**Lesson 1 – Stateless Sessions**

**Demo 1** - Stateless Session Introduction

Basic rules to check the validity of client insurance applications

Rules demonstrates use of logical operators and shows simple output

package org.sw.lesson01.demo1;  
  
import org.sw.facts.Application;  
  
rule "Invalidate application with Pre-existing conditions, underage, or unemployed"  
 when  
 Application( clientAge < 18 || !employed || PEC )  
 then  
 System.*out*.println("Application Invalid - Client is minor, unemployed, or has PECs");  
end  
  
rule "Valid Application"  
 when  
 Application( clientAge >= 18 && employed && !PEC )  
 then  
 System.*out*.println("Application Validated");  
end

**Demo 2 -** Splitting Rules

This file demonstrates better practice by dividing the conditions for invalidation into separate rules with single conditions

More descriptive outuput

package org.sw.lesson01.demo2;  
  
import org.sw.facts.Application;  
  
rule "Invalid application - underaged"  
 dialect "java"  
 when  
 Application( clientAge < 18 )  
 then  
 System.*out*.println("Application Invalid – client is underaged");  
end  
  
rule "Invalid application – unemployed"  
 dialect "java"  
 when  
 Application( !employed )  
 then  
 System.*out*.println("Application Invalid – client is unemployed");  
end  
  
rule "Invalid application – has pre-existing conditions"  
 dialect "java"  
 when  
 Application( PEC )  
 then  
 System.*out*.println("Application Invalid – client has PECs");  
end  
  
rule "Valid Application"  
 when  
 $application: Application( clientAge >= 18 && employed && !PEC )  
 then  
 System.*out*.println("Application Validated");  
end

**Demo 3 -** Variable binding and usage

These rules demonstrate how to bind variables and recall them for reporting purposes

Rules demonstrate use of ‘$’ before variable name for best-practice

Application ‘toString’ method returns number and name of application

Objects and object properties are bound to variables and used in RHS reporting

package org.sw.lesson01.demo3;  
  
import org.sw.facts.Application;  
  
rule "Invalid application from minor"  
 dialect "java"  
 when  
 $application: Application( $clientAge: clientAge < 18 )  
 then  
 System.*out*.println($application + ": Invalid – client age: " + $clientAge);  
end  
  
rule "Invalid application – unemployed"  
 dialect "java"  
 when  
 $application: Application( $employed: employed == false )  
 then  
 System.*out*.println($application + ": Invalid – unemployed: " + $employed);  
end  
  
rule "Valid Application"  
 dialect "java"  
 when  
 $application: Application( employed, clientAge >= 18 )  
 then  
 System.*out*.println($application + ": Validated");  
end

**Demo 4 –** Variable binding for property modification

These rules use variables to alter properties of fact objects

Enum object contains Object Statuses: APPROVED, DENIED, VALID, PENDING, and UNKNOWN to track and set status of fact objects

The importance of this step is denoting how to return results to java code rather than just printing data

package org.sw.lesson01.demo4;  
  
import org.sw.facts.Application;  
import org.sw.facts.Status;  
  
rule "Valid Application"  
 dialect "java"  
 when  
 $application: Application( !PEC , employed , clientAge >= 18 )  
 then  
 System.*out*.println($application + ": Validated");  
 $application.setStatus(Status.*APPROVED*);  
end  
  
rule "Invalid application - minor"  
 dialect "java"  
 when  
 $application: Application( clientAge < 18 )  
 then  
 System.*out*.println($application + ": Invalid – Client age: " + $application.getClientAge());  
 $application.setStatus(Status.*DENIED*);  
end  
  
rule "Invalid application - unemployed"  
 dialect "java"  
 when  
 $application: Application( !employed )  
 then  
 System.*out*.println($application + ": Invalid - Client unemployed");  
 $application.setStatus(Status.*DENIED*);  
end  
  
rule "Invalid application - Pre-existing Condition"  
 dialect "java"  
 when  
 $application: Application( PEC )  
 then  
 System.*out*.println($application + ": Invalid - Client has PEC");  
 $application.setStatus(Status.*DENIED*);  
end

**Demo 5 -** Inference and Stateless Session limitations

These rules make a slight change to the “Valid Application” rule, removing the conditions of validity and instead inferring validity if the application hasn’t been invalidated by other rules. This demonstrates the logic of inference

Running session demonstrates limitations of stateless sessions - all applications will be marked valid by the end as the session cannot be made aware of changes to the object inside session running

To use inference properly, Stateful Sessions are required

package org.sw.lesson01.demo5;  
  
import org.sw.facts.Application;  
import org.sw.facts.Status;  
  
rule "Invalid application - minor"  
 dialect "java"  
 when  
 $application: Application( clientAge < 18 )  
 then  
 System.*out*.println($application + ": Invalid – Client age: " + $application.getClientAge());  
 $application.setStatus(Status.*DENIED*);  
end  
  
rule "Invalid application - unemployed"  
 dialect "java"  
 when  
 $application: Application( !employed )  
 then  
 System.*out*.println($application + ": Invalid - Client unemployed");  
 $application.setStatus(Status.*DENIED*);  
end  
  
rule "Invalid application - Pre-existing Condition"  
 dialect "java"  
 when  
 $application: Application( PEC )  
 then  
 System.*out*.println($application + ": Invalid - Client has PEC");  
 $application.setStatus(Status.*DENIED*);  
end  
  
rule "Valid Application"  
 dialect "java"  
 when  
 $application: Application( status == Status.*PENDING* )  
 then  
 System.*out*.println($application + ": Not Invalidated - Approval Granted");  
 $application.setStatus(Status.*APPROVED*);  
end

**Lesson 2 –** **Stateful Sessions**

**Demo 1 –** the `modify` keyword

This demo fixes the error of last lesson by running the same rules within a stateful session

In contrast with Stateless Sessions, Stateful Sessions do not need to be explicitly declared when calling the Session and use a different KieSession import

Stateless sessions can simply get an array of data while stateful sessions must get the data and insert each object directly into the session after session has been launched

Stateless Session uses ‘execute’ while Stateful session uses ‘fireAllRules’

Stateless Session must be disposed of to prevent memory leaks

Rules are same as previous lesson except for using ‘modify’ keyword to make session aware of changes within session

Note: Moving last rule to start can demonstrate consistency

package org.sw.lesson02.demo1;  
  
import org.sw.facts.Application;  
import org.sw.facts.Status;  
  
rule "Invalidate application if not employed"  
 dialect "java"  
 when  
 $application: Application( !employed )  
 then  
 System.*out*.println($application + ": Invalid – Unemployed");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate application from minor"  
 when  
 $application: Application( clientAge < 18 )  
 then  
 System.*out*.println($application + ": Invalid – Minor");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalid application – has pre-existing conditions"  
 when  
 $application: Application( PEC )  
 then  
 System.*out*.println($application + ": Invalid – Client has PEC");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Valid Application"  
 when  
 $application: Application( status == Status.*PENDING* )  
 then  
 System.*out*.println($application + ": Approval Granted!");  
 modify($application) {  
 setStatus(Status.*APPROVED*)  
 }  
end

**Demo 2 –** modifying multiple properties

These rules demonstrate simply how to modify multiple object attributes by adding a string description to the ‘Reason’ attribute, using the ‘setReason’ method

Session reports reasons for denials after running

package org.sw.lesson02.demo2;  
  
import org.sw.facts.Application;  
import org.sw.facts.Status;  
  
rule "Invalidate application if not employed"  
 dialect "java"  
 when  
 $application: Application( !employed )  
 then  
 System.*out*.println($application + ": Invalid – Unemployed");  
 modify($application) {  
 setStatus(Status.*DENIED*),  
 setReason( "Client unemployed")  
 }  
end  
  
rule "Invalidate application from minor"  
 dialect "java"  
 when  
 $application: Application( clientAge < 18 )  
  
 then  
 System.*out*.println($application + ": Invalid – Minor");  
 modify($application) {  
 setStatus(Status.*DENIED*),  
 setReason( "Client underaged")  
 }  
end  
  
rule "Invalid application – has pre-existing conditions"  
 dialect "java"  
 when  
 $application: Application( PEC )  
 then  
 System.*out*.println($application + ": Invalid – Client has PEC");  
 modify($application) {  
 setStatus(Status.*DENIED*),  
 setReason( "Client has PEC")  
 }  
end  
  
rule "Valid Application"  
 dialect "java"  
 when  
 $application: Application( status == Status.*PENDING* )  
 then  
 System.*out*.println($application + ": Approval Granted!");  
 modify($application) {  
 setStatus(Status.*APPROVED*)  
 }  
end

**Lesson 03 – Cross-Products**

**Demo 1 –** Cross-Checking Claims against Policies

These rules demonstrate Cross-Product checks, a feature similar to SQL join (used for queries to multiple tables).

This demo introduces 3 new objects:

1. Claims – a claim on an existing insurance policy
2. Policies – An existing client policy with attributes such as coverage, premium, and payment status

1st 3 rules check a claim object against properties of Policy Objects to invalidate Claims.

package org.sw.lesson03.demo1;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Valid Claims"  
 dialect "java"  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end

**Demo 2 –** Advanced Cross Products

This demo introduces new ‘Client ID’ object for cross-validation with Applications

This demo splits basic Application validation and Cross-Product Application/ID validation into 2 files for organization

Cross-Product rules demonstrate additional cross-product concepts such as looking for expired IDs or ID’s with matching numbers but incorrect names

Running Session demonstrates that several clients are denied multiple times (10 applications generate @15 reports) showing that rules are firing unnecessarily

package org.sw.lesson03.demo2;  
  
import org.sw.facts.Application;  
import org.sw.facts.ClientID;  
import org.sw.facts.Status;  
import java.time.LocalDate;  
  
rule "Invalidate applications with missing ID"  
 dialect "java"  
 when  
 $application: Application( )  
 not ClientID( idNumber == $application.clientIdNumber )  
 then  
 System.*out*.println($application + " Invalid – No matching ID found");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate applications with incorrect name"  
 dialect "java"  
 when  
 $application: Application( )  
 $clientID: ClientID( idNumber == $application.clientIdNumber, name != $application.clientName )  
 then  
 System.*out*.println($application + " Invalid – Name [" + $application.getClientName() + "] not matching name on ID [" + $clientID.getName() + "]");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate applications with Expired ID"  
 dialect "java"  
 when  
 $application: Application( )  
 $clientid: ClientID( idNumber == $application.clientIdNumber, expiryDate.isBefore(LocalDate.now()) )  
 then  
 System.*out*.println($application + " Invalid – " + $clientid + " ID expired");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end

**Demo 3 –** Eliminating unnecessary rule firing

This set of rules eliminates repeat invalidation of already invalidated applications by adding a ‘status != Status.DENIED’ check to each application, meaning already invalidated applications will not be reevaluated

Launching Session shows that 10/10 applications are evaluated

package org.sw.lesson03.demo3;  
  
import org.sw.facts.Application;  
import org.sw.facts.ClientID;  
import org.sw.facts.Status;  
import java.time.LocalDate;  
  
rule "Invalidate Applications with missing ID"  
 dialect "java"  
 when  
 $application: Application( status != Status.*DENIED* )  
 not ClientID( idNumber == $application.getClientIdNumber())  
 then  
 System.*out*.println($application + " Invalid – No matching ID found");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate applications with incorrect name"  
 dialect "java"  
 when  
 $application: Application( status != Status.*DENIED* )  
 $clientID: ClientID( idNumber == $application.getClientIdNumber(), name != $application.clientName )  
 then  
 System.*out*.println($application + " Invalid – Name [" + $application.getClientName() + "] not matching name on ID [" + $clientID.getName() + "]");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate Applications with Expired ID"  
 dialect "java"  
 when  
 $application: Application( status != Status.*DENIED* )  
 $clientid: ClientID( idNumber == $application.getClientIdNumber(),  
 expiryDate.isBefore(LocalDate.now()) )  
 then  
 System.*out*.println($application + " Invalid – " + $clientid + " ID expired");  
 modify($application) {  
 setStatus(Status.*DENIED*)  
 }  
end

**Lesson 04 –** Execution Control, Fact Objects, and Truth Maintenance

**Demo 1 –** Object Insertion

This demo introduces new fact object ‘Payout’.

Event Stream Object provides reporting on when an agenda group is added or popped from stack. Since Kie/drools is open-source, source code can be opened and read.

This set of rules includes a rule to issue a payout object for approved applications

`insert` keyword is used to insert a new object into session

Because no execution control is used, running the session shows that payouts are issued for every claim, even though multiple claims are invalidated.

package org.sw.lesson04.demo1;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
  
rule "Valid Claims"  
 dialect "java"  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end  
  
rule "Issue Payout"  
 dialect "java"  
 when  
 $claim: Claim(status == Status.*APPROVED*)  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insert( payout );  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end

**Demo 2 –** Execution Control with Salience

This demo splits rules into 2 files for organization

‘Salience’ rule attribute sets precedence for execution with higher values firing first and lower values firing last.

Matching values are a roll of the dice.

Running Session demonstrates that rules now fire in proper order, eliminating payouts for invalid applications

package org.sw.lesson04.demo2;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
  
rule "Valid Claims"  
 dialect "java"  
 salience 20  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end

package org.sw.lesson04.demo2;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
  
rule "Issue Payout"  
 dialect "java"  
 salience 0  
 when  
 $claim: Claim(status == Status.*APPROVED*)  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insert( payout );  
end

**Demo 3 –** Agenda Groups

Using salience can be hard to manage for larger repositories of rules

Agenda groups streamlines execution control management by packing rules into groups that fire using a ‘stack’

In session launcher file, we see stacks defined. First, we get the session agenda, then we create the stack

Stack uses ‘first-in-last-out’ logic, first creating ‘main’ group for rules without group, then pushing the top agenda group to stack using ‘.setFocus()’ method

Then, when stack is built, top rules fire and are popped from the stack until we reach main – after main fires, stack is done and session ends

package org.sw.lesson04.demo3;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
  
rule "Valid Claims"  
 dialect "java"  
 agenda-group "valid-claim"  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 agenda-group "invalid-claim"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 agenda-group "invalid-claim"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 agenda-group "invalid-claim"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end

rule "Issue Payout"  
 dialect "java"  
 agenda-group "issue-payout"  
 when  
 $claim: Claim(status == Status.*APPROVED*)  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insert( payout );  
end

**Demo 4 –** Hybrid Controls

This demo shows how to reduce number of agenda groups by combining agenda groups with Salience

This time, all validation rules are grouped into a single agenda, with additional salience attribute – invalidation rules firing first and validation rules firing second

Finally, Payouts are issued in separate agenda group that fires last.

Note: Removing ‘issue-payout’ agenda group from rule and session results in rule firing same way as rules without group still fire in ‘main’ agenda group

package org.sw.lesson04.demo4;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
  
rule "Valid Claims"  
 dialect "java"  
 agenda-group "claim-validation"  
 salience 20  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 agenda-group "claim-validation"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 agenda-group "claim-validation"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 agenda-group "claim-validation"  
 salience 50  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end

**Demo 5 –** Fire single Rule from Activation Group

This demo introduces ‘activation-group’ attribute.

First 2 ‘ClaimValidation’ and ‘PayoutIssue’ files are same as previous session

‘Verdict’ file contains 2 rules that report whether or not policies were issued

Running Session includes prompt to set all claims to 1,000,000. Selecting ‘no’ will run session as usual, resulting in a report of issued payouts. Inputting ‘no’ will result in a report of no payouts issued

Lab Challenge – remove rule that invalidates claims with max payout exceeded. Instead, issue a new rule that issues payout equal to max coverage for claims that exceed max coverage

package org.sw.lesson04.demo5;  
  
import org.sw.facts.Payout  
import org.sw.facts.Claim  
import org.sw.facts.Status  
  
rule "Report Payouts Issued"  
 dialect "java"  
 activation-group "verdict"  
 salience 20  
 when  
 Payout()  
 then  
 System.*out*.println("Verdict: Payouts were issued this session");  
end  
  
rule "Report payouts "  
 dialect "java"  
 activation-group "verdict"  
 salience 10  
 when  
 Claim( status == Status.*DENIED* )  
 then  
 System.*out*.println("Verdict: No Payouts issued this session");  
end

**Lesson 5 – Truth Maintenance and Logical Fact Insertion**

**Demo 1 –** Logical Fact Insertion

Rules in this session much the same as lesson 4 demo 1 – no salience or agenda groups used

‘Issue Payout’ Rule, however, used `insertLogical` keyword to insert an object that is logically tied to the truth of the rule’s LHS ‘when’ statement – if this changes to false, object will be removed

Session launcher contains RuleRuntimeEventListener to report on object insertions, updates, or deletions

Running Session shows that Claims are approved and Payouts issued prematurely, but payout objects are deleted as claims are invalidated

package org.sw.lesson05.demo1;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
  
rule "Valid Claims"  
 dialect "java"  
 when  
 $claim: Claim( status == Status.*PENDING* )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 modify($claim) {  
 setStatus(Status.*APPROVED*)  
 }  
end  
  
rule "Issue Payout"  
 dialect "java"  
 when  
 $claim: Claim(status == Status.*APPROVED*)  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insertLogical( payout );  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 when  
 $claim: Claim( status != Status.*DENIED* )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 modify($claim) {  
 setStatus(Status.*DENIED*)  
 }  
end

**Demo 2 –** Fact Objects

A fact object is like a sticker or stamp that is added to a base object – it is usually a wrapper object for a base object in our java code.

As opposed to using attribute modification, using fact objects can help with State Management and maintaining separation of base objects (raw data) and fact objects (processed/contextual data).

This demo introduces new ‘fact object’ wrappers for ‘invalid’ products and ‘valid’ products.

The ‘Status’ enum attributes are replace with ‘Invalid’ and ‘Valid’ fact object wrappers

Invalid fact objects are inserted permanently while valid fact objects are inserted logically

Event Listener reports on the insertion of fact objects into the session

‘Valid Claims’ rule checks that no invalid fact objects match its base object before inserting valid fact object

‘Payout Issue’ issues payouts when ValidClaim objects are found

Firing session shows duplicate invalid claim objects

package org.sw.lesson05.demo2;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
import org.sw.facts.InvalidClaim;  
import org.sw.facts.ValidClaim;  
  
rule "Valid Claims"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not (InvalidClaim( claim == $claim) )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 insertLogical( new ValidClaim($claim) );  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim with unpaid premiums"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), paymentStatus == "Unpaid" )  
 then  
 System.*out*.println($claim + ": failed. Unpaid premiums in " + $policy);  
 insert( new InvalidClaim($claim) );  
end

rule "Issue Payout"  
 dialect "java"  
 when  
 $validClaim : ValidClaim( $claim : claim )  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insertLogical( payout );  
end

**Demo 3 –** Non-Existence Checks

This demo revisits the concept of eliminating repeat invalidations and unnecessary rule reruns by checking for the ‘non-existence’ of fact objects, using the ‘not’ keyword

Similar to the ‘Valid Claims’ rule, all rules now use a ‘not’ statement to check that no invalid claims already for a base object exist before issuing an invalid claim

Instead of using this check to issue a valid claim, it uses it to avoid repeat rule firing

Small change also made to final rule – it no longer checks for unpaid premiums in corresponding policy. We now invalidate policies separately and have a rule that checks for claims with invalid corresponding policies.

Firing Session shows that no duplicate fact object were issued this time

package org.sw.lesson05.demo3;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
import org.sw.facts.InvalidClaim;  
import org.sw.facts.ValidClaim;  
import org.sw.facts.InvalidPolicy;  
  
rule "Valid Claims"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not (InvalidClaim( claim == $claim) )  
 then  
 System.*out*.println($claim + ": not invalidated. Approval Granted");  
 insertLogical( new ValidClaim($claim) );  
end  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim with invalid policy"  
 dialect "java"  
 when  
 $invalidPolicy: InvalidPolicy( )  
 $claim: Claim( policyNumber == $invalidPolicy.policy.policyNumber )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 System.*out*.println($claim + ": failed. " + $invalidPolicy + " Invalid");  
 insert( new InvalidClaim($claim) );  
end

rule "Invalidate unpaid Policy"  
 dialect "java"  
 when  
 $policy: Policy( paymentStatus == "Unpaid" )  
 not InvalidPolicy( policy.policyNumber == $policy.policyNumber)  
 then  
 System.*out*.println("Policy [" + $policy + "] invalidated due to unpaid premiums");  
 insert( new InvalidPolicy($policy) );  
end

rule "Issue Payout"  
 dialect "java"  
 when  
 $validClaim : ValidClaim( $claim : claim )  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insertLogical( payout );  
end

**Demo 4 –** Rule reduction with Implicit Validation

Previously, we used rules to separately validate and invalidate objects by issuing ‘valid’ and ‘invalid’ fact objects.

I this demo, we streamline our rules by eliminating validation cases and ‘valid’ fact objects, using inference to logically insert Payouts for claims without matching invalid claims

This concept is similar to Blacklisting with an Implicit ‘Allow’ security policy

Running Session shows that no Valid Claim objects were inserted into session, however results remain unchanged

package org.sw.lesson05.demo4;  
  
import org.sw.facts.Claim;  
import org.sw.facts.Policy;  
import org.sw.facts.Status;  
import org.sw.facts.Payout;  
import org.sw.facts.InvalidClaim;  
import org.sw.facts.ValidClaim;  
import org.sw.facts.InvalidPolicy;  
  
rule "Invalidate claim without matching policy"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not Policy( policyNumber == $claim.getPolicyNumber() )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 System.*out*.println($claim + ": failed (no matching policy)");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim that exceeds coverage"  
 dialect "java"  
 when  
 $claim: Claim( )  
 $policy: Policy( policyNumber == $claim.getPolicyNumber(), coverage < $claim.getClaimAmount() )  
 then  
 System.*out*.println($claim + ": failed. Coverage in policy " + $policy + " insufficient.");  
 insert( new InvalidClaim($claim) );  
end  
  
rule "Invalidate claim with invalid policy"  
 dialect "java"  
 when  
 $invalidPolicy: InvalidPolicy( )  
 $claim: Claim( policyNumber == $invalidPolicy.policy.policyNumber )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 System.*out*.println($claim + ": failed. " + $invalidPolicy + " Invalid");  
 insert( new InvalidClaim($claim) );  
end

rule "Invalidate unpaid Policy"  
 dialect "java"  
 when  
 $policy: Policy( paymentStatus == "Unpaid" )  
 not InvalidPolicy( policy.policyNumber == $policy.policyNumber)  
 then  
 System.*out*.println("Policy [" + $policy + "] invalidated due to unpaid premiums");  
 insert( new InvalidPolicy($policy) );  
end

rule "Issue Payout"  
 dialect "java"  
 when  
 $claim: Claim( )  
 not InvalidClaim( claim.policyNumber == $claim.policyNumber)  
 then  
 Payout payout = new Payout( $claim.getPolicyNumber(), $claim.getClaimAmount() );  
 System.*out*.println("New Payout: " + payout + " Issued");  
 insertLogical( payout );  
end

**Lesson 6 –** **Working with Object Collections**

**Demo 1 –** Object Groups

This demo introduces new ‘FamilyApplication’ java class

FamilyApplication provides application for group of people and contains the following attributes:

* Application number (string)
* ClientIDNumbers (string list)
* ClientAges (Integer)
* startDate (LocalDate - unused)

Rule file ‘FamilyApplication’ demonstrates 2 possible rule structures:

* Primary rule demonstrates ‘memberOf’ keyword – it saves all numbers in FamilyApplication idNumbers list and looks for an expired ClientID whose number is ‘memberOf’ application’s ID list
* Commented Rule looks for an InvalidClientID fact object whose number is contained in the list of familyApplication ID numbers – this rule requires a separate, corresponding rule to invalidate expired ID’s
* LHS - Final rule finds a FamilyApplication object with no corresponding Invalid Application.
  + Saves all clientIdNumbers from familyApplication id list
  + Saves the name from corresponding ClientID.
* RHS – builds new Policy for each client ID number and name and inserts it permanently into session

/\*rule "Invalidate family applications with invalid ID"  
 dialect "java"  
 when  
 $familyApplication: FamilyApplication( )  
 InvalidClientID( $familyApplication.clientIdNumbers.contains(clientID.idNumber) )  
 then  
 System.out.println( "Set " + $familyApplication + " invalid - invalid ID found");  
 insert( new InvalidFamilyApplication($familyApplication) );  
end\*/  
  
rule "Invalidate family applications with expired ID"  
 dialect "java"  
 salience 50  
 when  
 $familyApplication: FamilyApplication( $ids: clientIdNumbers )  
 ClientID( $id: idNumber, isExpired() == true, $id memberOf $ids )  
 then  
 System.*out*.println( "Set " + $familyApplication + " invalid - expired ID found for ID: " + $id);  
 insert( new InvalidFamilyApplication($familyApplication) );  
end

/\*  
rule "Invalidate Expired ID"  
 when  
 $clientID: ClientID( isExpired())  
 then  
 System.out.println($clientID + " is expired.");  
 insert( new InvalidClientID($clientID));  
end\*/

rule "Issue New Policy to Valid Application"  
 dialect "java"  
 salience 20  
 when  
 $familyApplication: FamilyApplication()  
 not InvalidFamilyApplication(familyApplication.applicationNumber == $familyApplication.applicationNumber)  
 $clientIdNumber: String() from $familyApplication.clientIdNumbers  
 $clientID: ClientID(idNumber == $clientIdNumber, $name: name)  
 then  
 Random random = new Random();  
 int premium = 50 + random.nextInt(51); // Random number between 50 and 100  
 int coverage = premium \* 100;  
  
 Policy newPolicy = new Policy();  
 newPolicy.setPolicyNumber($familyApplication.getApplicationNumber());  
 newPolicy.setClientName($name);  
 newPolicy.setPremium(premium);  
 newPolicy.setCoverage(coverage);  
 newPolicy.setPaymentStatus("Unpaid");  
  
 insert(newPolicy);  
 System.*out*.println("New policy issued: " + newPolicy);  
end

**Demo 2 –** exists Keyword

This demo introduces only a small change to the previous rules, using the ‘exists’ keyword to avoid unnecessary reruns of FamilyApplication Invalidation rule.

‘exists’ is used to check if there exists at least one fact that meets the specified criteria.

rule "Invalidate family applications with single invalid ID"  
 dialect "java"  
 when  
 $familyApplication: FamilyApplication( )  
 exists InvalidClientID( $familyApplication.clientIdNumbers.contains(clientID.idNumber) )  
 then  
 System.*out*.println( "Set " + $familyApplication + " invalid - invalid ID found");  
 insert( new InvalidFamilyApplication($familyApplication) );  
end

**Demo 3 –** `forall` keyword

This demo illustrates how to check that a condition is true ‘for all’ objects in a collection, using `forall` keyword

Commented Rules used to illustrate rule expiry:

* ‘date-expires’ attribute sets rule expiry date
* ‘enabled false’ deactivates a rule the same as commenting

Invalidation rule now uses `forall` keyword to check that the condition ‘isExpired’ is true for all the ID’s with numbers in the FamilyApplication idNumbers list.

Policy Issue rule now uses ‘InsertLogical’ keyword to logically insert single policy to first name in application

Running rule prompts to set all ID’s as invalid:

* Inputting ‘no’ shows that both applications are now valid
* Inputting ‘yes’ invalidates all applications

Change expiry date of one ID in first application to invalidate application

import org.sw.facts.InvalidFamilyApplication  
import org.sw.facts.InvalidClientID;  
import org.sw.facts.ClientID;  
import org.sw.facts.Policy;  
import java.util.Random;  
  
/\*rule "Invalidate family applications with single Invalid ID"  
 date-expires "04-May-2010"  
 enabled false  
 dialect "java"  
 when  
 $familyApplication: FamilyApplication( )  
 exists InvalidClientID( $familyApplication.clientIdNumbers.contains(clientID.idNumber) )  
 then  
 System.out.println( "Set " + $familyApplication + " invalid - invalid ID found");  
 insert( new InvalidFamilyApplication($familyApplication) );  
end\*/  
  
rule "Invalidate family application with all IDs expired"  
 dialect "java"  
 when  
 $familyApplication: FamilyApplication( $ids : clientIdNumbers )  
 forall (  
 $clientIdNumber : String() from $ids,  
 ClientID( idNumber == $clientIdNumber, isExpired() )  
 )  
 then  
 System.*out*.println( "Set " + $familyApplication + " invalid as all client IDs are expired");  
 insert( new InvalidFamilyApplication($familyApplication) );  
end

**Demo 4 –** Additional ‘forall’ use-case

This last demo simply demonstrates an additional use-case for the `forall` keyword – issuing a report that all ID’s in session of of legal age 16 or higher

rule "Check for Child Applications"  
 dialect "java"  
 when  
 forall ( Application( clientAge >= 16 ))  
 then  
 System.*out*.println( "No applications from under 16 year olds" );  
end