to modify the **mySales** fact.

The other options on this screen include retract (deleting the fact from memory) and inserting, or logically inserting a new fact, which work in the same way as before.

Back in the guided editor, we can click on the 'pencil' icon and enter the formula (**=salesValues-10)** to make our rule look like the one in the screenshot back at the start of this chapter.

#### Looking behind the curtain—a text-based rule

Before we jump in and run our new rule, let's catch our breath and take a look at the source. Remember that when we were working through and building the above rule, we were actually building up a text-based rule behind the scenes. Clicking on the 'view source' button might make our rule a little bit clearer.

**rule "SalesDiscount"**

**dialect "mvel" when**

**mySales : Sales( salesValue : sales >= "100" )**

**then**

**mySales.setSales( salesValue-10 ); update( mySales );**

**end**

We can guess most of what this means, but running through it allows us to see what is going on.

* Name the rule (**SalesDiscount)**
* Use the **Mvel** style for writing our rule (slightly clearer than the Java dialect)
* When

We can find a row **Sales** of greater than **100**, keep a handle to it called **mySales**, and put the actual sales value into **salesValue**

* Then

Reduce the sales by **10**

Notify (**update**) all of the other rules that the sales has changed, so that they may fire (or not)

* End of rule

**A small problem...**

Now that we know what's going on, we go back to our test scenario that we created using Guvnor in the previous chapter—the one called **TestSales**. We would expect that when we click on the **Run Scenario** button (making no changes), we would get the same **results 100%** message. But that doesn't happen.

What happens is that the scenario fails with the message **Rule [SalesDiscount] activated 11 times. Expected 1 times.** What happened? How can our rule have fired 11 times if we only have one test scenario?

What actually happened when we ran the test scenario is this:

1. We put our sales object into the working memory (that is, passed it to JBoss Rules), with a total sales of 200.
2. Our discount sales rule kicks in.
3. The sales value is now updated by this rule to 190 dollars.
4. Drools detects that our sales object has been modified, and sees if any rule can be applied.
5. The same discount sales rule is found (as the sales value is still greater than 100).
6. The sales value is discounted by a further 10 dollars.
7. Repeat the last two steps until the sales value finally drops below 100 dollars.
8. No more rules fire and JBoss Rules exits.

So the rule is doing exactly what we asked it to do—applying a discount to orders with a sales value greater than 100 dollars. There are many instances in which we want this recursive behavior. In fact, it's one of the advantages of using a rule engine: It matches the business rule to the situation that it finds itself in, and then it fires.

What exactly is this working memory? Working memory is like a scratchpad. All of the information that we pass to Drools is stored here. When rules fire, the information in the working memory gets updated. And when Drools is finished, a copy of the working memory is passed to the rest of the program.

The important thing to remember is that Drools only knows about and fires its rules according the information that we explicitly place in the working memory.

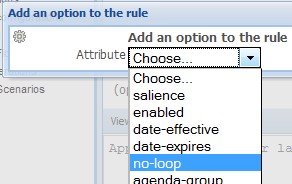
#### Rule attributes

However, what we want the rule to do is fire once and then stop. To be more precise, we do not want to have a rule that fires recursively in a tight loop. How do we express this intention in a rule?

Luckily, Drools has a feature called rule attributes. This allows us to state more information about our rule. The rule engine can use this extra information to modify its behavior.

In our example, we want to switch on the **no-loop** attribute—meaning that the rule will fire again only if some other rule has modified the working memory in the meantime. To express this in our rule using the guided editor in Guvnor, follow these steps:

1. At the bottom of our rule click on the '**+**' sign next to the options.
2. Choose **no-loop** from the drop-down menu.



1. Save and run the rule (via the test scenario). It now behaves as we expect.

The test passes it correctly saying that the rule has fired only once.

The **no-loop** attribute will stop the rule from being called; or rather it will stop a rule from changing a fact in working memory, which could otherwise cause the same rule to fire again.

What **no-loop** will not prevent is the looping of two (or more) rules; for example, Rule A causing Rule B to fire, which then causes Rule A to fire, and so on. Well-written business rules with appropriate restrictions (in the When part) will avoid this situation. For example, we can add a 'discount amount' column in our fact model. Our rule will check that this column was empty before firing and update the column when it fires. That way, we would avoid any chance of looping.

Congratulations, you've not only written a much more sensible business rule, but you’ve also touched upon your first bit of rule engine theory!

### More on the guided editor

When we were writing our rule using the guided editor, we came across a lot of options that we didn't use. Some of these are obvious, but it's worth running through what they are.

#### Possible comparisons

When we added a fact comparison, we chose the option of testing to see if our sales value was greater than 100. This comparison can also be a less than, equal to, not equal to, equal to or greater than, or equal to or less than test. For text fields (String), Guvnor will also give you the option of using **matches** and **sounds like**.

#### Condition types

When previously we added our first condition (using the '**+**' icon next to the **WHEN** statement), we chose the **Sales** fact from the **Add a condition to a rule** dialog that appeared. Not specifying a condition type means that the rule will fire every time a fact is in memory.

We have a few other options in the second drop-down box (**Condition type**) in this dialog:

* **This is no**: This option is a simple negative value. The rule will fire only if the fact does not exist at all in the working memory.
* **There exists**: The rule will fire once, no matter how many times the fact exists in working memory.
* **Any of**: The rule will fire, no matter how many times the facts exist in working memory. It differs from **There exists** because it can be applied to multiple types of facts at the same time.

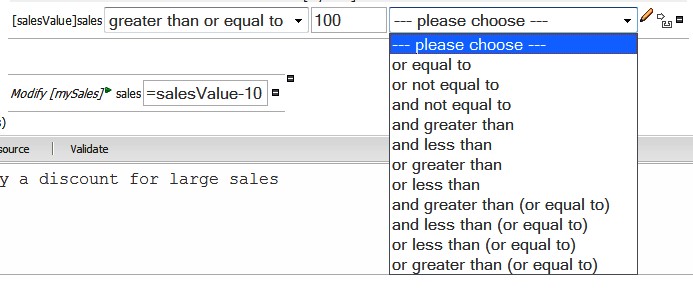
Once we add one of these condition types, we have the option in the main guided editor to specify (as before) the **Fact** (for example, **Sales**) that the condition applies to.

#### Add more options

On each part of the rule you'll see the **Add more options** icon (it looks like this).

This allows you to add additional conditions to the same line, based on the same value. You'll see the full list in the dropdown once you click on this icon, but it allows us to write more sophisticated conditions, such as:

* If **salesValue** is greater than 100 and less than 200, then give discount level 1
* If **salesValue** is less than 100 or greater than 100, then give discount level 2



Note the following differences between using the '' (add more options) icon and the large green '**+**' icon that we've been using until now:

1. The '**+**' icon only allows us to add **And** conditions (that is, all parts of the rule have be true before the rule fires). The **Add more options** allows **And** conditions as well as O**r** conditions. **Or** conditions are used when only one of the conditions needs to be true for the rule to fire.
2. The '**+**' icon allows comparisons based on multiple facts (for example, Sales and Existing customers). The **Add more options** only allows comparisons based on the same fact (for example, only Sales).

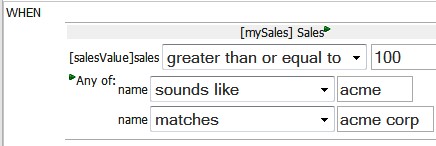
#### Multiple field constraints

When we clicked on the 'green arrow' icon next to **Sales**, we modified the constraints for the **Sales** object via the dialog box that appeared. At the time, we chose to add restrictions on a field (**salesValue**), but we have a few other options in this dialog.

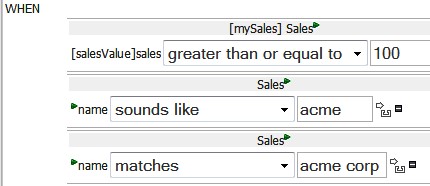
We can add a multiple field constraint, which will allow us to combine one or more conditions. This can be done as **All of** (and) or **Any of** (or). If you choose this option in the editor, you can see that these can be nested to build up sophisticated (but easy to get lost in) rules.

We also have the option to add a formula according to our condition. We won't go into the details of this here, as we will come across a similar formula in our text rules, before this chapter ends.

The following screenshot shows our rule once it has been updated with a multiple field constraint. In this case, our rule will fire only when the sales value is **greater than or equal to 100** and the name either **sounds like** **acme** or **matches** **acme corp**.



So what's the difference between multiple field constraints and just simply adding constraints using the '**+**' icon in the editor? Take a look at the same rule using the '**+**' icon to add the additional conditions in the following screenshot. When do you think this rule will fire?

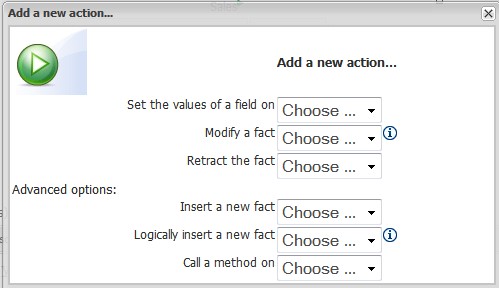


The answer is: never. What this rule is saying is that it should fire only when our **Sales** are **greater than or equal to 100** and the customer **name sounds like acme** and the customer **name** **matches acme corp**. The last two conditions are contradictory— our name must match **acme corp**, but if it does it won't sound like **acme**.

So, just by changing the icon that we clicked on, we've changed our rule from 'or' to 'and', thus giving a very different behaviour. Beware of subtleties such as these when building your rules, and test your rules using scenarios, to ensure that they behave as you expect.

#### The Then part

There are also a few options that in the Then part of the guided editor we skipped over. These are context-sensitive, so they may or may not appear, depending on what you have in the When part of the rule.



The options are:

* **Set the values of a field on**: If we set a variable in the When part, we can update that value here.
* **Modify a fact**: This is the same as setting the values, except that we notify the rule engine that the values have changed (which may cause another set of rules to fire, as in the last example).
* **Retract a fact**: Remove an item from working memory (which is akin to deleting a line from our spreadsheet). This may cause other rules to fire.
* **Insert a new fact**: This will create a new line on our spreadsheet (and may cause other rules to fire). This is the option that we used in the last chapter, in our first sales discount rule.
* **Logically insert a new fact**: This is the same as **Insert a new fact**, but the fact is removed again,as soon as this rule stops being true.
* **Call a method on**: This allows you to call a Java method (advanced). In this way, you can pretty much use the full power of Java to send messages, link to other systems, and so on, from your rules.

All of the options have dropdowns, which will provide appropriate selections from our fact model and will update the main editor, allowing you to build your rule step by step.

#### More rule options and attributes

In our last version of the sales discount rule, we added a rule option called **no-loop**, to ensure that the rule did not recursively cause itself to fire. You may have noticed that there were several other options in that dropdown, and you may be wondering what they are used for.

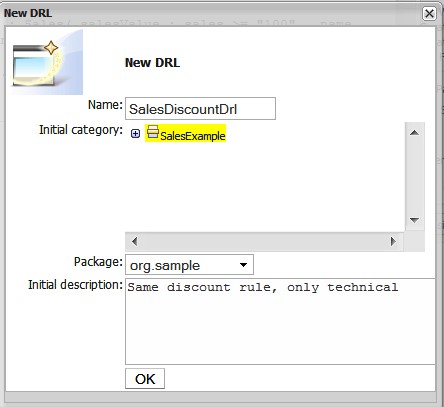
* **date effective** and **date expires** allow us to switch our rules on or off, depending on the current date.
* **dialect:** Rule can be written in different dialects—currently Java and MVEL, with more to come. Both of the currently-supported dialects follow the same 'when...then' format, although MVEL is slightly easier to read (unless, or sometimes even if, you are familiar with Java). **dialect** should not be confused with **DSL** (**Domain** **Specific** **Language**), which is a templating mechanism that allows near-English language rules. We'll look at DSL shortly.
* **duration** forces the rule to remain true for the specified duration before it fires. This can be useful for a commodity-trading application, where the price (of chocolate beans, for example) must remain high for a specified period of time before we'd fire a rule to sell it.
* **salience**: In general, you can't control the order in which multiple rules will be fired; either something is true, or it isn't. (Remember that this in one of the big differences from traditional programming languages.) However, we can give hints to the rule engine as to which rule we consider should be fired first. All other things being equal, the higher the salience for a rule, the more likely it is that this rule will be fired first.
* **agenda-group**, **auto-focus**, and **activation-group** are all used by Ruleflow, which we will cover in more detail in a later chapter. For the impatient, Ruleflow is a means by which you can group rules so that they are only available to be fired at specific stages in a business process.
* **no-loop**, as we already know, stops rule recursion.
* **lock-on-active** is a stronger version of **no-loop**, for use with Ruleflow.

### Text editing

We looked behind the scenes earlier, and viewed the text rule that Guvnor automatically builds up for us. That view was read-only, but it is possible to write text rules directly using the Guvnor editor—that's how we wrote our first 'Hello World' rule. Even better, it is possible to start writing our rule using the guided editor, and then switch to the text editor for the trickier parts. To do this, carry out the steps shown below:

1. View the source of the rule that we created in the guided editor. For this example, open the sales discount rule and then select **View Source**.
2. Copy the text of the rule from **rule** to **end**, inclusive.
3. In the side bar of Guvnor, create a new technical rule (**Rules | Create New | New DRL**). You'll be presented with a New DRL dialog box that looks familiar. Enter the values as shown in the screenshot below these steps, and then click on **OK**.
4. We will be shown the technical rule editor. This is similar to the guided rule editor except that in place of all the dropdowns, you have space for editing text.
5. Into the text editor, paste the text of the rule that you copied in step 2 earlier.

We can then save the rule and run it as before.



That's it. You've written (and run) your first text-based rule. But what is this DRL thing? A DRL file (often with the extension **.drl**) is the file where Drools stores its rules, in much the same way that a file with the extension .**doc** is a Word file, and a file with the extension **.xls** is an Excel file. Inside, it's a DRL file just a text file, which we can open in Notepad, or one of the other text editors, such as the JBoss IDE.

### Introduction to the JBoss IDE

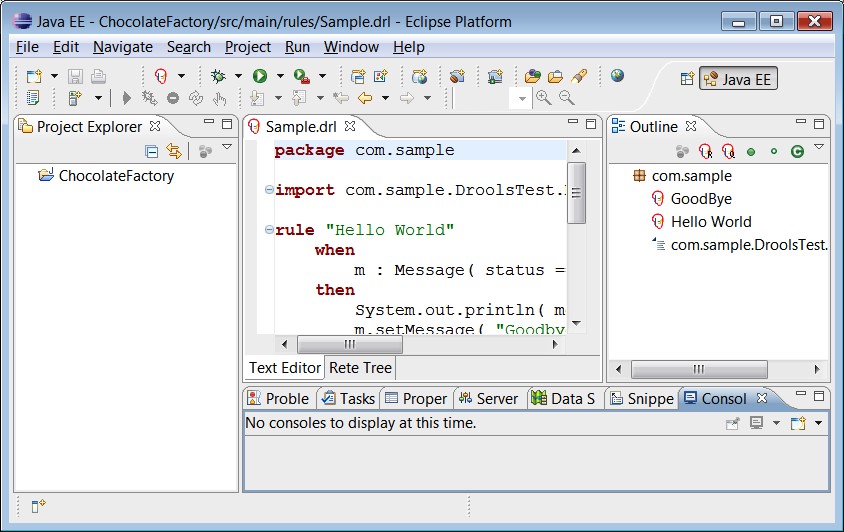
The text editor in Guvnor is useful, but for the moment, it is not as powerful as some of the other editors that are available. For example, it doesn't have syntax highlighting (coloring that makes the business rules easier to read) or the inline highlighting of errors. Compare the plain grey text in Guvnor with the following screenshot of the JBossIDE, where the rules are much easier to read.

So what is the JBoss IDE? There are a few ways to answer this question:

* An **IDE** is an **Integrated** **Development** **Environment**. This is a fancy way of saying 'you can edit all of your files in one place'.
* The JBoss IDE is based on the Eclipse open source platform. If you look at the applications on your desktop, you'll see that a lot of them have common functionality (open file, save file, and so on.). Eclipse provides this functionality not only for Java, but also for editors for other languages. There are even Eclipse-based financial applications.
* The whole aim of Eclipse is a platform for extensions, so basically the JBoss IDE = Eclipse + extensions. The extensions provided are not only for rules editing, but also for things such as workflow, storing information in databases (Hibernate), faster web development, and distributed applications (Seam).

Think of the IDE as a text editor that can edit multiple files at a time. Yes, it can do a lot more powerful things, such as allowing us to test and run our text-based rules, but we'll get to that later. Now, does a text editor seem quite as daunting?

We came across the JBoss IDE in Chapter 3, when we used it to edit our Java-based fact model. In this chapter we're going to concentrate on the rules-editing extensions.



Previously, we only looked at the IDE in passing (as we were using it to create just enough Java to get by). This time it is worth taking a more detailed look around, as we'll be using the IDE quite a bit in the next few chapters. The parts of interest in the above screenshot are:

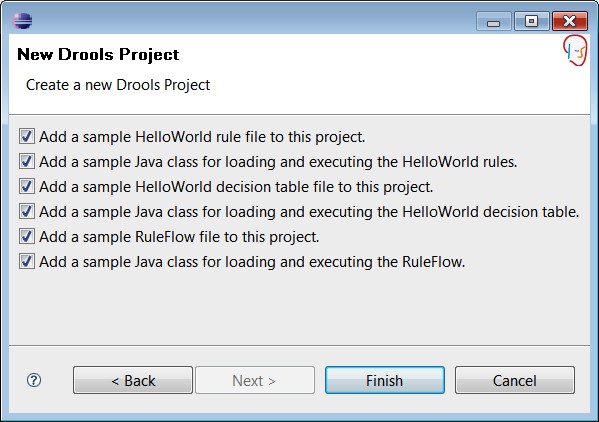
* Main menu bar: This is at the top of screen, and contains commands such as **File**, **Edit**, **Search**, **Window**, and **Help**. A lot of these commands (such as **File**, **Save**, and **Edit**) will be familiar to you.
* Menu icons: There is a set of icons just below the menu bar. Just like Word or

Excel, these are shortcuts to the functionality provided by the main menu bar.

* Main editor: In the centre of the screen is the main editing space. In the screenshot above, this is showing a rule (**Sample.drl**) being edited. If we have more than one file open at a time, multiple tabs will appear (in the same line as **Sample.drl**) which allow us to switch between the open files. There is an '**X**' on each tab, which allows you to close it. (You will be prompted if you want to keep any unsaved work.)
* The lefthand panel, by default, shows the project explorer (that is, a way to view all of the files of the project). Like most things in Eclipse, this panel is configurable. For example, selecting **Window | Show View | Other | General | Navigator** will cause a Windows Explorer type view to appear in the lefthand panel. The project explorer will remain open, and tabs are available to allow you to switch between these two views of the same project.
* The righthand panel, by default, shows an outline of the currently open file—in this case, the rule we are editing. This is also configurable. For example, dragging the **Outline** tab and dropping it onto the lefthand panel will cause it to be displayed there.
* Right-clicking will cause a context-sensitive menu to pop up, just like in other Windows applications. For example, right-clicking on the **Outline** tab and selecting **Fast view** will minimise the view to an icon in the bottom left of the screen, to save screen space. Don't worry; you can click on this icon to see its contents again (or right-click on it to restore it to the way it was previously).
* The Status Bar at the bottom of the screen is where the fast view icons migrate to. It also shows messages from Eclipse, such as if Eclipse is working on a background task (for example, validating a rule).
* The bottom set of tabs are also configurable (that is, we can drag and drop, or fast-view just like the left- and righthand panels). The two tabs shown in the screenshot that we're most interested in are **Console**, where anything we print out from our rules is displayed, and **Problems**, which shows any problem (for example, incorrect spelling) that we may have.

Now that you're interested in using the JBoss IDE to edit your rules, lets get started. This is similar to creating a new Java project (which we did in Chapter 3), that is:

1. Choose **File | New | Project** from the main toolbar, but this time choose **New Drools Project** as the next step.
2. Give the project a name (for example, **ChocolateFactory**), but click on **Next** (and not **Finish)**. The dialog that is displayed is shown in the screenshot that follows.
3. Take the existing samples. (Actually, the only samples we need right now are the first two, but selecting them now will save you having to recreate the project later.)



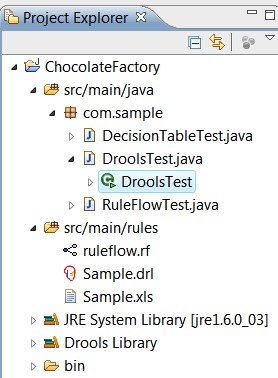
Next, click on **Finish**. Your JBoss IDE will look similar to the one we walked through.

### Hello World in the JBoss IDE editor

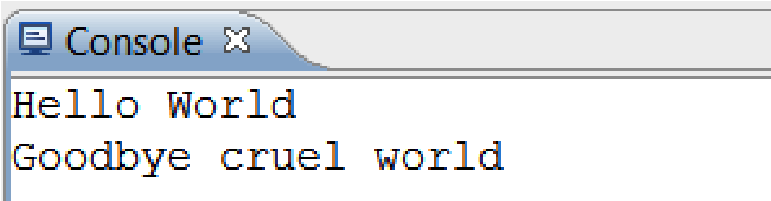
When we started with the Guvnor IDE editor, our first rule was 'Hello World'. It would be a pity to break a fine tradition, so let's write our first rule in the JBoss IDE in the same way. Even better, the steps that we just followed (to set up samples as part of a new Drools project) means that all the hard work of typing is done for us. All we need to do is poke around and understand what is going on.

Looking at the project explorer (on the lefthand side) and expanding the folders (using the small arrow icons) shows that quite a few folders and files have been added to our project. These follow a standard format, so once you understand them for this project, you'll be at home in bigger, more complex projects.

* **src/main/java**: this folder contains the text files holding the Java code (**\*.java**). We saw these in Chapter 3 when we were building our Fact Model to transport information into and out of the Rules Engine. Of course, Java is a lot more powerful than that. **DroolsTest**, a Java-based test, is highlighted in this folder. We will run it shortly.
* **src/main/rules**: This folder contains the text files containing the source for the rules.
* **JRE System Library**: This is a shortcut to the Java toolkit. Without the Java toolkit, the Java code wouldn't mean anything to the computer.
* **Drools Library**: This is a shortcut to the Drools toolkit. If the Drools toolkit is missing, the computer will not know how to convert the rules into code that it can understand.
* **bin**: Before we can run our rules, we must convert them from text that we understand to a format the computer understands. This step is called compilation, and the computer-format output is stored in the bin folder. In Guvnor this step is performed behind the scenes, but we are still notified of any errors in compilation (for example, spelling mistakes).



I promised that we were going to have a look at the HelloWorld rule. Let's run it first. Right-click on the **DroolsTest.java** file and, from the pop-up menu that appears, select **Run As | Java Application**. After a second or two, the following message will appear in the **Console**.(Remember the Console? It will be in the bottom set of tabs on your screen.)



**What just happened?**

In summary: we loaded some rules and passed them some information; the rules fired; and words were printed to the Console.

#### The business explanation

A longer explanation will gloss over what is happening in the Java file (**DroolsTest**), except to say that it loads the rule file (**Sample.Drl**), passes in a fact object (**Message**) with a status of 'Hello' and message text of 'Hello World', and then fires the rules.

So what happens when the rules fire? That's the bit we're interested in. So, open **Sample.Drl** and let's have a look. We'll see that one file contains two rules: **Hello** **World** and **GoodBye**. The first two lines of the file are a package statement (saying what folder it lives in) and an import statement.

What's this import statement? Our Java code passes to the rules the **Message** fact object, which contains important information.

Drools needs to know more about what sort of fact it is.

The import statement (standard in Java) tells Drools about the folder and file where it can find more information. Previously, Guvnor automatically generated the import statement for us.

**package com.sample**

**import com.sample.DroolsTest.Message;**

**rule "Hello World" when**

**m : Message( status == Message.HELLO, message : message ) then**

**System.out.println( message );**

**m.setMessage( "Goodbye cruel world" );**

**m.setStatus( Message.GOODBYE ); update( m ); end rule "GoodBye" no-loop true when**

**m : Message( status == Message.GOODBYE, message : message ) then**

**System.out.println( message );**

**m.setMessage( message );**

**end**

The intention of the business rules is fairly clear. Let's work through them line by line:

**rule "Hello World" when**

**We find a message fact, and its status is 'Hello' , put the fact in a box called 'm' and put our message text in a box called**

**'message' then**

**Print the contents of the 'message' box**

**Set our message text to 'Goodbye Cruel World'**

**Set our message status to 'GoodBye'**

**Notify the Rule engine that our message fact has changed end rule "GoodBye" no-loop true when**

**We find a message fact, and its status is 'Goodbye', put it in a box called 'm' and put our message text in a box called**

**'message' then**

**Print the contents of the 'message' box end**

So, this is what happens when the rules are called:

1. When Drools gets called, it starts with a message fact that has a status **Hello**. It also looks for rules that match this fact.
2. The first rule (**Hello** **World**) matches. It prints the message, updates its status to **GoodBye**, changes the message, and notifies Drools of the change in the fact status.
3. On hearing that facts have changed, Drools searches for rules that match this newly-updated fact and finds the **Goodbye** rule that applies to the new situation.
4. The contents of the message (**GoodBye** **Cruel** **World**) are printed.
5. There are no more matching rules to fire, so Drools finishes and returns the updated fact objects to the program that called it.

Easy when you know how, eh?

#### The technical explanation

Normally, as a business rule writer, you don't care about the system that calls your rules. You let somebody else worry about all that stuff. But if you're the 'somebody else' (for example, if you're a Java programmer wanting to know how the magic is done and how to deploy rules as part of a web application), read on. If you're not a 'techie', there's nothing going on 'behind the scenes' that Guvnor doesn't do for us automatically; so it may still be worth a peek.

Not surprisingly, the bit that you're interested in as a (Java) programmer is in the (Java) file **DroolsTest**. Open it in Eclipse. (Pressing *Ctrl+Shift+R*, and then typing **DroolsTest** is a quick way of opening it.) The key lines to watch out for are as follows:

* The **public** **static** **final** **void** **main(String[]** **args)** method is a Java convention. When we run **DroolsTest**, the starting point is in this method.
* The first line, **try**, and the **catch** line towards the end mean 'stop if you encounter a problem and do something'. In this case, the 'something' is to print out the details of the problem.
* The **readRule()** method contains the lines of code that we use to find (and load) our **Sample.drl** file, and convert it into a RuleBase (that is, a set of rules that are ready to fire). It may look complicated, but the steps that we take here are pretty much the same as if we had loaded the rules from a file or directly from Guvnor. We'll go into these steps in more detail later, when we cover other rule deployment options.
* The following two lines call the **readRule()** method and use it to get a working memory—a stateless or goldfish-type memory that remembers the results of our rules firing, and then forgets it as soon as we want it to forget:

**RuleBase ruleBase = *readRule*();**

##### WorkingMemory workingMemory = ruleBase.newStatefulSession();

* At the end of our file is the fact object **message** (**public** **static** **class** **Message**). It's the same idea as our **sales** object from the previous chapter (a placeholder for carrying information into and out of the rules), except that this time it shares the file with the rest of the code, instead of having a **.java** file all to itself.
* Using this message template, the following lines create an actual object (a bit like making a cookie using a cookie-cutter template), and then add the information that we want to pass:

**Message message = new Message(); message.setMessage( "Hello World" ); message.setStatus( Message.HELLO );**

* Finally, the following lines pass our message to the rules and tell Drools to start firing the rules as is appropriate (that is, the bit that we discussed in the previous section):

**workingMemory.insert( message ); workingMemory.fireAllRules();**

### Try it yourself

Although the business and technical explanations of our rules may appear daunting, the fact is that the same lines appear again and again—familiarity may not breed contempt, but hopefully, confidence! In fact, to prove the point, you're going to use the IDE to change the above rules. Change the rules so that instead of printing **Goodbye** **Cruel** **World**, it prints **Goodbye** **(whatever** **your** **name** **is)**. For the really advanced readers (those who have a taste for playing around with the **.java** files), do the same for the **Hello** message: change it to **Hello** **(insert** **name** **here)**.

Why should we use the JBoss IDE instead of Guvnor? It's your choice, but many people find that the text editor is more powerful. And if you're interested in the way we call rules (that is, you going to be the poor soul writing the Enterprise Java code as well), then this chapter has shown you that we need to load and call our rules; and you're probably already familiar with the Eclipse environment anyway.

So the IDE isn't that big and scary; it's just another way of wiring the same rules. Go ahead and play with it. If the worst happens, you can delete and start again.

## Summary

This chapter pushed the boundaries of what we can do with the Guvnor rules editor, and then brought in the JBoss IDE as an even more powerful tool for writing rules.

We started by using variables in our existing sales discount rules, to get around the problem of having negative sales. Then we came across rule attributes (for example, salience) to stop our rules from making changes that cause them to fire again and again. After testing this successfully, we looked at text-based rules—both in Guvnor and the JBoss IDE—that run the Hello World example in the new environment.

Now that we've written more complicated rules, in the next chapter we'll look at ways to make sure that our rules don't break—both now, and in the future.