# Data and Rules in Excel

At the start of the book we said 'If you can understand Microsoft Excel, then you should be OK'. To prove this, we're going to devote this entire chapter to Excel.

Almost everybody understands Excel spreadsheets, or their equivalent in OpenOffice and Google Docs. All of these are simple, grid-based editors that allow us to store, edit, and share information. Spreadsheets may not be perfect, but they're popular and well-understood. First, we're going to use them to hold the data that we supply to the business rules. Then we're going to use spreadsheets to hold the actual rules in a decision table format.

## Reading data from Excel

One of the basic problems that we face is getting information into and out of our business rules. Until now we've been using JavaBeans. These are great, especially if you're familiar with Java already. JavaBeans are also good if your information is already available from a technical source of data, such as a database, that Java can easily read. But JavaBeans can be time-consuming to create—more so if your information is already contained in an Excel spreadsheet.

Instead of having to create our Java Beans manually, wouldn't it be great if we could hand this spreadsheet to the Drools rule engine and say 'here you go—fire the rules against this information'. It would be even better if our rules could say something like 'If the value at cell A1 is greater than 100, then update the spreadsheet to say: value too high'. As a result of our rules firing, the spreadsheet could be updated with that message.

Fortunately, there is already an open source project called Apache POI that allows Java-based programs (such as the Drools rule engine) to read and write data from Excel spreadsheets. We'll use a simplified version, based on POI (from the Red-Piranha project), in our chapter so that we can read and write from Excel. That covers the 'store the data in Excel' requirement.

What's the difference between updating values using normal Excel formulae and updating the values using Drools, as we explain in this chapter?

For most spreadsheets, the power of Excel (and it is very powerful) is enough. But, pretty quickly, those formulae become very complex and difficult to understand, test, and maintain—the same problem of 'traditional coding' that we talked about in the previous chapters. Rules solve this problem by expressing your business rules in a clear and clean way.

If you're interested in loading high volumes of data from different formats (including Excel), fast-forward to the Smooks section in Chapter 12. The advantage of the Red-Piranha approach is that it allows us to change the formatting or colours in the spreadsheet. Smooks is better for large volumes of data, but doesn't give you the pretty colors!

### Business rules for this sample

The business rules that we'll walk through in this chapter are based on the following:

* The chocolate factory needs to buy cocoa beans to make candy bars
* Because the price of cocoa beans rises and falls almost daily (an entire building in Chicago is dedicated to this market), the chocolate factory decides to make a bit of side money by trading in chocolate
* We trust certain traders more than others, so we will buy from them at a higher price—that is, we have specific prices for each broker
* We also have different selling prices for different traders, as either they have a bulk-buying agreement, or we factor in the cost of delivery

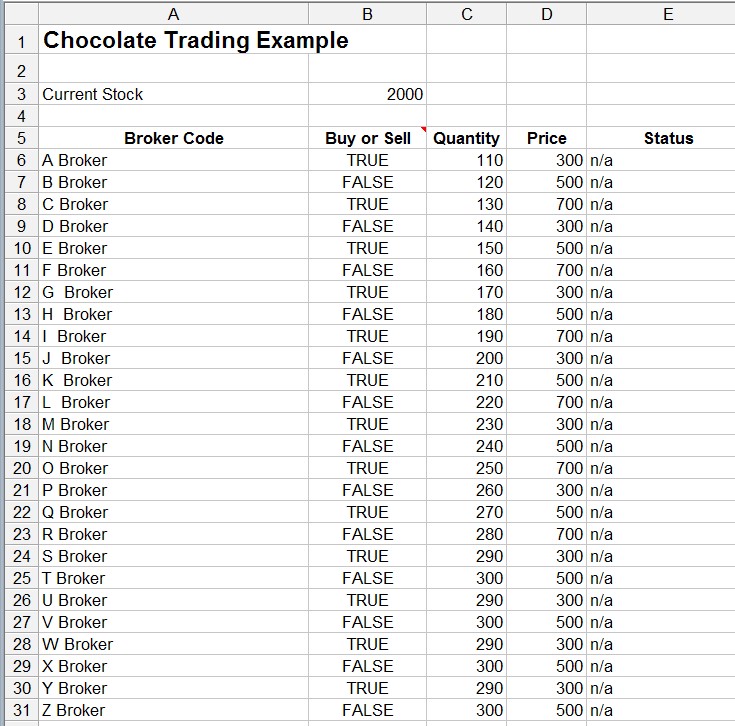
Of course, this is very simple compared to what really goes on in the market, but it's enough to show how to store both data and rules in Excel.

### Getting and running the sample

Our sample that shows how to read and write Excel data (as our fact model) can be downloaded (like the last example) from Google Code at **http://code.google. com/p/red-piranha**. The file you need is **droolsbook-chap8-sample\_01.zip**. . Like the previous example, unzip it to a location of your choice. The downloaded file should contain the Eclipse project, but (as before) you may need to execute the Maven command **mvn clean package** to download the required libraries to your computer.

#### Input

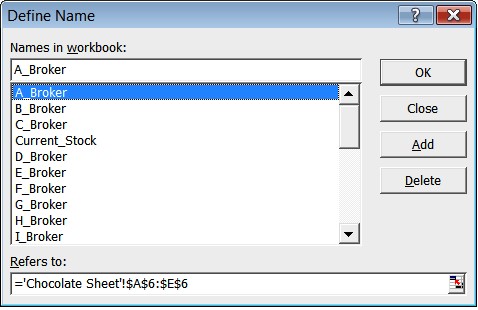
Open this project and look for the input spreadsheet. It is called **chocolate-data. xls**, and is found in the **src/main/resources** folder. When you open it, it looks something like the following screenshot—a fairly ordinary spreadsheet with one value at the top (**Current Stock**), and below that many rows, one per broker (**A Broker**, **B Broker**, and so on) with five columns. These lines are the various buy and sell offers that chocolate brokers will make to us. The five columns are contain **Broker Code**, **Buy or Sell**, **Quantity**, **Price**, and **Status** of the trade.



The screenshot below demonstrates named ranges. Referring to cells such as 'A1' or 'B12' is great, until you have to modify the spreadsheet when you need to update all your rules.

Rather than referring to cells 'A12-E12', we can use a name such as 'G Broker Values' to refer to a range (group) of cells. If we add or remove lines from the spreadsheet, the named range will continue to point to the same set of information.

To see the named ranges in Excel, select **Insert | Name | Define** from the toolbar. You should see the **Define Name** dialog box listing all of the ranges in the current spreadsheet. In effect, there is one named range for each broker (that is, one for each line in the Excel spreadsheet), and a named range for the current stock value.



#### Rules

Our rules for the first sample are pretty simple and are still in the standard format that we're used to. It just prints out the values and marks cells as modified. Remember, this first sample is all about reading data from Excel.

**package net.firstpartners.chap8; import net.firstpartners.drools.log.ILogger import net.firstpartners.exceldata.Cell; import net.firstpartners.exceldata.Range; global ILogger log; rule "log then modify cell values" when $cell : Cell(modified==false) then**

**$cell.setModified(true);**

**//Logging message**

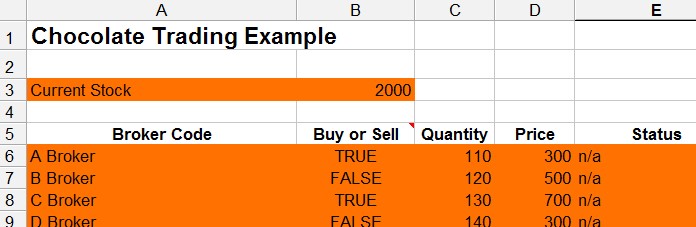
**log.info("initial cell value:"+$cell);**

**end**

#### Running the sample

Our main class is **ExcelDataRulesExample.java** (which is in the directory **src/ main/java/net/firstpartners/chap8**). Select this Java file in Eclipse and run it in the usual way (right-click on the file, and then choose **Run as Java application**). You'll see some output in the console, but the real change is in the Excel spreadsheet, which has been copied and then modified.

This output spreadsheet is called **chocolate-output.xls**, and can be found in the main project folder. When you open it you'll see that according to our simple rule, all cells having a range name have been marked as modified. In this case, the cells are highlighted to make it clear what is going on.



Copying the Excel spreadsheet and then modifying it means that even

if a rule deleted all the values by accident, you still have the original

spreadsheet to go back to and try again!

**What's going on?**

If you're not interested in the technical nuts and bolts, then skip ahead to the next section. Just remember that we can read and write information in Excel, and pass this information to our rules. Still with us? Then you're in for a treat, because we're going to explain what's just happened in the previous example. Along the way, you'll pick up information that will help you write rules that use Excel as a fact model to hold information for our business rules.

#### Under the covers

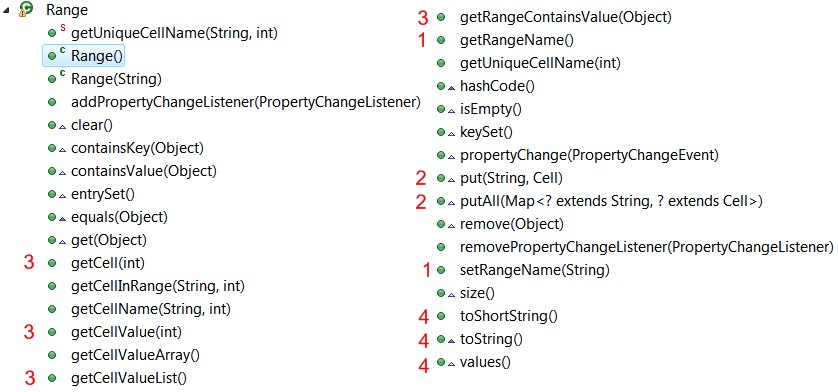
Our sample starts in the **ExcelDataExample.java** file. As before, the flow starts in the main method. It's fairly clear in this file where our input (**chocolate-data. xls**), output (**chocolate-output.xls**), and rules (**log-then-modify-rules.drl**) are. What is different from the previous sample is that we have a piece of code (**Rangeconvertor**) that loads the Excel spreadsheet and transforms it into a set of simple JavaBeans (Cells and Ranges). These JavaBeans are the fact model (containing the data) that we pass to our rules as normal.

Why not use the Cells and Ranges supplied by the Apache POI framework? The answer is that you can use them, but the Cells and Ranges based on Red-Piranha that we will use here are simpler. They also allow us to add methods (such as **getRangeContainsValue**) that make it easier to write rules.

We've already seen that the simple rule for this example matches against all cells, logs their contents, and marks them as modified. Our **RangeConvertor** also translates our (updated) Cells and Ranges back into a proper Excel spreadsheet. For clarity, any cell that we modify is highlighted—hence all our ranges are highlighted.

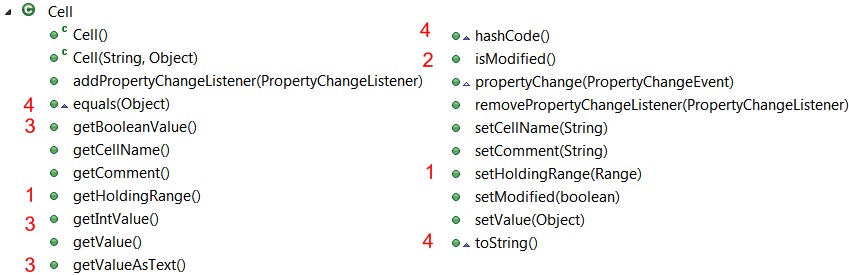
#### More on Cells and Ranges

When we write more sophisticated rules, we'll be matching against Cells and Ranges. These JavaBeans hold the information that we passed in via the Excel spreadsheet. Because our rules will be matching against these Cells and Ranges, it's worth looking at them in more detail.



Our **Range** JavaBean, like its equivalent in Excel, is used to hold a group of cells. The items marked on the above screenshot are the ones that we will discuss in more detail.

1. Like Excel, our **Range** has a name that we can access with the **getRangeName** and **setRangeName** methods.
2. A range is based on the Map interface. That is, our cells are stored within the range as a cell name, and then the actual cell itself. Our cell names follow the range name (for example, the first cell in the **some\_name** range would be **some\_name\_0**), but we have convenience methods that handle this for us.
3. Our own methods for working with cells are: **getCell** (the actual cell itself), **getCellValue** (a shortcut to the value held within the list), **getCellValueList** (a list of all the values within the range), and **getRangeContainsValue** (which searches the range and returns true if the value is found).
4. We have other methods to help us print (for example, **toString** and **toShortString**), methods to help us compare ranges (**hashcode** and **equals**), and other Map methods (**size**,**remove**, **isEmpty**, and so on).



One **Range** can contain one or more cells, depending on the Excel spreadsheet that is passed in.

1. A cell is aware of the Range that holds it (**getHoldingRange**, **setHoldingRange**).
2. When a cell is updated by any **setMethod**, the modified flag is tripped. We can read this with the **isModified** method. This is what is used later on to mark the modified cells (in Excel) are highlighted.
3. We can read or set a value with the **getValue** or **setValue** methods. We also have convenience methods such as **getBooleanValue**, **getIntValue**, and **getValueAsText** to get the values converted to yes/no, numbers, and text respectively.
4. Like the range, we implement **toString**, **equals**, and **hashcode** to make Cells easier to print (to the console) and easier to handle from our rules.

The actual code that converts between the Excel code and our JavaBean code (and back again) is contained in the **RangeConvertor.java** and **CellConvetor.java** files. We don't need to go into the details. But we should know that we loop through all of the named ranges in the Excel sheet, and copy the values to and from our JavaBeans as required. There are a couple of quirks in this conversion process, such as:

1. If we leave the **chocolate-output.xls** open in Excel, and run our sample again, we'll get the following error (to resolve it, simply close the file in Excel):

**Exception in thread "main" java.io.FileNotFoundException:**

**chocolate-output.xls (The process cannot access the file because it is being used by another process)**

1. Because of the way that Eclipse builds projects, if you change the input spreadsheet (**chocolate-data.xls**), you must save the Excel file (as normal) and then clean out any old versions by selecting **Project | Clean** from the Eclipse toolbar. If you don't do this before running the sample, any changes that you make will appear to be ignored.
2. If a cell is blank in Excel, then it doesn't exist. (It's a way to make the Excel file size smaller.) If a cell doesn't exist, then we can't update it with a value, even if the Rules have changed it—that's why have **n/a** in some cells rather than keeping them empty. This is a quirk in the Excel conversion code that a future version of Red-Piranha should correct.

Red Piranha is a knowledge management tool available as an open source project on Google Code, at **http://code.google.com/p/ red-piranha**.

For the purposes of this chapter, Red-Piranha makes it easier to manipulate data in Excel using Drools. Drools hides the complexity of Apache POI (the code that does the actual Excel manipulation), which means that your rules could update other table-based sources of information such as Google Docs, or web pages containing HTML tables.

You may have noticed that when we invoked the rule engine (via the **DtRuleRunner** file), we passed in both Ranges and Cells (that is, we called the **ranges.getAllRangesAndCells** method). Surely, this is duplication. Because Ranges contain Cells, why not pass only the Ranges so that the rule engine can automatically read all the Cells that are contained within them? The answer is: We *can* access a cell within a range using the notation **SomeRange.getCell(1).getValue()**. But as the rule engine cannot detect changes in these second-level JavaBeans, updates to Cells made in this way will not cause other rules to fire when they should.

This restriction applies not only to Ranges and Cells, but also to any 'nested' JavaBeans that you may write on your own. Later, we'll see ways of notifying Drools about the changes to the cells. But for the moment, it's best to pass both Ranges and Cells to the rule engine. Java is smart enough to realise that they are duplicates, so we only end up with a single copy of the cell being stored in memory.

Note that both Cells and Ranges implement property change listeners to help the rule engine detect changes in values. However, because your JavaBeans may not have these, we also notify the rule engine explicitly of any changes (via update and modify calls).

In this example, contents of Cells may change, but Ranges do not. That is, the value of a cell may be updated, but the shape or the name of a range is unlikely to change.

As an aside, you may have noticed that even though we have both Range and Cell values available (in working memory) to match against, (to keep the sample simple), the current rules only match against values contained in Cells.

### Sophisticated, but repetitive rules

Reading and updating an Excel file is a good trick, but our simple rule doesn't do much yet. How can we add more sophisticated rules? More importantly, if you've extended the rules from previous chapters, how do we write repetitive rules without a lot of copying and pasting? And how do we write rules in an Excel format, and not just the data, which is what we've done so far in this chapter?

Let's remind ourselves of the chocolate trading rules and write them in a more 'business rule' type format:

* When you get a **BUY** offer for **A\_Broker**, compare the price of the offer against the price that you are willing to pay. If the price is reasonable, make a note to execute a buy order.
* Put in place similar buy rules for each and every broker.
* When you have a note saying **BUY**, execute the order.
* When you get a **SELL** offer for **A\_Broker**, compare the price of the offer against the price that you are willing to accept. If the price is reasonable, make a note to execute a sell order.
* Put in place similar buy rules for each and every broker.
* When you have a note saying **SELL**, execute the order.

Although, there are only four types of rules (evaluate buy, execute buy, evaluate sell, and execute sell) there are potentially hundreds of duplicate rules. These rules may be simple compared to those in previous chapters, but they have a higher risk of error due to their large number. It would be better if we could set them out in a simple table format like the rules table shown in the following screenshot:

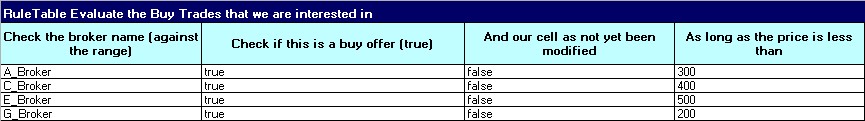
The two images in the screenshot below are actually the same table, except

split to fit into the page. Both are screenshots from the

**TradingRules.**

**xls**

file that you downloaded as a part of the Chapter 8 sample.



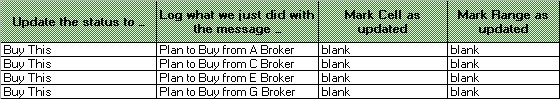
The first part of the table shows the usual 'if' conditions. In the column which is highlighted, we check the broker name, ensure that this is a buy offer, make sure that no other rule has processed this offer, and then ensure that the price is greater than what we're prepared to pay for.

Underneath, we have the actual lines of data we use as part of these 'if' conditions. We only show four, but there are 13 as a part of the sample table, and there could be many more. These lines combine with our rules, so we can read the first line as:

Check that our broker name is **A\_Broker**. Check that it is a buy offer, and that nobody else has taken up (modified) this offer. And if the buy offer price is less than 300 then …

If we didn't have the table format, we'd have had to write out in full each of the rules from 4 to 13, or however many we have. So the decision table format is ideal for the rules that are repetitive. The structure stays the same, but the values that we are checking against change from rule to rule.

That's the 'when' part of the table. What about the other part (the 'then' part)? In real life, the screenshot below (showing the 'then' part) follows to the right of the previous table.



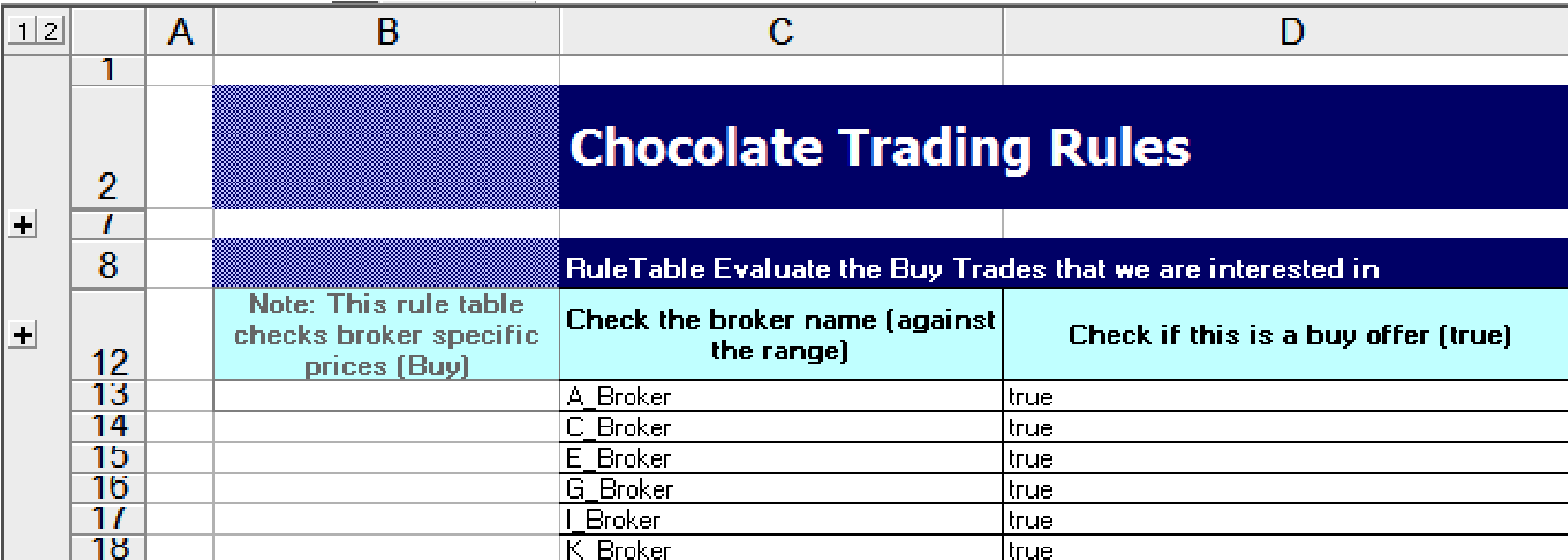
Our 'then' part follows a similar format, breaking the rules into the structure (the highlighted part) and the associated values that change for each rule. The 'then' part matching against our 'when' part (**A\_Broker** (the first line)) is:

* **Update the status to**\_**Buy This**
* Log a message about what we plan to do
* Notify the rule engine that both Cells and Ranges have been updated (to see if any other rules should be fired)

### Some Excel magic

The decision tables that you saw earlier are taken directly from our sample **TradingRules.xls**, but it looks like there is something missing. How does our technical rule engine understand the near-English language we have in the excel decision tables?

The answer is that we've used a bit of Excel magic to hide this technical complexity from the business users. In Excel 2003 and Open Office, the feature we use is called **Group and Outline** and is found under the **Data** item on the toolbar. Office 2007 has this feature under the **Group** item on the ribbon.

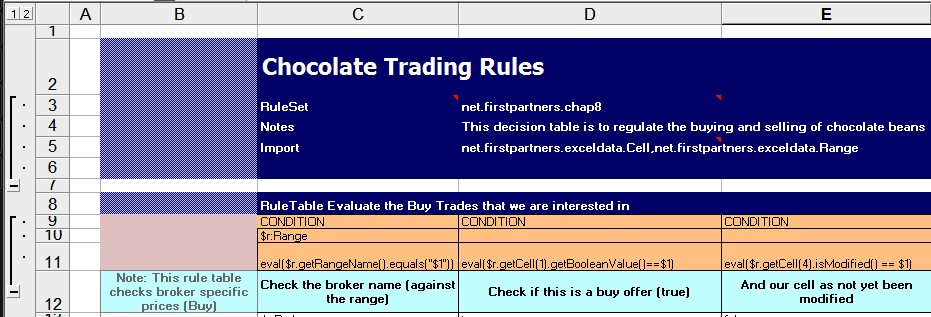


Clicking the '**+'** sign at the far left of the screen shows the hidden technical mapping.

With the lines unhidden, we can begin to see what is really going on.

Grouping and hiding columns is not the only visual trick that we can use to make data entry easier. We can use formatting and colors in cells, borders, and comments—all of which are used on this sample. We can also use more advanced features such as merged cells and dropdowns for selecting values from a predefined list.

In general, Drools just ignores these and reads the basic Excel table in black and white.



### Decision tables behind the scenes

A lot of the lines that we've just displayed contain the rule syntax that we've used in the previous chapter. We'll take a run through the information at the top of our decision table file and the first two decision tables. Remember that we're writing these rules to match against the **chocolate-data.xls** fact model (containing the data), which we loaded at the start of this example.

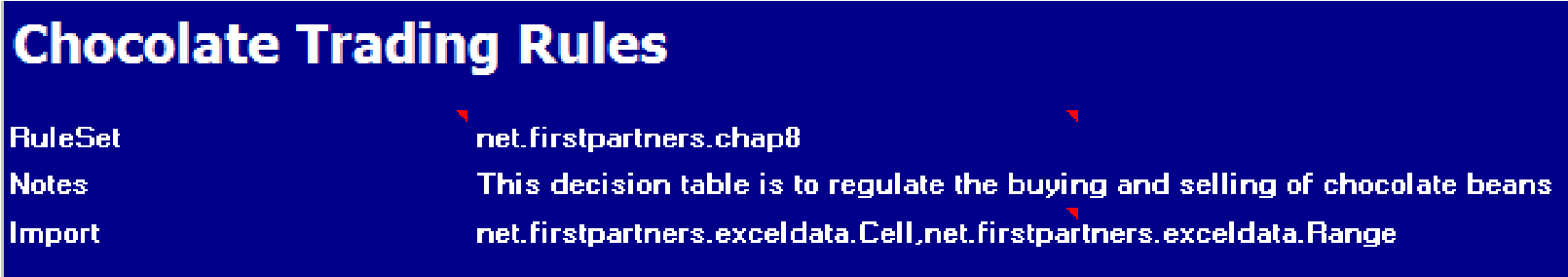
Keep in mind that decision tables are just a mechanism to help generate our rules. Behind the scenes, the rules themselves get translated into the rules syntax that we're familiar with.

#### Header information

In Chapters 5 and 6, we saw a list of items that go at the top of our rules files. These items can go into the header table, which always starts with a **RuleSet** declaration, as we can see in the screenshot below.

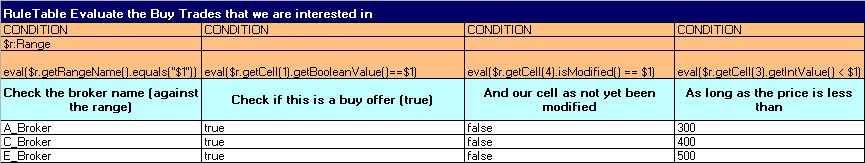
**RuleSet** lets Drools know where the header table begins, and everything else is ignored (for example the **Chocolate Trading Rules** title is not read). **RuleSet** also lets

Drools know which package these rules live in, although we do import additional JavaBeans for this example (the ones that represent our Ranges and Cells). Our **Notes** line is ignored as it means nothing to Drools. We could also have used items such as 'global' and 'function' in this part.



#### RuleTable—Evaluate the buy trades

Drools ignores everything else in the Excel file, until it comes to a section marked as **RuleTable**, where it expects to find rules laid out in the specific decision table format. The text following **RuleTable** is used to autogenerate the rule names, so be careful with the name that you use.



The first part of the decision table is the **CONDITION** cells, which makes up the 'when' part of the rule. In this case, we have one variable definition and four conditions to be matched before the rule fires. Note the use of parameter variables in the unhidden part of the rules. **$param1** or **$1** means take the first value from the cells below and use it as part of the comparison in the rule. The following table explains the different conditions shown in the above screenshot:

* **CONDITION $r:Range**: Only match against Ranges in working memory.
* **CONDITION eval($r.getRangeName().equals("$1"))**: Matches the name of this range against the parameter from the cell below (for example **A\_Broker**, **B\_Broker**, and so on).
* **CONDITION eval($r.getCell(1).getBooleanValue()==$1)**: Matches the

first cell of this range against the parameter (this value is **true** for all lines). Remember that the first cell of data is the **Buy** / **Sell** flag.

* **CONDITION eval($r.getCell(4).isModified() == $1)**: Checks that the fourth

(that is, last) cell in our Range—'modified' flag—is equal to the parameter (that is, this parameter is false for all lines). Remember that the last cell of **data** that we pass in (from the data spreadsheet, which was the first spreadsheet mentioned in this chapter) is **Status**, which starts out as **N/A**.

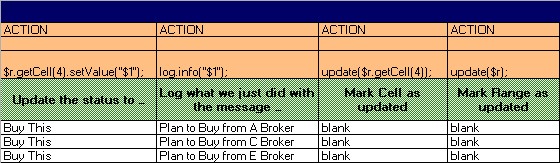
* **CONDITION eval($r.getCell(3).getIntValue() < $1)**: Checks that the third cell in our data range (the price that we offered for the chocolate beans) is less than our parameter (which varies from broker to broker).

Decision tables can use more than one parameter. For example, $2 means take the second value within the data cells in the same column of the spreadsheet. For example, if a data cell had values of 10, 20 then $1 would have a value of 10, and $2 would have a value of 20. (Note that the two values are separated by a comma.)

As a reminder, **eval()** can take any calculation, as long as it returns a true or false answer. It's great for text and calculations. The downside is that it's not as efficient, but that's OK for a small ruleset like the one in this example.

The second part of this decision table (**ACTION**) gives the 'then' part of the rules, which includes the following:

* **ACTION $r.getCell(4).setValue("$1");**: Sets the value on the fourth cell in the data range (the status) to our param, which is always **Buy This**
* **ACTION log.info("$1");**: Logs the message from the following table
* **ACTION update($r.getCell(4));**: Pings the rule engine to let it know that cell 4 in our data has been updated
* **ACTION update** **($r);**: Pings the rule engine to let it know that the entire Range has been updated



The above screenshot will generate three rules. Each rule will use the same structure, but will have the values embedded in it. To demonstrate this, the following is the rule that will be generated for the first line, for **A\_Broker**:

**rule "Identify the Buy Trades that we are interested in\_12"**

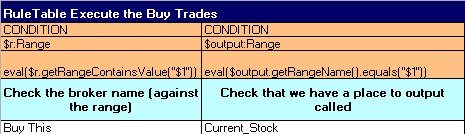
|  |  |
| --- | --- |
| **when** |  |
|  | **$r:Range(eval($r.getRangeName().equals("A\_Broker")))** |
|  | **eval($r.getCell(1).getBooleanValue()==true)** |
|  | **eval($r.getCell(4).isModified() == false)** |
| **then** | **eval($r.getCell(3).getIntValue() < 300)** |
|  | **$r.getCell(4).setValue("Buy This");** |
|  | **log.info("Plan to Buy from Y Broker");** |
|  | **update($r.getCell(4));** |
| **end** | **update($r);** |

There will be similar rules generated for each and every line in each of our decision tables. This sample generates 13 in total, one for each line of this table.

#### RuleTable—Execute the buy trades

We can have multiple RuleTables on one Excel spreadsheet. Each starts with the keyword **RuleTable**, and then follows the same decision table format. For example, take a look at the next decision table, which takes trades marked **Buy This** and actually executes the trade. (That is, the first table says we are interested in buying, but this second rule table is where money changes hands.)

The format (once we unhide or ungroup the technical parts) is the same as before, even if we've only one line of rules rather than the 13 we had previously. It also has two conditions and three actions.



The conditions are:

* **CONDITION: $r:Range , eval($r.getRangeContainsValue("$1"))**: Only matches lines of data that have been flagged with param 1 (**Buy This**).
* **CONDITION: $output:Range , eval($output.getRangeName()**

**.equals("$1"))**: Fires this rule only if we have a place to put our output (that is, a cell or range in our data sheet called **Current Stock**).



The actions are:

* **ACTION: $output.getCell(1).setValue($1);**: Increases the value of our current stock (the cell we previously identified as **$output**) by the amount of stock we have just bought. You can demonstrate the formula passed as params by using the calculation listed in the (white) param cell as **$output. getCell(1).getIntValue()+$r.getCell(2).getIntValue()**.
* **ACTION: $r.getCell(4).setValue("$1");**: Updates the status of this trade on the Excel spreadsheet to the param that we pass (that is, **Trade Executed**).
* **ACTION: log.info("$1");**: Logs what we just did to the console.

#### Other rule tables

We have two other rule decision tables in our sample **TradingRules.xls** file. These deal with identifying the sales offers that we wish to take up and the execution of those sales. The format of these two tables is very similar to the two rule tables above (**Evaluate Buy**, and **Execute Buy**). Of course, there is a minor difference—we are selling chocolate beans rather than buying.

#### Mixing rules and decision tables

While reading the previous rules Excel file, you may have wondered where the log object that we used in the previous rules actually comes from. In other chapters we had to declare a global variable (that our rules had a connection to the outside world, including the screen or console to actually print or log messages onto) and pass it in our logging object. In our **DecisionTable.xls** file we don't have this, but why?

The answer is that as we run this example, we load a second traditional rules file (**log-rules.drl**). This file is optional, as we could have used a global variable in the header section of our Excel rules file, but it's a good way of showing a mixture of rules and decision tables. After all, Drools translates them behind the scenes to be in the same format.

A good rule of thumb is to put the business rules stay in Excel and keep the more expressive, but more technical, 'plumbing' in the **.drl** (standard rule) format.

Looking at the **log-rule.drl** file, you'll see that it contains the imported file **global ILogger log** and two rules, which are:

1. A rule that matches against **unchanged** cells and logs (but does not modify) the contents.
2. A rule that matches against **changed** cells and logs (but does not modify) the contents.

Remember that both our Cell and Range JavaBeans have a modified **toString()** method to make logging easier. The logging object (**Ilogger**) also comes from the Red-Piranha project. It prints everything to the console, but also saves it (if we ask it to) as a part of our Excel output file.

### Running the Chocolate Trading example

To run the DecisionTable-based Chocolate Trading example, open

**ExcelDataRulesExample.java** in Eclipse (it's in the same directory as the sample that we ran in the previous chapter, **src/main/java/net/firstpartners/chap8**). Before we run it (by selecting menu option **Run | Run As | Java Application** from the Eclipse toolbar), we note that there are a couple of obvious similarities and differences in this example file:

* As before, our input file is called **chocolate-data.xls** and our output goes to **chocolate-output.xls**
* Unlike before, we load two rules files: the Excel decision table that we've just looked through (**TradingRules.xls**), and a standard rules file (**logrules.drl**) that shows that we can mix and match decision tables with the

'traditional' rule format that we're used to

Like most of the examples in the book, your PC will pause for a couple of seconds,

print out lots of messages to the console, then show the line, **ExcelDataRulesExample - Finished** when done.

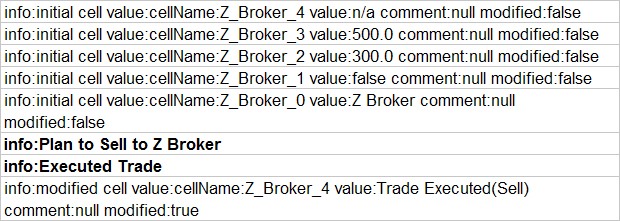
Opening the output file (**chocolate-output.xls**), we see that only some of the cells have been updated (highlighted) in line with the more sophisticated business rules in this example.



The following cells are updated:

1. The **Current Stock** is updated, as both our "execute buy" and "execute sell" rules modify the total.
2. Status fields for individual offers that we have taken up are highlighted, with a **Trade Executed (Buy/Sell)** message.
3. Status fields for individual offers that we have not taken up remain white, with the value unchanged.
4. All other cells remain white, as they have not been modified by our rules.

Logging information is printed to the Eclipse console. More usefully, there is a second tab in the output spreadsheet (**Chocolate-Output.xls**), with logging only from the rules. An extract of this is shown as follows (highlights in bold):



Before we go through what is happening when these rules fire, we'll go back to basics. We will look through the main log file (in the Eclipse console) to explain what just happened, from start to finish.

**What just happened?**

The first thing that we see in our console log when we run **ExcelDataRules.java** is that our sample finds the source Excel file (containing the buy and sell data) and converts it to Cell and Range JavaBeans (which we looked at in detail earlier)—just like the very first example in this chapter.

**ExcelDataRulesExample - found file:chocolate-data.xls**

It then logs all Cells and Ranges, and so on, for each broker and range listed in our Data Excel spreadsheet.

**============ Excel Cell Contents In ========= ExcelDataRulesExample - Range:A\_Broker**

**A\_Broker\_2: cellName:A\_Broker\_2 value:110.0 comment:null modified:**

**false A\_Broker\_3: cellName:A\_Broker\_3 value:300.0 comment:null modified: false**

**A\_Broker\_0: cellName:A\_Broker\_0 value:A Broker comment:null modified: false**

**A\_Broker\_1: cellName:A\_Broker\_1 value:true comment:null modified:false**

**A\_Broker\_4: cellName:A\_Broker\_4 value:n/a comment:null modified:false**

**…**

The sample then loads the 'traditional' style rule file, before loading the decision table.

**DtRuleRunner - Loading file: log-rules.drl**

**… found file:TradingRules.xls**

After it loads the decision table, it compiles it into the standard rule format. Our sample converts our Excel-based decision table to rules and logs the output—like the extract we see below. We saw one of the 26 evaluate buy-sell rules earlier in this chapter. This is the 'execute sell rule', once it has been translated. (This will probably be at the end of the set of the rules that is output to the log.)

**#From row number: 68 rule "Execute the Sell Trades\_68"**

**when**

**$r:Range(eval($r.getRangeContainsValue("Sell This")))**

**$output:Range(eval($output.getRangeName().equals**

**("Current\_Stock")))**

**then**

**$output.getCell(1).setValue($output.getCell(1)**

**.getIntValue()-$r.getCell(2).getIntValue());**

**$r.getCell(4).setValue("Trade Executed(Sell)");**

**log.info("Executed Trade"); end**

If we have an error (that is, because of incorrect decision table format), it normally gets flagged here. Often, it can help to look at the rule format that is printed out, to help you identify the error.

The next step in our example is to insert global variables, of which we have only one in this example—a handle to the object that we use for logging.

**Inserting handle to logger (via global)**

And then we call our rules.

**DtRuleRunner - === Calling Rule Engine ===**

At this point, in our rules, our Excel output matches the console output. As before, we have no control over the order in which our rules fire. For example, the first set of rules to fire concern the last broker in the list (**Z\_Broker**).

You may have noticed from the console output that rules are not executed in the order in which they appear on the decision table. Rather, just like the other rules that we have seen, they are executed in the order that the rule engine deems best.

Remember that your rules will fire if something is true. So if you find yourself writing rules that are sequence-dependant, think again.

The cells are updated with the help of the following steps:

1. The initial rule (from the drl file) logs the initial cell values.

**ExcelLogger - initial cell value:cellName:Z\_Broker\_4 value:n/a comment:null modified:false**

**ExcelLogger - initial cell value:cellName:Z\_Broker\_3 value:500.0 comment:null modified:false**

**ExcelLogger - initial cell value:cellName:Z\_Broker\_2 value:300.0 comment:null modified:false**

**ExcelLogger - initial cell value:cellName:Z\_Broker\_1 value:false comment:null modified:false**

**ExcelLogger - initial cell value:cellName:Z\_Broker\_0 value:Z**

**Broker comment:null modified:false**

1. The evaluate Sell rule fires (from the decision table).  **ExcelLogger - Plan to Sell to Z Broker**
2. The execute Sell rule fires (from the decision table).  **ExcelLogger - Executed Trade**
3. The modified cell logger fires (from the drl file).

**ExcelLogger - modified cell value:cellName:Z\_Broker\_4 value:Trade**

**Executed(Sell) comment:null modified:true**

1. This pattern repeats itself for each of the brokers until we have evaluated all of the buy/sell rules.
2. It is at this point (as the rules have finished firing) that Excel and the console log diverge again.

**DtRuleRunner - ==== Rules Complete ====**

1. We now log a snapshot of the cell values after all of the rules have fired.

**ExcelDataRulesExample - ==== Excel Cell Contents Out ====**

**… cell contents …**

Now, our JavaBean to Excel Data file convertor (in the **RangeConverter.java** file) outputs its messages. You'll notice that:

* Only the cells marked as modified will be updated.
* Only one update is made to the current stock (even though it has changed multiple times—once for each stock trade that we fired). This is because the Excel updater only converts the finished set of values.

**CellConvertor - UpdatingCell:B\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:Current\_Stock\_1 value:590 as Number**

**CellConvertor - UpdatingCell:F\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:H\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:J\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:K\_Broker\_4 value:Trade Executed**

**(Buy) as String**

**CellConvertor - UpdatingCell:M\_Broker\_4 value:Trade Executed**

**(Buy) as String**

**CellConvertor - UpdatingCell:R\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:S\_Broker\_4 value:Trade Executed**

**(Buy) as String**

**CellConvertor - UpdatingCell:T\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:V\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:X\_Broker\_4 value:Trade**

**Executed(Sell) as String**

**CellConvertor - UpdatingCell:Z\_Broker\_4 value:Trade**

**Executed(Sell) as String**

• Although it's not noted in the log file, behind the scenes we add the log output to our output Excel file, and save everything to disk, by starting with the line.

**// update the excel spreadsheet with our log file excelLogger.flush(wb, EXCEL\_LOG\_WORKSHEET\_NAME);**

8. Finally, we see a final message saying that everything is complete (and it is time to open the **chocolate-output.xls** file to see what the output looks like).

**ExcelDataRulesExample - Finished**

#### Have a go

The whole point of loading our rules and data from Excel is to make them easier to modify and maintain. So go on, have a go yourself! Modify the above example as follows:

* Change the sample so that it uses different quantities and prices, and see the effect.
* Add a check to our sell rules to ensure that we cannot sell more chocolate than we have in our current stock. Hint: Our **execute\_sell** rule already obtains a handle to the current stock. We can use this as part of an additional condition to the **evaluate\_sell** rule.
* Take a data spreadsheet from your business and use it instead of the data sheet (**chocolate-data.xls**) used in this sample. Hint: When writing the rules, try to make small changes to one rule at a time.

When 'having a go', remember to execute a 'clean project' command in Eclipse (if you are making modifications to Excel), and make sure that you don't keep your **chocolate-output.xls** file open (or you'll get the Java **FileNotFoundException** error that we highlighted earlier).

## Summary

This chapter, behind all of the details, was pretty simple. We used Excel spreadsheets (Cells and Ranges) as our fact model, instead of the write-your-own-JavaBean approach that we took earlier. Then we used Excel spreadsheets to hold decision tables in order to make repetitive rules easier to write.

In reality, we can use this new capability in three ways:

* Use Excel as our data model, with a standard rule (**.drl**) file
* Use Excel to hold our rules, and JavaBeans to hold our data
* Use Excel to hold both rules and data

There are several ways of handling rules and data. Use the one that makes it easy for you to handle your project. In the next chapter, we'll again try to make things simpler by using DSLs and Ruleflow.