

FICO® Decision Modeler

decision rules management system

3.1.5

USER GUIDE

FICO Decisions

© 2014-2020 Fair Isaac Corporation. All rights reserved. This documentation is the confidential, proprietary, and unpublished property of Fair Isaac Corporation ("FICO"). Receipt or possession of this documentation does not convey rights to disclose, reproduce, make derivative works, use, or allow others to use it except as set forth in a written software license agreement between you and FICO (or a FICO affiliate). Use of this documentation and the software described in it must conform strictly to the license agreement.

The information in this documentation is subject to change without notice. If you find any problems in this documentation, please report them to us in writing. Neither FICO nor its affiliates warrant that this documentation is error-free, nor are there any other warranties with respect to the documentation except as may be provided in the license agreement. FICO and its affiliates specifically disclaim any warranties, express or implied, including, but not limited to, non-infringement, merchantability and fitness for a particular purpose. Portions of this documentation and the software described in it may contain copyright of various authors and may be licensed under certain third-party licenses identified in the software, documentation, or both.

In no event shall FICO or its affiliates be liable to any person for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this documentation or the software described in it, even if FICO or its affiliates have been advised of the possibility of such damage. FICO and its affiliates have no obligation to provide maintenance, support, updates, enhancements, or modifications except as required to licensed users under a license agreement.

FICO is a registered trademark of Fair Isaac Corporation in the United States and may be a registered trademark of Fair Isaac Corporation in other countries. Other product and company names herein may be trademarks of their respective owners.

FICO® Decision Modeler 3.1.5 powered by Blaze Advisor

Deliverable Version: A

Last Revised: November 2020

Contents

CHAPTER 1:	
Welcome to FICO® Decision Modeler	17
What's New in FICO® Decision Modeler 3.1.5	17
Supported Web Browsers	18
CHAPTER 2:	
Introduction to Decision Modeler	19
CHAPTER 3:	
Decision Modeler Workspaces and Projects	21
Managing Workspaces and Projects	22
Creating New Workspaces	27
Viewing and Managing Workspace Settings	28
Downloading Workspaces	29
Managing Workspace Teams	29
Workspace Team Roles	30
Project Permissions	31
Capturing Data Using the Analytic Data Mart	32
Togglz Admin Console Settings	33
Limiting Data Capture in the Analytic Data Mart	34
Disabling Reporting Data in the Analytic Data Mart	35
Enabling Decision Service Rule Logging	36
Creating New Projects	36
Configuring a Project Using an XML Schema	37
Configuring a Project Using Default Settings	38
Configuring a Project With Entry Points	38
Viewing and Managing Project Settings	39
Viewing Project Links and the Client ID	39
Testing Links	40
Importing Projects	41
Importing Projects into a Workspace with Existing Projects	42
Modifying a Project Configuration	42
Managing Object Models in a Project Configuration	44

Business Object Models in the Workspace	44
Managing the XML Object Model	45
Managing the Java Object Model	45
Managing JAR Files	46
Troubleshooting JAR File Import Errors	47
Using Enumeration Display Labels in the Decision Logic	48
CHAPTER 4: Authoring Rules in a Project	51
The Project Editor	51
Routine Operations	53
Creating New Decision Entities and Folders	53
Naming Decision Entities and Folders	54
Importing Models	55
Renaming Decision Entities and Folders	55
Generating Reports	55
Compiling Decision Services	56
Quickly Finding Decision Entities	57
Quick Search Type Reference	59
Locating References to Decision Entities	60
Sharing Files Using FICO Drive	60
Adding Files to FICO Drive	61
Project Explorer Toolbar Commands	61
Keyboard Shortcuts in the Project Explorer	62
Version Management	63
Version Management Commands	63
Version History Page	64
Promoting Previous Versions	65
Releasing Locks	65
CHAPTER 5: Using Business Term Sets to Represent Classes	67
Creating Business Term Sets	68
Writing Term Calculations	68
Creating Value Lists	69
Mapping the Business Terms to Object Model Properties	69
Initializing Business Term Sets	71
CHAPTER 6: Authoring Logic in Rulesets	73
Creating Rulesets	73
Defining Rulesets	74

Using Variables in a Ruleset	74
Creating Local Variables	75
Using Patterns in a Ruleset	76
Creating Patterns	77
Unused Pattern Warnings in Rules	78
Dynamic Objects	79
Creating an Initialization Rule	80
Rule Builder Interface	81
Creating Rules	83
Copying and Pasting Expressions or Statements Within Rulesets	83
Defining Rules	84
Rule Conditions in a Ruleset	85
Creating Rule Conditions	86
Grouping Condition Expressions in the Rule Builder	87
Rule Actions in a Ruleset	87
Creating Rule Actions	88
Validation	89
Valid Values for Properties	90
CHAPTER 7: Authoring Rules in Decision Tables	93
Creating Decision Tables	94
Defining Decision Tables	94
Changing the Return Type	95
Creating Parameters	96
Editing Rules in a Decision Table	96
Inserting Condition and Action Expressions	97
Editing Decision Table Rule Names	98
Using Multiple Values in Decision Table Cells	99
Changing the Expression Format in a Decision Table Cell	100
Cutting, Copying, and Pasting Rows and Columns	100
Copying and Pasting Content in Decision Table Cells	101
Decision Table Profiling with Sample Data	101
Real-time Profiling	101
Dataset Requirements for Decision Table Profiling	102
Creating a Sample Dataset for Decision Table Profiling	103
Running Decision Table Profiling	103
Profiling Results Display Options	104
Decision Table Profiling Report	105
Generating Profiling Reports	106
Setting Profiling Options	106
Enabling and Disabling Decision Table Profiling	107

Filtering Rules in Decision Tables	107
Showing Filters	107
Clearing Filters	108
Exporting and Importing Decision Table Data	108
Permission Requirements for Exporting Decision Table Values	109
Exporting Decision Table Data to CSV Files	109
Formatting Requirements for Importing Decision Table Data	109
Importing Data Values from CSV Files into Decision Tables	110
Decision Table Keyboard Shortcuts	110
Compiled Table Optimization	112
 CHAPTER 8: Authoring Rules in Decision Trees	115
Decision Tree Interface	116
Creating Decision Trees	117
Creating Parameters	118
Using Variables in a Decision Tree	119
Creating Local Variables	120
Using Patterns in a Decision Tree	120
Creating Patterns	121
Unused Pattern Warnings for Decision Trees	122
Changing the Return Type	122
Inserting Splits	123
Inserting Levels	124
Removing Levels	124
Replacing Levels	125
Copying and Pasting Subtrees	125
Reordering Branches	125
Creating Conditions Using Or by Merging Branches	126
Using Any Other Value as a Condition Expression	127
Assigning Actions	127
Assigning an Expression	128
Changing an Action	129
Managing Decision Tree Variables	129
Repairing a Decision Tree	130
Decision Tree Profiling	131
Dataset Requirements for Decision Tree Profiling	131
Generating a Sample Dataset for Decision Tree Profiling	132
Previewing and Running Data Profiling	133
Displaying Decision Tree Profiling Results	133
Decision Tree Views	134
Node Views	135

Action Node Colors	136
Showing the Proportion of Actions in Decision Trees	136
Analyzing Decision Tree Profiling Results	137
Recounting Decision Trees	140
Exporting a Decision Tree	141
Decision Tree Limitations	142
CHAPTER 9: Authoring Rules in Scorecards	143
Reason Code Lists	144
Creating Reason Code Lists	144
Adding Reason Codes to a List	144
Creating Scorecards	145
Defining Scorecards	145
Setting the Precalculate Score Option	146
Writing Rules in Scorecards	147
Variables	147
Adding Variables	148
Deleting Variables	148
Baseline Score	148
Bins	148
Adding Bins	149
Deleting Bins	150
Tracking Unexpected Range Values	151
Validating Scorecard Values	151
Methods for Returning Reason Codes	152
CHAPTER 10: Importing PMML Models	153
Creating Scorecards from PMML Files	153
Examples Decision Services with a PMML Scorecard Model	155
Importing the Decision Service	155
Testing the PMML Scorecard Model with Decision Testing	156
Recreating an Example	156
Re-importing a PMML Scorecard Model	157
Importing Tree Ensemble Models with PMML Files	158
Invoking the Tree Ensemble Model	159
Supported PMML Mining Models	160
Model Builder Models	161
Size of the Segmentation Model	161
Runtime Performance	161
Example Decision Services with a Tree Ensemble Model	162

Importing the Decision Service	162
Testing the Tree Ensemble Model with Decision Testing	163
Recreating an Example	163
Re-importing a PMML Mining Model	164
Importing Other PMML Models	165
Supported PMML Models	165
Re-importing a PMML Model	166
Limitations	166
CHAPTER 11: Importing FSML Files	169
Importing Decision Trees from FSML	169
Opening the Wizard and Selecting an FSML File	170
Selecting and Renaming Business Terms	170
Naming the Decision Tree	171
Viewing the Import Summary	171
FSML Decision Tree Import Limitations	171
Example Decision Services with an FSML Decision Tree Model	172
Importing the Decision Service	172
Testing the FSML Decision Tree Model with Decision Testing	173
Recreating an Example	173
Re-importing an FSML Model	174
CHAPTER 12: Importing SAS Programs	175
Uploading a SAS License	175
Selecting Files to Import	176
Creating Functions from SAS Programs	177
Invoking the SAS Program	178
Example Decision Services with a SAS Program	178
Importing the Decision Service	179
Testing the SAS Program with Decision Testing	179
Recreating an Example	180
Re-importing a SAS Program	180
SAS to SRL Mapping Reference	181
Type Mapping	181
Data Step Statements	182
Data Set Options and Statements	184
Operations and Values in Expressions	185
Supported SAS Based Constructs	188
SAS Standard Functions Implemented in Carolina Runtime Library	191

CHAPTER 13:	Adding Procedural Code in Functions	193
Creating Functions	194	
Defining Functions	194	
Adding Local Variables in Functions	195	
CHAPTER 14:	Authoring Decision Logic Using SRL	197
Triggering Code Completion	198	
Code Completion for Common Use Cases	199	
Code Completion Lists	199	
CHAPTER 15:	Controlling the Execution Flow	201
Decision Flow Editor Interface	202	
Decision Flow Editor Toolbar Commands	203	
Decision Flow Floating Toolbar Commands	204	
Creating Decision Flows	205	
The Details Pane	206	
Decision Flow Variables	206	
Creating a Flow Variable	207	
Setting a Decision Flow Variable, Parameter, or Return Type	207	
Inserting Decision Flow Entities	209	
Defining Tasks	209	
Defining Subflows	210	
Defining Split Nodes	210	
Defining Loops	211	
Looping Through a Complex Type Collection Property	211	
CHAPTER 16:	Searching for Entities Using a Query	213
Creating and Running Standard Business Queries	213	
Query Field Definitions	214	
Additional Search Criteria	215	
CHAPTER 17:	Comparing Differences Between Entities	217
Creating and Running Comparison Queries	217	
Comparison Query Results	218	
Comparison Query Results Merge Actions	220	

Limitations in Comparison Queries	221
CHAPTER 18: Detecting Anomalies in Entities	223
Creating and Running Verification Queries	224
Verification Query Results	225
CHAPTER 19: Preparing a Project for Testing and Deployment	227
CHAPTER 20: Deploying Decision Services	229
Batch Processing	230
Requesting Access to a Decision Service	230
Obtaining the Client ID and Secret	231
Obtaining the Bearer Token	231
Using the Bearer Token for Testing	231
SOAP and REST Endpoints in Decision Services	232
Date, Time, and Time Zones for Decision Services	235
Visualizing Data	235
CHAPTER 21: Decision Testing and Analysis	237
Entry Point or Decision Entity Requirements	238
Global Variable Usage	238
Configuring Datasets in CSV Files	239
Dataset Requirements	240
Input and Expected Data Requirements	240
File Structure Requirements	242
Dataset Sizing Guidelines	243
Importing Data from the Analytic Datamart	244
Enabling Data Capturing in the Decision Management Platform	244
Generating Datasets	245
Running the Tests	246
Understanding the Results	248
Viewing Impact Analysis Reports	249
Analyzing the Results	250
CHAPTER 22: Unit Testing	253
Running the Testing Framework	254

Viewing the Testing Results	254
CHAPTER 23: XML Object Model Reference	255
Data Types	255
Enumerations	256
Unsupported Blaze Advisor Types	256
CHAPTER 24: Troubleshooting	257
Issues with Multiple Components in a Solution	257
Decision Table Profiling	258
Issue Attaching and Mapping Dataset Files	258
Dataset Files Contain Missing or Incorrect Data	258
Changes to the Dataset Do Not Appear in Rule Profiling	259
Decision Table Export and Import	259
Cell Coordinate Messages	259
"Import Cannot Proceed" Messages	260
Layout of Double-axis Decision Tables Misaligned When Printing	260
org.xml.sax.SAXException:No TargetNamespace	260
Verifying the Business Object Model in your Decision Service	261
Error Displays When Importing a Business Term Set as a Java BOM	261
Uploaded Blaze Advisor Project Fails Deployment	261
Handling FICO® Application Studio Issues Related to Enumeration Types	262
Handling FICO® Application Studio Issues Related to Duration Types	268
Web Service Request 503 and 502 Errors	268
WSDL No Longer Works for Decision Modeler	269
Decision Executor Appears in SOAP Links	269
Decision Testing: "No Records Were Found"	270
CHAPTER 25: Reference Guide	271
FICO Structured Rule Language Syntax	271
Notation Conventions	271
Language Contents	272
Object_Model	272
Expressions	279
Statements	290
Functions	300
Function_Body	300
Parameter_Bindings	301
Argument_List	301

Control Constructs	302
Exceptions	305
SRL Operators	307
Assignment_Operator	307
Boolean_Operator	307
Collection_Comparison_Operator	307
Comparison_Operator	308
Compound_Assignment_Operator	310
Numeric_Operator	310
String_Comparison_Operator	311
Operator Precedence	311
Keywords	312
Object-specific Keywords	312
Ruleset and Rule-specific Keywords	313
Event-specific Keywords	315
Control Construct-specific Keywords	316
Function-specific Keywords	316
Exception-specific Keywords	316
String-specific Keywords	317
Operators	318
Data Types	320
Special Values	321
Grammar	321
Naming Requirements and Conventions	322
Punctuation	322
Reserved Words	323
Writing Comments in SRL	325
Date, Time, Timestamp, and Duration Data Types	326
Calculating and Comparing Date, Time, Timestamp, and Duration Values	327
Creating a Date, Time, Timestamp, or Duration	328
Data Type Conversions for Date, Time, Timestamp, and Duration Values	333
Formatting Output Strings for Date, Time, Timestamp, and Duration	334
Money Data Type	334
Creating a Money Type	335
Converting Money Values	337
Calculating and Comparing Money Values	340
Collections	341
Arrays	342
Arrays in the Rule Builder	343
Array Statements	344
Set Operations	345
Associations	349

Association Statements	350
Dynamic Objects	351
Math Built-ins (NdMathBuiltIns)	352
Converting Values	353
math().ceil()	353
math().floor()	353
math().round()	354
math().truncate()	355
Calculating and Returning Values	356
math().abs()	356
math().arctan()	357
math().cos()	357
math().exp()	358
math().log()	358
math().mod()	359
math().power()	359
math().sin()	359
math().tanh()	360
Comparing Values	360
math().max()	360
math().min()	361
Date and Time Data Type Values	361
Date and Time Data Types	362
Returning the Default Calendar	362
Setting Default Date, Time, and Timestamp Formats	362
Valid Default Format Patterns	362
Calendar, Date, Time, Timestamp and Duration Operations	364
NdCalendar	364
Comparison and Calculation Methods	372
NdDate	376
NdDuration	383
NdTime	387
NdTimestamp	393
brUnit Assertions	401
brUnit Assertions Methods	402
The Difference Between the assert().assert and assert().check Methods	408
assert().fail Methods	409
Currency Built-in Functions	410
format(BigDecimal):string	410
parse()	410
Portable Built-ins (NdPortableBuiltins)	411
Using Dates with the Portable() Built-in Functions	413

portable().century()	414
portable().compareDates()	415
portable().date()	416
portable().dateMinusDays()	416
portable().datePlusDays()	417
portable().dateToInt()	418
portable().day()	418
portable().dayOfWeek()	419
portable().daysInMonth()	420
portable().ddyyyy()	421
portable().ddmmyyyy()	421
portable().diffInDays()	421
portable().diffInMonths()	422
portable().diffInYears()	423
portable().intToDate()	424
portable().isLeapYear()	425
portable().mmddyyyy()	425
portable().mmyyyy()	426
portable().month()	426
portable().time()	427
portable().verifyDate()	427
portable().verifyIntDate()	428
portable().verifyMonth()	428
portable().verifyYear()	429
portable().year()	429
portable().ymdToDate()	430
portable().yyyyddd()	431
portable().yyyyymm()	431
portable().yyyymmdd()	431
Converting Values with the Portable() Built-in Functions	432
portable().charToInt()	432
portable().compressString()	433
portable().concat()	433
portable().concatByContent()	434
portable().subString()	434
portable().toBoolean()	435
portable().toFloat()	436
portable().toInteger	438
portable().toLowerCase()	439
portable().toString()	440
portable().toUpperCase()	441
Testing Strings and Numeric Values Using Portable() Built-ins	442

portable().findChar()	442
portable().findString()	443
portable().is_in()	443
portable().isFloat()	444
portable().isInteger()	445
portable().max()	445
portable().min()	446
portable().not_in()	447
portable().selectValue()	448
Standard Built-in Functions	448
ignore()	449
print()	450
Scorecards and Reason Codes	450
NdReasonCode	450
NdScoredCharacteristic	451
NdScoreModelReasonCalculationOptions	452
NdScoreModelReturnInfo	453
addReason(NdReasonCode, real)	454
calculateReasons(NdScoreModelReasonCalculationOptions)	455
calculateReasons()	456

APPENDIX A: Exporting FICO® Blaze Advisor Projects 457

Deciding Which Projects to Export	457
Preparing Blaze Advisor Projects for Export	458
Reviewing Your Business Object Models	459
Avoiding Security Manager Exceptions with ObjectMapper	459
Setting the Classpath for Exported Projects	460
Entry Points in Rule Service Definitions	460
Excluding Content When Exporting Blaze Advisor Projects	461
Addressing Unsupported Functionality	462
Exporting Blaze Advisor Projects Using a Wizard	462
Importing Decision Modeler Projects into Blaze Advisor	463

APPENDIX B: Upgrading Decision Modeler Component Instances 465

Upgrading Decision Modeler Component Instances 3.1 or Later	465
Preparing Your Component for Upgrade	465
Upgrading the Component	465
Rolling Back an Upgraded Component	466
Upgrading Decision Modeler Component Instances 3.0 or Earlier	466
Downloading the Decision Logic and Noting the Component Settings	466

Downloading the Decision Logic in a ZIP File to Your Desktop	467
Noting the Togglz Admin Console Settings	468
Noting the Configure Data Reporting Settings	469
Creating a New Decision Modeler Component in Your Solution	469
Creating a New Component	470
Specifying the Component Settings and Uploading the Decision Logic to the New Component	470
Specifying the Togglz Admin Console Settings	470
Uploading the Decision Logic to the New Decision Modeler Component	471
Specifying the Configure Data Reporting Settings	471
Testing and Managing the New Decision Modeler Component	472
Testing the Decision Logic in the New Decision Modeler Component	472
Publishing the New Decision Modeler Component	472
Inserting the New Endpoints URL in the Client Application	472
Deleting the Older Decision Modeler Component	473
Using the New Component in the Projects Page	473
APPENDIX C: Contacting FICO	475
Product Support	475
Product Education	475
Product Documentation	475
Related Services	476
FICO Community	476
INDEX.....	477

CHAPTER 1

Welcome to FICO® Decision Modeler

FICO® Decision Modeler is a component for authoring, testing, deploying, and executing decision services. A *decision service* is a web service that is called by an enterprise business application to provide a decision. Decision Modeler evaluates incoming data against the decision logic and answers a business question for other services.

For example, an insurance application could call a decision service to determine whether or not an applicant meets the minimum requirements for further processing. After the call is made, the applicant's information is evaluated against the insurance company's business rules and returns a decision.

A decision service can be used for any business application that requires an automated decision. When an enterprise application needs a decision, a call is executed to the FICO® Decision Management Platform which manages scalability, multiple service requests, updates to the decision logic, and more.

Decision Modeler stores all versions of the decision logic, from those currently in development to the decision logic that has been approved for production. Decision Modeler helps you manage the life cycle of your decision logic.

What's New in FICO® Decision Modeler 3.1.5

This release contains the following new features.

Supported Versions of Projects

You can upload a Blaze Advisor project, repository, or workspace exported from Blaze Advisor 7.7.x, 7.6.x or 7.5.x or a project exported from Decision Modeler 2.0.x through 3.1.5 into a Decision Modeler workspace.

Decision Modeler is Available in French or English

The Decision Modeler interface, the Decision Modeler Projects Page, and the Decision Modeler documentation are available in French or English. Contact your Support representative for more details.

Upgrade to a New Version of Decision Modeler

If you have a Decision Modeler 3.1 component, you can use the DMP Component page to upgrade to your component to Decision Modeler 3.1.5.

Decision Table Enhancement

In the **Insert New Condition** and **Insert New Row** windows, the search for object properties is now case-insensitive.

Decision Testing Enhancement

In the **Configure Data** window, you can select a class and all properties are automatically added as input or result columns.

Change to Project Explorer Toolbar

Downloading rules is now a workspace-level operation so the Download Rules command is no longer available on the Project Explorer toolbar.

Supported Web Browsers

The following web browsers are supported:

- Microsoft® Internet Explorer® 11
- Mozilla® Firefox®
- Google Chrome™

 **Note** Support for Microsoft® Internet Explorer® 11 is deprecated.

CHAPTER 2

Introduction to Decision Modeler

FICO® Decision Modeler enables business users to easily develop, test, and deploy decision services in the cloud. A decision service is a deployed web service that provides a decision to an enterprise application based on business rules.

The logic for these decision services is developed in one or more projects that you import or create in Decision Modeler. After the projects are completed and thoroughly tested, they are deployed as decision services.

The first step in the process is to create a workspace, which is a container for one or more projects. After a workspace is created, you can create and configure one or more projects. Alternatively, you can use a wizard to import a Blaze Advisor project, workspace, or repository into a Decision Modeler workspace.

The next step is to configure the entry point that defines the type of data that will be passed into the decision service. For example, if the decision service will process college applications, the entry point function might include an input parameter of type CollegeApplication class and return an object of type CollegeApplication class with any modified values based on data evaluation. The entry point defines the format of the data that will be passed in for evaluation and the format of data that will be returned.

Because building a decision service is a team effort, you can specify a role for each team member in the workspace. For each project in a workspace, you can set the permissions that each team member has for a specific project. For example, Sue, a rule author, may have read-write permission for Projects A and B and read-only permissions for Projects C and D. While John, another rule author, may have read-write permission for Projects A and C and no access to Projects B and D.

After the permissions are set, your rule authors can open a project and edit any existing decision logic or author new decision logic. Decision Modeler contains several authoring editors, each of which is uniquely designed to support different types of decision logic. This is an overview of the available decision entity types:

- **Ruleset:** A set of sequential rules that performs a specific task.
- **Decision Table:** A tabular presentation of decision logic that is used when each combination of condition expressions and their action statements are based on the same business terms.
- **Decision Tree:** A tree-like graph used to model decisions and their possible consequences. It is a convenient way to visualize the different paths in a complex decision process.
- **Scorecard:** An entity used to predict the likelihood of future behavior about a customer or prospect based on historical data. Create your own scorecard or import a PMML scorecard.

- **Function:** An entity used to execute a set of procedural statements on a specific data context. SAS programs and other types of PMML models can be imported as functions and used in a project.
- **Decision Flow:** An entity that specifies the sequence in which data will be passed into the decision entities for evaluation.

There are queries to assist your rule authors during the development process.

- **Comparison Query:** This query compares two different versions of a decision entity and allows the promotion of an older version to be the current version.
- **Standard Business Query:** This query locates decision entities based on the specified criteria.
- **Verification Query:** This query searches for anomalies in your decision logic so they can be fixed before the testing phase.

Decision Modeler includes two testing modules that can be used for projects in the Development, Staging, or Production environments:

- For unit testing a portion or all of your decision logic in a project, you can use Decision Testing. You select an entry point and upload test data to compare the actual results against the expected results. Any tests that do not generate the desired outcome can be analyzed in detail to see if the problem lies with the data or the decision logic. After the data is run, reports are generated that show the distribution of the result values and any variances between the results and the expected data.
- For testing a decision service, you can use a batch job to test SOAP or REST endpoints. After a project endpoint is selected, you upload data and then schedule or run the job. The data can simulate the types and volume of data expected in a production environment. The data can be visualized in reports, if you have a Tableau license.

After a project is completed, the appropriate team member submits it to the Staging environment. After the project has been tested and, if it is approved for deployment to Production, the project is deployed as a decision service that is ready to be called by your enterprise application to return a value.

CHAPTER 3

Decision Modeler Workspaces and Projects

Create a workspace and add or import one or more projects in the Decision Modeler Projects page. The maximum number of projects that can be created and/or imported in a workspace is five.

The workspace that you create is a DMP component that is automatically added to a DMP solution of the same name. You can create one or more workspaces in Decision Modeler.

- All projects in the workspace share the same resources. If you have larger or more complex projects that may require more resources, FICO recommends that you create a dedicated workspace for each of those projects. If you have several smaller or simpler projects, those projects can share a workspace.
- Only projects with a Rule Service Definition (RSD) are listed during the import process and therefore are eligible for import.

There are two ways to add projects to a workspace.

- Import projects from a ZIP file from a supported version of Blaze Advisor or Decision Modeler. This method allows you to include and configure projects at the same time.
- Create new projects in the workspace on the Projects page.
If you create new projects in a workspace, you need to use the **Configure project** icon on the toolbar to perform one of the following actions:
 - Upload an XML object model (XML schema).
 - Configure a project with default settings.
If you configure a project with default settings, you can use the Project Configuration dialog from the RMA to import a JAR or XML schema and then return to the Configure Project dialog on the Projects page to configure the project using the imported JAR or XML file.

If you have access to a supported version of Blaze Advisor, you can import a single project in a workspace by connecting directly to Decision Modeler through the Blaze Advisor IDE. Contact FICO Product Support for details.

Related Links

- [Creating New Workspaces](#)
- [Creating New Projects](#)
- [Importing Projects](#)
- [Modifying a Project Configuration](#)
- [Managing Object Models in a Project Configuration](#)

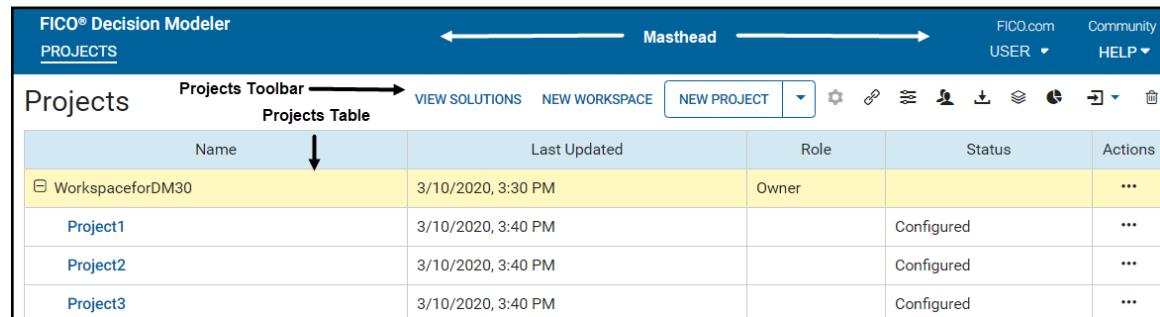
Managing Workspaces and Projects

Manage workspaces and projects using the Decision Modeler Projects page. The table on the Projects page displays information about the workspaces and the projects that you have either created or imported. The table also lists workspaces and projects where you have been assigned a specific role by another user.

When you create a workspace, you are the Owner and links, buttons and icons are available to you. When you select in a table row for a workspace or project, the links, buttons, and icons on the Projects page are appropriately enabled or disabled based on your role.

 **Note** In some cases, you may need to complete an action such as selecting a specific table row for an icon to become enabled or a contextual workspace or project dialog to become available.

The Projects page includes:



Name	Last Updated	Role	Status	Actions
WorkspaceforDM30	3/10/2020, 3:30 PM	Owner	Configured	...
Project1	3/10/2020, 3:40 PM		Configured	...
Project2	3/10/2020, 3:40 PM		Configured	...
Project3	3/10/2020, 3:40 PM		Configured	...

Figure 1: The Projects page with callouts for important features

Table 1: The Projects Toolbar

Name	Description
Learn	Opens the Learn about Decision Modeler page when you select Help > Learn . The Learn page contains guides and links to how-to videos for creating decision tables, decision trees, rulesets, and scorecards. You can also provision an example project that you can open from the Projects page.
View Solutions	Opens the Decision Management Platform (DMP) Solutions page in a new tab.

Table 1: The Projects Toolbar (continued)

Name	Description
New Workspace	Allows you to create a workspace. When a workspace is created, a Decision Modeler component and a solution with the same name are created in the DMP.
New Project	Allows you to create a project in your workspace when a workspace row is selected. Use the commands in the New Project drop-down list to: <ul style="list-style-type: none">■ New project Create a project. Each time you want to create a project, you use this command to launch the New Project dialog. After creation, the project must be configured either with default settings or with an XML object model and entry points.■ Import projects Import one or more projects using a ZIP file that you export from a supported version of Blaze Advisor or Decision Modeler.
Configure project	Opens the Project Configuration wizard where you can configure a project with an XML schema. Alternatively, you can configure a project using the default settings. Later you can return to this wizard to update its project configuration and redeploy it to the Development environment.
Project links	Allows you to view the project links and the client id and secret for a workspace: <ul style="list-style-type: none">■ Links Displays the SOAP and REST endpoint URLs for an available project Link type and Environment. If your project has a XML object model, you see SOAP endpoints. If your project has a Java object model, you see both SOAP and REST endpoints.■ Client ID Displays the client ID and secret used to obtain the bearer token for decision service deployment. The client ID and secret can be used for all the projects in your workspace.
Settings	Allows you to launch a workspace or project settings dialog depending on whether you select a workspace or project row prior to selecting the Settings icon. <ul style="list-style-type: none">■ Use the Workspace Settings dialog to select projects to include or remove from the workspace in the Development environment. In future, if you want to include a project you previously removed, you need to select it from the Workspace Settings dialog.■ Use the Project Settings dialog to view a Last Modified timestamp for each environment where the project has been submitted or deployed.
Manage workspace team	Allows you to add one or more team members to contribute to your workspace and the projects within it. You can search for user names to add to your team and click Save. The users you have added will see the shared workspace and its projects on their Projects page the next time they log in or refresh the page.

Table 1: The Projects Toolbar (continued)

Name	Description
Download workspace	Downloads a workspace as a ZIP file to save its contents. You can import these contents to a supported version of Decision Modeler using the New Project > Import projects command.
View batch jobs	Opens the Batch Jobs page, where you can define, run, or schedule batch processing jobs. After you upload a dataset, the data is evaluated using the decision service.
View reports	Opens the Tableau projects for the workspace if you have purchased a Tableau license with the Analytic Data Mart.
Manage deployment	<p>Allows you to select projects from a workspace to submit to Staging for testing or deploy to Production after approval.</p> <ul style="list-style-type: none"> ■ Submit to staging Select the projects that are ready to submit to the Staging environment and optionally enter a comment. <ul style="list-style-type: none"> - The Last Submission column shows a timestamp when the workspace contents were submitted. - The Changes column shows the status of each project in the workspace: Any new modification that you submit will override existing changes in the Staging environment: <ul style="list-style-type: none"> - Unchanged If the project has no modifications since the last time it was submitted to Staging. - Modified If the project was changed since the last time it was submitted to Staging. - Removed If the project was excluded from the workspace since the last time it was submitted to Staging. - Deleted If the project was deleted from the workspace since the last time it was submitted to Staging. ■ Change Status Approve or Reject workspace contents submitted for testing and optionally enter a comment. In the Change Status dialog, you cannot assign a status to a specific project. With the approval of the workspace contents, a Deployer can deploy the workspace contents to Production. Conversely, if workspace contents are rejected, the workspace can be edited in the Development environment and resubmitted to the Staging environment for testing. The Changes column shows the status of each project in the workspace: <ul style="list-style-type: none"> - Unchanged If the project has no modifications since the last time the status was changed. - Modified If the project has been modified since the last time the status was changed.

Table 1: The Projects Toolbar (continued)

Name	Description
	<ul style="list-style-type: none"> - Removed If the project was excluded from the workspace since the last time the status was changed. - Deleted If the project was deleted from the workspace since the last time the project status was changed. <p>■ Deploy to production Deploy approved workspace contents to the Production environment. The Changes column shows the status of each project in the workspace:</p> <ul style="list-style-type: none"> - Unchanged If the project has no modifications since the last time it was deployed. - Modified If the project has been modified since the last time it was deployed. - Removed If the project was excluded from the workspace since the last time it was deployed. - Deleted If the project was deleted from the workspace since the last time it was deployed.
Delete	<p>Allows you to delete a workspace or a project.</p> <p>Important There is NO UNDO functionality for this action.</p> <p>Tip To save your workspace prior to deleting it, FICO recommends that you use the Download workspace command.</p> <ul style="list-style-type: none"> ■ Select a workspace row and click the Delete icon. Important If you delete a workspace, all its projects are also deleted at the same time even if these projects have been deployed to Staging or Production. ■ Select a project row and click the Delete icon. If the project you deleted in Development has already been pushed to the Staging or Production environments, its status is Pending deletion in the Projects table and Deleted in the Submit to Staging dialog. <p>Note After you click Delete, the Delete Project dialog may remain for a short time as the deletion process begins and its progress is displayed in the Status column of the workspace row in the Projects table.</p>

Table 2: The Projects Table

Column Name	Description
Name	Displays the name you entered when you created the workspace or project or edited its property settings. To author rules, click the project name or use the Open command from the Actions menu.
Last Updated	Displays the timestamp when the project was last updated in the Development environment.
Role	Displays the role assigned to the user. If you create a workspace, you are automatically the Owner and have access to all workspace and project features. Possible user roles are Owner, Administrator,

Table 2: The Projects Table (continued)

Column Name	Description
	Approver, Author or Deployer. Each role allows a different level of access.
Status	<p>Displays the status for a project in the Development environment. Possible values include:</p> <ul style="list-style-type: none"> ■ Needs configuration The project has not yet been configured with an XML schema or an imported project. ■ Configured The project has been configured with an XML schema or an imported project. ■ Submitted The project has been submitted to the Staging environment. ■ Approved The project in the Staging environment has been approved for deployment to production. ■ Published The project has been published to the Production environment. ■ Pending removal The project has been hidden in the workspace, but there are still instances of the project in the Staging or Production environment. The removed project will remain visible with project-level operations disabled on the Projects page until the change has been deployed to Staging or Production. ■ Pending deletion The project has been deleted, but there are still instances of the project in the Staging or Production environment. The deleted project will remain visible with project-level operations disabled on the Projects page until the change has been deployed to Staging or Production.
Actions	Launches a drop-down menu containing many of the same commands as those available from the Projects page toolbar when you have selected a workspace row. If you have selected a project row, you see a subset of these commands.

Table 3: The Masthead

Name	Description
Projects link	Allows you to return to the Projects page from the Project editor page.
Username link	Allows user to select the Exit command from the drop-down menu to log out of Decision Modeler.
Help link	<p>Allows access to:</p> <ul style="list-style-type: none"> ■ Contents -- the Decision Modeler documentation ■ Learn -- the Decision Modeler Learn page ■ Privacy and Terms -- FICO Analytic Cloud Trust Center pages ■ About -- Decision Modeler copyright information

Related Links

- [Creating New Workspaces](#)
- [Creating New Projects](#)
- [Importing Projects](#)
- [Managing Workspace Teams](#)
- [Viewing and Managing Workspace Settings](#)
- [Viewing and Managing Project Settings](#)
- [Viewing Project Links and the Client ID](#)

Creating New Workspaces

On the Projects page you can create one or more workspaces and each of these workspaces can contain one or more projects that you create or import.

- The **Name** field can contain up to 100 characters. A valid name can include uppercase and lowercase letters, numbers, spaces, and underscores.
- The **Description** field can contain up to 400 characters.

To add projects to a workspace, select the workspace row. A maximum of five projects (imported and/or created) are allowed in a given workspace.:.

- If you want to create projects, click **New Project > Create project**.
After you create projects, you need to configure them.
- If you want to import projects, click **New Project > Import projects**.
When you import projects, the projects are created and configured with the object model they were exported with.

 **Note** If you create one or more projects in a workspace, and then you attempt to import one or projects using a ZIP file, the imported contents from will overwrite the projects you created in the workspace. If you want to import projects, FICO recommends that you create a separate workspace for this purpose.

Related Links

- [Decision Modeler Workspaces and Projects](#)
- [Managing Workspaces and Projects](#)
- [Creating New Projects](#)
- [Importing Projects](#)
- [Viewing and Managing Workspace Settings](#)
- [Managing Workspace Teams](#)

Viewing and Managing Workspace Settings

The Workspace Settings dialog provides information about a workspace and allows you to remove (hide) or include (show) projects.

-  **Note** To launch the Workspace Settings dialog, you must highlight a workspace row in the Projects page before selecting the **Settings** icon on the toolbar.

The workspace information that is provided in the dialog includes:

- **Name** The name of the workspace.
- **Solution name** The name of the solution where the workspace component is located on the DMP.
- **Product version** The Decision Modeler version used to create the workspace.
- **Creation date** The timestamp when the workspace was created.

If you are an Author, Deployer or Approver, you have a read-only view of the Workspace settings.

If you are the Owner or Administrator, you can use workspace settings to exclude or include projects for a workspace as long as the total number of projects is not greater than five. To remove a project from the Projects page, clear the check box next to its name and save your changes.

- When you import projects from a ZIP file, use the Workspace Settings dialog, you can change your workspace contents by including or excluding projects.
- The projects that you explicitly select will be deployed to the Development environment.
- If you have five projects in your workspace, use the Workspace Settings dialog to exclude one or more projects before creating or importing any new projects.
- If you have subprojects where there is a shared dependency with the other projects being imported into a workspace, those subprojects are automatically imported. The subprojects exempted from the maximum project count.

-  **Note** Projects with a status of pending removal or pending deletion do not count towards the limit of five projects per workspace.

Related Links

[Managing Workspaces and Projects](#)

[Creating New Workspaces](#)

[Importing Projects](#)

Downloading Workspaces

To preserve the contents of your workspace use the **Download workspace** command on the Projects page.

There are some differences in the workspace content downloaded as a ZIP file depending on which option you choose.

- **Include uncommitted changes but no version history.** The workspace content is downloaded along with any saved local changes. The version history for your entities is not included if you select this option.
- **Include version history but no uncommitted changes.** The workspace content is downloaded along with any version history for your entities. Your saved local changes that have not been committed are not included if you select this option.



Tip For best results, check in your changes to ensure that you have all the latest changes before downloading the workspace using this option.

After you download the ZIP file, you can upload it to a supported version of Decision Modeler.

Managing Workspace Teams

When you create a workspace, you are automatically assigned the Owner role, which gives you full access to the workspace and any projects you create or import into that workspace. As the Owner, you can share your workspace by adding other users to your team using the **Manage Workspace Team** dialog.

You can share a workspace and assign individuals with specific skills and responsibilities to appropriate team roles. For example, you could assign a Business Analyst the Author role so that they can create decision entities or you can assign a Quality Assurance Engineer to the Approver role so that they can test the projects in the Staging environment and then Approve or Reject them.

Each user can only have one role in a workspace and that role will apply to all projects in that workspace. Depending on the user's role, different permission levels can be set for each project. After you have added your team members to the workspace, you can assign a role for each user. After you assign each user a workspace role, you can edit their permissions for each project. After you click **Save**, the workspace and any projects will be available in each users' Project page the next time they login or refresh the page.



Note Only Owners and Administrators have read-write access for the **Manage Workspace Team** dialog. All other roles have read-only access.

Related Links

[Managing Workspaces and Projects](#)

[Creating New Workspaces](#)

Workspace Team Roles

Team members in a workspace are assigned a role: Owner, Administrator, Author, Approver, or Deployer. Each team member can be assigned only one role for a workspace.

This table describes each role and related access in the Decision Modeler interface.

Table 4: Workspace Roles and Access Description

Role Name	Role Description	Access Description
Owner (default)	<p>The Owner role is automatically assigned to the original creator of the workspace. There can only be one owner for a workspace and this role cannot be reassigned.</p> <p>An Owner can add or delete workspace users and set their roles, but they cannot delete or modify their own role. Additionally, an Owner can set project-level permissions for all workspace users.</p>	<p>All buttons, icons, and links on the Project Page and the Project editor are available to this user. In some cases, the Owner needs to perform an action such as clicking in a table row before a command is enabled on the Projects toolbar or in the Actions menu.</p>
Administrator	<p>The Administrator role is similar to the Owner role except that an Administrator cannot delete or modify their role or that of the Owners.</p> <p>There can be one or more Administrators on a team.</p> <p>Only Administrators can release the locks on decision entities checked out by Authors and other Administrators.</p>	<p>All buttons, icons, and links on the Project Page and the Project editor are available to this user. In some cases, the Administrator needs to perform an action such as clicking in a table row before a command is enabled on the Projects toolbar or in the Actions menu.</p>
Author	<p>After a project is created in a workspace, the Author configures the project and deploys it to the Development environment. The Author develops the decision entities in the Project editor and performs some testing before submitting the project to the Staging environment for more testing.</p> <p>Note There can be one or more Authors on a team.</p>	<p>Workspace permissions for the Author role are described in the Workspace Commands table below.</p>
Approver	<p>The Approver changes the status of a project in a workspace from Submitted to Approved or Rejected based on test results in the Staging environment.</p> <p>If the status is changed from Submitted to Approved, the project can now be deployed to Production.</p> <p>If the status is changed from Submitted to Rejected, the project is sent back to the Development environment where the Author makes any necessary changes and resubmits it to the Staging environment for testing.</p>	<p>Workspace permissions for the Approver role are described in the Workspace Commands table below.</p>

Table 4: Workspace Roles and Access Description (continued)

Role Name	Role Description	Access Description
	Note There can be one or more Approvers on a team.	
Deployer	After the Approver approves a project in the Staging environment, the Deployer deploys the project to the Production environment as decision service. Note There can be one or more Deployers on a team.	The Deployer does not have any access to the Project editor. Workspace permissions for the Deployer role are described in the Workspace Commands table below.

The actions described in this table become available when a workspace row is highlighted in the **Projects** table.

Table 5: Workspace-Level Commands

Command	Administrator/ Owner	Author	Approver	Deployer
New Workspace	X	X	X	X
New Project	X	X		
Import Projects	X			
Project Links	X	X	X	
Workspace Settings	X	Read-Only	Read-Only	Read-Only
Manage Team	X	Read-Only	Read-Only	Read-Only
Download Workspace	X	X		
View Batch Jobs	X	X	X	
View Reports	X	X	X	X
Submit to Staging	X	X		
Change Status	X		X	
Deploy to Production	X			X
Delete Workspace	X			

Project Permissions

Depending upon their assigned workspace roles, users can be assigned different permissions for each project in the workspace.

The project-level permissions that are available for each user role are listed in this table.

Table 6: Available Project Permission Levels

Role	Read-Write	Read-Only	No Access
Administrator/ Owner	X		
Author	X	X	X
Approver		X	X
Deployer			X

The commands for each level of project permissions are described in this table. These commands become available when a project row is highlighted in the **Projects** table.

Table 7: Project-Level Commands

Command	Read-Write	Read-Only	No Access
Open Project	X	X	
Configure Project	X		
Project Settings	X		
Project Links	X	X	
Delete Project	X		

Capturing Data Using the Analytic Data Mart

The Decision Modeler Projects Page provides a convenient way to deploy your projects to the Decision Management Platform (DMP). However, there are a few settings that can only be managed using the DMP. Most of these settings have to do with capturing and reporting data generated by the projects in your workspace. By default, capturing and reporting data occurs for all projects in all stages of the project life cycle.

 **Note** The Analytic Data Mart is available only to customers using the cloud version of Decision Modeler.

The DMP includes the Analytic Data Mart (ADM) for capturing and retrieving data. ADM captures data from the projects in your workspace and makes the data available for reporting and analysis. ADM consists of two data stores: the event store and the report store. The event store is a log of invocations and the report store is an analytical data store. Each time a project is invoked, event data about the invocation and environment is captured and persisted in the ADM. For example, the ADM captures data from web service deployments and batch jobs. Depending on the settings you choose for a project in a specific environment, the input and output payloads of the requests may also be captured.

In Decision Modeler, the data are available for use with Decision Testing and Tableau. In Decision Testing, the data logged in the ADM can be used to execute against new versions of a project. You can also import data from batch processing or web service deployments into Decision Testing.

If you have a license for Tableau, the data can be accessed to develop reports. If you need a Tableau license, contact your FICO representative.

Togglz Admin Console Settings

You can use the Togglz Admin Console settings to change the default behavior of the Decision Modeler projects in your workspace.

 **Note** If you want to disable or enable any of these settings, FICO recommends doing this before configuring new projects or importing a Blaze Advisor workspace with projects into your workspace.

If you disable the first two Togglz settings after configuring a project or importing a Blaze Advisor workspace, you will need to do the following:

- Open the **Configure Projects** wizard from the Projects page, navigate through all wizard pages without changing the settings, and click **Done** and then click **Close** to redeploy the projects.
- Resubmit, redeploy, and restart all Decision Modeler component instances so that the changes take effect in the Staging and Production stages of the decision service life cycle. This step will need to be performed by your Decision Management Platform administrator.

FICO Analytic Data Mart Event Logging

(Enabled by default) Disabling the FICO Analytic Data Mart Event Logging feature prevents any invocation data from being captured and persisted in ADM when the decision service is invoked. You should only disable this feature if you are sure you do not want to leverage any features that rely on the ADM data such as Tableau report generation or Decision Testing in Decision Modeler.

REST Payload for Single Input Uses Wrapper Element

(Enabled by default) For consistency, REST endpoints in Decision Modeler, whether they have a single input or multiple inputs, include a wrapper element. In previous releases, the REST payload for single inputs did not include a wrapper element. If you want the previous behavior, disable the **REST Payload for Single Input Uses Wrapper Element** option.

Compilation Job (Beta)

(Disabled by default) Enable the Compilation Job (Beta) feature to improve compilation performance. During the default import project process, projects are deployed to the Development environment, compiled and the resulting deployment files are added behind the scenes to your Decision Modeler workspace. Depending on the size and complexity of your projects, the default compilation process may take several minutes. However, if you enable this feature, a compilation job is used. The resulting deployment files are still added to the Decision Modeler workspace.

Disabling or Enabling Default Features using the Togglz Admin Console

You can enable disable or enable features in the Togglz admin console using the Decision Management Platform (DMP).

- Disable settings in the Togglz admin console if you do not want data logged to the Analytic Data Mart (ADM) or you do not want wrapping for a single input payload.
- Enable a setting in the Togglz admin console if you want to use a Compilation Job to improve performance during the project import process.

 **Note** If you want to disable or enable any of these settings, FICO recommends doing this before configuring the projects in your workspace.

- 1 On the Projects page, click **View Solutions**.
- 2 Click the solution name. The solution has the same name as the workspace.
- 3 In the **Solution** panel, click **Components**.
- 4 In the **Component name** column, select the component name and click the down arrow.
- 5 Select **Configure > Togglz admin console**.
- 6 Select one of the following options:
 - Click the **FICO Analytic Data Mart Event Logging** gear icon. If this option is disabled, then features that pull data in from the ADM such as Decision Testing or Tableau will not have data available.
 - Click the **REST Payload for Single Input Uses Wrapper Element** gear icon.
 - Click the **Compilation Job (Beta)** gear icon. If this option is enabled, then a compilation job is used instead of the default job to compile projects during the import process.
- 7 Clear the **Disabled** or **Enabled** check box and click **Done**.

Limiting Data Capture in the Analytic Data Mart

By default, when you configure a project or import a Blaze Advisor workspace, Analytic Data Mart (ADM) tables are created for each operation and each life cycle stage (Development, Staging and Production). If you do not need data capture for every operation or in every life cycle stage, you can edit the Data Capture settings in the FICO Decision Management Platform (DMP). Note that the Development stage is referred to as the Design stage in the DMP.

To access the DMP from the Decision Modeler Projects page, click **View Solutions**. The DMP opens in a new browser tab. Click on the solution name, which will have the same name as the Decision Modeler workspace. Click the **Settings** tab to open the **Data Capture** page to control the event data persisted to the ADM.

Events

Select or clear the input/output for specific environments (Design, Staging, or Production) to limit data capture for events such as rules fired events. The **Internal**

- **Input and Output of each step within the Decision Service** option does not apply to Decision Modeler decision services. For more information, see the Data Capture Settings topic in the Decision Management Platform Help accessible through the Help link on the Solutions page.

Important

- If you disable the input/output for a given environment on the Data Capture Settings dialog, then reporting data is not captured regardless of the settings selected in the **Configure Reporting Data** dialog.
- If the Togglz admin console setting for ADM event generation is disabled, then the Data Capture Settings are not applied.

Reports

When a decision service is invoked, the data is stored by the ADM but is not immediately available in Tableau. By default, event and report data is sent to Tableau once every 24 hours. The **RUN NOW** button allows you to run the reports manually and access your data in Tableau on-demand.

Disabling Reporting Data in the Analytic Data Mart

When you create or import a project in a Decision Modeler workspace, your workspace is automatically enabled for reporting data in the Development environment. When projects are submitted to the Staging or Production environments, data reporting for those environment becomes enabled.

You can disable reporting data at runtime for all environments or specific environments using the **Configure Report Data** window in the FICO Decision Management Platform (DMP). Note that the Development stage is referred to as the Design stage in the DMP.

To access the DMP from the Decision Modeler Projects page, click **View Solutions**. The DMP opens in a new browser tab. Click on the solution name, which will have the same name as the Decision Modeler workspace. Click **Components** in the panel. Click the menu next to component and select **Configure Reporting Data** to open the window.

If the menu command is disabled, this means that the workspace does not contain a project that has valid endpoints or that the FICO Analytic Datamart Event Logging setting is disabled in the Togglz admin console.

For more information, see the Configure Component Reporting Data topic in the Decision Management Platform Help accessible through the Help link on the Solutions page.

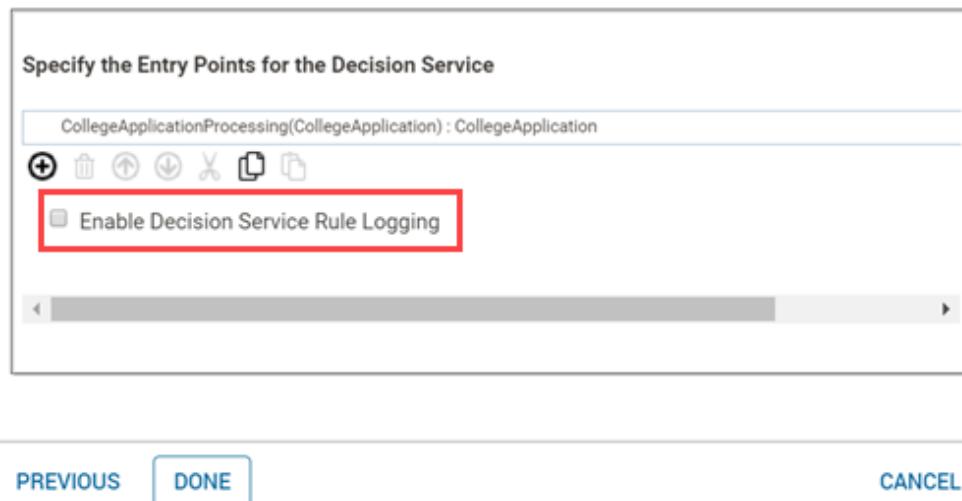
Enabling Decision Service Rule Logging

Rule events data are stored in the Analytic Data Mart (ADM) when you enable the option in the **Configure Project** wizard on the Decision Modeler Projects page.



Note

- This feature should not be enabled if you plan to run the decision service in a high load production environment.
- FICO recommends that you do not select this option for viewing rule events because this support is currently limited.



To enable rule event logging in a project:

- For a project imported from Blaze Advisor, select the **Configure Project** command to launch the **Configure Projects** dialog. Select the **Enable Decision Service Rule Logging** check box and click **Done**.
- For a project created in Decision Modeler, you can select the **Enable Decision Service Rule Logging** check box when you configure the project using an XSD file. If you do not select the option when the project is configured, you can do this later. Select the **Configure Project** command to open the dialog, click **Next** twice, select the check box, and click **Done**.

Creating New Projects

Create a Decision Modeler project by entering a name and a description.

- The **Name** field can contain up to 100 characters. A valid name can include uppercase and lowercase letters, numbers, spaces, and underscores. A name cannot start with a number.
- The **Description** field can contain up to 400 characters.

A Decision Modeler project contains the logic that can be deployed as a decision service, alone or with other projects in a workspace. Before you can author decision logic in a project, you must configure it.

Related Links

- [Decision Modeler Workspaces and Projects](#)
- [Managing Workspaces and Projects](#)
- [Creating New Workspaces](#)
- [Configuring a Project Using an XML Schema](#)
- [Configuring a Project Using Default Settings](#)
- [Configuring a Project With Entry Points](#)

Configuring a Project Using an XML Schema

Configure a project by importing an XML schema. The schema must include a namespace or an error occurs during the import process.

 **Note** The **Close** button is available on each wizard page. If you click **Close** before you import a schema, you can exit the wizard without configuring a decision service. However, if you click **Close** after you have imported a schema, but before you have selected an input and output type and entry points, the project is configured with default settings. Return to the Configure Project wizard in the Projects page at any time to update the configuration.

- 1 Select the project row and click the **Configure Projects** icon on the Projects page.
 - 2 Click **Next** and click **Browse** next to the **Import XML Schema** field to select the schema (.xsd) and click **Open**.
 - 3 (Optional) Click **Preview** to review the content of the selected XML schema.
 - 4 Click **Next**.
 - 5 Select an **Input and Output Type** from the drop-down list for your decision service and click **Next**. The input type defines the data that the decision service receives and the output type defines the returned modified data. The default string input and output types are used if you do not explicitly select an input and output type.
 - 6 Ensure that the entry points displayed are the ones you want to use or select new entry points.
The entry points are used in the deployed decision service to provide data for evaluation.
-  **Note** If you need to select an entry point, avoid selecting an entry point with an interface because an entry point with this type of signature cannot be used to invoke a decision service.
- 7 (Optional) Select **Enable Decision Service Rule Logging**. This option is applicable only to customers using the cloud version of Decision Modeler. When the check box is selected, rule events during service invocation are logged to the Analytic Data Mart. Note that this option should not be used in a high-load production environment.

8 Click **Done** and **Close**.

Decision Modeler deploys it to the Development environment. To begin authoring rules, click the project name to open the Project editor.

Related Links

[Creating New Projects](#)

Configuring a Project Using Default Settings

Configure a project using the default settings. In future, you can return to the Configure Project dialog on the Projects page to update your project .

1 Select a project row and click the **Configure Projects** icon on the Projects page.

2 Click **Next** two times.

3 Retain the default entry points.

The entry points are used in the deployed decision service to provide data for evaluation.

4 (Optional) Select **Enable Decision Service Rule Logging**. This option is available only to customers using the cloud version of Decision Modeler. When the check box is selected, rule events during service invocation are logged to the Analytic Data Mart. Note that this option should not be used in a high-load production environment.

5 Click **Done** and **Close**.

Decision Modeler generates a decision service and deploys it to the Development environment. To begin authoring rules, click the project name to open the Project editor.

Related Links

[Creating New Projects](#)

Configuring a Project With Entry Points

When you configure a project you created in Decision Modeler using an XML schema and select an input and output type, depending on your requirements, you may select different entry points than those provided by default. For example, you may want to configure your project with specific entry points to maximize your web service throughput for batch/boxcar processing.

In the Development environment as part of the configuration process for each of the projects you create, Decision Modeler generates two default entry points that can be used to invoke a decision service. Your client implementation or Java client determines which of these entry points are used.

The entry points are generated based on the XML schema and the input and output type you selected for the project configuration. By default, there is one entry point

for a single decision service invocation and one entry point for batch processing decision service invocations. For example:

- `processWith<NameOfTheInputOutputType>()` is the entry point that accepts and returns a single document of the selected input and output type with each invocation of the web service.
- `processBatchWith<NameOfTheInputOutputType>()` is the entry point that accepts a list of documents of the selected input and output type and returns a list of documents of the same type.

This boxcar (batch) entry point allows a client to process multiple documents with a single invocation of the web service. Using this entry point can improve throughput per second (TPS) by an order of magnitude or more compared to processing using multiple calls to the single document entry point when the list contains a large number of input documents (for example, 1000 input documents).

If you configure a project with default settings and you select `string` as the Input/Output type, the entry points that are generated for you by default are:

- `processWithDecisionFlow()` for the entry point that accepts and returns a single document.
- `processBatchWithDecisionFlow()` for the entry point that accepts a list of documents.

If you have imported projects in your workspace and you want to view the entry points of a project, you need to launch the **Configure project** wizard and edit the entry points page.

Related Links

- [Deploying Decision Services](#)
- [Decision Testing and Analysis](#)
- [Creating New Projects](#)

Viewing and Managing Project Settings

The Project Settings dialog provides last modified information about the project for a given environment. You can also use this dialog to modify the project name and description when the project is in the Development environment.

Related Links

- [Managing Workspaces and Projects](#)

Viewing Project Links and the Client ID

The **Project links** command on the **Projects** toolbar allows all roles except Deployers to view the web service URLs and the client ID and secret for requesting bearer tokens for a given deployment environment.

- **Links**

In the **Links** tab, each project exposes decision services as SOAP (for XML BOMs) web service endpoints and REST (for Java BOMs) web service

endpoints. You can view the URL links for each project in your workspace by selecting a project from the **Project** drop-down list. As a convenience for the REST web services, a `swagger.json` is listed directly after the REST endpoints.

The URL links for a project are different in the Development, Staging and Production environments. The URL links for a project in a specific environment are not available until the project has been deployed to that environment at least once. For example, the endpoints are available in the Development environment as soon as you import a project or create a new project and configure it. The endpoints for the Staging environment become available as soon as the project is submitted to Staging. The same holds true for the URL links in the Production environment.

- **Client ID**

The client ID and secret are used to request a token when deploying a project as a decision service. The client ID and secret are available as soon as you create a workspace.

Related Links

- [SOAP and REST Endpoints in Decision Services](#)
- [WSDL No Longer Works for Decision Modeler](#)
- [Obtaining the Client ID and Secret](#)
- [Managing Workspaces and Projects](#)

Testing Links

You can test URL endpoints at any time using a supported web browser.

The URL links for a project are different in the Development, Staging and Production environments. To locate the URL links for your project, select the project and open the **Project Links and Client ID** dialog.

If a project is configured with an XML object model, a SOAP endpoint is available. You can copy the SOAP URL and paste it into a URI field of a browser to view the contents of the WSDL.

If a project is configured with a Java Object Model, both a SOAP endpoint and one or more REST endpoints are available. To view the contents of the REST endpoint, use the `swagger.json` listed directly after the REST endpoints. You can also copy and paste the JSON into a Swagger online editor to view it in a more readable JSON format

Related Links

- [SOAP and REST Endpoints in Decision Services](#)
- [WSDL No Longer Works for Decision Modeler](#)

Importing Projects

You can import a ZIP file containing one or more projects to a workspace. The ZIP file may contain an exported Blaze Advisor workspace or a workspace downloaded from Decision Modeler.



Note

- The **Close** button is available on each wizard page. If you click **Close** before you upload a ZIP file, you can exit the wizard without importing projects.
- For best results, FICO recommends that the ZIP file you use for import contains at least one compilable project containing a Rule Service Definition (RSD) with a valid entry point.

- 1 Click **New Projects > Import Projects** on the Projects toolbar.
- 2 Drag and drop a ZIP file or use the **local file browse** link to select the ZIP file from your local machine and click **Open** and click **Next**.
- 3 If you imported a ZIP file with more than one project, you can select up to five projects you want to use in the Decision Modeler workspace and click **Next**. Only projects that contain a RSD file are listed in the wizard. If there are any projects that should be listed, you need to return to Blaze Advisor and add an RSD file to those projects and re-export the ZIP file. In the wizard, you can:
 - Select the check box next to the project name for any projects you want to use in your workspace.
 - Clear the check box next to the project name for any projects you do not want to use in your workspace.
- 4 Review entry points for each project that you selected and click **Import**. The entry points are used in the deployed decision service to provide data for evaluation.

Related Links

[Decision Modeler Workspaces and Projects](#)

[Managing Workspaces and Projects](#)

[Creating New Workspaces](#)

[Viewing and Managing Workspace Settings](#)

[Managing Object Models in a Project Configuration](#)

[Modifying a Project Configuration](#)

Importing Projects into a Workspace with Existing Projects

If you import a ZIP file with one or more projects into a workspace with existing projects, the project(s) in the ZIP file will overwrite the existing contents in workspace.

 **Note** If you import a ZIP file and for some reason the import fails, any existing contents in the workspace are removed.

 **Important** FICO recommends that you use the **Download workspace** icon on the Projects page to save a copy of any workspace.

The lifecycle status of existing projects is replaced when you import another set of projects into the workspace.

If you have a workspace with existing projects in various lifecycle environments and you import another set of projects into this workspace, any lifecycle information for the existing projects including pending removal, pending deletion, submitted, approved, or deployed is removed and replaced with the lifecycle information for the newly imported projects.

Previously submitted projects will continue to be functional in the Staging or Production environments and their project links will be available in the Staging and Production tabs in the Project Links and Client ID dialog.

Modifying a Project Configuration

Modify the configuration of the Decision Modeler project you have created or imported and redeploy it to the Development environment using the Configure Project wizard from the Projects page.

The Configure Project wizard can be used to do the following:

- Replace an XML schema.
- Select new input and output types.
If you have added XML schemas or Java classes to your project in Project Configuration wizard, you return to the Projects page and use the Configure Project wizard to select an input and output type and redeploy your project.
- Select new entry points.
If you select new entry points for your project, you must redeploy your project.

 **Note** If a project you created in Decision Modeler was originally configured with default settings, you can return to the Configure Project wizard pages to upload an XML schema.

The **Configure Project** wizard guides you through the process for modifying a project.

- 1 Select the row of a project you created or imported into a workspace and click the **Configure Project** icon on the Projects page.
 - To modify a project you created by uploading an XML schema:
 - 1 Click **Browse** next to the **Import XML Schema** field to select the schema and click **Open**.
 - 2 (Optional) Click **Preview** to review the content of the selected object model.
 - 3 Click **Next** and proceed to Step 2.
 - If you have added a new XML schema or Java class(es) using the Project Configuration wizard, click **Next** and proceed to Step 2.
- 2 Select an **Input and Output Type** from the drop-down list for your project and click **Next**. The input type defines the data that the project receives and the output type defines the returned modified data.
- 3 Ensure that the entry points displayed are the ones you want to use or select new entry points.

The entry points are used when the project is deployed as a decision service to provide data for evaluation.

 **Note** If you need to select an entry point, avoid selecting an entry point with an interface because an entry point with this type of signature cannot be used to invoke a decision service.

- 4 (Optional) Select **Enable Decision Service Rule Logging**. When selected, rule events during service invocation are logged to the Analytic Data Mart. Note that this option should not be used in a high-load production environment.
- 5 Click **Done and Close**.

Decision Modeler redeploys the project to Development environment. To begin authoring rules, click the project name to open the Project editor.

Related Links

[Decision Modeler Workspaces and Projects](#)

[Importing Projects](#)

Managing Object Models in a Project Configuration

Depending on the configuration of the projects in your workspace, there may be more than one way to add, remove, or replace the object model.

Use the **Project Configuration** icon on Global Command Bar in the Project editor if you have any of the following cases:

- An XML object model that you want to update or replace.
- Any additional XML schemas that you want to add.
- A Java object model that you want to update or replace.
- Any JAR files you want to add, remove, or replace.
- A default configuration where you want to add either an XML or Java object model.

To update or replace an XML schema for projects created in Decision Modeler, you can use the **Configure Project** dialog that is opened from the Projects page to update or replace an XML schema. However, if you have imported additional schema files in the **Project Configuration** dialog or you want to use a schema file from one of the other projects created and configured in the Projects pages, you need to return to the Projects page to complete the changes to your configuration.

For projects imported into Decision Modeler, each project you include in your workspace is automatically configured using the object model imported with the project. However, if you import additional XSD or JAR files in the **Project Configuration** dialog and you want to use a different schema or class for the configuration of an imported project, you need to return to the Projects page to complete the configuration.

Related Links

- [Decision Modeler Workspaces and Projects](#)
- [Importing Projects](#)

Business Object Models in the Workspace

Multiple projects in a workspace are supported in Decision Modeler 3.0 or later. Any XML schema or Java class included in or added to a project is treated as a global Business Object Model (BOM) and is visible to all projects in the workspace.

Each project can be configured with its own XML schema or Java class or share a BOM with other projects in the workspace. If a project is removed or deleted, the XML or Java BOM used to configure it remains available in the workspace.

If the same XML schema or Java class is inadvertently uploaded twice into the same workspace, when you compile a project you may see a compilation error. To resolve this issue, open the Project Configuration dialog from the Project Editor page to remove the BOM and refresh the project.

Managing the XML Object Model

Manage the XML object model by using the **Project Configuration** dialog in the Project editor to add, remove, or replace an XML schema.

 **Note** If a user deletes a project created and configured with an XML schema, the XML object model is still globally available in the workspace and all of its class appear in the Input/Output drop-down list. To avoid confusion after you create a new project in the same workspace, FICO recommends that you open the project in Project editor and remove all the unnecessary XML object models from the Project Configuration wizard prior to configuring the project.

- 1 In the Project editor, click the **Project Configuration** icon on the Global Command Bar and select **Manage XML object model**.
 - 2 To remove a schema, select the object model and click **Remove** and then click **Done**.
 - 3 To add a schema, click **Add**.
 - a Select **Add object model from .xsd file** and click **Choose File**.
 - b Select the XSD file, click **Open**, and then click **Include**.
 - c (Optional) Click **View** to see the details of the object model.
 - d Click **Close**.
 - e Return to the Projects page.
 - 4 If you have replaced any existing schema files in your XML object model, in the project you created or imported in Decision Modeler, you must redeploy the project to the Development environment to use the object model changes in your project.
-  **Note** You must ensure your decision entities are updated with the object model changes.
- 5 Return to the Projects page and select the project in the table and select **Configure Project** from the Action menu or select the command from the toolbar.
 - a Review the input and output type to ensure that the types are correct or select new input and output types, and click **Next**.
 - b Review the entry points, click **Done** and click **Close**.

Managing the Java Object Model

Manage the Java object model using the **Project Configuration** wizard from the Project editor to add, remove, or replace the Java classes in your project.

- 1 In the Project editor page, click the **Project Configuration** icon on the Global Command Bar and select **Manage Java object model** to view the classes in your project.

- 2 To add a class, click **Import Java Classes**.
The classes are added to the default Java Business Object Model (BOM) in your project.
 - a On the **Imported Java Classes** page, expand the JAR file and select the check box next to the name of the class you want to add.
 - b Click **Close**.
 - c If you do not see the classes you want to add, you may need to upload a JAR file to make those classes available to your project. Select **Manage JAR Files**.
- 3 To delete a class, select the check box next to the class name and click the **Delete** icon and click **Close**.
FICO recommends the following best practices when deleting classes:
 - If one or more of your decision entities reference the class(es) you plan on deleting, edit your decision entities to remove these references prior to deleting any class(es) to avoid compilation errors.
 - Whenever possible, delete classes from the Java object model prior to removing the related JAR file. If you remove the JAR file prior to deleting its classes, you see a "failed to load" error. If you intended to remove all the classes related to the JAR file, click **Remove Classes**. However if you removed the JAR file in error, click **Leave Unchanged** and return to **Manage JAR File** to reinstate the deleted JAR file.
- 4 If you have replaced any existing Java classes in your Java object model, in the project you created or imported in Decision Modeler, you must redeploy the project to the Development environment or use the object model changes in your project.
 - a Return to the Projects page and select the project in the table and select **Configure project** from the Actions menu or select the command from the toolbar.
 - b Review the input and output type to ensure that the types are correct or select new input and output types, and click **Next**.
 - c Review the entry points, click **Done** and click **Close**.

Related Links

[Uploaded Blaze Advisor Project Fails Deployment](#)

[Troubleshooting JAR File Import Errors](#)

[Error Displays When Importing a Business Term Set as a Java BOM](#)

Managing JAR Files

Manage the Java classes available to your projects using the **Project Configuration** wizard from the Project editor to add, remove, or replace JAR files.



Note All imported Java classes in a JAR file must be in a Java package.

- 1 In the Project editor page, click the **Project Configuration** icon on the Global Command Bar and select **Manage JAR files**.

- 2 To add or replace one or more .jar files, you can either drag and drop files or click **Browse** to select files from your local machine.
The JAR files are added to the project classpath.
- 3 To delete a JAR file, select the check box next to the .jar file name and click the **Delete** icon.
If your Java Business Object Model (BOM) contains references to classes in the JAR file that you want to remove, return to **Manage Java object model** and remove the classes prior to deleting the JAR file; otherwise, you may encounter compilation errors.
- 4 To select classes from an imported JAR file, return to **Manage Java object model**.

Related Links

- [Managing the Java Object Model](#)
[Avoiding Security Manager Exceptions with ObjectMapper](#)
[Troubleshooting JAR File Import Errors](#)
[Uploaded Blaze Advisor Project Fails Deployment](#)
[Error Displays When Importing a Business Term Set as a Java BOM](#)

Troubleshooting JAR File Import Errors

Under certain circumstances when you import JAR files, you may see one or more error messages. The error messages are displayed with the fully qualified name of the class where the error occurred.

Table 8: JAR File Import Errors

Error Message	Problem	Solution
Initialization error. May be related to an unavailable class.	When a definition for a class is not available at runtime, it usually means that the class being added to the Java Business Object Model (BOM) is dependent on a class that is not available on the project classpath. Most commonly, this happens when a class references another class that is not in any of the current JAR files. It can also happen when a class references a static member of a class that is not in any of the current JAR files.	Ensure that all JAR files containing any classes and dependencies are uploaded to the project.
Name of imported class conflicts with an existing class.	When a class is added to the Java Business Object Model (BOM), its SRL reference name is derived from the name of the Java class without the package. For example, com.example.package.Variable is referred to locally as Variable. Therefore, you cannot import two classes with the same class name, even if they are in different packages. If you have already imported com.example.package.Variable, you cannot import com.example.differentpackage.Variable.	If you need to import both classes and you have access to the source code, you need to change the name of one of the classes so that both classes can be imported.

Table 8: JAR File Import Errors (continued)

Error Message	Problem	Solution
		Note Decision Modeler does not support changing the name of a local class. Any changes to a class name must be done prior to uploading the JAR file.
Name of implicitly imported class conflicts with an existing class.	When a class is imported to the Java Business Object Model (BOM), any classes that are dependent on that class are not currently in the BOM are also imported. This is called implicit importing. Implicitly imported classes are used internally but are not directly reference-able through the BOM. When this error occurs, it is because the name of an implicitly imported class used by the current class has the same local name as a class that has already imported, either explicitly or implicitly.	If you need both classes, and have access to the source code, you need to change the name of one of the classes. Note Decision Modeler does not support changing the name of a local class. Any changes to a class name must be done prior to uploading the JAR file.

Using Enumeration Display Labels in the Decision Logic

By default, if an enumeration in your Java object model contains display labels, the display labels are displayed when selecting condition or action values in some decision entities.

Display labels are available when authoring decision logic in these decision entities:

- Decision tables
- Scorecards
- Rulesets when using the Rule Builder
- Functions when using the Rule Builder
- Decision trees when authoring action expressions
- Decision flow when editing branches in a split

The display labels are only for display in the decision entity editors and are not used in the underlying decision logic. In this example, the full name of the first enumeration item is `AccountType.premiumPlus` and the display label is `Gold`. `Gold` represents `AccountType.premiumPlus` or `premiumPlus`.

```
public enum AccountType {
    premiumPlus("Gold"), premium("Silver"), value("Bronze");
}
```

When passing in values to test your decision service with decision testing or batch, you must pass in the full enumeration item name such as AccountType.premiumPlus or the enumeration item name such as premiumPlus. Using enumeration display labels as values in a dataset will result in errors.

If you are authoring decision logic using SRL, you cannot use display labels. Use the full enumeration item name or the just the enumeration item name.



Note If you are uncertain as to the full name of an enumeration item, use the enumeration property in a condition or action in a decision entity and generate a report. Select the **Include Rule SRL** option in the **Generate Report** window to ensure that the underlying decision logic for your decision entities is available in the report.

Related Links

[Generating Reports](#)

CHAPTER 4

Authoring Rules in a Project

The decision logic encapsulated in your business processes represent the facts, policies, and procedures in your day-to-day operations and therefore your competitive advantage.

In Decision Modeler, the decision logic can be written in entities such as rulesets, decision tables, decision trees, and scorecards. Each Decision Modeler project can include multiple entities and folders. When you open a project from the Projects page, you are ready to begin authoring or editing rules.

The Project Editor

When you open a project, you see the Project editor where you can create, author, and test your decision logic.

This is how the Project editor appears:

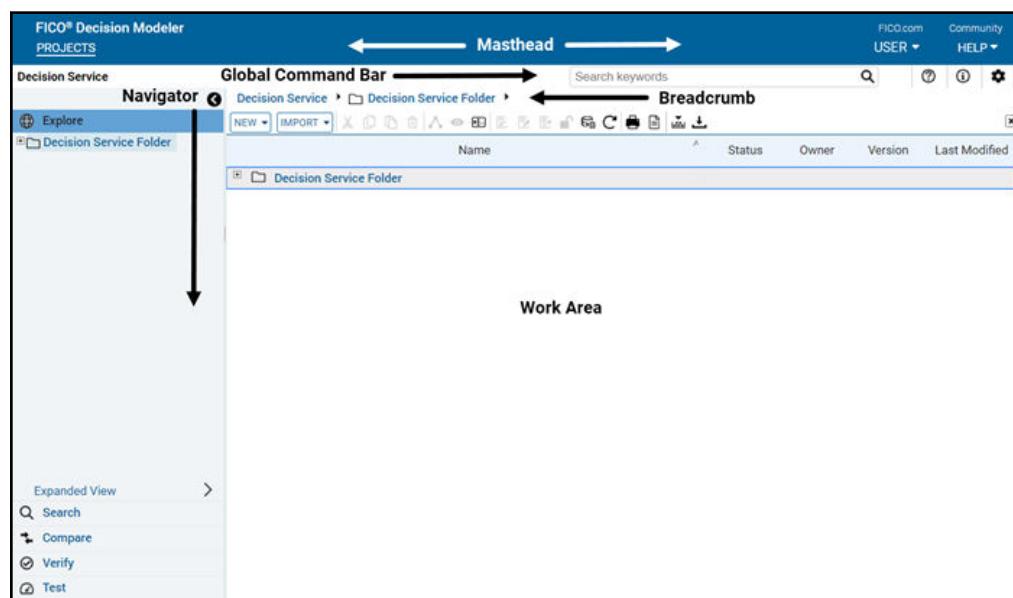


Figure 2: The Project editor with callouts for important features

Table 9: The Global Command Bar

Icon Name	Description
<Project Name>	If you create and configure a project using Decision Modeler 2.2 or later, you see the project name on the Global Command Bar in the Project editor. The project name is not displayed if you open a project provisioned with a Decision Modeler version that is 2.0 or earlier.
Quick Search	Allows you to quickly locate one or more decision entities that match a simple text query.
Help	Opens the Decision Modeler Help system.
Session info	Opens a window where you can see information about the repository and workspace for the project.
Project Configuration	Opens a wizard where you can manage an XML or Java object model. Note that you cannot change the input and output types, entry points, or redeploy your project in this wizard. Return to the Configure Project wizard on the Projects page for this purpose.

Table 10: The Navigator Pane

Name	Description
Explore pane	Allows you to traverse the hierarchy of folders and open an entity for viewing or editing.
Expanded View	Opens a tabular view of the folders and decision entities in the Project Explorer in the Work Area. This is where you can perform common operations on the decision entities and folders, such as renaming, previewing or copying entities.
Search pane	Allows you to open and run an existing query in the Query editor or create a new one.
Compare pane	Allows you to open and run an existing comparison query in the Comparison Query Editor or create a new one.
Verify pane	Allows you to open and run an existing verification query in the Verifier or create a new one.
Test pane	Upload a dataset to Decision Testing to test whether or not the rules you authored are generating the expected results. Also, if the decision service was configured using an exported Blaze Advisor project and it included brUnit test cases, you can run them here using brUnit.

Table 11: The Work Area

Name	Description
Breadcrumb	Allows you to quickly traverse the path to a decision entity. The Breadcrumb displays the path to the entity that is currently selected in the Work Area. You can select a decision entity name or folder name in the path. You can also use the Show navigation menu icon that displays after each item in the path to open other decision entities and folders in the hierarchy.
Work Area	Allows you to perform edit operations. When a decision entity is opened for editing, the appropriate editor appears in the Work Area.

Table 11: The Work Area (continued)

Name	Description
	For example, a decision tree opens in the Decision Tree editor and a Standard Business Query opens in the Query editor. Open the Project Explorer view in the Work Area by clicking on Expanded View . The Project Explorer is where you can create new entities, and name, rename, or delete entities and folders.

Navigate back to the Projects page or exit Decision Modeler by using the links on the masthead. These links are also available from the Projects page.

Table 12: Masthead

Name	Description
Projects link	Click the Projects link to return to the Projects page. If you have unsaved edits, you are prompted to save them prior to leaving the Project editor.
Username link	Click on the Username link to select the Exit command from the drop-down menu. When you exit Decision Modeler, you can also exit your FICO Analytic Cloud account at the same time.

Routine Operations

Use the Project Explorer to perform routine operations such as previewing an entity, renaming an entity, or creating subfolders. The Project Explorer is a tabular view of the entities and folders displayed in the **Explore** pane.

To open the Project Explorer, expand the **Explore** pane, and click the **Expanded View** command. To perform a routine operation, select an entity or a folder and then select a command from the Project Explorer toolbar or the Context toolbar. Only those commands that are allowed for that entity or folder are displayed. To access the Context toolbar, select a row containing a specific entity or folder and then click the blue orb at the end of the row.

The only commands applicable to folders are Rename and Delete. The Delete command is available only when a folder is empty.

 **Note** The list of folders appears in alphabetical order. To reverse the order, click the triangle in the **Name** column header.

Creating New Decision Entities and Folders

In the Project Explorer, create a new decision entity or folder using the **New** drop-down menu.

Before opening the **New** drop-down menu, select the folder where you want the decision entity or folder stored. The new item is added in the selected folder.

Custom Entity

If you uploaded an exported Blaze Advisor project to configure your decision service and it included templates, use the **Custom Entity** command to create new template instances.

Naming Decision Entities and Folders

There are two different types of names for decision entities: the display name and the reference name. Folders only have a display name and the length of the name cannot exceed 256 characters.

When you create a decision entity, a screen appears requesting a name. Your entry in the **Name** field initially represents the following:

- The display name of the decision entity that you see in the tree view and in the expanded view of the Project Explorer.
- The reference name when you want to call the decision table from another decision entity such as a decision flow or a ruleset.

Because a reference name is used to call a decision entity, there are naming conventions that must be followed. A valid reference name is a unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %. If you enter a special character, it becomes an underscore in the reference name. However, there are no such restrictions for the name of the decision entity in the Project Explorer. So if you enter a name such as Account 9*# when you first create the decision entity, the space and special characters show in the decision entity name in the Project Explorer, but are displayed as Account____ in the **Reference name** field in the editor for the decision entity.

At any time, you can change the reference name in the editor for a decision entity or change the display name using the **Rename** command in the Project Explorer. A change to the display name updates the reference name and vice versa.

If you change the reference name of a decision entity and it is called by another decision entity, you must update the expression that calls the decision entity; otherwise, an error is thrown. For example, if you change the reference name and the decision entity is referenced in a decision flow task, an error is thrown the next time you open the decision flow. To fix the issue, open the task and select the new reference name.

Related Links

[Naming Requirements and Conventions](#)

Importing Models

In the Project Explorer, import an FSML, PMML, or SAS model using the **Import** drop-down menu.

Before opening the **Import** drop-down menu, select the folder where you want the decision entity containing the model stored. The new decision entity or entities are stored in the folder.

Renaming Decision Entities and Folders

Rename decision entities and folders in the Project Explorer.

Decision entities must be checked out in order to rename them; however, this is not required for folders.

- 1 In the **Explore** pane, click the **Expanded view** icon.
- 2 Select the decision entity or folder that you want to rename.
- 3 If the item to be renamed is a decision entity, click the **Check Out** icon.
- 4 Perform one of the following to enable the name field:
 - Click the **Rename** icon on the Project Explorer toolbar or on the context toolbar.
 - Press the **F2** key.
- 5 Enter a new name for the decision entity or folder.
- 6 Press the **Esc** key to cancel the change or press the **Enter** key to register the change.
- 7 If the item that was renamed was a decision entity, click the **Check In** icon.

Generating Reports

Generate a report to view the relationships between decision entities. After the HTML report is generated, download it and open it from your file management system.

 **Note** Decision tree information is available in the reports, but a picture of the decision tree is not.

- 1 From the **Explore** pane, click the **Expanded view** icon.
- 2 Click the **Generate Report** icon on the Project Explorer toolbar.
- 3 In the Generate Reports dialog, do the following:
 - a In **File Name Prefix**, enter a name for the report.
 - b (Optional) Select the **Include Rule SRL** check box if you want to see the rule SRL when you click on a rule in the report.
 - c (Optional) Select the **Include Cross Reference** check box if you want the report to analyze the rules.

- d Click **OK**.
- 4 When the **File download** window appears, click **Save**.
If any errors occur while generating the report, the error details are stored in the text file within the ZIP file.
- 5 Click **Close**.
- 6 Locate the ZIP file in your file management system and unzip the file.
- 7 Click on the <File Name Prefix>Contents.html file to open the report.

Related Links

[Using Enumeration Display Labels in the Decision Logic](#)

Compiling Decision Services

Compile your decision service to ensure that it does not generate errors. The Compile icon is available on the Project Explorer toolbar and on the editor toolbars for all decision entities, business term sets, and reason code lists. If compilation errors occur, the errors are reported at the bottom of the page.

-  **Note** If you are using Microsoft Internet Explorer, you can use the **Copy errors to clipboard** icon directly above the error to copy the error message to the browser clipboard. If you click the Copy error to clipboard icon, a message appears asking you to allow the web page to access your clipboard. Click **Allow access** to paste the error message to a text file or email message. This feature is not available in other supported browsers.

Quickly Finding Decision Entities

Use the search field on the Global Command Bar to quickly locate one or more decision entities that match a simple text query. When a string is entered, the search returns any decision entity that contains the string in its name or in the content. After the search returns the results, the names of any decision entities that match the criteria are displayed in a drop-down list. Selecting a decision entity from the list of results opens the decision entity. (To search for an entity based on multiple criterion such as the author name, creation date, or display text use a Standard Business Query in the Search pane.)

In the search field in the Global Command Bar, you can search using keywords, wildcard matching, fuzzy matching, boolean queries, phrase queries or combinations of these search types. All search criteria are case-insensitive.

Table 13: Keyword Search

To locate this...	Enter this text in the search field	Comments
A decision entity by its display name (as shown in the Project Explorer) or its reference name (as shown in a decision entity editor). You can also search based on a string in the content of a decision entity editor. For example, a parameter name in a function, a reason code in a reason code list, SRL in a rule, a property name, or a property value.	Enter text that appears in the name or the body of a decision entity.	Quotation marks for entries with spaces are not required. When searching by name, the list of decision entities will also include references to the decision entity containing the search string. For example, if you enter Evaluate Initial Criteria to locate a ruleset, not only do you see the name of the ruleset in the drop-down list, but also the names of any entities that reference the ruleset, such as a decision flow.
A specific decision entity by display name. The display name is the name of a decision entity as shown in the Project Explorer.	Enter the following: name:<display name>"	Using quotation marks returns only those decision entities with an exact match.
All decision entities in a folder.	Enter the following: path:<folderName>	The search returns all decision entities in the folder, including any decision entities stored separately. The search returns No Results if the folder is empty. Quotation marks for folders with spaces in the name are not required.
A decision entity based on its type. See Quick Search Type Reference on page 59 for a complete list of types.	Enter the following: type:function	It returns all decision entities that match the specified type.

Table 13: Keyword Search (continued)

To locate this...	Enter this text in the search field	Comments
	Replace function with another decision entity type, such as ruleset.	

Table 14: Wildcard and Fuzzy Matching

To search using a partial search	Enter this text in the search field	Comments
To search using a sequence of characters, enter the characters followed by an asterisk *.	Enter the following: <characters>*	The * should not be used as the first character.
To search using one or two characters, enter the characters followed by a question mark ?.	Enter the following: <characters>?	The ? should not be used as the first character.
To match the nearest spelling matches, use the tilde ~.	Enter the following: <characters>~	The ~ should not be used as the first character.

You can also use Boolean queries with any combination of AND/OR. For example: (word1 or word2) or (word3 or word4).

To look for exact sequence of words, enclose the text in double quotes. For example: "word1 word2".

Combinations of search criteria can be used such as type:ruleset name:decision. By default, the space between the criteria indicates that these criteria are joined by or. You can use multiple criteria with or without using or. This is an example: age type:query name:"Assign Card".

If a search locates a match in a subinstance and that subinstance is stored separately, the match returns the name of the subinstance. If the subinstance is not stored separately, the name of the parent instance is returned. For example, if you create a ruleset using the **New** menu, the rules are not stored separately so the name of the parent instance is returned. However, business terms are stored separately so you will see the names of individual business terms returned as a search match.

Quick Search Type Reference

When you search for a decision entity by type using the search field on the Global Command Bar, all decision entities that match the specified type are returned. All search criteria are case-insensitive.

 **Note** The keyboard shortcut to move the focus to the search field is Ctrl + Shift + F.

Table 15: Keyword Search

To locate this decision entity type:	Enter this text in the search field:
business term set	type:"business term set"
comparison query	type:comparison Alternatively, you can use this criteria: type:"comparison query"
comparison query, standard business query, verification query	type:query
decision flow	type:flow Alternatively, you can use this criteria: type:"decision flow"
decision table	type:table Alternatively, you can use this criteria: type:"decision table"
decision tree	type:tree Alternatively, you can use this criteria: type:"decision tree"
decision tables, decision trees, and decision flows	type:decision
function	type:function
PMML mining model	type:"pmml mining model"
reason code list	type:"SRL named object"
ruleset	type:ruleset
SAS model	type:"sas program"
standard business query	type:search Alternatively, you can use this criteria: type:"search query"
scorecard	type:scorecard
verification query	type:verification Alternatively, you can use this criteria: type:"verification query"

Locating References to Decision Entities

Use the **Show usage** command to display a list of decision entities that reference a particular decision entity. For example, an action statement in a rule may call a decision entity, such as a decision table or function, to return a value or to perform some type of procedural operation. Knowing where a decision entity is referenced can help to minimize any errors that are introduced when modifying a decision entity or deleting it.

To see a list of the decision entities that reference a particular entity, do one of the following:

- In the Project Explorer, select the row containing the decision entity and click the **Show usage** command on the Project Explorer toolbar or the Floating toolbar.
- Open the editor for the decision entity and click the **Show usage** command on the editor toolbar.

A list of references is displayed in the **Usage** pane. If the name of an entity is a link, you can preview the decision entity or open it in an editor. If the name is not a link, this means that the entity is a fixed decision entity in the underlying project. This happens only if the project was created by uploading an exported Blaze Advisor project as a component.

A reference to a reason code list appears only in the **Usage** pane if a reason code is used in a variable in the scorecard. While a reason code list is associated with a scorecard, it is the individual reason codes that are referenced by the variables.

 **Note** In this release, references to decision entities in decision tree action statements do not appear in the Usage pane.

Sharing Files Using FICO Drive

FICO Drive is an Amazon S3 bucket that stores files that can be used in a decision service, such as a PMML file. The files stored in FICO Drive are accessible to all users who have access to the same solution on the Decision Management Platform.

 **Note** For the best editing experience when using FICO Drive, open your project with Google Chrome from the Decision Modeler Projects page.

Using FICO Drive, you can add new files, create folders, and select files to import into Decision Modeler or download files to your local machine.

FICO Drive is accessed from the **FICO Drive** link in one of the following import wizards in Decision Modeler:

- FSML Decision Tree
- PMML Scorecard Model
- PMML Mining Model
- Other PMML Models
- SAS Program

You can also select a dataset from FICO Drive to use with decision testing.

 **Note** All files in FICO Drive are displayed by default; however, you can filter the list by selecting a file type in the **Filter** drop-down menu.

Adding Files to FICO Drive

Add files to FICO Drive to make them available to all users using the same solution. For best results, do not upload more than 1 GB at a time.

For the best editing experience when using FICO Drive, open your project with the Decision Modeler Projects page in Google Chrome.

- 1 Click the **FICO Drive** link.
- 2 (Optional) Create a folder to store the file.
- 3 Select the root or a folder where you want to store the file.
- 4 Click the down arrow and select **Add File**.
- 5 Scroll down to the bottom of the dialog to locate the drop zone.
- 6 Drag a file onto the drop zone or click **Select a file from your computer**.

Project Explorer Toolbar Commands

The commands on the Project Explorer toolbar are used to perform routine operations such as creating, deleting, and previewing decision entities.

Table 16: Project Explorer Toolbar Commands

Command	Icon	Description
New		Creates a new decision entity.
Import		Allows the import of file that contains a model.
Cut		Cuts one or more decision entities.
Copy		Copies one or more decision entities.
Paste		Pastes one or more cut or copied decision entities. After a decision entity is cut and pasted, the name of the decision entity that was cut is displayed with a line through it until it is checked in. Decision entities in a folder must have unique names. If a decision entity is copied and pasted to a folder where a decision entity with the same name exists, the new decision entity is automatically appended with a number starting at 1.
Delete		Deletes one or more decision entities or folders. A folder can be deleted only if it is empty. You cannot delete the root folder.
Show Usage		Opens a pane that shows the list of decision entities that reference the selected decision entity.
Preview		Opens a window where the selected decision entity can be previewed.

Table 16: Project Explorer Toolbar Commands (continued)

Command	Icon	Description
Rename		Makes a decision entity or folder name editable. Press Enter to register the name change.
Check In		Checks in changes to one or more decision entities.
Check Out		Checks out one or more decision entities.
Cancel Check Out		Cancels the check out of one or more decision entities.
Release Lock		Releases the lock on an entity that is checked out by another user. This command is only available to Administrators or Owners. It is only enabled when an entity is currently checked out by another user.
Update		Overwrites decision entities and folders in your workspace with the latest versions from the repository.
Refresh		Refreshes your workspace with changes that were made by another team member using the decision service.
Print		Prints the list of decision entities in the Project Explorer.
Generate Report		Generates a report so you can view the relationships between the decision entities in the decision service.
Compile		Compiles the decision service and reports any compilation errors.

Related Links

[Naming Decision Entities and Folders](#)

Keyboard Shortcuts in the Project Explorer

Use keyboard shortcuts to perform common tasks in the Project Explorer.

Table 17: Project Explorer Shortcuts

Action	Shortcut
Edit or rename selected item	F2
Cancel an edit or rename operation	Esc
Select multiple non-adjacent rows	Ctrl-click
Select multiple adjacent rows	Shift-click
Move the focus to the search field on the Global Command Bar	Ctrl + Shift + F

Note You can select multiple items to delete, copy, cut, or paste. However, no other operations are allowed for multiple selections.

Version Management

Decision entities are stored in a versioned repository that can be accessed by other users. As a result, all versioning commands, such as check in and check out are explicit operations.

When you open Decision Modeler and configure a decision service, the workspace contains the latest versions of the entities and folders from the repository. However, you must periodically update your workspace to ensure that it contains the latest versions.

In Decision Modeler, you check in and check out decision entities to create each version. If you save changes to a decision entity but you do not check in the changes, those changes are not available to other users with access to the same decision service.

The entity version history is maintained. You can view previous versions and promote a previous version to replace the current working version.

Version Management Commands

Use version management commands to check in, check out, and promote previous versions of entities.

The version management commands appear on the following toolbars:

- Project Explorer toolbar
- Context toolbar in the Project Explorer
- Editor toolbars

Table 18: Versioning Command Descriptions

Command	Icon	Description
Check in		Checks in one or more entities.
Check out		Checks out one or more entities for editing.
Cancel Checkout		Cancels a check out of one or more entities.
Promote		Promotes an older version to replace the current working version.
Update		Updates the workspace with the latest versions of entities from the repository. This icon is located only on the Project Explorer toolbar.

When an entity is checked out, the **Edit Privilege** icon appears next to it in the **Explore** pane and the Project Explorer.

You will notice the following behavior when performing check in and check out operations:

- If an entity is opened first and then checked out, it is automatically enabled for editing.
- If the entity is checked out first in the Project Explorer and then opened, click the **Edit** icon to enable it for editing.
- If an entity is checked out and closed, when you open it, click the **Edit** icon to enable it for editing.

In the Project Explorer, you can perform versioning operations, such as check in or check out, on multiple entities by selecting the row for each entity and then clicking the appropriate versioning command on the Project Explorer toolbar. You cannot perform versioning operations at the folder level.

Version History Page

The History page contains information about the working copy or previous versions of an entity. It also allows you to view and compare those other versions.

The History page contains a working copy status table and a version history table.

Working Copy Status				
View	Base Version	Owner	Last Modified	Status
	4	Not checked-out	4/20/20 9:45:31 AM	
Version History Compare				
Compare	View	Version	Author	Check-in Date
<input type="checkbox"/>		4	User1	4/20/20 9:36:41 AM
<input type="checkbox"/>		3	User1	4/20/20 9:05:29 AM
<input type="checkbox"/>		2	User1	4/20/20 8:39:49 AM
<input type="checkbox"/>		1	dmpadmin	4/20/20 8:34:38 AM
Comments				
				Deleted rule 3.
				Updated rule 1.
				Changed ruleflow.

Figure 3: History Page

Working Copy Status This table shows the status of the base version in the workspace. Click the **History** icon in the **View** column to view the contents of the working copy.

Version History This table displays information for all versions. Click the **View** icon next to an item in the Version History Table to open it and view the contents. To compare two different versions, select the check box for each version in the **Compare** column, and then click the **Compare** link. Note: You cannot compare two versions of a decision tree in this release.

There are two ways to open the History page:

- Open a decision entity, select **History** from the **Version** drop-down menu.
- Open the Project Explorer, select the row containing the decision entity, and click the **History** icon in the **Version** column.

Promoting Previous Versions

To use an older version as the latest version, promote the older version to be the base working copy. Promoting a previous version overwrites the current working copy with the contents of the older version.

If a previous version that references other entities is promoted, you may need to promote the previous versions of those entities created in the same time period to avoid any compatibility issues.

- 1 Open the entity for which you want to promote a previous version.
- 2 Click the **Check Out** icon on the editor toolbar.
- 3 From the **Version** menu in the upper right section of the page, select the number of the previous version to promote.
- 4 Click the **Promote** icon.
- 5 Click **OK**.
- 6 Click the **Check In** icon to check in the promotion.

The promoted entity is now the latest version, and the version has been updated by 1. The Version History and Working Copy Status tables do not indicate which version has been promoted, so it is recommended to note the promoted version number for your records.

After the promoted entity is checked in, the former working copy is preserved as a revision.

Releasing Locks

An Administrator or an Owner can unlock an entity checked out by another user in a project.

When a lock is released, the entity is reverted to its previous checked in state. This means that all the changes made by the user after the entity was checked out are discarded, and the entity can be checked out by other users.



Note When a user checks out and deletes an entity without checking in the change, the Administrator or Owner can release the lock. However, the entity is still checked out to the user with no write access and they must cancel their check out. As a result, the entity is not deleted.

Complete the following steps to release a lock on an entity:

- 1 Select the entity that you wish to unlock in the Project Explorer or open the entity.
- 2 Click the **Release Lock** icon on the toolbar.

The entity is reverted to its previous checked in state. Any changes that were made to the entity are reverted and the entity can be checked out by other users.

CHAPTER 5

Using Business Term Sets to Represent Classes

A business term set is a global class variable used to represent a class in your project. The individual business terms in a business term set are primitive-type properties that can be used in your decision logic in the same way that you use object properties from an external class. For example, you could use business terms in your decision logic to evaluate conditions or to assign a value to a property. Any values stored in business terms that you want used in a deployable decision service must be mapped to properties in your external object model.

You can map business terms to properties in your external object model using the Business Term editor or using a decision entity such as a function or ruleset. Just because a business term is used in the decision logic does not mean that it must be mapped to an external object property. For example, if business terms are used in a calculation, you may only want to map the business term storing the result of the calculation to a property in your external object model.

Another use of business term sets is to prototype or extend an existing object model. You can experiment with changes to the object model using business term sets and then make your final changes in the external object model and reload it using the appropriate wizard. You can then delete the business term sets if they are no longer necessary.

Business term sets are also used to represent an object model for a PMML or FSML model when the model is imported into the project. For example, when a PMML scorecard is imported into the project, the model is rendered as a scorecard and the variables (also known as characteristics) are represented by business terms in a business term set. You can also use those business terms to author additional decision logic.

When you create a business term set or one is created when a model is imported, a global variable representing the business term set is available when authoring decision logic. This variable allows you to use business terms when authoring decision logic with the Rule Builder or with FICO's Structured Rule Language (SRL). In the Rule Builder, the variable and its properties are available in the Item Selector, where you can author decision logic using drag and drop functionality. The variable is also available in SRL when authoring in the advanced builder editor. To reference the variable in the SRL, use the variable name. The variable name is always the name of the business term set with var prepended to the name. For example, if the business term set is LoanApplicant, the variable will be varLoanApplicant.

To ensure that your project compiles, you must initialize the global variable in a decision entity using FICO's Structured Rule Language (SRL). Alternatively, if you

configured the project with an XML schema or an imported Java class, you can associate the business term set with the input variable (for the decision service) using the Initialization entity.

Related Links

[Preparing a Project for Testing and Deployment](#)

[Verifying the Business Object Model in your Decision Service](#)

Creating Business Term Sets

Use the **New** menu on the Project Explorer toolbar to create a business term set.



Note Only terms of primitive types such as boolean, date, duration, integer, money, real, string, time, and timestamp are supported.

- 1 In the **Project Explorer**, select the folder where you want the business term set created.
- 2 Click the **New** drop-down menu
- 3 In the dialog, select **Business Term Set**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 Click **Create**.
- 6 (Optional) To extend an existing object model, use the **Input class** menu to select an existing class.
- 7 In the Business Terms table, click the **Add** icon to add a business term. Click on the term to define it.
- 8 Use the **Parent View** icon on the toolbar to return to the Business Term Set page where you can add more business terms or add value lists.
- 9 Check in your changes.

Related Links

[Naming Decision Entities and Folders](#)

Writing Term Calculations

To assign a value to your business term, create a business term calculation. Use either the Rule Builder or the Advanced Builder to write assignment statements.

Create a term calculation to do one or more of the following operations:

- Map a term to a property in your input data model. For example:
`Guarantor.age=input.PersonInfo.age;`
- Calculate a term's value based on other terms. You can use a control construct that eventually assigns a value for the term. For example, suppose

you have a `monthlySalary` term, you could write the following control construct to assign a value to `monthlySalary`:

```
...
if (age > 30 and age < 40
then monthlySalary = 4000.00;
else if (age >= 40 and < 50) then monthlySalary = 5000.00;
...
```

You can use special value keywords when creating assignment statements to test for the presence of a value not specified in the payload, and the property is used in the decision logic. The keywords are `available`, `unavailable`, `known`, `unknown`, and `null`.

 **Note** In the Rule Builder, the maximum depth at which you can select a property is 5. A workaround is to create an intermediate variable that can be used to reach the desired property.

Related Links

[Authoring Decision Logic Using SRL](#)

[Reference Guide](#)

Creating Value Lists

A value list is a set of string values that you can associate with a business term of type string. When a business term of type string is used in a ruleset, decision table, or decision tree, you can select a value from the list.

Because a value list is created in a business term set, it cannot be shared across other business term sets in the project. A value list can only be associated with string type business terms in the same business term set.

Create a value list by adding a link to the Value Lists table. Use the link to go to the page where you can name the value list and define the values. For each value added to the list, enter a unique internal value and a unique display value. The display value appears in the drop-down list when a value is selected in a decision entity such as a ruleset, scorecard, decision table or decision tree. When authoring decision logic using the advanced builder editor in a decision entity, you can reference the internal value or the display value in the SRL.

Mapping the Business Terms to Object Model Properties

To use the values that are stored in business terms in a deployable project, the business terms must be mapped to properties in your external object model. You do not have to map all business terms in your project, just those business terms storing the values that you want used during deployment.

If the business term set extends an imported class, there is a property in the business term set called `termSetInput`. The property type is the class that was

extended. In this figure, there is a business term set called theWebsiteScoreGBDTIn. Since it extends the SportsFanPromotion class, there is a termSetInput property of type SportsFanPromotion. The property is available when authoring in SRL or when using the Rule Builder.

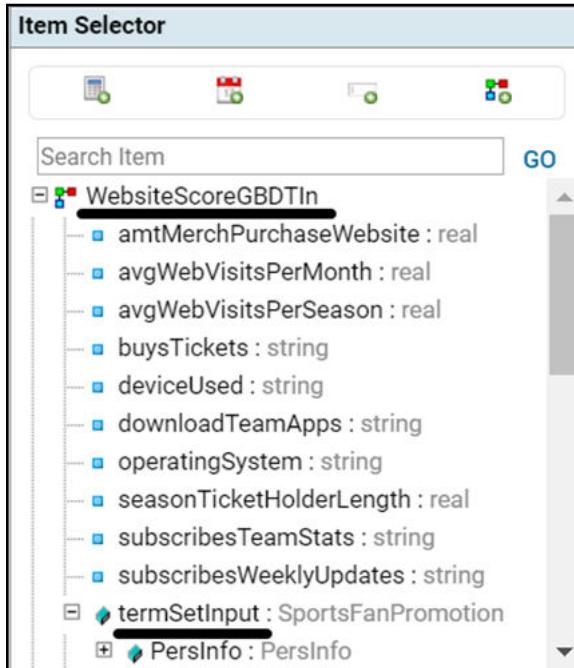


Figure 4: Business Term Set with termSetInput Property

You can map a series of individual business terms to properties in your object model using a decision entity such as a function or ruleset, as shown in this figure:

A screenshot of the Function Body editor. At the top, it says "Function Body" and has "COPY" and "Edit in Rule Builder | Advanced Builder" buttons. The main area contains two lines of code: "varWebsiteScoreGBDTIn.buysTickets = varWebsiteScoreGBDTIn.termSetInput.WebsiteUsage.buysTickets" and "varWebsiteScoreGBDTIn.deviceUsed = varWebsiteScoreGBDTIn.termSetInput.WebsiteUsage.deviceUsed". Both lines have the "termSetInput" part underlined in red.

Figure 5: Mapped Business Terms

Alternatively, you can open the Business Term editor for an individual business term and map a business term to a property in the Term Calculation field.

Related Links

[Importing Tree Ensemble Models with PMML Files](#)

[Importing Decision Trees from FSML](#)

[Example Decision Services with a Tree Ensemble Model](#)

[Preparing a Project for Testing and Deployment](#)

[Example Decision Services with an FSML Decision Tree Model](#)

Initializing Business Term Sets

Because a business term set is a global variable, it must be initialized in the decision logic. You can initialize the variable using the Rule Builder or the advanced editor in a decision entity such as a ruleset or function.

In a project configured with an XML schema or Java class, if the business terms are mapped in the Business Term editor, you can use the **Initialization entity**. The Initialization entity is not available when a project is configured by uploading a Blaze Advisor project.

BUSINESS TERM SETS

Hover over the ellipses icon to add the business term sets you want to initialize for this decision service.

The business term sets listed here will be initialized before each execution.

To create a term set, open the Explore expanded view and select the "New" icon in the toolbar.

*** Term Set: SocialMediaScore_ScorecardIn ▾ Input assignment: varSocialMediaScore_ScorecardIn termSetInput property is input ▾

Figure 6: Example of an Initialized Business Term Set

The business term set fields in the **Initialization entity** are only available after a business term set is created in the project.

CHAPTER 6

Authoring Logic in Rulesets

Author logic in a ruleset to maintain a set of rules that work together to achieve a particular business decision. For example, a ruleset could contain rules that work together to generate a premium for an automobile insurance quote.

Each rule in a ruleset contains the following:

- One or more conditions to test
- One or more actions to execute when one or more of the conditions are satisfied

This is an example of a rule:

```
if theCustomer.age is greater than 18 and theCustomer.drivingRecord is "excellent"  
then theCustomer.valueProposition is "high"
```

Rules are defined based on the business terms in the object model and any local variables you create in the ruleset.

Create rule conditions and actions by dragging and dropping object and variable properties in the **Rule Builder** interface. Alternatively, you can write condition or action expressions using FICO's Structured Rule Language in the **Advanced Builder**.

Creating Rulesets

Create a ruleset to author a set of rules that will be evaluated based on the data passed into the decision service.

- 1 In the **Project Explorer**, select the folder where you want the ruleset created.
- 2 Click the **New** drop-down menu.
- 3 Select **Ruleset**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 Click **Create**.

When you open or create a ruleset, the **Ruleset Definition** page appears. From the **Ruleset Definition** page, you can define the ruleset and its associated parameters, rules, and local variables.

Related Links

[Naming Decision Entities and Folders](#)

Defining Rulesets

Use the **Ruleset Definition** page to define the ruleset.

It contains the following fields and sections:

Reference name	A unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.
Description	(Optional) A description of the ruleset.
Return type	The type of value the ruleset returns. Set the return type to void if the ruleset does not return a value.
Parameters	(Optional) Parameters that you can use to define rules in the ruleset.
Variables	(Optional) Variables that you can use to define rules in the ruleset.
Patterns	(Optional) Patterns that you can use to define rules that evaluate each available object of a particular type, or each available object of a particular type in an array.
Initialization Rule	(Optional) Initialization statements for the rules in the ruleset.
Rules	The rules associated with the ruleset. You can create one or more rules in a ruleset. Up to five rules can be displayed in the rule table before a new page is created. You can use the Next Page and Previous Page arrows to navigate between pages.

Using Variables in a Ruleset

A variable is a placeholder for storing a value in a ruleset.

You can create variables of the following types:

Primitive Data Types A data type such as boolean, integer, real, or string.

Object Types Object types from the imported business object model.

After you create a variable, you can use the variable in initialization rules, condition expressions, or action statements. In an initialization rule, you can initialize the value of a property using a variable. In a condition, you can test the value of a variable. In an action, you can assign a value to a property based on the value of a variable.

Variable of Primitive Type Example

You can create a variable of a primitive type to perform a calculation based on the values in the object properties and then use the calculated value in a rule.

For example, suppose you have MortgageApplication as the input object type for the decision service. Then you want to create a variable named IncomeAfterPayments of type real. This variable subtracts the applicant's monthly debt from the applicant's monthly salary. totalMonthlySalary is a property of the PersInfo sub-object type and totalMonthlyDebtPayment is a property of the Applicant sub-object type.

You can use the following initialization for IncomeAfterPayments:

```
input.PersInfo.totalMonthlySalary - input.Applicant.totalMonthlyDebtPayment
```

 **Note** In this example, the project was configured by uploading an XML schema so input is a variable that is initialized with the input type of the decision service.

Then, you can use IncomeAfterPayments as a property in the condition expressions and action statements for rules in this ruleset.

Variable of Object Type Example

You can create a variable of an object type when you want to create rules with properties in a sub-object type.

For example, suppose LoanApplication is the input object type for the project. Applicant is an object type within the LoanApplication object type. You want to create rules with only the Applicant properties. So, you create a variable named NewApplicant. Then, you initialize the properties in NewApplicant so that they map to the Applicant properties.

You can click the **Switch to advanced builder** link to open the **Advanced Builder** and initialize the values that you plan to use in the rules. The initialization for NewApplicant could look like the following:

```
an Applicant initially
{firstName=input.Applicant.firstName,
 housingRatio=input.Applicant.housingRatio,
 lastName= input.Applicant.lastName,
 loanAmount= input.Applicant.loanAPR }
```

 **Note** In this example, the project was configured by uploading an XML schema so input is a variable that is initialized with the input type of the decision service.

Then, create rules with the initialized properties of NewApplicant. For example, the property input.Applicant.firstName can be referenced as NewApplicant.firstname.

Creating Local Variables

You can create one or more local variables that can be used by the rules in a ruleset.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check out** icon.

- If the ruleset is already checked out, click the **Edit** icon.
- 4 Click the arrow next to the **Local Variables, Patterns and Initialization Rule** section to expand it.
- 5 Click the **Add** icon on the context toolbar of the variables or patterns table. A default entity name is added as a link to the table. The entity name is appended with a number starting with 1.
- 6 Click the link in the **Name** column.
- 7 In the **Name** field, enter a name that is unique within the ruleset.
- 8 In the **Type** drop-down list, select the variable type.
- 9 In the **Initialization** section, enter the initial value for the variable.
- To initialize a variable of *primitive* data type use the Rule Builder:
 - 1 Click the **Open Editor** icon.
 - 2 Drag a property from the **Item Selector** to the default group or drag an empty field and provide a valid value.
 - 3 Click **Done**.
 - To initialize a variable of *object* type, you need to use the Advanced Builder.
 - 1 Click **Switch to Advanced Builder**.
 - 2 Use Structured Rule Language (SRL) to initialize the variable. For example:

```
an Applicant initially
  {firstName=input.Applicant.firstName,
   housingRatio=input.Applicant.housingRatio,
   lastName= input.Applicant.lastName,
   loanAmount= input.Applicant.loanAPR }
```
- 10 Click the **Save** icon or the **Check in** icon.
- 11 To return to the **Ruleset Definition** page, click the **Parent View** icon.

Related Links

[Rule Builder Interface](#)

[Authoring Decision Logic Using SRL](#)

Using Patterns in a Ruleset

Use a pattern to write rules that evaluate each available object of a particular type or each available object of a particular type in a specific array.

A pattern is a way to refer to multiple instances of a specific class or interface. Create a class-based pattern and refer to it in the rules to evaluate each object that matches the pattern. When a pattern is encountered in a rule, it is treated as a shorthand notation for defining a rule for every object that matches the pattern.

For example, suppose you have a class called Account, and have some number of account objects, account1, account2, ..., that are available to a ruleset. You can

declare a pattern and use it in the rules. Writing a rule once is the equivalent to writing a rule for each available object.

You can also add constraints to a pattern that limit which objects the pattern refers to. For example, if the objects are bank accounts, you can constrain the pattern to operate only on checking accounts.

A pattern can refer to the following:

- All instances of a class or interface.
- The instances of a class or interface in a specific collection.
- The instances of a class or interface in a specific collection that meet certain constraints.
- The instances of a class or interface that meet certain constraints but are not in a specific collection.

Creating Patterns

Create one or more patterns to write rules that evaluate each available object of a particular type or each available object of a particular type in a specific array.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 4 Click the arrow next to the **Local Variables, Patterns and Initialization Rule** section to expand it.
- 5 Click the **Add** icon on the context toolbar of the variables or patterns table.
- 6 Click the link in the **Name** column.
- 7 Select the **Pattern** radio button.
- 8 In the **Name** field, enter a name that is unique within the ruleset.
- 9 In the **Type** drop-down list, select the class type.
- 10 (Optional) Click the **Save** icon to save the pattern type selection.
- 11 (Optional) In the **In collection** field, if you have created a collection containing objects of the same type that you selected from the **Type** drop-down list, you can select the collection from the drop-down list.

Selecting a collection limits the pattern matching to the objects in that collection.

- 12 (Optional) In the **Such that** field, enter a constraint clause for the pattern using an expression that refers to the properties of the pattern type.

Alternatively, you can also select the **Switch to Advanced Builder** link to enter a constraint clause for the pattern using Structured Rule Language syntax in the **Advanced Builder**.

- a Click the **Open Editor** icon.
- b Drag a property from the **Item Selector** to the default group or drag an empty field and provide a valid value.
- c Click **Done**.

The constraint can include more than one test as in this example:
theAccount.type does not exactly match "savings" and
theAccount.name is equal to "Premium"

- 13 Click the **Save** icon or the **Check in** icon.

- 14 To return to the **Ruleset Definition** page, click the **Parent View** icon.

Related Links

[Rule Builder Interface](#)

[Authoring Decision Logic Using SRL](#)

Unused Pattern Warnings in Rules

When a project is compiled, a warning message is displayed when a rule condition in a ruleset references a pattern and the corresponding rule action does not. If you want a rule to fire for every object that satisfies the condition, use the `ignore()` built-in function in the action statement. The `ignore()` built-in function instructs the rule engine to repeat the rule action for each pattern match that satisfies the rule condition even if the pattern match is not referenced in the action.

You can use the `ignore()` built-in function when authoring rules in the Rule Builder or in SRL. This is an example of a rule that would generate a warning if the `ignore()` built-in was not used in the action statement:

```
if transcriptDetails.gradePointAverage is larger than or equal to 0
then {
    newApplicant.Applicant.GPA = totalGPA / transcriptCounter;
    ignore(transcriptDetails);
}
```

The signature for the function is `ignore(Object)`, where `Object` is the name of the class-based pattern.

Related Links

[ignore\(\)](#)

[Creating Patterns](#)

Dynamic Objects

Dynamic objects are instances of external classes defined in functions, rule actions, and initialization statements. Create object instances to hold known values, intermediate results of rule processing, or as test objects while authoring rules.

Dynamic objects are created or mapped into the project during rule processing. Create a dynamic object by using a variable or property assignment statement using the assignment (=) operator or by calling a `newInstance()` method of a class.

You can map dynamic objects as a result of calling a method that returns an object or as property values of objects that you explicitly map. Dynamic objects are available to the rules as long as there is a reference to them.

An object has a reference to it if one of the following applies:

- It is assigned to a variable.
- It is assigned to the property of another object.
- It is stored in a collection.

Use dynamic objects when one of the following applies:

- You do not know how many objects to create until rule processing occurs, and you do not have to save the objects after processing. For example, you might need to create a temporary account object on the basis of information that a user provides.
- You need to create an object to process information gathered during rule processing. For example, you might need to create an object if the user chooses an out-of-stock item and you want the rules to suggest another alternative to the customer.

Create a dynamic object using one of the following options:

- In the Rule Builder for an initialization rule or rule action, drag and drop a class-type property, variable, or pattern to the canvas. When an assignment statement is created, a new dynamic object is created and assigned to the class-type property, variable or pattern. A dynamic object of an external class type can be created only if the external class has a zero argument constructor.
- In the Advanced Builder editor, instantiate the object and then assign it to an *object property* using Structured Rule Language (SRL).


```
objectName.objectPropertyName = a className initially
    { property1=value, property2=value, ... }
```
- In the Advanced Builder editor, instantiate the object and then assign it to a *variable* using Structured Rule Language (SRL).


```
variableName = a className initially
    { property1=value, property2=value, ... }
```
- In the Advanced Builder editor, you can use the `newInstance()` function to call a constructor to create an *object of an external class using arguments* that

you specify. Then assign it to an object property or variable where the arguments must be of the same type the external class constructor specifies.

```
objectName.objectPropertyName = a className initially  
extClassName.newInstance(argument1, argument2, ...)
```

The `newInstance()` function lets you instantiate objects with arguments if the class has no zero-argument constructors. You can call this function with the appropriate arguments as part of an initialization rule or action statement. For example:

```
newProduct=product.newInstance(size, name)
```

When a local object is declared in a ruleset or a function, only statements and expressions within that decision entity can access or change the value of that object. In a ruleset, to create a reference to an object without an object reference, declare a pattern or variable of its type, and use it to write your rules.

Related Links

[Rule Builder Interface](#)

[Authoring Decision Logic Using SRL](#)

Creating an Initialization Rule

Use an Initialization rule to create statements that assign initial values for object properties. The initialization rule is the first rule that is evaluated in the ruleset so the initialization of property values occurs before any rules are fired. Therefore, you usually complete the initialization rule before you start writing the other rules. Because you can only enter action statements in the initialization rule, patterns are not available.

 **Note** The maximum depth at which you can select a property is 5. A workaround is to create an intermediate variable that can be used to reach the desired property.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 4 Click the arrow next to the **Local Variables, Patterns and Initialization Rule** section to expand it.
- 5 Click the **Open editor** icon in the **Initialization Rule** section.
Alternatively, you can also select the **Switch to Advanced Builder** link to enter Structured Rule Language syntax using the **Advanced Builder** editor.
- 6 In the **Rule Builder** interface, select a property, a functional, a built-in function, or a built-in construct from the **Item Selector** and drag it to the default group. If you declare one or more temporary variables, a new category is added to the Item Selector. When you are ready to use a temporary variable, drag and drop it from the **Temporary Variables** category.

- 7 Enter a value for the property.
- 8 Repeat steps 5-7 for any other properties that you want to initialize.
- 9 (Optional) If you need to add an operator, date, or empty field, drag and drop these elements to the statement you are creating.
- 10 Click **Done** to return to the **Ruleset Definition** page and save your changes.
- 11 Click the **Save** icon or the **Check In** icon.

Related Links

- [Rule Builder Interface](#)
[Authoring Decision Logic Using SRL](#)

Rule Builder Interface

The **Rule Builder** interface provides the ability to initialize variables and author expressions and statements using a drag and drop interface. This is the default interface when authoring conditions and actions in rulesets and when authoring action expressions in a decision tree. The interface is also available in the Function editor by clicking the **Switch to rule builder** link.

 **Note** There is no undo command in the Rule Builder interface to revert the delete operation. However, you can click **Cancel** to discard unsaved changes.

The **Rule Builder** interface has four main components:

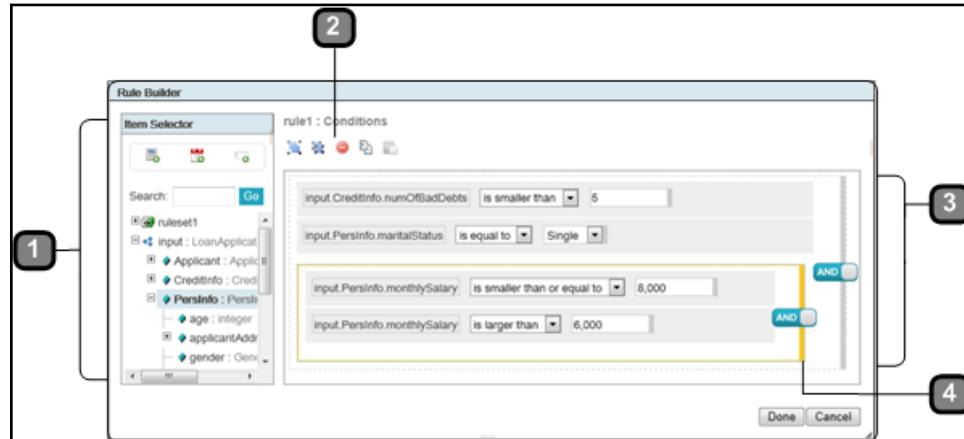


Figure 7: Rule Builder Interface

Item	Description
1	The Item Selector displays the entities associated with your object model. This could be parameters or variables and, in the case of rulesets, patterns. It includes a search field and a toolbar with options to add operators, dates, empty fields, or

Item	Description
1	<p>dynamic objects that you can use to create more complex expressions. These are the operations that you can perform with the toolbar commands:</p> <ul style="list-style-type: none"> ■ Drag and drop an operator and an empty field to add it to an expression. ■ Drag and drop a date to replace an existing date field or one that may have been accidentally deleted. ■ Drag and drop an empty field to replace an existing field or one that may have been accidentally deleted. ■ Drag and drop a dynamic object to replace an existing field or one that may have been accidentally deleted. When the dynamic object is dragged, the fields where the object can be used are highlighted. (Dynamic objects are not applicable when building rule conditions or creating constraints in the "Such that" field in the Pattern editor.) ■ Drag and drop a built-in function. ■ Drag and drop another functional. ■ Drag and drop static properties and methods. ■ Drag and drop a temporary variable declaration and then drag and drop the temporary variable. ■ Drag and drop a return statement.
2	<p>The Rule Builder toolbar displays commands that you can use to author decision logic. These are the operations that you can perform with the toolbar commands:</p> <ul style="list-style-type: none"> ■ (For condition expressions only.) Group expressions when you need to specify the order of operations for mathematical expressions. Click the outline of the first expression and then press Ctrl before clicking on the outline of the second expression. ■ (For condition expressions only.) Ungroup expressions. Click on the selector (see Item 4) for the expressions and then click the Ungroup command. ■ Delete an expression or a group of expressions. ■ Copy and paste an expression or a group of expressions.
3	<p>The default group appears on the right side. It displays the object and local variable properties that you drag and drop from the Item Selector. In rule conditions, the default group can also contain subgroups.</p>
4	<p>A selector is a vertical border on the right side on a date, field, group, or operator in the default group. When you click the selector of an item, the Rule Builder interface highlights the selector and the rectangle that encompasses the selected item.</p> <p> Note To change the order of the expressions, use the selector to drag and drop an expression or a group of expressions to another location.</p>

Related Links

- [Creating Local Variables](#)
- [Creating Patterns](#)
- [Dynamic Objects](#)
- [Creating an Initialization Rule](#)
- [Defining Rules](#)
- [Creating Rule Conditions](#)
- [Creating Rule Actions](#)
- [Assigning an Expression](#)

Creating Rules

Create one or more rules for the ruleset.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 4 Scroll down to the **Rules** section.
- 5 Click the **Add** icon to add a rule to the ruleset.
A default rule name is added as a link to the **Rules** table. The default rule name is rule appended with a number starting with 1. You can change the rule name when you define the rule.
- 6 (Optional) Repeat step 5 to create additional rules.
- 7 Click the **Save** icon or click the **Check In** icon.

After you create a rule, click the rule name to define it.

Copying and Pasting Expressions or Statements Within Rulesets

You can copy and past a condition expression or an action statement from one rule to another rule in the same ruleset.

- 1 On the Rule Definition page, select the condition expression or the group selector for the condition group or action statement. Then, click the **Copy** icon. The rule does not need to be enabled for editing when you copy an expression or statement.
- 2 Click the **Parent View** icon.
- 3 In the Rules section, click the name of the rule where you want to paste the expression, group, or statement.

- 4 Perform one of the following operations to enable the rule for editing:
 - If the ruleset is not checked out, click the **Check out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 5 Based on where you want to paste the content, click the **Open editor** icon in the Conditions or Actions section.
- 6 Click the group selector for the group where you want to add your expression, group, or statement. Then click the **Paste** icon.
- 7 Click **Done**.
- 8 Click the **Save** icon or click the **Check In** icon.

Defining Rules

The **Rule Definition** page defines each rule in the ruleset.

The Rule Definition page contains the following fields and sections:

Rule name	A unique rule name. The name must be unique in the ruleset. A valid name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.
Effective Date	(Optional) A date range. You can enter an effective from and to date using the MM DD, YYYY format. Alternatively, you can use the calendar widget in the Effective from and To fields to select dates. All values for effective dates are inclusive. For example, if you specify an effective date to be from 'June 1, 2020', the rule will be in effect on June 1, 2020.
Description	(Optional) A description for the ruleset.
Conditions	A condition consists of one or more condition expressions organized into one or more groups. You can use the Rule Builder or the Advanced Builder editor to create condition expressions.
Actions	An action consists of one or more action statements. You can use the Rule Builder or the Advanced Builder editor to create action statements.

Related Links

[Rule Builder Interface](#)

[Authoring Decision Logic Using SRL](#)

Rule Conditions in a Ruleset

Each rule in a ruleset must have a condition. A condition consists of one or more condition expressions organized into one or more groups. A rule is not complete until you create a corresponding rule action for a condition.

By default, condition expressions are added to the default group in the **Rule Builder** interface. When you add more than one condition expression to a group, the condition expressions can be joined by an AND or an OR operator.

You can create a subgroup after you have at least two condition expressions in the default group. Up to two levels of grouping are allowed in a condition in addition to the default group that contains all condition expressions.

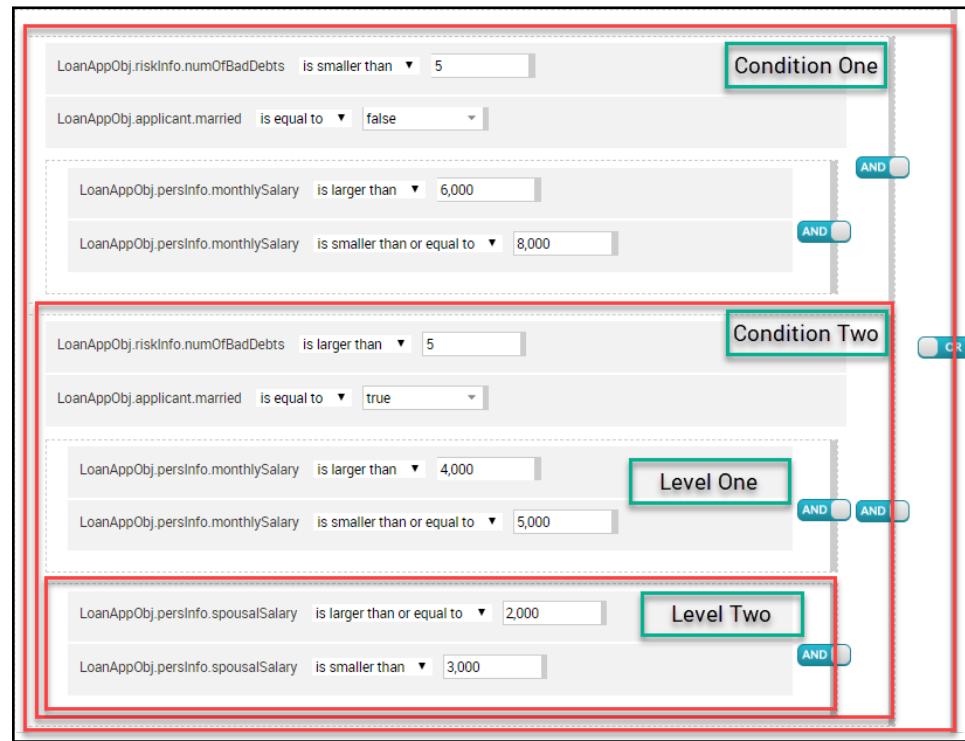


Figure 8: Example of Grouping Levels

In this example, you see two groups of condition expressions in the default group joined by an OR operator. Therefore, if one condition is true, then the action is executed.

Within each group of condition expressions there are condition expressions joined by an AND and at least one set of condition expressions that are joined by an AND that test a range of values.

The first condition is true when the following criteria is met:

- LoanAppObj.numOfBadDebts < 5
- LoanAppObj.Married = false
- LoanAppObj.income > 6000 AND LoanAppObj.income <= 8000

The second condition is true when the following criteria is met:

- LoanAppObj.numOfBadDebts < 5
- LoanAppObj.Married = true
- LoanAppObj.income > 4000 AND LoanAppObj.income <= 5000
- LoanAppObj.spousalIncome > 2000 AND LoanAppObj.spousalIncome < 3000

Related Links

[Valid Values for Properties](#)

Creating Rule Conditions

Use the **Rule Builder** or the **Advanced Builder** editor to author one or more expressions for a rule condition. The **Rule Builder** offers a structured way to author rules, while the **Advanced Builder** editor allows you to write expressions using Structured Rule Language (SRL) constructs.

 **Note** The maximum depth at which you can select a property is 5. A workaround is to create an intermediate variable that can be used to reach the desired property.

These instructions explain how to create rule conditions with the Rule Builder.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check Out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 4 In the **Rules** section, click on the rule name to open the **Rule Definition** page.
- 5 Click the **Open editor** icon in the **Conditions** section.
Alternatively, you can also select the **Switch to Advanced Builder** link to enter Structured Rule Language syntax using the **Advanced Builder** editor.
- 6 Select a property from the Item Selector and drag it to the default group. As you drag a property, a solid rectangle replaces the dotted rectangle around any field where it is valid to add the property.
After you have successfully added a property to the default group, you see an appropriate list of operators for that data type and an empty field or a drop-down list that you can use to enter or select a value.

- 7 Repeat step 6 to add additional properties to your default group.
When you add condition expressions to the default group, each expression is joined to the other expression with an AND operator. However, you can click on the AND operator to change it to an OR operator.
- 8 (Optional) Select a built-in function and drag it to the default group.
- 9 When you have completed your condition, click **Done**.
- 10 On the **Rule Definition** page, click the **Save** icon or click the **Check In** icon.
- 11 To return to the Ruleset Definition page, click the **Parent View** icon.

Related Links

[Authoring Decision Logic Using SRL](#)

[Rule Builder Interface](#)

Grouping Condition Expressions in the Rule Builder

Group condition expressions to test a range of values or to evaluate the group of expressions as a block. You can create a subgroup after you have at least two condition expressions in the default group.

- 1 In the **Rule Builder** interface, click on the outline of the first expression and then press **Ctrl** before clicking on the outline of the second expression.
A yellow outline appears around each expression you select.
- 2 Click the **Group/Add Parenthesis** icon from the editor toolbar.
By default, expressions are grouped with an **AND** operator icon.
- 3 (Optional) To change the **AND** operator icon to an **OR** icon, click the **AND** operator icon.

Rule Actions in a Ruleset

A rule action consists of one or more action statements that assign a value to a property as a consequence of a condition expression evaluating to true or false. Action statements are fired from the top to bottom.



Note If there is more than one action statement that assigns a value to a property, the last value assignment overwrites any assignment that was previously made.

Related Links

[Valid Values for Properties](#)

Creating Rule Actions

Use the **Rule Builder** or the **Advanced Builder** editor to author one or more statements for a rule action. The **Rule Builder** offers a structured way to author rules, while the **Advanced Builder** editor allows you to write statements using Structured Rule Language (SRL) constructs.



Note The maximum depth at which you can select a property is 5. A workaround is to create an intermediate variable that can be used to reach the desired property.

A rule action consists of one or more statements that do one or more of the following:

- Modify the values of object properties.
- Call functionals.
- Invoke built-in functions.
- Create new objects in working memory.
- Delete dynamic objects from working memory.
- Invoke Java methods on Java objects.
- Declare temporary variables for use within the action. Temporary variables can be of simple, complex, array or fixed array type.
- Return a value. (A rule can only return a value if the ruleset has a return type.)

Because temporary variables must be declared before they can be used, when you declare a temporary variable, each variable declaration automatically appears near the beginning of the action body. In Advanced Builder, FICO recommends as a best practice that you add any temporary variables near the beginning of the action body prior to any assignment statements or expressions.

In the Rule Builder, when you drag and drop a return statement, it automatically appears as the last statement in an action. In the Advanced Builder, it must be the last one in the sequence of statements in the action.



Note If you need to write complex statements, such as ones involving loops or nested conditions in an action statement, use the **Advanced Builder** editor.

- 1 Locate the ruleset in the **Explore** pane or the **Project Explorer**.
- 2 Click the ruleset to open it in the editor.
- 3 Perform one of the following operations to enable the ruleset for editing:
 - If the ruleset is not checked out, click the **Check Out** icon.
 - If the ruleset is already checked out, click the **Edit** icon.
- 4 In the **Rules** section, click on the rule name to open the **Rule Definition** page.
- 5 Click the **Open Editor** icon in the **Actions** section.
Alternatively, you can also select the **Switch to Advanced Builder** link to enter Structured Rule Language syntax using the **Advanced Builder** editor.

- 6 Select a property, a functional, a built-in function, or a built-in construct from the **Item Selector** and drag it to the default group.
If you declare one or more temporary variables, a new category is added to the Item Selector. When you are ready to use a temporary variable, drag and drop it from the **Temporary Variables** category.
- 7 Enter the required values to complete the statement.
- 8 When you have finished the action statements, click **Done**.
- 9 On the **Rule Definition** page, click the **Save** icon or click the **Check In** icon.
Until you save the rule, an asterisk (*) remains displayed next to the file name in the **Explore** pane.
- 10 To return to the **Ruleset Definition** page, click the **Parent View** icon.

Related Links

[Authoring Decision Logic Using SRL](#)

[Rule Builder Interface](#)

Validation

The **Rule Builder** interface displays immediate feedback when an expression or a statement is invalid. It ensures that the business terms that you add to the right side of the expression or statement are valid for the data type expected by the left side and its operand.

You will notice the following behavior when you build expressions and statements with the **Rule Builder**:

- If an invalid entry is made in a field or the field is empty, a thin red line appears around the area where the problem exists. If you hover over the area displaying an error, a tooltip displays describing why the expression is not valid. The red line persists until you enter a value with the correct data type.
- While dragging a business term to the default group, if you do not see the solid outline around the target location or your cursor is displayed as a **Not Allowed** icon, you are prevented from adding the business term to your expression. This is usually because the business term is read-only or what you are trying to add is not a business term.



Note Read-only business terms are allowed in condition expressions or in the right side of an assignment statement.

- When you have an object or sub-object in the **Item Selector**, you will not be able to drag the entire object to the default group. Instead, expand the object to display its business terms, then drag and drop the business terms to the default group.

Valid Values for Properties

The data type of a global object, variable, or local variable property determines the types of values that are valid in an initialization expression, condition expression, or action statement.

The format of the value you can enter is enforced by the system for both expressions and statements.

Property Type	Allowed Values	Additional Operators and Values
boolean	<ul style="list-style-type: none"> ■ true (default) ■ false 	none
date	<p>Use the calendar widget to select a date or enter a date in the short, medium, long, or full format of the display locale. For example:</p> <ul style="list-style-type: none"> ■ 9/30/2020 (short) ■ Sep 30, 2020 (medium) ■ September 30, 2020 (long) ■ Wednesday, September 30, 2020 (full) <p>The system automatically reformats the date to the medium format of the display locale. However, if you enter a date such as Sept 29, 2020, you see a validation error.</p>	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply)
duration	<p>A value that defines an amount of time in minutes, hours, days, months, and years. For example:</p> <ul style="list-style-type: none"> ■ 7 days ■ 2 months ■ 1 hour ■ 15 minutes <p>If you define an amount of time as a number of weeks, the weeks are converted to the equivalent number of days.</p> <p>If you define an amount of time as a number of seconds and the value is less than 60 seconds, there is no change. However, if the value is equal to or greater than 60 seconds, it will be converted to the number of minutes and seconds.</p>	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply) <p>If you want to specify a duration of time such as a number of hours, you can use a duration property type with a time value and a minus (-) sign with another time value. For example:</p> <ul style="list-style-type: none"> ■ object.durationProperty is equal to 9:00:00PM - 9:00:00AM for a duration of 12 hours ■ object.shippingTime is equal to 9:00:00AM - 6:00:00PM for a duration of -9 hours.

Property Type	Allowed Values	Additional Operators and Values
integer	<p>A numeric value that is whole. For example:</p> <ul style="list-style-type: none"> ■ 1000 ■ -1000 	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply) ■ / (divide) ■ div (division with integer values only) ■ mod (modulo) <p>You can use either / or div with integer values depending on the value you want to return.</p> <ul style="list-style-type: none"> ■ If you want to return an integer quotient, use div. For example: 17 div 5 returns 3. ■ If you want to return a quotient with a fractional, use /. For example: 17/5 returns 3.4
money	<p>A numeric currency value that is formatted based on the conventions of the display locale along with the ISO code or symbol. The system automatically reformats the value to display the ISO code before the numeric value. It also displays separators based on the conventions of the display locale. For example:</p> <ul style="list-style-type: none"> ■ USD 12.50 ■ USD 12,000 <p>You see a validation error if you enter a numeric non-currency value such as 12 or 12.5. Enter \$12 and the system will format it as USD 12.</p> <p>Note A parenthesized value, such as (USD 40), is considered a negative value.</p>	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply) ■ / (division) ■ div (division with integer values only) ■ mod (modulo)
real	<p>A numeric value that contains decimal places. For example:</p> <ul style="list-style-type: none"> ■ 1000.1 ■ -1000.1 	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply)

Property Type	Allowed Values	Additional Operators and Values
		<ul style="list-style-type: none"> ■ / (division) ■ div (division with integer values only) ■ mod (modulo)
time	A time value that is expressed in hours, minutes and seconds.	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply)
timestamp	A digital record of an occurrence of an event on a specific time, hours, minutes, seconds, on a given date.	<p>Depending on the first literal value, you can use one of the following operators to define a value for this property type:</p> <ul style="list-style-type: none"> ■ + (plus) ■ - (minus) ■ * (multiply)
object	A variable of type object cannot be used in a rule and cannot be dragged and dropped.	none
string	<p>Any value you enter for a string property is treated as a string.</p> <p>Note You do not need to enter single or double quotes for string values.</p>	<p>You can use the equality and relational operators for this property type including:</p> <ul style="list-style-type: none"> ■ is equal to ■ is not equal to ■ contains text ■ is contained within text ■ is not contain within text ■ is equal to (ignoring case) ■ is not equal to (ignoring case) ■ contains text (ignoring case) ■ is contained within text (ignoring case) ■ is not contain within text (ignoring case)

Related Links

[Rule Conditions in a Ruleset](#)

[Rule Actions in a Ruleset](#)

CHAPTER 7

Authoring Rules in Decision Tables

Use a decision table to author rules when the rules are regular, meaning that they share the same condition and action expressions.

A decision table is designed to accommodate a large number of rules that evaluate data against the same object properties, where the primary differences among the rules are the actual condition and action values. In a decision table, all of the condition expressions are connected with "and", which means that all of the condition expressions in a rule must be true to fire the action. If all or some of your condition expressions must be written using "or", write your decision logic with a ruleset or a decision tree instead.

The following layout types are supported:

- Single-axis (Columns)
- Double-axis

Single-axis (Columns) decision table

In a Single-axis (Columns) table, each *column* consists of a group of condition or action expressions that are based on the same object property, primitive data type or enumeration. Each *row* is a rule in the decision table.

In this example, the conditions are in columns A - C and the action is in column D.

A	B	C	D
Total Income	Rolling Over Balances	Credit History	Card Type
1 4,000 <= .. < 60,000	true	Fair	Standard
2 60,000 <= .. < 80,000	true	Good	Standard

Figure 9: Single -axis (Columns) Decision Table

The first rule in the table is "If the applicant's total income is between \$4,000 and \$60,000, the customer is rolling over balances from other credit cards, and their credit history is fair, then offer the Standard card type."

Double-axis decision table

In a Double-axis decision table, at least one *row* and one *column* are conditions. The cells in the matrix of the table are the action expressions. Each action expression represents one rule.

In this example, Total Income, Rolling Over Balances, and Credit History represent condition expressions and Card Type represents the action expression. There are four rules in this table.

A	B	C	D
	Total Income	40,000 <= .. < 60,000	60,000 <= .. < 80,000
Rolling Over Balances		Card Type	
1 true	Fair	Standard	Standard
2 true	Good	Standard	Gold

Figure 10: Double-axis Decision Table

The first rule reads the same as the one used in the Single-axis (Columns) table. Only the visual layout changed.

Creating Decision Tables

Use the **Decision Table** command on the **New** menu to create a Single-axis or Double-axis decision table.

- 1 In the **Project Explorer**, select the folder where you want the decision table created.
- 2 Click the **New** drop-down menu.
- 3 Select **Decision Table**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 In the **Table type** field, do one of the following:
 - Retain **Single-axis**, and click **Create**.
 - Select **Double-axis**. Additional fields appear allowing you to select a return type and create a parameter. If void is retained as the return type, click **Next** to select a property on which to base the action. Click **Select**, and then click **Create**.

In the Decision Table editor, you can change the return type and create one or more parameters.

Related Links

[Naming Decision Entities and Folders](#)

[Changing the Return Type](#)

[Creating Parameters](#)

Defining Decision Tables

The Decision Table editor contains the following fields:

Reference name	A unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and
----------------	--

underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.

Description	(Optional) A summary of what the decision table evaluates.
Return Type	By default, the return type is <code>void</code> ; however, you can change it to a primitive type or an enumeration type.
Parameters	Parameters are a way of declaring the types and names of any argument values or objects to be passed to a decision entity at runtime. Parameters can be a primitive, enumeration, or class type.

Changing the Return Type

By default, the return type is `void`; however, you can change the return type to return a primitive or enumeration type instead. If a decision table already contains action expressions based on one return type, and the return type is changed, modifications may need to be made to the action expressions.

Single-axis (Columns) Decision Table

When the return type is `void`, you can insert any number of action columns based on properties in your object model. The same holds true when the return type is a primitive or enumeration type except that the last action column must be based on the **Return** parameter or an error is thrown. The **Return** parameter must only be used once in a decision table.

If an action column is based the **Return** parameter and the return type is changed to another primitive or enumeration type, the values are reset to the default for the new return type. Alternatively, if the return type is changed from a primitive or enumeration type to `void`, the values are cleared in the **Return** column and you must delete the column. Add a new action column using the **Insert New Column** command on the Decision Table editor toolbar.

Double-axis Decision Table

The return type is initially set in the **Create Decision Table** wizard when the decision is created. This table describes the expected behavior if the return type is changed in the Decision Table editor.

Table 19: Expected Behavior When Changing the Return Type

Original Return Type	New Return Type	Result
Void	Primitive or Enumeration	Values in the action cells change to the default value for the new return type. For example, 0 for integer values or the current date for date values.
Primitive or Enumeration	Primitive or Enumeration	Values in the action cells change to the default value for the new return type.

Table 19: Expected Behavior When Changing the Return Type (continued)

Original Return Type	New Return Type	Result
Primitive or Enumeration	Void	Values in the action cells are cleared. Click the Update Action icon on the Decision Table toolbar to select another property.

Creating Parameters

You can create one or more primitive, enumeration, or class type parameters to pass data into a decision table. When you create a parameter, it can be used to create condition and action expressions.

- 1 Locate the decision table in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision table to open it in the editor.
- 3 Perform one of the following operations to enable the decision table for editing:
 - If the decision table is not checked out, click the **Check out** icon.
 - If the decision table is already checked out, click the **Edit** icon.
- 4 Click the **Add** icon on the context toolbar of the Parameters table.
- 5 Click the link.
- 6 In the **Name** field, enter a name that is unique within the decision table.
A parameter name cannot contain spaces or special characters such as "% \$:/\".
- 7 In the **Type** drop-down list, select the parameter type.
- 8 Click **Save** icon or the **Check In** icon.
- 9 Return to the decision table by clicking the **Parent View** icon on the toolbar.

Editing Rules in a Decision Table

Inserting or deleting a condition or action expression modifies the construction of the rules. When you add a condition or an action expression, the expression is added to each rule. Likewise, deleting a condition or action expression deletes an expression from each rule.

In this figure, there are two condition expressions (Deductible column and Network Member column) and one action expression (Approved Claim Percent column).

Deductible	Network Member	Approved Claim Percent
USD 2,000	true	90
USD 2,000	false	75

Figure 11: Single-axis Decision Table with Two Conditions and One Action

The decision table defines the following rules:

- If the deductible is 2,000 USD for a network member, then the claims are paid at 90%.
- If the deductible is 2,000 USD for a non-network member, then the claims are paid at 75%.

If another condition expression (ServiceType column) is added to the decision table, a condition expression is added to each existing rule.

Deductible	Network Member	Service Type	Approved Claim Percent
USD 2,000	true	Office_Visit	90
USD 2,000	false	Ambulance	75

Figure 12: Single-axis Decision Table with One Additional Condition

The decision table now defines the following rules:

- If the deductible is 2,000 USD for a network member and the service type was an office visit, then the claims are paid at 90%.
- If the deductible is 2,000 USD for a non-network member and the service type was a call for an ambulance, then the claims are paid at 75%.

The same concepts apply to a Double-axis decision table. This table shows the same two rules, but with a different layout.

	Deductible	USD 2,000
Network Member	Service Type	Approved Claim Percent
true	Office_Visit	90
false	Ambulance	75

Figure 13: Double-Axis Decision Table with One Additional Condition

Inserting Condition and Action Expressions

Each condition and action can be based on a primitive type, an enumeration or a property from your object model. The selection of a primitive type, enumeration or object property is made in the **Insert New Column** window.

If the project was configured with an XML schema, in addition to any parameters created in the Decision Table editor, you have access to the global variable used for the input type, and any business term sets. If you uploaded a project exported from Blaze Advisor, in addition to any parameters you created, you have access to any global objects and global variables in the project and any business term sets. If you later modify the name or type of a parameter, variable, object or business term on which a condition or action column is based, the column or row where it was used is outlined in red until the problem is fixed.

This table shows the commands used to add condition or action expressions to each type of decision table layout.

Table 20: Condition and Action Expressions

Decision Table Layout	Command Names
Single-axis (Columns)	Insert New Column or Insert New Row
Double-axis	In the Context menu the commands are Insert New Row (above the column headers) or Insert New Column (adjacent to one of the existing condition columns). On the Decision Table editor toolbar, the commands are Insert New Condition Row or Insert New Condition Column . The Update Action command on the toolbar is enabled when the return type is void . This command opens a window where you can change the property on which an action column is based.

- 1 Locate the decision table in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision table to open it in the editor.
- 3 Perform one of the following operations to enable the decision table for editing:
 - If the decision table is not checked out, click the **Check Out** icon.
 - If the decision table is already checked out, click the **Edit** icon.
- 4 Depending upon the layout of the decision table, perform one of these operations:
 - In a Single-axis decision table, click the **Add Column** icon on the Decision Table editor toolbar.
 - In a Double-axis decision table, click the **Add Condition Row** or **Add Condition Column** icon on the Decision Table editor toolbar.
- 5 (Optional) In the **Search** field, enter the name of an object property and press **Enter**. The search is case-insensitive. If the property is found, the object type containing the property is highlighted.
- 6 In the **Property Picker**, select an object property on which to base the condition or action expression.
- 7 Enter a label for the column header in the **Display Name** field.
- 8 Select one of the commands to specify where to place the new column or row.
- 9 Click the **Save** icon or click the **Check In** icon.

Related Links

[Changing the Return Type](#)

Editing Decision Table Rule Names

Each decision table rule has a name in the **Rule Id** column that can be edited at any time. A valid name consists of one or more letters (A–Z, a–z), digits (0–9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %. Every rule must have a unique name in the decision table.

This is an example of how the Rule Id column appears in a decision table:

	Rule Id	Deductible
1	ClaimRule1	USD 600
2	ClaimRule2	USD 0 <= .. <= USD 200

Figure 14: Rule Id in a Decision Table

If a rule is cut and pasted from one row to another, the rule name and the number remain unchanged. However, if a rule is copied and pasted, it is assumed that this is a new rule and the rule number is incremented by 1.

 **Note** This functionality is not applicable for Double-axis decision tables.

Using Multiple Values in Decision Table Cells

For condition cells based on a string or enumeration, you can enter or select a set of unique values. At execution time, the condition will be met if the value passed into the condition cell matches one of the values.

- 1 Locate the decision table in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision table to open it in the editor.
- 3 Perform one of the following operations to enable the decision table for editing:
 - If the decision table is not checked out, click the **Check out** icon.
 - If the decision table is already checked out, click the **Edit** icon.
- 4 Create a condition based on a string or enumeration property.
- 5 Double-click in a condition cell. Use the drop-down menu to select "is one of" to enable support for multiple values.
- 6 Perform one of the following operations:
 - For enumerations, select a value and press **Enter**. Alternatively, erase the default entry, enter the first letters of the desired enumeration item, and then click **Enter**.
 - For string values, enter a value and then click **Enter**.
- 7 Repeat step 6 as necessary.
- 8 (Optional) To reorder a value, double-click in the cell and place the focus on the double bars next to a value and move it to the desired location.
- 9 (Optional) To delete a value, select it, and click the **Delete** icon.
- 10 (Optional) Click the **Save** icon or click the **Check in** icon.

Changing the Expression Format in a Decision Table Cell

Decision Modeler includes several expression formats for numeric data types. You can select from among several expression formats for object properties with numeric data types. For example, you could select > **integer value** to indicate that the property is less than a particular integer value.

In addition to the various expression formats, there is an option to use **Not Applicable**, which means that the condition expression in that cell will always resolve to true regardless of the value that is passed in to the cell.

- 1 Locate the decision table in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision table to open it in the editor.
- 3 Perform one of the following operations to enable the decision table for editing:
 - If the decision table is not checked out, click the **Check out** icon.
 - If the decision table is already checked out, click the **Edit** icon.
- 4 Perform one of the following operations to open the context menu:
 - Click in the cell and click the down arrow.
 - Click in the cell and press **Ctrl + Down** (directional arrow).
- 5 Under **Cell Formats**, select an expression. Alternatively, using your keyboard, press the number next to the expression.
- 6 Double-click in the cell to edit the values.
- 7 Click the **Save** icon or click the **Check in** icon.

Cutting, Copying, and Pasting Rows and Columns

You can insert a row or column with copied content in the decision table. The paste operation does not replace an existing row or column.

The Cut, Copy, and Paste commands are available on the Decision Table editor toolbar and from the context menu. Using keyboard shortcuts, you can select multiple rows or columns to be cut or copied and pasted as long as the rows or columns are contiguous.



Note Copying and pasting individual values from a decision table to another application such as Microsoft Excel or a text editor is supported; however, copying and pasting rows, columns, or ranges of values is not supported.

Related Links

[Decision Table Keyboard Shortcuts](#)

Copying and Pasting Content in Decision Table Cells

Although there are some restrictions, you can copy the contents of an entire cell or copy a value and then paste it into another cell.

The Copy and Paste commands on the Decision Table editor toolbar are for copying and pasting condition or action columns. Use the commands on the Context menu or keyboard shortcuts to copy and paste content in a decision table cell.

The following restrictions apply when you copy and paste content in a decision table cell:

- Copying and pasting the entire contents of a cell is applicable only to cells that are based on a same data type.
For example, if you copy a condition cell containing > 6000 and paste it into another condition cell containing = 5000, the = 5000 is overridden with > 6000. Not only has the cell value changed, but the expression format has changed too. For cells based on a string type, if the value was selected from a list, you can only paste the cell content to other cells in condition or action columns that use the same list.
- Copying and pasting a value is applicable only when the values are of a compatible type. If you copy and paste a value that is not of the correct type, a warning message appears stating that value is invalid.

 **Tip** To see if a cell will accept a pasted value, click in the cell and open the context menu to see if the Paste command is enabled.

Related Links

[Decision Table Keyboard Shortcuts](#)

Decision Table Profiling with Sample Data

Use profiling to verify that the decision table generates the expected results based on a sample dataset.

When profiling is run, the data in the dataset is evaluated by the condition expressions. The amount of data that matches the conditions is displayed in the Decision Table editor as a percentage or numeric count of the data. You can run the rule profiler as needed or use real-time profiling to provide feedback as the values are edited in the decision table.

The dataset file must be a CSV (comma-separated values) file. You can create your own dataset in an external application, such as text editor, or you can use the Rule Profiling menu to generate a sample CSV dataset file and then modify the values.

Real-time Profiling

By default, profiling occurs in the background while the decision table is edited.

As changes are made to the values in the condition cells, the profiling results are updated in the decision table to reflect those changes. Real-time profiling is

triggered by modifications such as changing cell values or expression formats or by adding and deleting rules.

If you generate a report while the real-time results update feature is enabled, the results in the report reflect the current state of the decision table.



Note Unless you save the decision table before generating the report, the current state may not be the saved state.

You can disable real-time profiling using the **Set Display Options** dialog. Open the **Set Display Options** dialog by clicking the down arrow next to the **Rule Profiling** icon on the Decision Table editor toolbar. If real-time profiling is disabled, profiling continues and the current results remain visible in the decision table. However, the profiling results are not displayed for any new rows added to the decision table and the profiling results that were already displayed are not updated based on any edits to the decision table.

Dataset Requirements for Decision Table Profiling

The dataset used for profiling must be a CSV file that conforms to specific requirements. You can create your own dataset in an external application or generate a sample dataset and modify the values.

The CSV file must have the following characteristics:

- The data must be listed in columns. Each row is a set of condition values for one rule.
- The first row in the file must contain the condition labels.
- The number of condition labels in the file must be at least equal to the number in the decision table.
- Each subsequent row in the file must contain the exact same number of values. The only exception is for a string value, which can have an empty value.
- Each column in the file must contain data of the correct data type for the corresponding condition in the decision table.
- If a condition expression can contain multiple values, add a row for each possible value. This is an example of a condition expression that can contain more than one value.

```
if ((claim.serviceType ="Office_Visit") or (claim.serviceType ="X_Rays")) or  
(claim.serviceType ="FollowUp_Visit"))
```

This is how the values might appear in the dataset. The only change in each row is another possible value for the Service Type condition.

Deductible	Claim	Service Type	Network Member
USD 600	USD 400	Office_Visit	FALSE
USD 600	USD 400	X_Rays	FALSE
USD 600	USD 400	FollowUp_Visit	FALSE

- The encoding for the CSV file must be Unicode (UTF-8).

When a dataset is attached, the headers in the first row of the CSV file are compared to the condition labels in the decision table. If there are sufficient

matching headers in the dataset, these headers are mapped to the corresponding condition labels and profiling is ready to run. The data columns in the dataset can be in any order because the mapping is based on the condition labels specified in the dataset's header row. This allows you to use data where the columns in the data file may not be in the same order as the decision table.

If the dataset does not contain sufficient matching headers and mapping is incomplete, an error message is displayed. If the dataset contains more data columns than the number of condition labels in the decision table, the extra data columns are ignored as long as there are sufficient matching headers.

The number of condition headers in the dataset must match the number in the decision table at the time the profiler is run. Attaching a dataset then altering the decision table structure, such as removing or adding a condition, will cause the dataset to become detached. If the dataset is detached, update it, and attach it again.

Related Links

[Decision Table Profiling](#)

Creating a Sample Dataset for Decision Table Profiling

You can create a sample dataset based on the current decision table. It shows the correct format for the dataset and includes a column header and column of data for each condition in a decision table. After the sample dataset is generated, you can modify the values as required.

- 1 Click the **Rule Profiling** icon to open the **Rule Profiling** menu.
- 2 Select **Generate Sample Data**.
- 3 Open or save the file.
If you do not save the file to a specific directory, it is automatically saved to `Users\<yourUserName>\Downloads`
The file name is `FileName_Dataset.csv`. The sample dataset file has `_Dataset` appended to it to distinguish it from the `FileName.csv` file generated using the **Export Decision Table** command.

Running Decision Table Profiling

To run the profiler, attach a dataset to run against the conditions in the decision table. Only one dataset can be attached to a decision table. The dataset remains attached until you attach another valid dataset or you close the Decision Table editor.

- 1 Click the **Rule Profiling** icon.
- 2 Select **Run Profiling Data**.
- 3 Click **Browse** and navigate to a dataset.

- 4 Select the file and click **Open**.

If you select a file type other than a .csv file or if the file cannot be mapped correctly, the **Run sample data** dialog displays an error message. Click **Browse** again and select another dataset file.

- 5 Click **Run Data** to start profiling.

The Run Data option is not enabled unless a valid dataset is attached and all of the headers are mapped to the corresponding conditions in the decision table.

The profiler displays a status message during profiling and then displays the results in the decision table in one or more of the following formats: bar chart, numeric count, or percentage. The last valid dataset is retained until a new valid dataset is attached to the decision table or the Decision Table editor is closed.

 **Note** Attaching a dataset and then removing or adding a condition detaches the dataset. If a dataset becomes detached, update the dataset, and attach it again.

Profiling Results Display Options

Rule profiling results are displayed based on the decision table type and the display options.

There are three different options for displaying the profiling results:

Charts The length of the yellow bar is proportional to the percentage of data that matches the conditions in a rule.

Percentage The percentage of the total number of value sets in the dataset that match the conditions. A value set is equal to one row of data in the dataset.

Counts The total number of value sets in the dataset that meet the conditions.

The bar chart and percentage are the default settings, as shown in this example of a Single-axis (columns) decision table:

	Matches	A	B	C
1	49%	<= 0.25	N/A	20
2	51%	> 0.25 <= 0.4	N/A	-10
3	0%	> 0.4	N/A	-50
4	55%	N/A	<= 0.4	20
5	45%	N/A	> 0.4 <= 0.5	-10

Figure 15: Decision Table with Rule Profiling Results

You can configure the display options to include the count (the total number of value sets in the dataset that meet the conditions) in the Set Options Display window.

The way the profiling results are displayed depends upon the decision table layout.

Table 21: Display Results Based on Layout Test

Decision Table Type	Display Results
Single-axis (Columns)	Rule profiling results are displayed in a separate frozen Matches column next to the row index column.
Double-axis	Rule profiling results are displayed within the action cells.

Related Links

[Setting Profiling Options](#)

Decision Table Profiling Report

The report includes columns for the Rule ID and the corresponding count and percentage results. All three columns are displayed by default and can be hidden or displayed as desired using the check boxes on the left side.

Display:	Rule ID	Count	Percent
<input checked="" type="checkbox"/> Rule ID	5 - ClaimRule5	6	6%
<input checked="" type="checkbox"/> Count	53 - ClaimRule53	6	6%
<input checked="" type="checkbox"/> Percent	57 - ClaimRule57	6	6%
	8 - ClaimRule8	5	5%
	29 - ClaimRule29	5	5%
	4 - ClaimRule4	4	4%
	7 - ClaimRule7	4	4%
	11 - ClaimRule11	4	4%
	39 - ClaimRule39	4	4%
	59 - ClaimRule59	4	4%

Figure 16: Decision Table Profiling Report

The Rule ID column displays the rule index plus the user-defined rule label, if the rule label is visible in the decision table; otherwise it refers to the rule index. The columns display the corresponding profiling results in descending order. By clicking the up or down arrow in a column, you can sort the results in ascending or descending order.

The report display varies depending upon the decision table layout:

- For a Single-axis (Columns) table, the Rule ID column shows the row index number. For example: 1, 2, 3, . . . , 10, 11, 12, . . . , 100, 101, and so on.
 - For a Double-axis table, the Rule ID column shows a letter plus a number. For example: A1, A2, A3, . . . , A10, A11, . . . , B1, B2, and so on.
-  **Note** When the Decision Table editor is closed, the attached dataset file or related profiling information is not saved.

Generating Profiling Reports

You can generate a report containing the profiling results after profiling is run on a decision table.

- 1 Click the **Rule Profiling** icon.
- 2 Select **Generate Report**.
- 3 From the **Show entries** drop-down list, select the number of entries to display.
- 4 Select the **Rule ID**, **Count** and **Percent** check boxes to hide or show those columns of data in the report.
- 5 (Optional) Click a column header to sort the report data. You can only change the sort order for one column at a time.

Setting Profiling Options

Set the profiling options in the **Set Display Options** window. The selections made in this window are persisted only until the Decision Table editor is closed.

- 1 Click the **Rule Profiling** icon.
- 2 Select **Set Display Options**.
- 3 (Optional) Select or clear **Enable real-time profiling results**.
- 4 (Optional) Select or clear **Show charts**.
- 5 (Optional) Select or clear **Show percentage**. This option displays the percentage of the total number of value sets in the dataset that match the conditions.
- 6 (Optional) Select or clear **Show counts**. This option displays the total number of value sets in the dataset that match the conditions.
- 7 Click **Save** to update the display setting.

Related Links

[Real-time Profiling](#)

Enabling and Disabling Decision Table Profiling

Profiling is enabled by default. When profiling is disabled, the profiling results are removed from the decision table. The current dataset remains attached and you can still generate and view the profiling report.

- 1 Click the **Rule Profiling** icon.
- 2 Select one of the following:
 - **Disable Profiling:** The profiling results are cleared from the decision table and the menu item toggles to **Enable Profiling**.
 - **Enable Profiling:** The profiler is run again using the attached dataset (CSV file), and the results are displayed in the decision table. The menu item toggles back to **Disable Profiling**.

Filtering Rules in Decision Tables

In the Decision table editor, you can apply filters to the columns to control which rules are displayed.

You will notice the following behavior when filters are applied:

- When you select or enter an expression in a filter, only those rules that contain the expression are displayed. If there are no rules that contain the expression in the filter, then all of the rules in the table are hidden from view.
- If filters are used in other header columns, you can only filter the rules based on the ones that are displayed as a result of the first filter.
- Each subsequent filter that is applied adds further constraints to the display.
- Configured filters are not saved with the decision table.

For Double-axis decision tables, the filters are available only for condition columns and the actions matrix.

Showing Filters

You can filter rules in decision tables.

- 1 Click the **Show Filters** icon to enable the filtering fields.
- 2 Click the arrow in the column where you want the filter applied.
- 3 Perform one of the following operations and click **OK**:
 - Select an expression keyword and then enter the appropriate values.
For example, if **Between** is selected, enter the boundaries of the values you want matched in the rules.
 - Select an expression from the list.
If there are cells in a condition column that contain multiple values, the values are presented in the filter dialog as one entry with each value separated by a comma.

- 4 Repeat Steps 2 and 3 for all header columns where you want filters applied.

Clearing Filters

You can clear filters in a column or clear all the filters at the same time.

- ▶ Perform one of the following actions:
 - Click the arrow in the column where you want to clear the filter and click **Remove Filters**.
 - Select the column where you want to clear the filter and backspace through the entry and press **Enter**.
 - Select the filter entry in the column where you want to clear the filter, press **Delete**, and then press **Enter**.
 - Click the **Clear Filters** icon on the Decision Table editor toolbar.



Note This option clears filters for all columns in the decision table at the same time.

Exporting and Importing Decision Table Data

You can export and import decision table data in the Decision Table editor.

When decision table data is exported, both a **.csv** file containing the values and a **.properties** file containing configurable arguments, such as file paths, folder names, and so forth are automatically generated. The files are generated in a ZIP file.

You can export decision table data in one of these two formats:

Default Format This format creates new cells in the target file with values matching those of the corresponding cells in the decision table.

Split Cells Format This format splits value cells into one or more CSV table cells to accommodate a range of values; for example, **10,>= .. >=,5**. Table cells with just one or two values receive an empty cell or cells to balance the column count in the CSV file. The split cells can be modified easily; for example **EMPTY, >, VALUE** could be changed in the CSV file to **VALUE, > .. >, VALUE**.

Make	Model	Year	Rating
Ford	Falcon	1,974	-0.5
Ford	Falcon	1,977	3.7
Chevrolet	Impala	<=	1,960
Chevrolet	BelAir	>=	1,955
Chevrolet	BelAir	<	1,955

Figure 17: Exported data in the Split Cells format

After you export the decision table data, you can modify the data and then import the modified data into the decision table.

Permission Requirements for Exporting Decision Table Values

Before exporting decision table values to a CSV file, verify the permission requirements.

Review the following permission requirements:

- You have write permission to the disk and existing files.
- The CSV file to replace is closed.

Exporting Decision Table Data to CSV Files

You can export decision table values to a CSV file. The decision table does not need to be checked out to perform this operation.

- 1 Click the **Export Decision Table** icon on the Decision Table editor toolbar.
- 2 Select **Default Format** or **Split Cells Format**.
- 3 (Optional) Enter a description in the **Comments** field.
- 4 Click **Finish**.
- 5 In the dialog that appears, select one of the following options:
 - **Open** to open the ZIP file.
 - **Save** to save the file.
 - **Cancel** to stop the operation.

After you export the decision table, you can update and reimport the values into the decision table. However, you must know which format was used to export the data.

Formatting Requirements for Importing Decision Table Data

Before importing decision table data, verify that these formatting requirements are met.

- Any format change in a cell (for example, from < to <=) is an allowed expression format in that cell.
- The condition or action format matches in both the CSV file and decision table. For example, a condition column formatted for real values does not contain string values.
- The number of condition and action (columns or rows) is the same in the CSV file and in the decision table.

- No header columns or rows are missing or empty.
 - The encoding for the CSV file must be Unicode (UTF-8).
-  **Note** Any changes to the names of header columns or rows in the CSV file are ignored.

Importing Data Values from CSV Files into Decision Tables

You can import data into a decision table using a CSV file. When you import decision table values, you must know the format that was used to create the decision table values.

- 1 Locate the decision table in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision table to open it in the editor.
- 3 Perform one of the following operations to enable the decision table for editing:
 - If the decision table is not checked out, click the **Check Out** icon.
 - If the decision table is already checked out, click the **Edit** icon.
- 4 Click the **Import Decision Table** icon on the Decision Table editor toolbar.
- 5 Click **Browse** to locate the .csv file to import and click **Open**.
- 6 Under **Import Format**, select one of the following formats:
 - Default Format
 - Split Cells Format
- 7 (For Single-axis decision tables only) Under **Additional options**, select one of the following options:
 - **Replace Existing Data** to overwrite all current data in the decision table with the imported data.
 - **Append Data** to add the imported data to the decision table.
- 8 Click **Finish**.
- 9 Click the **Save** icon or click the **Check In** icon.

Decision Table Keyboard Shortcuts

Keyboard shortcuts are available in the Decision Table editor.

Row

When a row is selected, these keyboard shortcuts allow you to perform the following operations:

Table 22: Row Shortcuts

Shortcut	Operation
Up directional key	Selects the row above the current position.
Down directional key	Selects the row below the current position.
Shift + Up directional key	Extends the selection to the adjacent rows above the currently selected row.
Shift + Down directional key	Extends the selection to the adjacent rows below the currently selected row.
Delete	Deletes selected rows.
Ctrl + C	Copies selected rows.
Ctrl + X	Cuts selected rows.
Ctrl + V	Pastes at selected row position.

Column

When a column is selected, these keyboard shortcuts allow you to perform the following operations:

Table 23: Column Shortcuts

Shortcut	Operation
Left directional key	Selects the column to the left of the current position.
Right directional key	Selects the column to the right of the current position.
Shift + Right directional key	Extends the selection to the adjacent columns to the right of the selected column.
Shift + Left directional key	Extends the selection to the adjacent columns to the left of the selected column.
Delete	Deletes selected columns.
Ctrl + C	Copies selected columns.
Ctrl + X	Cuts selected columns.
Ctrl + V	Pastes at selected column position.

Cell

When a cell is selected, these keyboard shortcuts allow you to perform the following operations:

Table 24: Cell Shortcuts

Shortcut	Operation
Shift + F10	Opens the context menu for that cell.
F2/[A-Za-z0-9]/ENTER	Makes the value or values editable in that cell.
Ctrl + Down (directional key)	Opens a menu of other possible expression formats for that cell.

Table 24: Cell Shortcuts (continued)

Shortcut	Operation
Home	Moves the focus to the first cell in the row.
End	Moves the focus to the last cell in the row.
Enter	Moves the focus to the cell below.
Page Up	Moves the focus to the header cell.
Page Down	Moves the focus to the last cell in a column.
Ctrl + Space	Selects the entire column.
Shift + Space	Selects the entire row.
Shift + Arrow (directional keys)	Selects contiguous rows or columns.

Compiled Table Optimization

Compiled Table Optimization provides for a smaller memory footprint, improved throughput and a faster compilation time.

By default, Compiled Table Optimization is available for use with projects containing decision tables of any size, but it may significantly improve the performance of projects with larger decision tables.

When a project with one or more decision tables is compiled, Compiled Table Optimization is applied automatically. If you compile a project with one or more decision tables and you see Compiled Table Optimization warnings, this means that a decision table in your project contains some unsupported template structures, SRL expressions, or data types. If this is the case, another optimization is applied instead. If there is more than one decision table in your project, during the compilation process, only those decision tables that contain unsupported constructs will use default metaphor optimization and the others will continue to use Compiled Table Optimization.

If you want to use Compiled Table Optimization on all your decision tables, you need to edit any decision tables associated with the warnings and remove the unsupported constructs, expressions or types described below.

Best Practices for using Compiled Table Optimization

The following are recommended best practices for ensuring that Compiled Table Optimization is used for all decision tables in your project:

- Avoid using any of the constructs, expressions, or types in condition or actions cells that are listed in the **Limitations** section below.
- If functions or method calls must be used, use a local variable to store the result of the call and use the variable in the conditions of the decision table.
- If patterns must be used, use an intermediate function or ruleset to loop through elements in a collection and focus the decision table on the object type contained within the collection.

Limitations

Compiled Table Optimization does not support decision tables with the following constructs, expressions, or types in condition or action cells:

- Duration data type
- Functional or method calls in conditions
- Built-in Functions in conditions
- Comparisons between a variable, static object or parameter to another variable, static object or parameter in conditions
- Change detection
 - If rules in rows or columns depend on values modified in previous rows or columns, the change will not be detected.
- Inferencing
- Patterns
- Instances based on Decision Table Templates created using Blaze Advisor versions prior to 7.2.5.



Note When a known limitation is encountered, a warning message is displayed and the Compiled Table Optimization is not applied to the current decision table. The default optimization is applied for these decision tables.

CHAPTER 8

Authoring Rules in Decision Trees

A decision tree is a tree-like graph used to model decisions and the possible outcomes of those decisions. A decision tree is a convenient way to visualize the different paths in a complex decision process.

You can create a decision tree by selecting decision variables from your object model properties and business terms or by importing a FSML model.

The Start node is on the left side of the tree. Each vertical band is known as a "level" and corresponds to a decision variable in your object model. Each condition node (displayed as a circle) on a single level represents the segmentation of the same decision variable. The leaf node at the end of each branch is the action (outcome) of each decision.

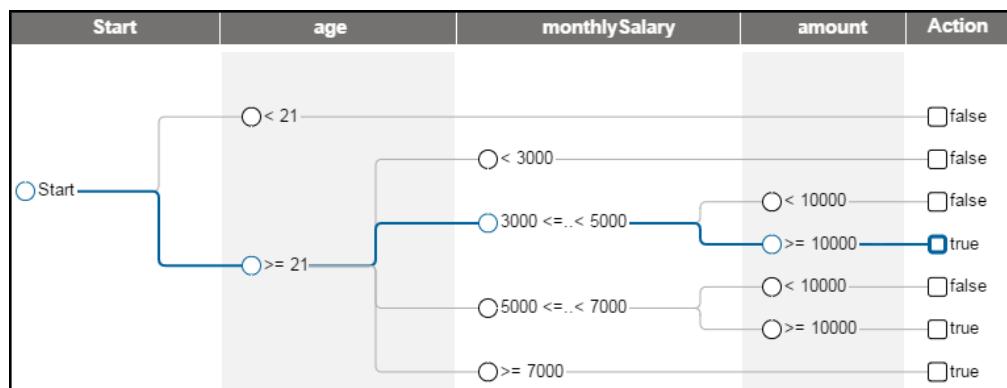


Figure 18: Decision Tree with Condition and Action Nodes

You use a wizard to select the decision variables to use for the condition levels and the action level in the decision tree. A decision variable can represent an object property in your imported object model, a business term, or a parameter. The decision tree supports these data types for decision variables:

- boolean
- date
- enumeration
- integer
- real
- string

You can use a text string and String type with a value list.



Note The money, duration, time and timestamp data types are not supported so properties and business terms of these types are not available in the wizard.

Decision Tree Interface

The decision tree interface consists of the Decision Tree editor toolbar and a canvas to create and edit the decision tree.

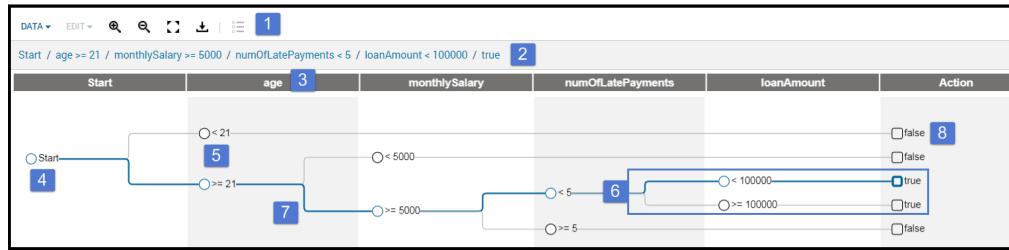


Figure 19: Decision Tree with Levels, Splits, Branches, and Action Nodes

Table 25: Decision Tree Element Descriptions

Area	Element	Description
1	Decision Tree editor toolbar	This toolbar contains commands that let you generate a sample dataset, run profiling, edit the decision tree, change the magnification of the decision tree, download the decision tree as an FSML file, and manage the variables.
2	Node path	When a node is selected in the decision tree, the element names for a given series of nodes are displayed.
3	Level	Each level represents a condition or action variable. If you created and exported a decision tree that contained a "." in a level name in a previous release and you import and open the decision tree in this release, the "." in the level name will be replaced by a " _ ". Note The level names cannot be edited.
4	Start node	To begin building a decision tree, select the start node to see the available commands in the Edit drop-down menu.
5	Split node	A split node and its branches represent condition values.
6	Branch	The values in a split.
7	Subtree	Splits with branches and nodes. A subtree is made up of branches each with their own set of nodes. You can assign an action node to the branch nodes on the last level.

Table 25: Decision Tree Element Descriptions (continued)

Area	Element	Description
8	Action node	Each action node represents a value assigned to the action variable. When you assign an action for the first time, an Action level is created. End nodes without an action assignment are considered incomplete. For example, the branch with < 21 is an incomplete node. Incomplete nodes do not cause compilation errors.

Creating Decision Trees

Create a decision tree and select variables for the conditions and actions using the wizard. In a decision tree, an object model property is known as a variable.

If you imported an XML schema to use as your object model, you will see a global variable for the input type, and any business term sets you created. If you uploaded a project exported from Blaze Advisor, you have access to any global objects and global variables in the project, and any business term sets. Additionally, you can create parameters in the Decision Tree editor.

- 1 In the **Project Explorer**, open the **New** drop-down menu and select **Decision Tree**.
- 2 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
Decision tree names are automatically validated and any errors are displayed on the wizard page. If a name includes spaces, the spaces are replaced with underscores in the decision tree name. If you call the decision tree from a function using SRL, include the underscores in the name.
- 3 (Optional) Enter a **Description** that summarizes what the decision tree evaluates.
- 4 (Optional) Select a **Return type**.
If a type other than void is selected, you do not need to specify the Action variable. You can change the return type after creating the tree in the Decision Tree editor.
- 5 (Optional) Add **Parameters**.
Parameter names cannot contain spaces or special characters such as "%\$:/\". Validation checks for SRL naming and uniqueness occur automatically and any error messages are displayed on the wizard page. See [Creating Parameters](#) on page 118.
- 6 Click **Next**.
- 7 Select one or more variables to use as the conditions.
If your object model contains several classes that have properties with the same name, to avoid confusion, move the properties of the first class to the condition variables list before moving the properties of the second class to the list.
When you move the property of the first class to the conditions variable list, it is displayed with the property name. When you move the property of the second class that has the same name, it is displayed with the partially qualified

property name. For example, `class.propertyName`. This name will be displayed as the decision tree level label.

 **Note** Parameters can be searched for by name. If the parameter is of a class type, you cannot search for specific class properties.

- 8 Click **Next**.
- 9 Select one variable to use as the action.
- 10 Click **Done**.

Related Links

[Naming Decision Entities and Folders](#)

Creating Parameters

You can create parameters in the Create Decision Tree wizard or in the Decision Tree editor. A parameter can be used as a boolean, date, integer, real, string, enumeration, or class type.

The decision tree will go into repair mode for one of the following reasons if a parameter used as a condition or action variable is changed:

- The data type is changed.
 - The parameter is deleted.
 - The parameter type is invalid. For example, a business term set is removed or its name is changed.
- 1 Locate the decision tree in the **Explore** pane or the **Project Explorer**.
 - 2 Click the decision tree to open it in the editor.
 - 3 Click the **Edit** icon on the toolbar to enable editing.
 - 4 Perform one of the following operations to enable the decision tree for editing:
 - If the **Check out** icon appears on the toolbar, click the icon.
 - If the **Check out** icon is not on the toolbar, click the **Edit** icon.
 - 5 Click the **Add** icon on the context toolbar of the Parameters table.

A default parameter name is added as a link to the table. The parameter name is appended with a number starting with 1.
 - 6 Click the link.
 - 7 In the **Name** field, enter a name that is unique within the decision tree.
 - 8 In the **Type** drop-down list, select the parameter type.
 - 9 Click **Save**.
 - 10 Return to the decision tree by clicking the **Parent View** icon on the toolbar.

Using Variables in a Decision Tree

A variable is a placeholder for storing a value in a decision tree.

You can create variables of the following types:

Primitive Data Types A data type such as boolean, integer, real, or string.

Object Types Object types from the imported business object model.

After you create a variable, you can use the variable as a condition or action variable in your decision tree. As a condition, you can test the value of a variable. As an action, you can assign a value to a property based on the value of a variable.

Variable of Primitive Type

You can create a variable of primitive type to perform a calculation based on the values in the object properties and then use the calculated value as a condition.

For example, suppose you have MortgageApplication as the business object model for the decision service and you want to create a variable named IncomeAfterPayments of type real. This variable subtracts the applicant's monthly debt from the applicant's monthly salary. totalMonthlySalary is a property of the PersInfo sub-object type and totalMonthlyDebtPayment is a property of the Applicant sub-object type.

You can click **Edit in Advanced Builder** to initialize the IncomeAfterPayments variable to newApp.PersInfo.totalMonthlySalary - newApp.Applicant.totalMonthlyDebtPayment. Then, you can use IncomeAfterPayments as a condition variable in a decision tree.

Variable of Object Type

You can create a variable of an object type when you want to use the object properties as a condition or action.

For example, suppose LoanApplication is the input object type for the project. Applicant is an object type within the LoanApplication object type. You want to create conditions and actions with only the Applicant properties. So, you create a variable named NewApplicant. Then, you initialize the properties in NewApplicant so that they map to the Applicant properties.

You can click **Edit in Advanced Builder** to initialize the values that you plan to use in the decision tree. The initialization for NewApplicant can look like the following:

```
an Applicant initially
{firstName=input.Applicant.firstName,
 housingRatio=input.Applicant.housingRatio,
 lastName= input.Applicant.lastName,
 loanAmount= input.Applicant.loanAPR }
```

 **Note** In this example, input is a variable that is initialized with the input type of the project.

Then, create conditions and actions with the initialized properties of NewApplicant.

Creating Local Variables

You can create one or more local variables that can be used as conditions or actions in a decision tree.

- 1 Locate the decision tree in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision tree to open it in the editor.
- 3 Perform one of the following operations to enable the decision tree for editing:
 - If the decision tree is not checked out, click the **Check out** icon.
 - If the decision tree is already checked out, click the **Edit** icon.
- 4 Click the arrow next to the **Decision Tree** section to expand it.
- 5 Click the **Add** icon on the context toolbar of the variables or patterns table.
A default entity name is added as a link to the table. The entity name is appended with a number starting with 1.
- 6 Click the link in the **Name** column.
- 7 In the **Name** field, enter a name that is unique within the decision tree.
- 8 In the **Type** drop-down list, select the variable type.
- 9 In the **Initialization** section, enter the initial value for the variable.
 - To initialize a variable of *primitive* data type, use the Rule Builder.
 - 1 Click **Edit in Rule Builder**.
 - 2 Drag a property from the **Item Selector** to the default group or drag an empty field and provide a valid value.
 - 3 Click **Done**.
 - To initialize a variable of *object* type, use the Advanced Builder.
 - 1 Click **Edit in Advanced Builder**.
 - 2 Use Structured Rule Language (SRL) to initialize the variable. For example:

```
an Applicant initially
  {firstName=input.Applicant.firstName,
   housingRatio=input.Applicant.housingRatio,
   lastName= input.Applicant.lastName,
   loanAmount= input.Applicant.loanAPR }
```
- 10 Click the **Save** icon or the **Check in** icon.
- 11 To return to the **Decision Tree** page, click the **Parent View** icon.

Using Patterns in a Decision Tree

Use a pattern in a condition or action variable to evaluate each available object of a particular type or each available object of a particular type in a specific array.

A pattern is a way to refer to multiple instances of a specific class or interface. Create a class-based pattern and refer to it in the condition or action variables to

evaluate each object that matches the pattern. When a pattern is encountered in a condition or action, it is treated as a shorthand notation for defining a rule for every object that matches the pattern.

For example, suppose you have a class called Account, and have some number of account objects, account1, account2, ..., that are available to a decision tree. You can declare a pattern and use it in the condition or action variables of a decision tree. Referring to a pattern in a condition or action variable of a decision tree is equivalent to writing a rule for each available object.

You can also add constraints to a pattern that limit which objects the pattern refers to. For example, if the objects are bank accounts, you can constrain the pattern to operate only on checking accounts.

A pattern can refer to the following:

- All instances of a class or interface.
- The instances of a class or interface in a specific collection.
- The instances of a class or interface in a specific collection that meet certain constraints.
- The instances of a class or interface that meet certain constraints but are not in a specific collection.

Creating Patterns

Create one or more patterns to use as a condition or action variable to evaluate each available object of a particular type or each available object of a particular type in a specific array.

- 1 Locate the decision tree in the **Explore** pane or the **Project Explorer**.
- 2 Click the decision tree to open it in the editor.
- 3 Perform one of the following operations to enable the decision tree for editing:
 - If the decision tree is not checked out, click the **Check out** icon.
 - If the decision tree is already checked out, click the **Edit** icon.
- 4 Click the arrow next to the Decision Tree section to expand it.
- 5 Click the **Add** icon on the context toolbar of the variables or patterns table.
- 6 Click the link in the **Name** column.
- 7 Select the **Pattern** radio button.
- 8 In the **Name** field, enter a name that is unique within the decision tree.
- 9 In the **Type** drop-down list, select the class type.
- 10 (Optional) Click the **Save** icon to save the pattern type selection.
- 11 (Optional) In the **In collection** field, if you have created a collection containing objects of the same type that you selected from the **Type** drop-down list, you can select the collection from the drop-down list.

Selecting a collection limits the pattern matching to the objects in that collection.

- 12 (Optional) In the **Such that** field, enter a constraint clause for the pattern using an expression that refers to the properties of the pattern type.

Alternatively, you can also click **Edit in Advanced Builder** to enter a constraint clause for the pattern using Structured Rule Language syntax in the **Advanced Builder**.

- a Click **Edit in Rule Builder**.
- b Drag a property from the **Item Selector** to the default group or drag an empty field and provide a valid value.
- c Click **Done**.

The constraint can include more than one test as in this example:
theAccount.type does not exactly match "savings" and
theAccount.name is equal to "Premium".

- 13 Click the **Save** icon or the **Check in** icon.

- 14 To return to the **Decision Tree** page, click the **Parent View** icon.

Related Links

[ignore\(\)](#)

[Unused Pattern Warnings in Rules](#)

Unused Pattern Warnings for Decision Trees

When a project is compiled, a warning message is displayed when a condition in a decision tree references a pattern and the corresponding action does not. If you want a rule to fire for every object that satisfies the condition, use the `ignore()` built-in function in the action statement. The `ignore()` built-in function instructs the rule engine to repeat the action for each pattern match that satisfies the condition even if the pattern match is not referenced in the action.

You can use the `ignore()` built-in function when authoring decision logic in the Rule Builder or in SRL. This is an example that would generate a warning if the `ignore()` built-in was not used in the action statement:

```
if transcriptDetails.gradePointAverage is larger than or equal to 0
then {
    newApplicant.Applicant.GPA = totalGPA / transcriptCounter;
    ignore(transcriptDetails);
}
```

The signature for the function is `ignore(Object)`, where `Object` is the name of the class-based pattern.

Changing the Return Type

You can specify the return type of the decision tree in the **Create Decision Tree** wizard or in the Decision Tree editor. The return type can be void, boolean, date, integer, real, string, enumeration, or class type.

If `void` is selected as the return type, you can specify an Action variable. If any other data type is selected, you will not be able to specify an Action variable. You can change the return type to specify an Action variable using the **Manage Decision Tree Variables** dialog in the Decision Tree editor.

It is a best practice to change the return type before creating the Action level in the decision tree. If the return type is changed after the Action level has been created, the decision tree will go into Repair Mode. To fix the decision tree, do one of the following:

- Change the return type back to the original type.
- Remove the Action level.
- If the return type was changed to void, select an Action variable in the **Manage Decision Tree Variables** dialog.

Inserting Splits

Add a split to a decision tree when you want to create a set of branches for condition nodes. You can add a split to the Start node or to any condition node in a decision tree by selecting **Insert Split** in the **Edit** drop-down menu. Each split consists of two or more branches.

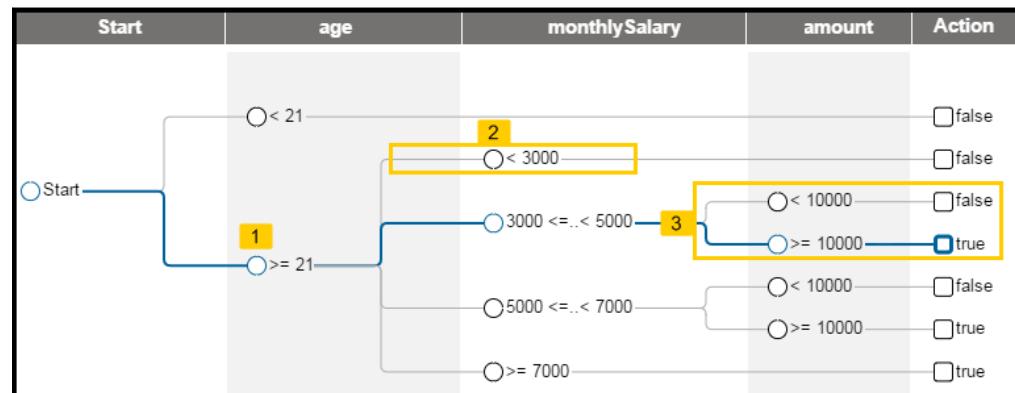


Figure 20: Decision Tree with Splits, Branches, and Subtrees

Item	Label
1	Split
2	Branch
3	Subtree

Select the Start node or a condition node to insert a split. You can add a split node that creates condition nodes on an existing level or create a new level while creating the split. A split with branches is also known as a subtree, which can be edited, cut, copied or pasted.

When you insert a split, enter a **Branch threshold** value or a string value and click **Insert** for each branch you want to create. Click **Apply** to add the branches to the decision tree.

- If you have entered numeric values, the ranges for each condition node corresponding to a branch are automatically generated as soon as you click

Apply. Additional thresholds are also added to ensure that there are no gaps or overlaps for numeric types.

- If string values are entered, each value is displayed as a branch along with the **Any other value** node. The Any other value branch is used to manage incoming string values that do not match any of the string values. The branches and the Any other value branch are added as soon as you click **Apply**.
- If you selected a condition variable of type enumeration, each enumeration value is displayed as a branch as soon as you click **Apply**.

After you have created several splits and subtrees, you have the option to collapse a subtree or the entire tree for a focused view of the tree. In the **Edit** drop-down menu, select **Collapse subtree** or **Collapse entire tree**.

There are a few options to expand the tree and subtrees. In the **Edit** drop-down menu, for collapsed subtrees, select **Expand subtree** or **Expand all subtrees**. For collapsed trees, select **Expand all**.

Inserting Levels

All condition nodes on a single level represent a segment of the same decision variable. When you create a decision tree, it does not contain any levels. Add a level using the **Insert Level** command in the **Edit** drop-down menu. You can also insert a level at the same time you insert a split using the **Insert Split** command.

The **Insert Level** command is available only when you first begin building a tree. After you begin adding splits, the Insert Level command is no longer available. However, you can add additional levels by selecting a condition node and then clicking the **Insert Split** command. An action level is created when you assign an action for the first time in a tree.

If you need to change the data type of a condition variable that is used as a level in the tree, it is a best practice to change the data type before creating its nodes. However, if you find you need to change the data type of a condition variable after you have added the nodes, the tree may go into Repair Mode and you may need to remove the nodes in the level before removing the level and replacing it with the new level.

You can view a level's properties, such as the variable and its type, by selecting the level (outlined in blue) then selecting **Properties** in the **Edit** drop-down menu.

 **Note** The display names of the levels are not editable.

Removing Levels

To remove a level, remove all of the nodes in the level first. After the nodes have been removed, select the level and then click **Edit > Remove empty level** or select the command from the shortcut menu.

Replacing Levels

You can replace the condition variable used for a level as long as the variable is replaced with one of the same type. For example, if a string variable is used for the level, you can only replace it with another string variable.

 **Note** String value list and enumeration data types can be replaced for each other.

- 1 Select the level you want to replace.
- 2 Click **Edit > Replace Level** or select the command from the shortcut menu.
- 3 Select a variable of the same data type from the drop-down menu or click the icon to open the **Select Decision Key Variable** dialog. Select a variable and click **Apply**.
- 4 Edit the values.
- 5 Save the changes.

Copying and Pasting Subtrees

A split with two or more branches is known as a subtree. You can copy a subtree from a given condition node and paste it to another condition node. You usually copy and paste subtrees when you want to repeat the same series of conditions on another branch using the same levels.

To copy a subtree, select a branch with a condition node that has the subtree you want to copy and use the **Copy subtree** command in the **Edit** drop-down menu.

To cut a subtree from one branch node to another branch node, select the **Cut subtree** command. Then select the condition node where you want to paste the subtree and select the **Paste subtree** command. The **Paste subtree** command is enabled only if you have copied or cut a subtree from a branch node.

When you cut and paste subtrees, duplicate levels may be created. Duplicate levels are permitted in decision trees.

Reordering Branches

Reorder the branches to change the display order in the tree. Use the **Edit Split** dialog to reorder the branches in a condition node. To open the **Edit Split** dialog, select the condition node and then select **Edit Split** from the **Edit** drop-down menu or from the shortcut menu.

The dialog displays the order of the branches as they are displayed in the decision tree. Note that the 0 branch is the top branch of the split. You can only reorder branches in levels of boolean, enumeration, or string type.

Branch	Values
0	Emergency
1	Office_Visit
2	Any other value

Figure 21: Changing the Display for the Branches

To reorder the branches, select the row with the branch number that you want to move. Drag and drop it on top of the branch number where you want it to appear. The order of the branches automatically readjusts. The tooltip changes to a green checkmark when the reordering operation is valid.

Creating Conditions Using Or by Merging Branches

Merge the branches to combine condition expressions that are connected with "or". For example, to author an expression such as `thePatient.serviceType` is Emergency or `thePatient.serviceType` is Office_Visit, merge the branches. You can only merge branches in levels of enumeration or string type.

Use the **Edit Split** dialog to merge the branches in a condition node. To open the **Edit Split** dialog, select the condition node and then select **Edit Split** from the **Edit** drop-down menu or from the shortcut menu. From the **Branch** column, drag and drop a branch row on top of another branch row to merge them. The tooltip changes to a green checkmark when the merge operation is valid, as shown in this figure. Note that the **Any other value expression** in a string level can never be merged.

Branch	Values
0	Emergency
1	Office_Visit
2	Any other value

Figure 22: Valid Merge Operation

To undo the operation, click the unmerge icon in the row to separate the expressions.

Branch	Values		
0	Emergency, Office_Visit		
1	Any other value		

Figure 23: Icon Used to Unmerge the Branches

Using Any Other Value as a Condition Expression

The **Any other value** expression is automatically added when you create a split with a condition variable of type string. It is used to manage situations where incoming data does not match the criteria used on the other branches of a split.

If you decide not to assign an action to an **Any other value** condition node, it is treated as an incomplete node and will not be evaluated. The incomplete node will not cause a compilation error.

Assigning Actions

Each decision tree has one action variable. An action is assigned to the last condition node for a given series of branch nodes. You can assign a value or a value based on an action expression or a list of action expressions to an action node.

- 1 Select the condition node where you want to assign an action.
- 2 Select an action by doing one of the following:
 - Click the **Edit** drop-down menu.
 - Right-click on the condition node.
- 3 Select one of the following options:
 - **Assign value:** Assign a specific value for the action. Depending on the data type of the action variable, you can select or enter a value.

Note This is how values were assigned to action nodes in previous releases.
 - **Assign expression:** Create an action expression or list of action expressions. A return value is not necessary, but if one is specified, its type should compatible with the Action variable type.
- 4 Assign a value or create an expression or expressions for the action. For a value, the value must match the data type of the Action variable. For an expression, the evaluated value must match the data type of the Action variable.
- 5 Click **Apply**.

Assigning an Expression

The Assign Expression dialog provides the ability to author action expressions using the Rule Builder, a drag and drop interface, or the Advanced Builder, a Structured Rule Language (SRL) editor. Action expressions or statements assign a value to a property as a consequence of a series of branch nodes (conditions) evaluating to true or false. Action expressions or statements are fired from top to bottom. If there is more than one action statement that assigns a value to a property, the value assignment overwrites any assignment that was previously made.

 **Note** There is no undo command to revert the delete operation. However, you can click Cancel to discard unsaved changes.



Figure 24: Assign Expression Dialog

This dialog has five main components:

Table 26: Assign Expression Dialog Components

Item	Description
1	The Expression label is the text that will display on the Action node in the decision tree. This is a required field.
2	The Switch to Advanced Builder button switches to an interface where you can create action expressions using SRL. You can use the Advanced Builder when you need to initialize a variable that is an object type from the imported business object model. Important When you switch to the Advanced Builder, all existing expressions will be ported to the Advanced Builder. However, if you choose to switch back to the Rule Builder, any expressions cannot be ported back.
3	The Rule Builder toolbar displays commands that you can use to author decision logic.
4	The Rule Builder Item Selector displays the entities associated with your object model, which can be parameters or variables. Note The maximum depth at which you can select a property is 5. A workaround is to create an intermediate variable that can be used to reach the desired property.
5	A Rule Builder selector is a vertical border on the right side of a date, field, group, or operator. When you click the selector of an item, the item is highlighted.

Related Links

[Authoring Decision Logic Using SRL](#)

[Rule Builder Interface](#)

Changing an Action

You can change an action to a literal value or an expression when an action has already been applied to a condition node. When you change an action, the decision tree will not go into Repair Mode.

- 1 Select the condition node where you want to change the action.
- 2 Open the menu by doing one of the following:
 - Click the **Edit** drop-down menu on the toolbar.
 - Right-click on the condition node.
- 3 Select one of the following options to change the action:
 - **Change to expression:** Change a value to an action expression. If this option is selected, the current value will be deleted.
 - **Change to value:** Change an action expression to a value. If this option is selected, the calculation in the expression will be deleted.
- 4 Assign a value or create an expression for the action.
For a value, the value must match the data type of the Action variable. For an expression, the evaluated value must match the data type of the action variable.
- 5 Click **Apply**.

Managing Decision Tree Variables

The **Manage Decision Tree Variables** dialog allows you to view and search for the available variables from your object model and parameters and to add or remove condition and action variables for a decision tree.

You can search for variables using the text box. However, only variables that can be used as a condition or action variable will appear in the search results. Classes cannot be used as condition or action variables. Only their properties can be selected as condition or action variables.

To remove a condition variable or replace an action variable that is already being used in a decision tree, you must first delete the level where it is used. Otherwise, the variable name is disabled in the dialog and it cannot be removed or replaced.

 **Note** To remove a level, you must delete all of the nodes in the level first.

If **void** is selected as the return type, you can specify an Action variable. If any other data type is selected, you will not be able to specify an Action variable. To specify an action variable, change the return type to a type other than void and remove the Action level from the tree.

Related Links

[Removing Levels](#)

Repairing a Decision Tree

The decision tree may open in Repair Mode if certain changes are made to variables in the decision tree.

The decision tree opens in Repair Mode, if any levels or nodes are outlined in red. A decision tree enters Repair Mode if any changes have been made to the following entities used in the decision tree:

- Names or data types of object model properties.
- Return type is not compatible with the action variable data type.
- Names of enumeration values.
- Names of business term sets where its business terms have been used as condition or action variables.
- Names or data types of business terms that have been used as condition or action variables.
- The name of a value that comes from a value list associated with a business term used as a condition or action variable.
- Names or types of parameters, variables, or patterns that are used as condition or action variables.
- Deleting parameters, variables, or patterns that are used as condition or action variables.
- Change the return type. If your action is displayed in Repair Mode as a result of a change to the return type, open the **Manage Decision Tree Variables** dialog and select a new action variable. When the dialog is closed, the action variable is immediately applied to the action level and you must edit the nodes.

While the decision tree still displays in the editor, the Copy, Cut and Paste and the Insert Split and Insert Level commands are not available; however, the Manage Decision Tree Variables icon is available. Any levels or nodes that are in error are outlined in red.

The commands that are specifically used for repairing the decision tree are displayed in the **Edit** drop-down menu. Depending on the change that triggered the level to display an error, follow the instructions in the messages that appear when you hover over the level displaying in Repair Mode. For example, if the variable type is changed from string to integer, you may need to clear the subtree and remove the empty level before inserting a new split.

To replace the *levels*, use variables of the same data type as the ones you want to replace. To repair *nodes*, use values with the same data types as the previous values.

 **Note** Values of type string with a value list and enumeration data types can be replaced for each other.

 **Note** If you have replaced your object model with another object model that is similar in structure and content, you may not be able to repair your decision tree until you remove the older object model using the Project Configuration dialog, and then reconfigure the project.

 **Note** In some circumstances, if you have a business term that is associated with a property of the input class, and you replace the object model, the next time you open the tree and open the Manage Condition Variable dialog, you may see a warning about the object model but the tree is not in Repair Mode. In this case, you can edit the business term set, replace the input type, and edit the term to remove the reference to the unavailable property. The next time you open the tree, it will be in Repair Mode.

Related Links

[Verifying the Business Object Model in your Decision Service](#)

Decision Tree Profiling

Use decision tree profiling to see how data flows through the decision tree logic and how the actions are distributed for a given sample dataset.

You can use profiling for decision trees that are created manually or are a result of a FSML import. When the profiler runs, the condition nodes in your decision tree evaluate each record in a dataset. The data that matches the conditions and actions are calculated and displayed in the Decision Tree editor as a record count.

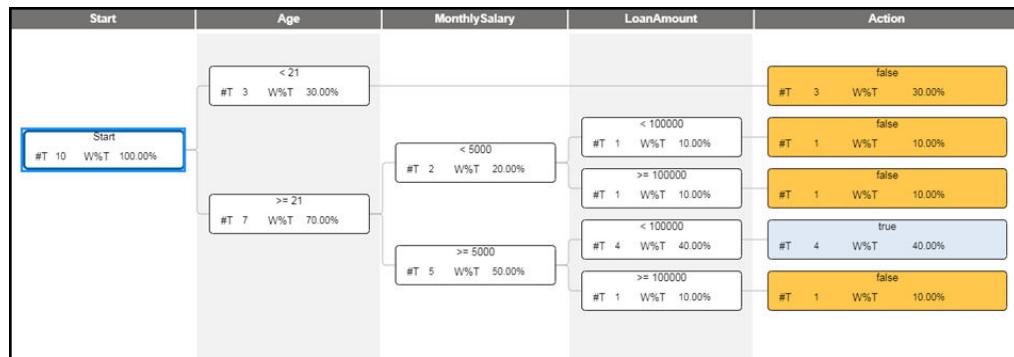


Figure 25: Decision Tree Displaying in Tree View

Dataset Requirements for Decision Tree Profiling

The dataset used for decision tree profiling must be a CSV (comma-separated values) file that conforms to specific requirements. You can create your own dataset in an external application or generate a sample dataset and modify the values.

The CSV file must have the following characteristics:

- The first row in the file must contain the condition labels.
- The number of condition labels in the file must be at least equal to the number in the decision tree.

- Each subsequent row in the file must contain the exact same number of values. The only exception is for a string value, which can have an empty value.
- Each column must contain data of the correct data type for the corresponding level in the decision tree.
- The encoding for the CSV file must be Unicode (UTF-8).
- For any date type values, the sample dataset file uses the default language and locale setting on your browser. For example:
 - For en-US, the date format is mm/dd/yyyy.
 - For fr-FR, the date format is dd/mm/yyyy.
 - For nl-NL, the date format is dd-mm-yyyy.



Note If you have a sample dataset generated using a previous Decision Modeler version that only supported the mm/dd/yyyy date format, if you run decision tree profiling in a browser locale that does not recognize this format, you may see errors. To avoid these errors, edit the date values to use the format supported by the browser locale.

The headers in the first row of the CSV file are compared to the condition labels in the decision tree. If the headers in the dataset match the levels in the decision tree, mapping occurs and profiling is ready to run.

The columns in the dataset do not necessarily need to be in the same order as the conditions in the decision tree. If the dataset does not contain sufficient matching headers and mapping is incomplete, an error message displays.

Generating a Sample Dataset for Decision Tree Profiling

Generate a sample dataset based on the current decision tree. The generated CSV file shows the correct format for the dataset and includes a column header and column for each condition in a decision tree. After the sample dataset generates, you can modify the values as required.

1 (Optional) Select **Data > Generate sample data** to generate and download a CSV file populated with columns from your tree. Enter rows with values you want to profile.

Alternatively, you can write your own dataset in an external application such as a text editor or Microsoft Excel.

2 Open or save the file.

If you do not save the file to a specific directory, it is automatically downloaded to `Users\<yourUserName>\Downloads` in Windows.

By default, the filename takes the following form:
`<DecisionTreeName>.csv`.

Previewing and Running Data Profiling

To run the decision tree profiler, attach a dataset (.csv file) to run against the conditions in the decision tree. Only one dataset can be attached to a decision tree.

After you upload the dataset and run the profiler, the dataset is saved in memory as long as the editor remains open. Therefore, if you edit the decision tree, you can recount the tree using the same dataset and update the statistics based on changes you make.

 **Note** The profiling data is persisted as long as the profiling views editor remains open.

- If you plan to edit your decision tree in the profiling views editor, FICO recommends that you check out the decision tree and enable Edit mode *prior* to running the profiler.
- If you check out a decision tree or change the version you are viewing after you run the profiler, you see a warning message.
- If you enable or disable Edit mode or edit a parameter, the profiling views editor closes and the statistics are no longer available until the next time you run the profiler.

- 1 Select **Data > Run profiling data**.
- 2 On the **Run Profiling Data** page, drag and drop a CSV dataset file onto the drop zone or click **local drive** or **FICO Drive** to browse for the file to add.

 **Note** The dataset file must be smaller than 100MB.

- 3 Select the file and click **Open**.

If you select a file type other than a CSV file or if the file cannot be mapped correctly, an error message displays. Click the links in the drop zone again and select another dataset file.

You see a dialog containing a preview of the dataset file. The first 1000 rows in the dataset are displayed with the total number of records in the dataset displayed in the upper right side above the preview table.

- 4 Click **Run Data** to start profiling.

The Run Data button is disabled until a valid dataset is uploaded and all of the headers are mapped to corresponding levels in the decision tree.

New sets of views are now available to help you examine the statistics for your decision tree.

Displaying Decision Tree Profiling Results

After the decision tree is profiler is run, several views are displayed to allow you to examine the statistics in more detail.

These views are available as long as the decision tree remains open and when you close the editor, the profiling results are not persisted. To view the statistics, you need to run the profiler each time.

Decision Tree Views

Use the different views to create, edit, and analyze your decision tree nodes. After you run the profiler, the **Views** menu appears on the Decision Tree editor.

After you run the profiler and edit the tree in Tree view, the **Recount** icon is available in any of the views. The **Export** icon is also available in all four views, but it behaves differently depending on the view. In Tree view, you can export the tree as an FSML file. In all of the other views, you can export a CSV file that contains the view statistics.

Table 27: Decision Tree Views

View Name	Description
Tree view	The default view where you create and edit the decision tree after you run profiling.
Profile view	Allows you to see and sort the condition variables you have selected for the tree. All the conditions appear in tabular format where the tree hierarchy is preserved. Action values and expressions are visible in this view.
Leaf view	Allows you to see and sort all the leaf nodes in the tree. You can see the leaf and action nodes in this view. This view is particularly helpful if you have a large tree with many leaf nodes. Action values and expressions are visible in this view.
Action summary	Allows you to view a summary of all action values in the decision tree. Action expressions are not visible in this view. Note If the Action is of date type, statistics will not display.

Decision Tree View Options

You can use the following column options to display the decision tree profiling statistics in the Profile view, Leaf view, and Action summary.

In any of the columns, change the order of the statistics by selecting **Sort Ascending** or **Sort Descending**. You can also hover over the column heading to see a tooltip with more information.



The screenshot shows a decision tree structure with various nodes and their properties. A context menu is open over the 'Order' column header, displaying options: 'Sort Ascending', 'Sort Descending', 'Columns', and a checked checkbox labeled 'Order'. The 'Action' column contains values like 'false' and 'true'. The '#T' and 'W%T' columns show counts and percentages. The 'Action' column has checkboxes for 'Action', '#T', '%T', 'W#T', and 'W%T', with 'Action' and '#T' checked.

Name	Order	Action	#T	W%T
Start	1			
└ Age < 21	2			
└ Age >= 21	3			
└ MonthlySalary < 5000	4			
└ LoanAmount < 100000	5	false		
└ LoanAmount >= 100000	6	false		
└ MonthlySalary >= 5000	7			
└ LoanAmount < 100000	8	true		
└ LoanAmount >= 100000	9	false		

Figure 26: Profiling Options in a Column

 **Note** Not all columns are available in each view.

Table 28: Profiling Options

Context	Statistics Column Heading	Statistics Description
	Name	Node name
	Order	Order that the node appears in the tree
	Action	Action node with display color displayed.
For all decision nodes	#T (Default)	Total number of raw counts. The count represents the number of records evaluated by a given node.
	%T	Percent of total counts
	W#T	Weighted total number of raw counts
	W%T (Default)	Weighted percentage of total counts The percentage of records evaluated by a given node.

Variables are not weighted so the weighted percent and counts are equivalent to the raw percent and counts.

Exporting Tree View Statistics

Use the **Export to CSV file** icon on the Profile view, Leaf view, and Action summary to export the view statistics in a CSV file.

- 1 Click **Export to CSV file** from a view toolbar.
The Export to CSV file dialog allows you to export the view statistics.
- 2 Enter a name for the file.
The filename is appended with _<viewname>. For example, if you choose to export the statistics in Profile view, the .csv file name is appended with _profile.
- 3 Click **Save**. By default, if you do not choose a file location, the file is downloaded to `Users\<yourUserName>\Downloads`.

Node Views

Use the different node views to display the profiling statistics in Tree view.

The **Counts** and **Counts-labels** views have a **Color by Action** option. This option displays the percentage of actions used in the conditions nodes. If you edit the decision tree, use the **Recount** icon to update the statistics.

Table 29: Node Views

View Name	Description
Compact	Displays the condition and action nodes without any statistics or labels. This view allows you to revert to a node view with no data

Table 29: Node Views (continued)

View Name	Description
	statistics except for colored action nodes. The action nodes display colors to visually group similar actions.
Counts	Displays the count statistics with the W%T (Weighted percentage of total counts).
Counts-labels	Displays the count statistics with the #T (Total number of raw counts) and the W%T (Weighted percentage of total counts) labels in Tree View by default.

Action Node Colors

The action node colors are visible in each tree view and are used to group the same action. The action nodes are automatically assigned colors and the colors cannot be edited.

The leaf nodes and parent nodes display the color of the related action values when the **Color by Action** option is selected in the **Counts** or **Counts-labels** views. Leaf nodes that do not have an action display with no color. Action expressions are also not colored.

 **Note** If there are two actions in a tree with the same display name, the action nodes use the same color.

Showing the Proportion of Actions in Decision Trees

Show the proportion of actions in each of the condition nodes in your tree by using the **Color by Action** option on the **Views** drop-down menu. When enabled in Tree view, the proportions appear in each node displaying a proportion of the related action color. You can also hover over each action color to view the percentage of the condition assigned to that action.

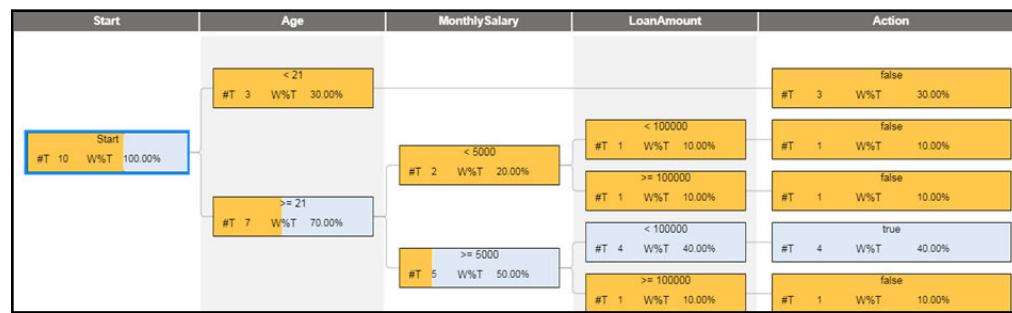


Figure 27: The Tree view with Color by Action selected

To show action proportions in your decision nodes in **Tree view**, you must first run profiling on your decision tree. Proportions only appear on leaf nodes with assigned actions and their parent nodes. By default, to optimize tree editing performance, action proportions are not enabled.

 **Note** Certain tree edits, such as cutting a subtree or copying and pasting a subtree, removes the statistics in the affected decision nodes. Assigning actions and recounting the tree restores action proportions in these nodes.

By default, the **Counts-labels** node view displays the information on the leaf nodes. The **Color by Action** option also displays in the **Counts** node view.

- 1 Open the **Views** drop-down menu.
- 2 Select **Tree view**.
- 3 Select **Counts** or **Counts-labels**.
- 4 Select **Color by Action**.
- 5 Hover over the colored bars to view the percentage of that condition that is assigned to a specific action.

 **Note** The action value and its percentage appear in the tooltip. Action colors appear from left to right by their rank, with any unassigned proportion, if any, displaying to the far right on the node with no color. Modified action ranks automatically result in the reordering of the proportion node colors.

Analyzing Decision Tree Profiling Results

Use the **Profile view**, **Leaf view** and **Action summary** to analyze the profiling results.

Example of Using the Profiling Results

Suppose you have a decision tree for evaluating applicants for a loan.

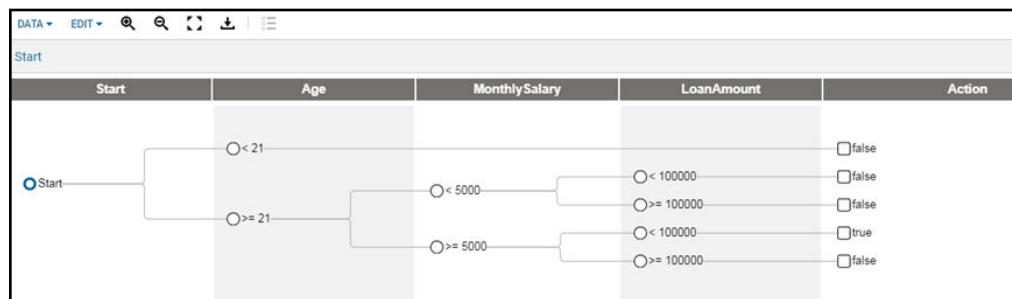


Figure 28: Decision tree for evaluating applicants for a loan

In order to run decision tree profiling, you create a CSV file with the following values.

	A	B	C
1	Age	MonthlySalary	LoanAmount
2	21	5001	99000
3	20	3000	100001
4	19	2900	880000
5	25	5001	89000
6	19	5400	88000
7	40	6000	60000
8	36	5001	70000
9	50	10000	300000
10	33	4999	49999
11	47	4000	199999

Figure 29: Decision tree profiling input data values

Profile view

After decision profiling is run, the **Profile view** displays the results as a file directory the Start and parent nodes represented as folders, the leaf nodes represented as files, and the statistics and action node values displayed in columns.

Name	Order	Action	#T	W%T
Start	1		10	100.00%
Age < 21	2	false	3	30.00%
Age >= 21	3		7	70.00%
MonthlySalary < 5000	4		2	20.00%
LoanAmount < 100000	5	false	1	10.00%
LoanAmount >= 100000	6	false	1	10.00%
MonthlySalary >= 5000	7		5	50.00%
LoanAmount < 100000	8	true	4	40.00%
LoanAmount >= 100000	9	false	1	10.00%

Figure 30: Profile view of decision tree profiling results

The statistics provide feedback on how many records in the dataset were processed by the decision tree nodes using a count and a percentage of the total number of records. If you run the profiler using the values similar to the dataset

above, the information in the **#T** column and **W%T** column can be analyzed in the following way:

- At the **Start** node you see that there are 10 rows that were evaluated representing 10 counts and 100% of the total records in the dataset.
- The **Age < 21** node evaluated 3 rows (3, 4, and 6) that resolved to False representing 3 counts and 30% of the total number of records. When the nodes resolve to an action value of True or False, the processing does not continue.
- The **Age >= 21** node evaluated 7 rows (2, 5, 7, 8, 9, 10, and 11) representing 7 counts and 70% of the total number of records. These 7 rows were evaluated further:
 - The **MonthlySalary < 5000** node evaluated 2 rows (10 and 11) representing 2 counts and 20% of the total number of records. These 2 rows were evaluated further:
 - The **LoanAmount < 100000** node evaluated row 10 that resolved to False representing 1 count and 10% of the total number of records.
 - The **LoanAmount >= 100000** node evaluated row 11 that resolved to False representing 1 count and 10% of the total number of records.
 - The **MonthlySalary >= 5000** node evaluated 5 rows (2, 5, 7, 8 and 9) representing 5 counts or 50% of the total number of records. These 5 rows were evaluated further:
 - The **LoanAmount < 100000** node evaluated 4 rows (2, 5, 7, and 8) that resolved to True representing 4 counts or 40% of the total number of records.
 - The **LoanAmount >= 100000** node evaluated row 9 that resolved to False representing 1 count or 10% of the total number of records.

If you edit the decision tree while in profiling mode, you can click **Recount** to update the counts and percentages. You can also edit the dataset and select **Data > Running profiling data** to upload it and run the profiler to see new results based on your edited values.

Leaf view

In **Leaf view** you can see the same statistics from the perspective of the leaf nodes however, the **#T** (Total number of raw counts) and **W%T** (Weighted percentage of total counts) reflected in this view are the same as those in Profile view :

Name	Order	Action	#T	W%T
Age < 21	2	false	3	30.00%
LoanAmount < 100000	5	false	1	10.00%
LoanAmount >= 100000	6	false	1	10.00%
LoanAmount < 100000	8	true	4	40.00%
LoanAmount >= 100000	9	false	1	10.00%

Figure 31: Leaf view of decision tree profiling results

Action summary

In **Action summary** you can also see the same statistics from the perspective of the action value nodes. The action expression nodes are not visible in this view.

-  **Note** If the Action is of date type, statistics will not display.

Action	#T	W%T
false	6	60.00%
true	4	40.00%

Figure 32: Decision tree profiling results Action summary

Recounting Decision Trees

Recounting the decision tree profiling statistics using the existing dataset ensures that the most recent statistics display.

By default in Tree view, the **Counts-labels** node view displays the set of statistics from the most recent profiling run. You can also select **Counts** node view to display statistics without the labels.

When you edit the decision tree, the **Recount** icon becomes enabled. If there are no significant decision tree edits that would prevent you from using the existing dataset, you can use **Recount** to recalculate the statistics for your tree.

-  **Note** If your decision tree has significant changes, such as a datatype change to a decision tree level or a new column, you may see an error message asking you to replace the dataset file. If this is the case, you need to make changes to your dataset and repeat the steps to run the profiler.

- 1 Click **Recount** in one of the tree views.
- 2 (Optional) View the updated statistics in any of the tree views.

Exporting a Decision Tree

Export a decision tree as a FICO Strategy Markup Language (FSML) 2.5 file to exchange tree models with other FICO products. You can import a FSML model as a decision tree, make some edits to the imported decision tree and then export the decision tree as an FSML file to import it to another FICO product.

Use the **Download tree** icon on the Decision Tree toolbar to export decision tree to a FSML 2.5 compliant XML file. Only types that are supported by FSML can be successfully exported. The exported file includes any levels, splits and nodes used in the decision tree as well as any unused variables; however, the generated FSML file does not include the description text, return type, or parameter statements.

The following table explains how decision tree property types are converted when the decision tree is exported as an FSML file.

Table 30: Conversion of Decision Tree Property Types

Parameter, Variable or Pattern Property Type	Converted Property Type
String	String
Boolean	String
Real	Integer
Integer	Integer
Enumeration	String

Unsupported types are handled in a couple of different ways. Most unsupported types are automatically converted to the categorical type. For example, if your decision tree contains a boolean type, the boolean type is converted to a categorical type with true and false values. Expressions are not supported in FSML, so if your decision tree contains expressions, you will have to remove all expressions from the decision tree before you can export. Some unsupported types such as *date* type cause the export process to fail and are not converted.

 **Note** If you export a decision tree as an FSML file and later import the file as a FSML Decision Tree, you need to be aware of the following behavior:

- FSML only supports actions of string value list or enumeration type therefore when a decision tree containing other action types is exported, the actions are converted to a string value list.
- If you have conditions of string type in a decision tree, upon export to a FSML file the conditions are converted into the categorical type with a string value list or enumeration. If the FSML file is imported as a FSML Decision Tree, you will find that each split contains all the values available for a given condition level because FSML only supports categorical strings.
- If you assign a value to only one of your action nodes prior to exporting the entity as a FSML file, upon re-import you cannot modify or add any new values to your decision tree actions. This occurs because the action type

is converted to a string value list. To modify or add any new action values, you need to first add the new values to the value list.

Related Links

[Decision Tree Limitations](#)

Decision Tree Limitations

These limitations exist for the decision tree in this release.

- The money, duration, time, and timestamp data types are not supported for condition and action variables.
- The words, `action`, `start`, `input`, and Any other value are reserved words in the Decision Tree editor. If your object model includes properties with these names and you want to use those properties as decision variables in your decision tree, change those properties names in your object model and then reconfigure your project with the updated object model before creating a decision tree. Note that if you update the object model, this may impact the decision entities that you have already created.
- In a generated report, a picture of the decision tree is not available.
- The decision tree is not supported for use with comparison queries.
- The decision tree is only partially supported with the Verifier. When the Verifier is run on a decision tree, the Usage Tests are not run.
- Group separators are not available for integer and real values used in condition or action nodes.

Related Links

[Limitations in Comparison Queries](#)

[FSML Decision Tree Import Limitations](#)

CHAPTER 9

Authoring Rules in Scorecards

A scorecard is a graphical representation of a mathematical formula used to predict outcomes, typically for business customers or prospects, based on facts known about them in the present. A target field is defined when creating a scorecard, which is the outcome to be predicted. Scorecards can predict two types of outcomes:

- The likelihood that an event will happen or will not happen (binary outcome).
- The amount of an event (continuous outcome).

A scorecard generates a score that is the calculation of the predicted outcomes. Scoring the likelihood that an event will happen results in a number that is proportional to the likelihood that the event will or will not occur. Scoring the estimate of a future amount results in that predicted value. The following examples describe these different types of scores:

- Your credit score is proportional to the likelihood that you will be able to pay your financial obligations on time. The higher your score, the greater the likelihood that you will pay.
- You can generate a score that represents the predicted revenue in the next 6 months.

The scorecard uses known facts about the customer or prospect as its calculation criteria. The known facts are referred to as scorecard *variables*. A variable can be a numeric value, a string value, or an item from a pre-defined list of alternatives.

Each variable contains one or more *bins*. At runtime, each bin in a variable evaluates the data value passed to the variable and when a value matches a bin, a partial score is assigned to that variable. The value used for the partial score is contained in the **Partial score** cell in each bin.

A scorecard is additive, which means it calculates the sum of all partial scores assigned to variables to produce an overall score. This overall score has predictive power and represents an assessment, opinion, or prediction about the customer or prospect.

Related Links

[Limitations in Comparison Queries](#)

[Creating Scorecards from PMML Files](#)

[Reason Code Lists](#)

Reason Code Lists

In addition to returning a score, you can configure a scorecard to return a reason code for each bin that is fired. Reason codes provide users with reasons as to why a particular score was assigned during evaluation.

A reason code list contains a rank, a name, a code, and a string message. A reason code list is created using the Reason Code List editor. When a reason code is created in a rule project, an instance of the NdReasonCode support class is generated. You can write a print statement in a function that displays the reason code name, reason code, and messages resulting from an evaluation.

You can add a reason code list to a scorecard and set options and methods for determining which reason codes are returned.

In the Reason Code List editor, you can create as many reason codes as needed so that your business users can select a reason code name from a drop-down list. When a reason code name is selected in the Scorecard editor, the corresponding reason code message is also added. When you create a reason code list, the reason codes are listed in priority order with the most important reason codes appearing near the top of the list. Although priority order or rank is the default for returning reason codes, you can choose from among several other options and return methods.



Note You should consult with legal counsel when using a reason code list to ensure that it complies with legal requirements.

Creating Reason Code Lists

Use the **Reason code list** command on the **New** menu to create a reason code list. You can refer to the list from the Scorecard editor.

- 1 In the **Project Explorer**, select the folder where you want the reason code list created.
- 2 Click the **New** drop-down menu.
- 3 Select **Reason Code List**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 Click **Create**.

Adding Reason Codes to a List

Create one or more reason codes for the reason code list.

- 1 Locate the reason code list in the **Explore** pane or the **Project Explorer**.
- 2 Click the reason code list to open it in the editor.

- 3 Perform one of the following operations to enable the reason code list for editing:
 - If the scorecard is not checked out, click the **Check out** icon.
 - If the scorecard is already checked out, click the **Edit** icon.
- 4 Scroll down to the **Reason codes** table.
- 5 Click the **Add** icon to add a reason code to the list.
- 6 Click the name to open the reason code.
The rank number is determined by the order in which the reason code appears in the **Reason codes** table. You can change the order by using the **Move Up** and **Move Down** commands on the **Reason codes** table.
- 7 (Optional) Enter a description of the reason code.
- 8 In the **Message** field, enter the text that you want to appear when this reason code is returned.
- 9 Click the **Parent View** command to return to the Reason Code List editor.
- 10 (Optional) Repeat step 5 to create additional reason codes.
- 11 Click the **Save** icon or click the **Check In** icon.

Creating Scorecards

Create a scorecard to author a set of bin expressions that will be evaluated based on the data passed into the project.

- 1 In the **Project Explorer**, open the **New** drop-down menu.
- 2 Select **Scorecard**.
- 3 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 4 Click **Create**.

Related Links

[Naming Decision Entities and Folders](#)

Defining Scorecards

The Scorecard editor contains the following fields:

Reference name	A unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.
Description	(Optional) A summary of what the scorecard evaluates.

Return type	By default, NdScoreReturnInfo is the return type. This is a built-in Blaze Advisor support class which contains the score, any reason codes, the variable count, and additional data about the calculations performed by the scorecard for each transaction.
Initial score	A numeric value that is used to set the minimums score of the scorecard.
Allow expressions	Select this check box if you want to enter an SRL function or a numeric value in the Partial Score column for each bin row in a scorecard.
Precalculate score	Select this check box if you want the score and score reasons to be calculated within the final bin (rule) in the scorecard.  Note If you will be using Decision Testing, you need to select this option to see the reason codes in the test results.
Possible reason codes	Reason codes provide users with meaningful reasons why a particular score was assigned during evaluation. A reason code usually contains a rank, a name, a code, and a string message. If you have created one or more reason code lists, you can select one from this drop-down list.
Number of reason codes to return	If you have selected a reason code list, specify an integer value for the number of reason codes that will be associated with each bin (rule) that fires after the scorecard rules are evaluated. One or more reason codes are returned with the scorecard value. The default number of reason codes is 1; however, you can enter any value as long as it does not exceed the number of variables in your scorecard.
Method for including reason codes	If you have selected a reason code list, use the default settings for returning reason codes or use one of the calculate methods from the drop-down list to determine how the reason codes are returned.
Parameters	Create parameters based on your object model. Parameters are used to pass values to the variables in the scorecard at runtime. Parameters can be a class type, string, real, integer, or enumeration type. A parameter is a graphical way of declaring the types and names of any argument values or objects to be passed to a ruleset.
Variables	A variable is a known fact about the customer. It can be a numeric value, a string value, an item from a pre-defined list of alternatives, or a property of a Java or XML class.
Bins	The possible values and the rule conditions for a variable.

Setting the Precalculate Score Option

The Precalculate score option is used when you want change how the score and score reasons are calculated during processing.

By default, the values you enter in the columns for a bin are used to generate the SRL rule conditions. Unlike other metaphors, such as decision tables or decision trees, where you set values for rule conditions and actions, in a scorecard you set the properties and values for rule conditions and at the end of scorecard

processing, the partial scores for each characteristic are summed to return an overall score.

If you select the Precalculate score option, the final rule of the scorecard checks if the option is set in `NdScoreModelReasonCalculationOptions` and ensures that the score and the score reasons are calculated before the end of scorecard processing. If you do not select the Precalculate score option, when score processing is completed, `NdScoreModelReturnInfo` returns the characteristics and if you want to also return the top reasons, you need to explicitly call the `calculateReasons()` method.

Select the Precalculate score option when testing a scorecard entry point where you want to return reason codes in Decision Testing.

If you select the Precalculate score option, be aware that the performance of your scorecard may be negatively affected.

Related Links

[NdScoreModelReturnInfo](#)

[NdScoreModelReasonCalculationOptions](#)

Writing Rules in Scorecards

Enter values for the variables and their bins to create rule conditions. Each bin within a variable is a rule condition. When the input matches one of the bins, the partial score of the matching bin is assigned to the variable. At the end of scorecard processing, the partial scores of each variable are summed to return an overall score. The resulting score can be used by other decision entities.

Variables

Every scorecard must have at least one variable. A variable is a known fact about the customer. It can be a numeric value, a string value, an item from a pre-defined list of alternatives, or a property of any Java or XML class.

Variables are defined by its **Baseline Score**, **Type**, and **Bins**. A **Description** can also be added to summarize what the variable evaluates.

- The **Type** is the data type of the variable and determines the bin values that can be added.
- The **Baseline Score** is used to determine the reason code for the variable. Each variable is divided into bins, which represent the possible values and rule conditions of the variable.
- The format of the **Bins** can vary from bin to bin within the same variable.

Adding Variables

Add variables in a scorecard by selecting them from the **Add Variable** dialog.

To add a variable to the scorecard, make the scorecard editable, click **Add Variables** to open the dialog and select a variable. Optionally, the **Display Name** field can be used to enter a user-defined name for the variable.

Deleting Variables

Delete a variable from a scorecard. If you want a record of a variable and its values before you delete it, you can use the **Print** icon in the editor. Note that there is no Undo command for delete operations.

- 1 Select the check box next to the header row of the bin table for the variable you want to delete.
- 2 Click **Delete Variable**.

Baseline Score

The baseline score can be the maximum, the minimum, the mean, or an arbitrary value that defines a typical point of comparison of the partial scores in the variable. Each variable can have a different baseline score. If you are not using a calculation to return reason codes, you do not necessarily need to enter a value in this field.

If a partial score for a variable is generated that is far from the baseline score, then calculating the reason code will indicate why the total score is higher or lower than expected. For example, if the baseline score of a variable was 50 and the partial score determined was 35, then the reason code would return why 15 points were lost from the total score.

Bins

A variable is divided into bins. Bins define the possible values and the rule conditions for a variable.

For example, an Age variable might contain bins for Young Adult, Adult, and Senior age ranges. Bins are configured so that a variable value can fall into only one bin. The **All Other** bin for each variable ensures that all possible values are covered.

Add one or more bins to a variable. The values entered in the bin cells are used to generate the rules. The numerical values are automatically sorted from the greatest to the least (top to bottom).

Table 31: Bin Columns

Column	Description
Values	Enter a value that will be used to evaluate the incoming data. This can be a range of values, string values, or enumeration values.

Table 31: Bin Columns (continued)

Column	Description
Label	(Optional) Enter a text string to describe the evaluation being performed. This text is for internal use only.
Partial Score	Enter a real number for the partial score. The number of decimal places that are displayed in a Scorecard editor depends on the design of the scorecard. The default behavior is to display numeric values up to six decimal places. If you are entering a numeric value of type <code>real</code> , it must be a 64-bit floating-point number. If the Allow Expressions option is selected, you can enter SRL for the partial score.
Unexpected	Select the check box in the Unexpected cell when you need to track the occurrence of unexpected values. By default, the All Other bin is generated and the Unexpected check box is selected.
Reason code	(Optional) The Reason Code column appears only if a reason code list has been added to the scorecard.

Adding Bins

There are various ways to add bins to a scorecard. You can add multiple bins at a time, bins with ranges, categorical bins, and multi-value bins. After a bin has been added, you can edit the value and operator in the cell.

Multiple Bins

You can add multiple bins at one time by entering the bin values separated by semicolons (;). These values can be numeric or string values. For example, entering "100; 200; 300" creates the following bins:

Values	Label (optional)	Partial score	Unexpected
100		0	<input type="checkbox"/>
200		0	<input type="checkbox"/>
300		0	<input type="checkbox"/>

Figure 33: Scorecard with multiple bins

Range Bins

You can add bins with range values by entering the bin values separated by a space. These can be numeric values only. For example, entering "5 10 15" creates the following bins:

Values	Label (optional)	Partial score	Unexpected
5 <=..< 10		0	<input type="checkbox"/>
10 <=..< 15		0	<input type="checkbox"/>

Figure 34: Scorecard with range bins

 **Note** Because a space indicates a range value, spaces between numbers is not supported. If your locale uses a space for the thousand separator, do not include the space. For example, enter "10000" instead of "10 000".

After the bins are added, you can change the operator to one of the following:

- \leq
- $<$
- \geq
- $>$

Categorical Value Bins

Categorical bins are string values. You can add bins with a single value or with multiple values. Select a bin value in the **Available Values** drop-down menu to add a single value bin. To add a single bin with multiple variables, select all of the desired values in the drop-down menu.

Multi-Value Bins

Add a multi-value bin by entering the bin values separated by a vertical bar (|). These can be numeric or string values. For example, entering "10 15 | 20 25" creates the following bin:

Values	Label (optional)	Partial score	Unexpected
10 <=..< 15		0	<input type="checkbox"/>
20 <=..< 25			

Figure 35: Scorecard with a multi-value bin

Deleting Bins

Delete one or more bins from a variable.

- 1 Select the check box next to a bin row.
You can select one or more rows at the same time.
- 2 Click **Delete Bins**.
When you delete a bin, all its settings, its label, partial score or reason code are also deleted.

Tracking Unexpected Range Values

Use the **Unexpected** check box in a scorecard to test the validity of a resulting score.

If you mark one or more bins in a variable as **Unexpected** and at runtime a value matches a bin marked as Unexpected, a counter is incremented. The existence of unexpected values may indicate that the resulting score is invalid.

By default the **All Other** bin in each variable has the **Unexpected** check box selected. If a value that is passed to a variable cannot be evaluated by any of the bins, then the partial score in the **All Other** bin is assigned and the occurrence of an unexpected value is tracked.

Validating Scorecard Values

The scorecard values are automatically validated when a variable or bin is added or edited.

The scorecard detects the following types of errors:

- Validation errors
 - For example, changes to a parameter name or type used in a variable will throw an error.
- Data type errors
 - For example, a string value is entered as a Bin value that allows only real values.
- Overlapping value errors.
 - For example, the value ≥ 3.8 of a bin overlaps with the values $3.6 \leq \dots < 4$ of a second bin.
- Duplicate expression errors
 - For example, two bins use the same expression format and the same range value, such as ≥ 4 .
- Upper bound value is less than the lower bound value
 - For example, a range has the lower and upper bound values of $3.6 \leq \dots < 2$.

If an invalid value is detected, you see the following error indicators:

- The variable name is marked in red and a tooltip displays the reason for the error.
- The bin values causing the error are highlighted in red.
- When entering a bin value, the text box is highlighted in red and a tooltip appears with the reason for the error.

Methods for Returning Reason Codes

If you associate a reason code list with your scorecard, you can select a method how reason codes are returned.

By default if a reason code list is associated with a scorecard, at runtime the reason codes are returned by their reason code list rank (**Use default rank as defined in the reason code list**). Alternatively, if you are not using the **Allow expressions** option, you can also use a calculation based on the minimum or maximum score value in a scorecard variable to return reason codes.

Each variable in a scorecard can have a different minimum or maximum partial score.

- **Calculate by distance from minimum value**

The minimum score is the lowest possible partial score you entered into the Score column for a scorecard variable. For example, if the minimum bin score in your variable was 10 and a variable was passed a value that fired a bin with the score of 20, the equation based on the minimum score method would be: `score reason distance = minimum score - score`. For this example, the equation with values would be: `score reason distance = 10 - 20`, resulting in a score reason distance of -10.

- **Calculate by distance from maximum value**

The maximum score is the highest possible partial score you entered into the Score column for a scorecard variable. For example, if the maximum bin score in your score card was 50 and a variable was passed a value that fired a bin with the score of 20, the equation based on maximum score method would be: `score reason distance = maximum score - score`. In this example, the equation with values would be: `score reason distance = 50 - 20` resulting in a score reason distance of 30.

- **Calculate by points from mean**

You can enter a value for the Baseline score for each variable in the scorecard. The baseline score value is usually provided by your modelers and is considered to be a neutral value for score purposes. For example, if the baseline score in your variable was 30 and a variable was passed a value that fired a bin with the score of 20, the equation based on the points from mean method would be: `score reason distance = baseline score - score`. For this example, the equation with values would be: `score reason distance = 30 - 20` resulting in a score reason distance of 10.

If you are using the **Allow expressions** option, you can select the **Use default rank as defined in the reason code list** or the **Calculate by points from mean** method.

CHAPTER 10

Importing PMML Models

Import a Predictive Model Markup Language (PMML) model to use its contents to augment your decision logic.

Developed under the auspices of the [Data Mining Group](#), PMML is an open standard XML-based markup language that provides a way for applications to define statistical and data mining models and to share models between PMML-compliant applications. It allows users to develop models within one vendor's application, and use other vendors' applications to execute, visualize, analyze, or evaluate the models.

Creating Scorecards from PMML Files

Create a scorecard model by importing a scorecard PMML file with the **Import PMML Scorecard** wizard.

 **Note** A scorecard PMML file must comply with the [PMML 4.2.1 Specification](#) in order to be imported with the **Import PMML Scorecard** wizard.

Scorecard models are widely used in the financial industry for their interpretability and ease of implementation, and because the model variables can be mapped to a series of reason codes which provide explanations of each score.

When a PMML scorecard is imported, the model is rendered as a scorecard in the project. If a reason code list is detected during the import operation, a reason code list is also generated. You edit the scorecard and assign reason codes in the Scorecard editor. You can edit the reason code list in the Reason Code List editor.

- 1 In the **Project Explorer**, select the folder where you want to import the PMML scorecard model.
- 2 Click the **Import** drop-down menu and select **PMML Scorecard Model**.
- 3 On the first page of the **Import PMML Scorecard Model** wizard, drag and drop a .pmml, .xml, or .zip file onto the drop zone or click **local file browser** or **FICO Drive** to browse for files to add.

 **Note** A .zip file can contain one or more valid scorecard .xml or .pmml files. Each of the files must be smaller than 100 MB.

- 4 Click **Next**.

- 5 In the **Select Entities** page, complete the following substeps:
 - a All the entities are selected by default. If you already imported the PMML model and did not change the object model, you can clear the **ModelIn** class check box.
 - b Optionally, you can select the following options:
 - **Overwrite entities in the current project**
 **Tip** Clear this option if:
 - The name of the scorecard or reason code list is the same as an existing one in your project, and you want to create a new scorecard and reason code list. Otherwise, the existing entities will be replaced by the ones in the PMML file.
 - You want to merge the reason code list in the PMML file with the reason code list that is already in the project. If this option is selected, the existing reason code list will be deleted and a new reason code list will be imported.
 - **Add entities to subfolder**
 **Note** By default, the subfolder name is the same as the model name and the model is created in the subfolder. If the check box is cleared, the entities are created outside of a folder.
 - **Add Reason Code List to subfolder**
 **Tip** If your PMML file contains a reason code list, select this option if you want the reason code list to be imported in the same folder as the scorecard. Otherwise, the reason code list will be imported outside of the folder.
 - **Use model name as prefix when generating entity names.** This creates unique names for each imported entity.
 - c Click **Next**.
- 6 The **Import Summary** page displays a read-only report of the selections made in the previous wizard pages. Click **Done** to import the PMML file.

When a PMML Scorecard Model is imported, the following entities are created in a folder:

Entity	Name	Description
Model Instance file	ModelName_Scorecard	This is the main function used to invoke the model in the project. The Model Instance is rendered as a scorecard.
Input Business Term Set	ModelNameIn	Contains the input type of the scorecard.
Reason code list (Optional)	ReasonCodeListName	A reason code list is only imported in the folder if the scorecard contains a reason code list and the Add Reason Code List to subfolder option is selected.

The scorecard, business term set, and reason code list are automatically checked in. You will need to check them out first in order to edit them.

Related Links

- [Authoring Rules in Scorecards](#)
- [Validating Scorecard Values](#)

Examples Decision Services with a PMML Scorecard Model

To see how a PMML Scorecard appears in a decision service, download one or both of these ZIP files. This is a description of each example:

-  **Note** Contact FICO Support for the sample ZIP files.
- **PMMLScorecardWithMappedBusinessTerms.zip**
This example shows how to use the business term set editor to map the business terms to the corresponding object model properties.
 - **PMMLScorecardWithMappedFunctions.zip**
This example shows how to map the scorecard's input business term set to the input object model using a function.

Each of the ZIP files contains the following:

- A deployable decision service containing a PMML Scorecard.
- The `SocialMediaScore_Scorecard.pmm1` file.
- The `SportsFanPromotion.xsd` that contains the XML Business Object Model (BOM).
- The `SportsFanDataset.csv` file that contains values you can use to test the output of the model.

Importing the Decision Service

- 1 Download the ZIP file to your desktop and unzip it. If you cannot download the file, contact FICO Support.

 **Important** Do not unzip the enclosed ZIP file with `DecisionService` in the file name.
- 2 Create a workspace in Decision Modeler.
- 3 Import the project by uploading the ZIP file with `DecisionService` in the file name.
 - a Select the workspace and click **Import Projects**.
 - b Click **Local File Browser** and select the ZIP file with `DecisionService` in the file name.
 - c Click **Next** and select the project name.

- d Click **Next** and then click **Import**.

Testing the PMML Scorecard Model with Decision Testing

You can test the PMML Scorecard Model by running sample data through it using Decision Testing.

To run the test data:

- 1 Open the project and expand the **Test** pane.
- 2 Under the **Test** pane, click the **Decision Testing** link to open the **Decision Testing** panel.
- 3 In the drop-down menu, select **processWithDecisionFlow(SportsFanPromotion):SportsFanPromotion**.
- 4 Upload the **SportsFanDataset.csv** file.
- 5 Click **Run Test**. After the tests are run, you can download the results as a CSV file on to your desktop.

Related Links

[Dataset Requirements](#)

Recreating an Example

When you download an example from the Help system, the ZIP file not only contains the example, but it also contains an XML schema and the original model file that you can use to recreate the example yourself.

Download one of the examples and unzip the contents. If you cannot download an example, contact FICO Support.

These are the steps required to recreate one of the examples with an imported model.

- 1 Create a project.
- 2 Configure the project using the XSD file that was included in the ZIP file.
- 3 Import the model that was included in the ZIP file using the wizard.
- 4 Map the input class properties for the imported model to the external object model. You can do this using the Business Terms editor or using a function.
The examples show how to map the input class properties to the external object model.
- 5 If the mapping was done with the Business Terms editor, initialize the business term set(s) using the **Initialization entity**.
- 6 Add one or more tasks in the decision flow. The number of tasks that you add depends upon the model type.
The examples show how to setup your task(s) on a decision flow.

- 7 Compile the project.
Fix any errors before moving on to the next step.
- 8 Use the CSV file to upload the dataset and run values through the project.

Re-importing a PMML Scorecard Model

You can use the Import PMML Scorecard model wizard to re-import your PMML Scorecard model.

When a model is imported again, the generated entities can overwrite the existing entities so long as the check boxes next to the model and the business term set are selected in the wizard. You do not have to delete these entities prior to re-importing your PMML Scorecard model.

 **Note** Models imported prior to Decision Modeler 3.0 have a prefix of the for the global variable name of the business term set. As of Decision Modeler 3.0, the prefix is now var so when you re-import the model, the prefix for the business term sets will be var. Compiling the project will generate errors unless you update the business term mappings. For example, `the SocialMediaScore_ScorecardIn.audienceComposition = input.SocialMedia.audienceComposition` must be changed to `var.SocialMediaScore_ScorecardIn.audienceComposition = input.SocialMedia.audienceComposition`.

Follow these steps to re-import a model in your project:

- 1 Select the folder where you want to re-import the model.
- 2 Import the model using the wizard. Clear the check box next to the business term set if you do not want a new business term set created.
- 3 By default, the model and the business term sets are generated in a subfolder. Clear the **Add entities to subfolder** check box if you do not want this behavior.
- 4 Click **Next**. Click **Done**.
- 5 If you chose to import a new business term set with the model, you will need to update the mapping of the business terms to the external object model.
 - If you mapped the business terms using the Business Terms editor, you will need to redo the mapping for each business term.
 - If you mapped the business terms using a function, the same function will continue to compile and run as long as the business terms did not change. If you removed any business terms, you must remove the business terms from the existing mapping function. If you added any new business terms, you must add the business terms mapping to the function.
- 6 If the mapping was done using the Business Terms editor and your project includes an Initialization entity, initialize the business term sets using the Initialization entity.
- 7 Compile the project to ensure that no errors were introduced into the project.

Importing Tree Ensemble Models with PMML Files

Import a Tree Ensemble model using the **Import PMML Mining Model** wizard.



Note A PMML Mining Model must comply with the [PMML 4.1 Specification](#) in order to be imported using the **Import PMML Mining Model** wizard.

A Tree Ensemble model is a type of predictive model that produces a score. It combines predictions from many models to arrive at their ultimate prediction or estimation for a given record.

Tree Ensemble models use a weighted average of the predictions from many simple decision trees to make predictions. They are constructed iteratively. In the first iteration, a single decision tree is trained and evaluated. In each subsequent iteration, another decision tree is trained with more emphasis on the rows that were poorly predicted by the weighted ensemble of the trees built thus far.

- 1 In the Project Explorer, select the folder where you want to add the model.
- 2 Click the **Import** drop-down menu and select **PMML Mining Model**.
- 3 On the **Import PMML Mining Model** page, drag and drop one or multiple .pmml, .xml, or .zip files onto the drop zone or click **local file browser** or **FICO Drive** to browse for files to add.



Note If a .zip file is selected, the .pmml or .xml files will be extracted. The extracted files must validate and be smaller than 100 MB in order to be imported.

- 4 Click **Continue**.

All of the decision entities that will be generated for the models are displayed.

- 5 Click **Done** to import the models.

When a PMML Mining Model is imported, the following entities are created in a folder:

Entity	Name	Description
Function	calculate modelName	This is the main function used to invoke the model in the project.
Input Business Term Set	modelNameIn	Contains the input type of the main model function. If changes are made to the business terms, the function may not compile.
Output Business Term Set	modelNameOut	Contains the output type of the main model function. If changes are made to the business terms, the function may not compile.

Related Links

[Mapping the Business Terms to Object Model Properties](#)

Invoking the Tree Ensemble Model

To execute a Tree Ensemble model in a project, you must call the function containing the model. You can call the function from another decision entity or add the function as a task in a decision flow. When you call the function from another decision entity, you must call it using the SRL name, which is displayed in the **Reference name** field in the Function editor.

To call the function using a decision flow, create a task and associate the task with the function.

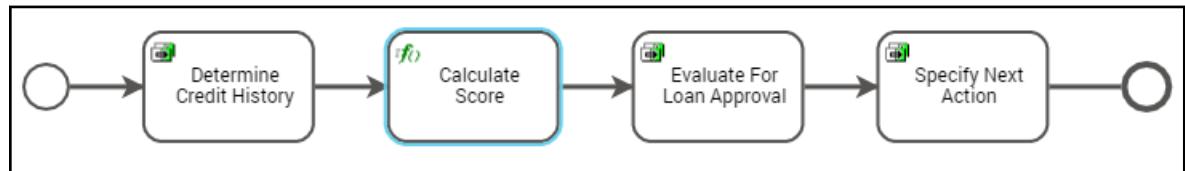


Figure 36: Decision flow with the task associated with the function highlighted

The value of the task parameter must evaluate to the same type as the *input* business term set, and the value of the return type must evaluate to the same type as the *output* business term set, as shown in this figure:

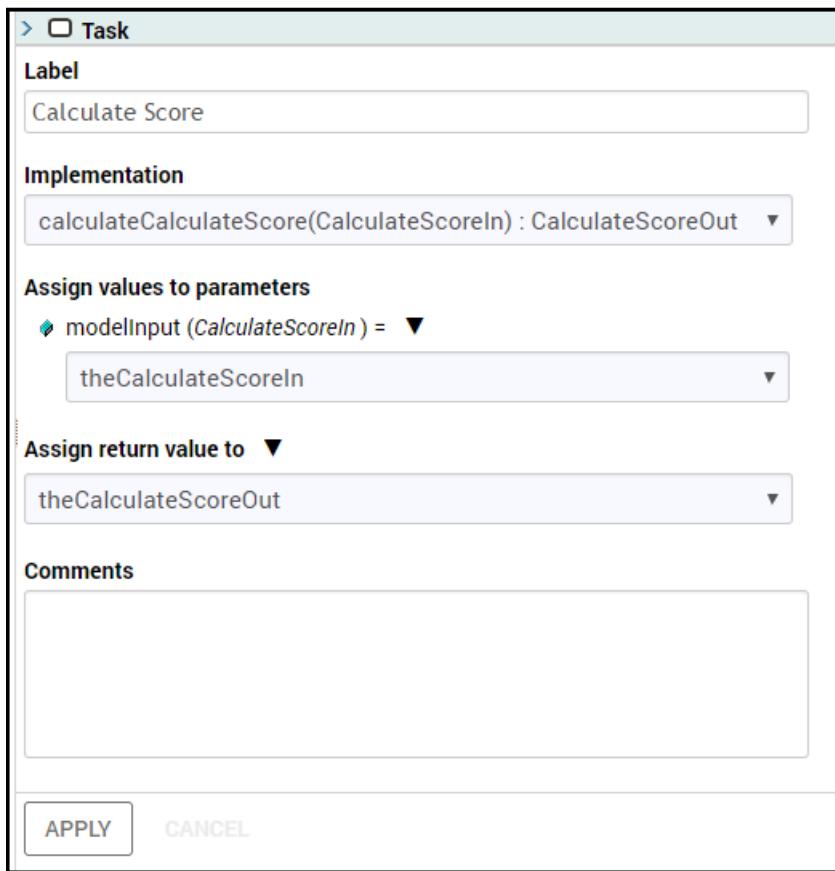


Figure 37: Task details for the Calculate Score task

Supported PMML Mining Models

The **Import PMML Mining Model** wizard supports importing mining models that conform to PMML 4.1 Specification. There are two kinds of mining models: Segmentation models and Model Composition models.

 **Note** Only Segmentation models are supported. Model Composition has been deprecated by the Data Mining Group.

Segmentation models consist of two or more models that are scored separately and then the scores are combined using a "multipleModelMethod". Only Tree Ensemble Models are supported and can be part of a segmentation model. If a model other than Tree Ensemble is in the PMML file, an error is thrown.

 **Note** If you have a PMML mining model that does not fit the criteria outlined in this section, you can still import the model using the **Import Other PMML Models** wizard.

Table 32: PMML Model Support

Product Name	Model Types Supported
Decision Modeler 2.x or later	Scorecard models that comply with the PMML 4.2.1 Specification.

Table 32: PMML Model Support

Product Name	Model Types Supported
	Segmentation tree ensemble models that comply with the PMML 4.1 Specification.

FICO supports the following methods:

- Regression function models
- sum
- average
- weighted-Average
- median
- max
- modelChain
- selectFirst
- selectAll
- Classification function models
- majorityVote
- weightedMajorityVote

 **Note** Model types and methods not listed are not supported.

Model Builder Models

Model Builder produces PMML that conforms to the Mining model specification, specifically GBDT models (Gradient Boosted Decision Tree) and Random Forest.

Size of the Segmentation Model

Segmentation models can contain any number of models; however, there is a practical limit as to the size of a model that can be imported successfully. For example, if a model file cannot be opened in a typical text editor, it may be too large to be imported. The maximum file size is 100 MB.

Runtime Performance

A model inside a segmentation model will be scored depending on whether the condition for that model is satisfied. In the case of a GBDT (Gradient Boosted Decision Tree) model, the condition is true, each model will be scored. The score of each model is summed and then returned. In this case, assuming the size of each model is relatively equal, the runtime performance is approximately linear and depends on the number of models in the segmentation model.

Example Decision Services with a Tree Ensemble Model

To see how a Tree Ensemble Model appears in a decision service, download one or both of these ZIP files. This is a description of each example:

-  **Note** Contact FICO Support for the sample ZIP files.

- **ExampleWithMappedBusinessTermSet.zip**
This example shows how to use the business term set editor to map the business terms to the corresponding object model properties.
- **ExampleWithMappedFunctions.zip**
This example shows how to map the business terms to the corresponding object model properties using a function.

Each of the ZIP files contains the following:

- A deployable decision service containing a Tree Ensemble Model.
- The `CalculateScore.pmm1` file that was generated using FICO® Analytic Modeler Scorecard. This is a Tree Ensemble Model that determines the likelihood a customer will pay back a loan based on their financial history.
- The `LoanApplication.xsd` file that was used to create the XML object model.
- The `CalculateScoreData.csv` file that contains values you can use to test the output of the model.

Related Links

[Mapping the Business Terms to Object Model Properties](#)

Importing the Decision Service

To view the imported model in a decision service, do the following:

- 1 Download the ZIP file to your desktop and unzip it. If you cannot download the file, contact FICO Support.

 **Important** Do not unzip the enclosed ZIP file with `DecisionService` in the file name.
- 2 Create a workspace in Decision Modeler.
- 3 Import the project by uploading the ZIP file with `DecisionService` in the file name.
 - a Select the workspace and click **Import Projects**.
 - b Click **Local File Browser** and select the ZIP file with `DecisionService` in the file name.
 - c Click **Next** and select the project name.
 - d Click **Next** and then click **Import**.

Testing the Tree Ensemble Model with Decision Testing

You can test the model by running sample data through it using Decision Testing. Both decision services call the model function using a decision flow so these instructions apply to both examples.

To run the test data:

- 1 Expand the **Test** pane.
- 2 Under the **Test** pane, click the **Decision Testing** link to open the **Decision Testing** panel.
- 3 In the drop-down menu, select **processWithDecisionFlow(LoanApplication):LoanApplication**.
- 4 Upload the `CalculateScoreData.csv` file.
- 5 Click **Run Test**. After the tests are run, you can download the results as a CSV file on to your desktop.

Related Links

[Dataset Requirements](#)

Recreating an Example

When you download an example from the Help system, the ZIP file not only contains the example, but it also contains an XML schema and the original model file that you can use to recreate the example yourself.

Download one of the examples and unzip the contents. If you cannot download an example, contact FICO Support.

These are the steps required to recreate one of the examples with an imported model.

- 1 Create a project.
- 2 Configure the project using the XSD file that was included in the ZIP file.
- 3 Import the model that was included in the ZIP file using the wizard.
- 4 Map the input class properties for the imported model to the external object model. You can do this using the Business Terms editor or using a function.
The examples show how to map the input class properties to the external object model.
- 5 If the mapping was done with the Business Terms editor, initialize the business term set(s) using the **Initialization entity**.
- 6 Add one or more tasks in the decision flow. The number of tasks that you add depends upon the model type.

The examples show how to setup your task(s) on a decision flow.

- 7 Compile the project.
Fix any errors before moving on to the next step.
- 8 Use the CSV file to upload the dataset and run values through the project.

Re-importing a PMML Mining Model

You can use the Import PMML Mining Model wizard to re-import your PMML Mining model.

When a model is imported again, the generated entities do not override the existing entities. If you want your re-imported model and business term sets to have the same name as the original entities, delete the existing entities before re-importing the model.



Note Models imported prior to Decision Modeler 3.0 have a prefix of the for the global variable name of the business term set. As of Decision Modeler 3.0, the prefix is now var so when you re-import the model, the prefix for the business term sets will be var. Compiling the project will generate errors unless you update the business term mappings. For example, `the SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition` must be changed to `var.SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition`.

Follow these steps to re-import a model in your project:

- 1 Delete the existing model and the business term sets by checking them out, deleting them, and checking in the changes. Because a PMML Mining model is always imported in a subfolder, delete the subfolder that contained the model.
- 2 Import the model using the wizard.
A new model and new input and output business term sets will be created in a subfolder.
- 3 Update the mapping of the business term sets of the imported model to the external object model.
 - If you mapped the business terms using the Business Terms editor, you will need to redo the mapping for each business term.
 - If you mapped the business terms using a function, the same function will continue to compile and run as normal so long as the business terms did not change. If you removed any business terms, you must remove the business terms from the existing mapping function. If you added any new business terms, you must add the business terms mapping to the function.
- 4 If the mapping was done using the Business Terms editor and your project includes an Initialization entity, initialize the business term sets using the Initialization entity.
- 5 Compile the project to ensure that no errors were introduced into the project.

Importing Other PMML Models

Use the **Import Other PMML Models** wizard to import a model that conforms to PMML 3.0 to 3.2 or 4.0 to 4.4 Specification. A PMML Executor license is required in order to import a PMML model with the Other PMML Models wizard. If you need a license or your license has expired, contact FICO Support.

- 1 In the **Project Explorer**, select the folder where you want to import the model.
 - 2 Click the **Import** drop-down list and select **Other PMML Model**.
 - 3 On the **Import Other PMML Models** page, drag and drop a single .pmml, .xml, or .zip file onto the drop zone or click **local file browser** or **FICO Drive** to browse for a file to add.
-  **Note** If a .zip file is selected, the .pmml or .xml file will be extracted. The file must validate and be smaller than 100 MB in order to be imported.
- 4 Click **Continue** to view the **Summary** page.
All of the decision entities that will be generated for the model are displayed.
 - 5 Click **Done** to close the wizard and complete the import of the PMML model.

When a PMML Model is imported, the following decision entities are created in a folder:

Entity	Name	Description
Function	calculate modelName	This is the main function used to invoke the model in the project.
Input Business Term Set	modelNameIn	Contains the input type of the main model function. If changes are made to the business terms, the function may not compile.
Output Business Term Set	modelNameOut	Contains the output type of the main model function. If changes are made to the business terms, the function may not compile.

Supported PMML Models

FICO supports importing PMML models that conform to PMML 3.0 to 3.2 or 4.0 to 4.4 Specification with the **Import Other PMML Models** wizard.

Note that these model types are not supported:

- Anomaly detection model
- Baseline model
- Bayesian network

- Gaussian process
- Sequence rules
- Text model
- Time series model

 **Note** For more information on the types of models that are supported and their general structure, see the Data Mining Group's website at www.dmg.org.

Re-importing a PMML Model

You can use the **Overwrite Existing Entities** check box in the upper right corner of the **File Selection** page of the **Import Other PMML Models** wizard to update previously imported models by overwriting the existing entities in the selected directory.

If the **Overwrite Existing Entities** check box is selected, the existing entities in the selected directory will be overwritten. If the **Overwrite Existing Entities** check box is not selected, the existing entities will not be overwritten. Instead, the generated entities will be added to the folder. If any duplicate names exist, the generated entity names will be modified to make them unique in the folder.

 **Note** Models imported prior to Decision Modeler 3.0 have a prefix of the for the global variable name of the business term set. As of Decision Modeler 3.0, the prefix is now var so when you re-import the model, the prefix for the business term sets will be var. Compiling the project will generate errors unless you update the business term mappings. For example, `the SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition` must be changed to `var.SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition`.

Limitations

The limitations described in this section exist for importing PMML models.

Built-in Functions

The PMML 3.2 Specification supports a set of built-in functions that can be used to perform low-level operations. FICO supports all built-in functions defined in the specification, except for the following functions:

- log10
- sqrt
- trimBlanks
- formatNumber
- formatDate
- dateDaysSinceYear

- dateSecondsSinceYear
- dateSecondsSinceMidnight

Reserved Words

As a best practice, the following Java reserved words should not be used as input or output field names in any PMML model:

- abstract
- assert
- boolean
- break
- byte
- case
- catch
- char
- class
- const
- continue
- default
- double
- do
- else
- enum
- extends
- false
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long

- native
- new
- null
- package
- private
- protected
- public
- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- true
- try
- void
- volatile
- while



Note Using any of these words as an input or output field name in a PMML model will result in a compilation error.

CHAPTER 11

Importing FSML Files

Import a FICO Strategy Markup Language (FSML) .xml file to create a decision tree.

FSML 1.0, 2.0, and 2.5 are supported in this release. The FSML file exported from the following FICO products are supported:

- FICO® Model Builder for Decision Trees
- FICO® Xeno (now known as Decision Tree Pro)

Importing Decision Trees from FSML

This section describes each page of the **Decision Tree from FSML** wizard that is used to create a decision tree from an FSML strategy file.

When a decision tree is created from an FSML file, a business term set is created for the Input and another one for the Output of the tree. The term sets must have a unique name in a project, and each business term must have a unique name within the term set. The term sets and their business terms can be used as variables in other decision entities in a project.

 **Note** If a variable in the FSML file is defined as an integer data type in the Data Dictionary and the variable is compared against a real data type value in a condition of the decision tree, the variable will be defined as a real data type in the business term set.

When importing an FSML file, the label of a condition node in the decision tree will be replaced with the actual condition that is being tested. For example, if a condition tests that age is between 18 and 35, the new label "18 <=..<35" will be created.

The decision tree will be imported in Repair Mode if the FSML file contains a value list with empty values. You can add the missing value to the FSML file, or, after the FSML has been imported, you can add the missing values to the value list by either clicking **Repair Level** or by editing each split in Repair Mode.

Related Links

[Mapping the Business Terms to Object Model Properties](#)

[FSML Decision Tree Import Limitations](#)

Opening the Wizard and Selecting an FSML File

Open the wizard and import an FSML-format .xml file to create a decision tree. The .xml file must contain a valid FSML strategy in order to be uploaded.

- 1 In the Project Explorer, select the folder where you want to create the decision tree.
- 2 Click the **Import** drop-down menu.
- 3 Select **FSML Decision Tree Model**.
- 4 On the **Import FSML Model** page, drag and drop an .xml or .zip file onto the drop zone or click **local drive** or **FICO Drive** to browse for files to add.
 **Note** A .zip file can only contain one valid FSML-format .xml file and the file must be smaller than 100 MB.
- 5 Click **Next**.

Selecting and Renaming Business Terms

Specify the attributes of the business terms that will be created for the decision tree. You can rename the Input and Output term sets and the business terms. The business term names, types, values, and descriptions come from the FSML file dataset. The Input and Output term sets can be accessed from the Project Explorer in the same folder that the decision tree is created in after it has been imported.

If you want to rename the term sets or business terms, be aware of the following naming restrictions:

Property	Restriction
Term Set	A valid name consists of one or more letters (A-Z, a-z), digits (0-9), and some special characters, such as <code>_</code> . It cannot contain the following special characters: <code>\$"^; '</code>
Business Term	A valid name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (<code>_</code>). It cannot contain spaces, punctuation marks, or special characters, such as <code>\$</code> or <code>%</code> .

By default, it is assumed that you want to use all of the business terms as properties in the input argument of the decision tree. Therefore, all of the business terms are selected as Input and will be added to the Input term set.

If you do not select any variables, the output defined in the FSML strategy file is added to the Output term set and named Action. By default, it is assumed that only the Action variable is required for the return argument, so no properties are selected in the **Output** column. If you select business terms to use as output, they will be added to the Output term set.

Naming the Decision Tree

The name of the Strategy is the default decision tree name. Enter a new name for the new decision tree and, optionally, enter a description of the decision the decision tree generates. A valid name consists of one or more letters, digits (0-9), and underscores (_).

-  **Note** If the Strategy name contains spaces, the spaces will be converted into underscores (_).

Viewing the Import Summary

The content of the report depends on the choices you made in the previous pages of the wizard.

- To complete the export operation, click **Done**.
- If you need to modify the export, click **Previous** to return to the previous page.
- If you do not want to complete the operation, click **Cancel**.

FSML Decision Tree Import Limitations

There are some limitations to the FSML decision tree import in this release.

- An FSML file can only contain zero or one strategy. If the file does not have a strategy, the business terms and an empty tree will be imported. If the file contains more than one strategy, an error is displayed.
- Multiple minimum/maximum ranges for variables defined in the Data Dictionary or Decision Area definitions of the FSML file are not supported. For example, variables can have a maximum and minimum value associated with them. Only a single pair of maximum/minimum range is supported.
- FSML files that contain alphanumeric variable definitions are not supported. As a workaround, you can change these variables to string types with category values.
- The FSML decision tree does not support the following items associated with variables and actions:
 - ID
 - MaxLength
 - Description
 - Increment (on a numeric value)
 - Color

-  **Note** The imported decision tree is subject to the same limitations as a decision tree that is created from the **New** drop-down menu.

Related Links

[Importing Decision Trees from FSML](#)

[Decision Tree Limitations](#)

Example Decision Services with an FSML Decision Tree Model

To see how an FSML Decision Tree Model appears in a decision service, download one or both of these ZIP files. This is a description of each example:

-  **Note** Contact FICO Support for sample ZIP files.
- [FSMLExampleWithMappedBusinessTermSets.zip](#)
This example shows how to use the business term set editor to map the business terms to the corresponding object model properties.
 - [FSMLExampleWithMappedFunctions.zip](#)
This example shows how to map the business terms to the corresponding object model properties using a function.

Each of the ZIP files contains the following:

- A deployable decision service containing an FSML Decision Tree.
- The `PromotionsFinal_Reals.xml` file.
- The `SportsFanPromotion.xsd` that contains the XML Business Object Model (BOM).
- The `SportsFanDataset.csv` file that contains values you can use to test the output of the model.

Related Links

[Mapping the Business Terms to Object Model Properties](#)

Importing the Decision Service

- 1 Download the ZIP file to your desktop and unzip it. If you cannot download the file, contact FICO Support.

 **Important** Do not unzip the enclosed ZIP file with `DecisionService` in the file name.
- 2 Create a workspace in Decision Modeler.
- 3 Import the project by uploading the ZIP file with `DecisionService` in the file name.
 - a Select the workspace and click **Import Projects**.
 - b Click **Local File Browser** and select the ZIP file with `DecisionService` in the file name.
 - c Click **Next** and select the project name.

- d Click **Next** and then click **Import**.

Testing the FSML Decision Tree Model with Decision Testing

You can test the FSML Decision Tree Model by running sample data through it using Decision Testing. Both decision services call the model function using a decision flow so these instructions apply to both examples.

To run the test data:

- 1 Expand the **Test** pane.
- 2 Under the **Test** pane, click the **Decision Testing** link to open the **Decision Testing** panel.
- 3 In the drop-down menu, select **processWithDecisionFlow(SportsFanPromotion): SportsFanPromotion**.
- 4 Upload the `SportsFanDataset.csv` file.
- 5 Click **Run Test**. After the tests are run, you can download the results as a CSV file on to your desktop.

Recreating an Example

When you download an example from the Help system, the ZIP file not only contains the example, but it also contains an XML schema and the original model file that you can use to recreate the example yourself.

Download one of the examples and unzip the contents. If you cannot download an example, contact FICO Support.

These are the steps required to recreate one of the examples with an imported model.

- 1 Create a project.
 - 2 Configure the project using the XSD file that was included in the ZIP file.
 - 3 Import the model that was included in the ZIP file using the wizard.
 - 4 Map the input class properties for the imported model to the external object model. You can do this using the Business Terms editor or using a function.
The examples show how to map the input class properties to the external object model.
 - 5 If the mapping was done with the Business Terms editor, initialize the business term set(s) using the **Initialization entity**.
 - 6 Add one or more tasks in the decision flow. The number of tasks that you add depends upon the model type.
The examples show how to setup your task(s) on a decision flow.
 - 7 Compile the project.
- Fix any errors before moving on to the next step.

- 8 Use the CSV file to upload the dataset and run values through the project.

Re-importing an FSML Model

You can use the Import FSML model wizard to re-import your FSML model.

When a model is imported again, the generated entities do not override the existing entities. If you want your re-imported model and business term sets to have the same name as the original entities, delete the existing entities before re-importing the model.



Note Models imported prior to Decision Modeler 3.0 have a prefix of the for the global variable name of the business term set. As of Decision Modeler 3.0, the prefix is now var so when you re-import the model, the prefix for the business term sets will be var. Compiling the project will generate errors unless you update the business term mappings. For example, `the SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition` must be changed to `var.SocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition`.

Follow these steps to re-import a model in your project:

- 1 Delete the existing model and the business term sets by checking them out, deleting them, and checking in the changes.
- 2 Import the model using the wizard.
A new model and new input and output business term sets will be created.
- 3 Update the mapping of the business term sets of the imported model to the external object model.
 - If you mapped the business terms using the Business Terms editor, you will need to redo the mapping for each business term.
 - If you mapped the business terms using a function, the same function will continue to compile and run as long as the business terms did not change. If you removed any business terms, you must remove the business terms from the existing mapping function. If you added any new business terms, you must add the business terms mapping to the function.
- 4 If the mapping was done using the Business Terms editor and your project includes an Initialization entity, initialize the business term sets using the Initialization entity.
- 5 Compile the project to ensure that no errors were introduced into the project.

CHAPTER 12

Importing SAS Programs

Import a SAS program to convert it into an SRL function and use it in your decision service. A SAS program is a sequence of steps with each step in the program performing a specific task.

You can import a single-step or multi-step SAS program. The SAS program is converted into an SRL function and the input and output variables are converted to business term sets.

Uploading a SAS License

A SAS license must be uploaded to your project in order to import a SAS program. If you need a license or your license has expired, contact FICO Product Support.

- 1 Create a JAR file containing the license file.
 - a Open the command prompt.
 - b Change directories to the folder where your carolina.lic file is stored.
For example, cd C:\Users\username\Desktop\SAS License
 - c Set the path to your JDK's bin folder.
For example, path C:\Program Files\Java\jdk1.8.0_31\bin
 - d Enter the following command to create the JAR file: jar cf <filename>.jar carolina.lic
- 2 Open the Decision Modeler project.
- 3 Upload the JAR file to your project.
 - a On the Project Editor page, click the **Project Configuration** icon on the Global Command Bar.
 - b Select **Manage JAR files**.
 - c Add the JAR file by dragging and dropping the file into the drop zone or clicking **local drive** and selecting the file from your local machine.

Selecting Files to Import

At least one program file (.sas) and one data file (.sas7bdat) are needed to import a SAS program. No more than two data files (.sas7bdat) can be imported. The following files can be imported and they must be valid:

- SAS program (.sas). This can be a single-step or multi-step SAS program.
- Input data (.sas7bdat).
- Validation data (.sas7bdat).

 **Important** The name of the data files (.sas7bdat) must be lowercase; otherwise, the files cannot be uploaded.

If two program files are being imported, they must be related to each other. They will be converted into a single SRL function. SAS program files must contain the following:

- A LIBNAME statement.
- A SET statement.

Files can be imported from your local drive, and they can be selected individually or imported in a ZIP file. If you attempt to import files that have already been imported in the same folder, the options are to overwrite the existing files or close the dialog and select a different folder. If you choose to import the files in a different folder, the names of the files will be made unique by generating a numeric value in the middle of the entity name.

To ensure a successful and accurate translation, follow these recommendations about the structure and content of a single SAS program and dataset:

- 1 The SAS program must be contained within a single file.



Note Through the use of %include statements, it is possible for a SAS program to be divided among many files; and while this may make sense to its original author, the division of code across files can add complexity for viewers other than the original author.

- 2 The SAS program must contain one or more data steps, and the new effect of the program must be to read data from the input dataset, and create the output dataset. Within the data steps, the SAS author may describe any and all of the typical scoring operations, such as data pre-processing, conditional model selection, score computations, and any post-scoring adjustments.
- 3 The SAS program and the two SAS datasets should all reside in the same directory, and the LIBNAME statements defining the location of the two datasets should be assigned to ".", the current working directory.



Note If the SAS program files are imported in a ZIP file, the LIBNAME statement does not need to be modified and the data files can reside in the location specified by the LIBNAME statement.

- 4 Do not use type V9 in the LIBNAME statement in the output dataset with a multi-step SAS program. This overwrites the SAS benchmark dataset, and the SAS program will be validated against an intermediary dataset, rather than the dataset supplied.

- 5 The input fields (name and data types) required by the SAS program should match the names and data types presented within the input dataset.
- 6 The output dataset should contain both the input fields plus any new fields created by the SAS program.
- 7 The input and output datasets must contain the same number of records, and the records must be identically ordered from top to bottom in both files.

Creating Functions from SAS Programs

Use the **Import SAS Program** wizard to create a function containing the SAS program. A SAS license must be added to the project before importing a SAS program.

- 1 In the Project Explorer, select the folder where you want to add the SAS program.
 - 2 Click the **Import** drop-down menu and select **SAS Program**.
 - 3 On the **Import SAS Program** page, drag and drop .sas, .sas7bdat, or .zip files onto the drop zone or click **local drive** or **FICO Drive** to browse for files to add.
-  **Note** If a .zip file is selected, the .sas and .sas7bdat files will be extracted. The files must be smaller than 100 MB in order to be imported.
- 4 (Optional) Select the main SAS program in the **Main SAS Program** drop-down menu if you are importing more than one .sas file.
 - 5 (Optional) Clear the **This is a multi-step SAS program** check box if you are importing a single-step SAS program.
 - 6 Click **Continue**.
All of the decision entities that will be generated for the SAS program are displayed.
 - 7 Click **Done** to import the SAS program.

When a SAS program is imported, the following decision entities are created in a folder:

 **Note** The names of the entities are converted based on the name of the SAS program. A "P" is prepended to the name and a suffix identifying the type of entity is appended, such as **Group Instance**, **InputObject**, and **OutputObject**. If the name of the SAS program contains dashes (-) or spaces, they will be converted to underscores (_).

Entity	Name	Description
Group Instance	ProgramName Group Instance	Contains the top-level function that contains the SAS program used to invoke the program in the project and input and output objects.
Business Term Set	ProgramNameInputObject	Contains the input type of the main SAS program function. If changes are made to the

Entity	Name	Description
		business terms, the function may not compile.
Business Term Set	ProgramNameOutputObject	Contains the output type of the main SAS program function. If changes are made to the business terms, the function may not compile.

Invoking the SAS Program

To execute a SAS program in a project, the function containing the SAS program must be called. The function can be called from another decision entity, or it can be added as a task in the decision flow.

 **Important** Before invoking the SAS program in a project, the input and output business term sets must be mapped to the project's XML or Java Business Object Model (BOM).

When the function is called from another decision entity, it must be called by its SRL name. The SRL name is displayed in the **Reference name** field in the Function editor.

To call the function using the decision flow, create a task and associate the task with the function. Both the input and the output business term sets are used as parameters, with the output business term set used as the return type.

Example Decision Services with a SAS Program

To see how a SAS program appears in a decision service, download one or both of these ZIP files.

This is a description of each example:

 **Note** Contact FICO Support for the sample ZIP files.

- **SASExampleWithMappedBusinessTermSets.zip**
This example shows how to use the business term set editor to map the business terms to the corresponding object model properties.
- **SASExampleWithMappedFunction.zip**
This example shows how to map the business terms to the corresponding object model properties using a function.

Each of the ZIP files contains the following:

- A deployable decision service containing a SAS program.
- The **HRattrition.sas** file that was generated using FICO® Analytic Modeler Scorecard Professional.

- The `hr_attrition_in.sas7bdat` input data file.
- The `hr_attrition_out.sas7bdat` output data file.
- `DefaultTermLibrary-HR.jar` file that contains the Java Business Object Model (BOM).
- The `employeeAttritionData.csv` file that contains values you can use to test the output of the model.

 **Note** While you can view a decision service with an imported SAS program, you must have a SAS license to import a SAS program. If you are interested in obtaining a license, contact FICO Support.

Importing the Decision Service

To view the imported model in a decision service, do the following:

- 1 Download the ZIP file to your desktop and unzip it. If you cannot download the file, contact FICO Support.

 **Important** Do not unzip the enclosed ZIP file with `DecisionService` in the file name.
- 2 Create a workspace in Decision Modeler.
- 3 Import the project by uploading the ZIP file with `DecisionService` in the file name.
 - a Select the workspace and click **Import Projects**.
 - b Click **Local File Browser** and select the ZIP file with `DecisionService` in the file name.
 - c Click **Next** and select the project name.
 - d Click **Next** and then click **Import**.

Testing the SAS Program with Decision Testing

You can test the SAS program by running sample data through it using Decision Testing. Both decision services call the model function using a decision flow so these instructions apply to both examples. To run the rest data:

- 1 Expand the **Test** pane.
- 2 Under the **Test** pane, click the **Decision Testing** link to open the **Decision Testing** panel.
- 3 In the drop-down menu, select **processWithDecisionFlow(HRTermset) : HRTermset**.
- 4 Upload the `employeeAttritionData.csv` file.
- 5 Click **Run Test**. After the tests are run, you can download the results as a CSV file on to your desktop.

Related Links

[Dataset Requirements](#)

Recreating an Example

When you download an example from the Help system, the ZIP file not only contains the example, but it also contains an XML schema and the original model file that you can use to recreate the example yourself.

Download one of the examples and unzip the contents. If you cannot download an example, contact FICO Support.

These are the steps required to recreate one of the examples with an imported model.

- 1 Create a project.
- 2 Configure the project using the XSD file that was included in the ZIP file.
- 3 Import the model that was included in the ZIP file using the wizard.
- 4 Map the input class properties for the imported model to the external object model. You can do this using the Business Terms editor or using a function.
The examples show how to map the input class properties to the external object model.
- 5 If the mapping was done with the Business Terms editor, initialize the business term set(s) using the **Initialization entity**.
- 6 Add one or more tasks in the decision flow. The number of tasks that you add depends upon the model type.
The examples show how to setup your task(s) on a decision flow.
- 7 Compile the project.
Fix any errors before moving on to the next step.
- 8 Use the CSV file to upload the dataset and run values through the project.

Re-importing a SAS Program

You can use the Import SAS program wizard to re-import your SAS program.

When a model is imported again, you can choose to overwrite the existing SAS program and business term sets. You do not have to delete the entities before re-importing your SAS program.

 **Note** Models imported prior to Decision Modeler 3.0 have a prefix of the for the global variable name of the business term set. As of Decision Modeler 3.0, the prefix is now var so when you re-import the model, the prefix for the business term sets will be var. Compiling the project will generate errors unless you update the business term mappings. For example, `theSocialMediaScoreIn.audienceComposition = input.SocialMedia.audienceComposition` must be changed to

```
var.SocialMediaScoreIn.audienceComposition =
input.SocialMedia.audienceComposition.
```

Follow these steps to re-import a model in your project:

- 1 Import the model using the wizard.
When the warning message appears alerting you that the SAS program was already imported into the folder, select the option to overwrite the files. This step will overwrite the existing model and business term sets.
- 2 Update the mapping of the business term sets of the imported model to the external object model.
 - If you mapped the business terms using the Business Terms editor, you will need to redo the mapping for each business term.
 - If you mapped the business terms using a function, the same function will continue to compile and run as long as the business terms did not change. If you removed any business terms, you must remove the business terms from the existing mapping function. If you added any new business terms, you must add the business terms mapping to the function.
- 3 If the mapping was done with the Business Terms editor, initialize the business term sets using the Initialization entity.
- 4 Compile the project to ensure that no errors were introduced into the project.

SAS to SRL Mapping Reference

FICO's Structured Rule Language (SRL) is an easy to-understand, English-like language. This section defines how the SAS program syntax will be translated into SRL.

Type Mapping

DATA Step

SAS Type	SRL Type
Character	string
Numeric	real

Input and Output

Input and output SAS data sets are mapped to the SRL classes: InputObject and OutputObject.

SAS Data Set Type	SRL Type
Character	string
NUMERIC with date format	date

SAS Data Set Type	SRL Type
NUMERIC with time format	time
NUMERIC with datetime format	timestamp
NUMERIC other	real

Data Step Statements

This table shows how SAS statements are mapped to SRL.

SAS Statement	SAS Example	SRL Example	Comments
Assignment	X=Y;	step.X=step.Y;	
Array			fixed array
DELETE	DELETE	return false;	
DO list	DO j=2,5,7; PUT j=; END;	<pre> l_1 is a fixed array of 3 real initially { it[0] = 2, it[1] = 5, it[2] = 7}; for each real in l_1 do { step.J = it; print ("" step.J); step.J += 2; } </pre>	
DO iterative	DO j=1 TO X+Y BY 2; PUT j=; END;	<pre> l_1 is a real. l_1 = x+y; step.J = 1; while (step.J < l_1) do { print ("" step.J); step.J += 2; } </pre>	Temporary expression is required because SAS loop boundaries are evaluated only once at the beginning of the loop.
DO WHILE	DO WHILE J>0; ... END;	<pre> while (step.J > 0) { ... } </pre>	
DO UNTIL	DO UNTIL J<0; ... END;	<pre> until (step.J < 0) { ... } </pre>	
IF subsetting	IF (X>Y);	<pre> if (not (step.X > step.Y)) return false; </pre>	
IF THEN ELSE	IF (X<1) THEN Y=2; ELSE Z=3;	<pre> if (step.X < 1) then { step.Y = 2;} else { step.Z = 3; } </pre>	

SAS Statement	SAS Example	SRL Example	Comments
LINK	<pre> Z = 0; LINK LBL; Z = Z+2; RETURN; LBL: Z=3; RETURN; </pre>	<pre> function loop for { step : a Data } returning a boolean is { step.Z = 0; temp_b=lbl(step); if (not temp_b) return false; step.Z = step.Z+2; return true; } ... function lbl for { step : a Data } is { step.Z = 3; return true; } </pre>	The target of the LINK statement is generated as a separate function.
RETURN	RETURN	return true;	
RETAIN			In initialization function, do not assign missing values.
SELECT (conditional)			Sequence of if-then-else.
SELECT selector	<pre> SELECT X; WHEN (1) Z=10; WHEN (2) Z=20; ELSE Z=30; END </pre>	<pre> select step.X case 1 : step.Z=10; case 2 : step.Z=20; otherwise : step.Z=30; </pre>	SRL allows the use of the SELECT statement for both numeric and character data, similar to the switch statement in Java.
sum statement	X + 3;	X = NdSas.sum(step.X, 3);	It is similar to X=X+3; but differs in the way it treats missing values. If x is missing, 3 will be assigned, otherwise, increment by 3.
WHERE			See WHERE=options for data steps.

Data Set Options and Statements

Option	SAS Example	SRL Example	Comments
DROP			No properties are generated in the Input/Output class definitions.
KEEP			Only specified properties are generated in the Input/Output class definitions.
RENAME	rename=(x=y)	step.Y=input.X	
WHERE	where=(X>0)	<pre>function calculate ... returning a boolean is { if (not (input.x > 0)) then { return false. } ... return true; } function main is { b is a boolean. ... b = calculate (d, i, o). }</pre>	This allows the filtering of the records when reading from the input data set or writing to the output data.

The DATA set statements DROP, KEEP, and RENAME are processed for the output data set similar to the corresponding options above.

Subsetting IF statement, WHERE statement, and WHERE= data set options for the input or the output data set maps to the same functionality in the generated SRL code. If the condition specified by the statement or option is not satisfied, the output record is not generated and the calculate() function returns null.

Operations and Values in Expressions

The SAS language operations and constants mapping summary is presented in this table. The implementation details are discussed in the sections following the table.

SAS Expression Type	SAS Example	SRL Example	Comments
Comparison	x < y	X < Y	
	x = y	X = Y	
truncation on assignment		s=NdSas.truncate(r,4)	
IN operator	2 in (1,2,3)	NdSas.inNumberList(2,1,2,3)	
	"abc" in ("cde", "x")	NdSas.inStringList("abc", "cde", "x")	
date literal	'17Jan2020;d	NdSas.dt(20200117)	
time literal	'12:13:14'	NdSas.t('12:13:14')	

Missing Values

NUMERIC missing values, including special missing values, are replaced with large negative numeric values configured from the external property file. The values: ., .A, ..., .Z will be mapped to values ordered: . < . < .A < ... < .z to preserve SAS semantics for the comparison operators with the special missing values. The side effect of the arithmetic operations with the missing values is supported.

The property file is expected to be available during model execution.

SAS CHARACTER missing value is mapped to a single space string.

Date, Datetime, and Time Constants

SAS date/time constraints ('...t', '...d', '...dt') are evaluated and converted to SRL normalized form. In the generated code, function calls to NdSas.t, NdSas.d, and NdSas.dt are added to convert the value on the fly to real constants.

NdSas.t function

```
function t
for {
    t : a time.
}
returning a real
is {
    //Java/COBOL implementation
}
```

NdSas.d function

```
function d
for {
```

```
    d : a integer.  
        }  
returning a real  
is {  
    //Java/COBOL implementation  
}
```

Numeric Relational Operations

SAS numeric comparison operations <, <=, >, >=, ^=, = have semantics which are different from SRL, in particular missing values are assumed to be lower than any other numeric values.

In Operator

Functions `inStringList` and `inNumberList` implement the IN operator for `string(CHARACTER)` and `real(NUMERIC)` variables respectively:

NdSas.inStringList function inStringList

```
function inStringList  
for {  
    s : a string.  
    ar : a fixed array of string.  
}  
returning a boolean  
is {  
    //Java/COBOL implementation  
}
```

NdSas.inNumberList function

```
function inNumberList  
for {  
    s : a real.  
    ar : a fixed array of real.  
}  
returning a boolean  
is {  
    //Java/COBOL implementation  
}
```

String Truncation

SAS CHARACTER variables are fixed size strings, where the extra space at the end is filled with spaces (blanks in SAS documentation) and those extra spaces are ignored in the comparison operations and most function calls. The SRL language requires that there is no fixed size string type so SAS CHARACTER variables are mapped to SRL strings. To maintain the semantics of the fixed size strings in the assignment statement, the calculated value on the right side should be truncated to the variable length on the left side using the `truncate()` function.

NdSas.truncate function

```
function truncatenorm  
for {  
    s : a string.  
    n : a integer.
```

```

}
returning a string
is {
    //Java/COBOL implementation
}

```

Type Conversion

SAS automatically converts between NUMERIC and CHARACTER types in the expressions. SAS does not have a Boolean type and numeric values are used in AND, OR logical operations and in the condition of an IF expression (similar to C language). The SRL language requires the explicit conversion between boolean, real, and string. The implicit SAS conversions are replaced with the explicit function calls: numeric, character and bool.

NdSas.numeric function

```

function numeric
for {
    s : a string.
}
returning a real
is {
    //Java/COBOL implementation
}

```

NdSas.character function

```

function character
for {
    x : a real.
}
returning a string
is {
    //Java/COBOL implementation
}

```

The character() function returns the character representation of a numeric value formatted by SAS format Best12.

NdSas.bool function

```

function bool
for {
    x : a real.
}
returning a string
is {
    //Java/COBOL implementation
}

```

The bool() function returns true for non-zero, non-missing values.

Supported SAS Based Constructs

The supported constructs are shown in this table.

Statements	Options
Assignment	
ABORT	
ARRAY	multidimensional, lower and upper bounds initial value list (*) meaning <i>implicit size</i> <code>_NUMERIC_</code> , <code>_ALL_</code> , <code>_CHARACTER_</code> <code>_TEMPORARY_</code>
ATTRIB	LABEL FORMAT INFORMAT LENGTH
BY	
CALL MISSING	
CALL	
SYMPUT	
DATA	DROP, KEEP, RENAME
DELETE	
DO statement	
DO statement, iterative	start TO stop by increment list, such as <i>index=value, value, value</i> WHILE option UNTIL option
DO UNTIL	
DO WHILE	
DROP	
END	
ERROR	
FORMAT	
IF, subsetting	
IF, then, else	
%INCLUDE	
INFORMAT	

Statements	Options
KEEP	
LENGTH	
LIBNAME	DISK, V9
LINK	
MERGE	DROP, KEEP, RENAME, IN
OPTIONS	
OUTPUT	
RETURN	
RENAME	
RETAIN	
RUN	
SELECT	
SET	DROP, KEEP RENAME, IN
Sum statement	
WHERE	

Proc	Statement	Option	Comments
FORMAT			
	VALUE		
		fuzz	
	INVALUE		
		fuzz	
		just	
		upcase	

Statements	Comments
%ABORT	
%DO	
%DO iterative	
%DO UNTIL	
%DO WHILE	
%END	
%IF %THEN %ELSE	
%INPUT	
%GLOBAL	

Statements	Comments
%GOTO	%GOTO from outside the blow into the middle of the block is not supported.
%LET	
%LOCAL	
%MACRO	
%MEND	
%PUT	
%RETURN	
%SYMDEL	
%SYSEXEC	

Statement	Comments
%BQUOTE, %NRBQUOTE	
%EVAL	
%INDEX	
%LENGTH	
%QSCAN	
%QSUBSTR	
%QSYSFUNC	
%QUOTE, %NRQUOTE	
%UPCASE	
%SCAN, QSCAN	
%STR, %NRSTR	
%SUBSTR	
%SUPERQ	
%SYSEXIST	
%SYMGLOBAL	
%SYMLOCAL	
%SYSEVALF	
%SYSFUNC	
%SYSGET	
%SYSPROD	%SYSPROD(BASE) = 1, %SYSPROD(CAROLINA)=1, all other return 0.
%UNQUOTE	
%UPCASE	

SAS Standard Functions Implemented in Carolina Runtime Library

Please see documentation at the following link: <http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000245860.htm>

Character Functions

cat	find	propcase
cats	findc	put
catt	index	putc
catx	indexc	putn
char	indexw	repeat
choosec	input	reverse
chooseen	inputc	scan
coalesce	inputn	scanq
coalescec	kscan	strip
collate	left	substr
compare	length	substrn
compress	lengthc	transtrn
count	lengthm	tranwrd
countc	lengthn	trim
countw	lowercase	upcase

Date/Time Functions

datdif	intck	second
date	intnx	time
datejul	juldate	timepart
datepart	juldate7	today
datetime	mdy	week
day	minute	weekday
dhms	month	year
hms	qtr	yyq
hour		

Math Functions

abs	digamma	nmiss
arcos	divide	npv
arsin	dur	ordinal

atan	durp	perm
atan2	exp	pvp
ceil	fact	rannor
ceilz	floor	ranuni
compound	floorz	round
convx	fuzz	rounde
convxp	largest	roundz
cos	log	sign
cosh	log10	sin
daccdb	log2	sinh
daccdbsl	max	smallest
daccsl	median	sqrt
daccsyd	min	std
dacctab	missing	stderr
depdb	mod	sum
depdbsl	modz	tan
depsl	n	tanh
depsyd	netpv	var
deptab		

CHAPTER 13

Adding Procedural Code in Functions

A function groups a sequence of related statements. Optionally, a function returns a value. You can create a function to execute a set of procedural statements on a specific data context. Functions are often useful for performing calculations, providing additional flow control, or performing other procedural tasks.

You can write functions that, in response to an explicit execution call, do one of the following:

- Modify the value of object properties.
- Create and delete objects.
- Invoke methods on objects.
- Apply decision entities, such as rulesets and decision tables.
- Invoke other functions.

You can also write an entry point function containing a method to invoke a project. To be accessible during project configuration, the function must be checked in and contain a parameter and a return type that is the same as the input and output type selected.

Functions can access and operate on:

- Global variables, patterns, and named objects.
- Values passed as parameter arguments.
- Local variables or named objects.

Functions do not have direct access to execution flow variables. You must pass execution flow variables to functions as parameter arguments.

You can execute functions from:

- Rule actions.
- Rule conditions (when the function returns a compatible result and does not create or modify the state of an object).
- Function statements.
- Tasks in a decision flow.



Note If you use a function in a *rule condition* (for example, to return a value in the test expression), the function will execute each time it needs to evaluate the rule. This can occur whether or not the function's arguments changed.

Make sure that repeatedly executing a function from a rule condition does not cause unintended side effects. For example, this rule condition causes `myFunction()` to execute again whenever `value1` or `value2` changes:

```
if myFunction(value1) > value2
```

A function consists of one or more statements that are executed when the function is called. These statements can include:

- Loops (control constructs that determine which actions to execute and the number of times to execute them).
- Assignments.
- Temporary variables that are declared and used within the function.
- Statements that apply a decision entity.
- Calls to Java methods.
- Calls to other functions.

 **Tip** There are several built-in functions that you can call directly from a function to provide various useful services and access to several support classes.

Creating Functions

Create a function when you need to write procedural code.

- 1 In the **Project Explorer**, select the folder where you want the function created.
- 2 Click the **New** drop-down menu and select **Function**.
- 3 Select **Function**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 Click **Create**.

When you open or create a function, the **Function Editor** appears. From the **Function Editor**, you can add parameters and enter Structured Rule Language (SRL) directly in the function body using code editor features. If you would prefer to use a drag and drop interface to write your decision logic, click **Switch to Rule Builder**.

Related Links

[Naming Decision Entities and Folders](#)

[Dynamic Objects](#)

Defining Functions

After you have created a function, define it in the drop-down panel.

The panel contains the following fields and sections:

Reference name	A unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.
Description	(Optional) A description of the function.
Return type	The type of value the function returns. Set the return type to void if the function does not return a value.
Parameters	One or more parameter declarations that specify the types and names of containers for any values you pass to the function.
Function Body	Use FICO's Structured Rule Language (SRL) to create statements in the editor.

Related Links

[Authoring Decision Logic Using SRL](#)

Adding Local Variables in Functions

When you declare a local variable of complex, simple, array, or fixed array type within a function body, the scope of the variable is local to the function. Any local variables are no longer available after the function returns a value or terminates.

Declare local variables in a function

If you use the default Advanced Builder interface, FICO recommends that you declare any local variables near the beginning of the function prior to creating any statements or expressions using those variables. Declare any local variables for the function with either one of these syntactical statements:

```
variableName1 is a primitive_datatype;
variableName2 is some className;
variableName3 is an array of string;
variableName4 is a fixed array of integer;
```

For example:

```
{
totalAmount is a real;
className is some Account;
currentStatus is an array of string;
teamNumber is a fixed array of integer;

}
```

If you use the Rule Builder interface, you can drag **Declare temporary variable** to the default group in the function body to declare a temporary (local) variable and write statements and expressions that use it. When you declare a temporary variable in the Rule Builder, the declaration statements are automatically added near the top of the function body.

CHAPTER 14

Authoring Decision Logic Using SRL

In some of the editors, there is an interface that you can use to author decision logic using FICO's Structured Rule Language (SRL), which is an easy-to-understand, English-like language. The interface is referred to as the **Advanced Builder** editor.

The **Advanced Builder** editor is available by default in the Function editor and when you click the **Switch to advanced builder** link from one of these editors:

- Ruleset (rule conditions, rule actions, and variable initialization)
- Business Term (calculations)
- Decision Tree (assign expressions)

The **Advanced Builder** editor features include:

- Line numbers.
- Code completion triggered based on user activity.
- Auto-completion to ensure matching parenthesis.
- Syntax errors that are indicated by an error icon next to the line number. If you hover over the error icon, a tool tip appears with an error description.
- Syntax coloring that helps you to distinguish between the different SRL syntax elements by highlighting them in different default colors.
- Hot-keys for undo (**Ctrl-Z**) and redo (**Ctrl-Y**).

When inserting calls to the built-in functions or class properties in SRL, use the fully qualified name of the function or class. For example, to use the built-in `max(integer, integer)` in the Math built-in functions, enter `math.max(integer, integer)`.



Note Nested statements should be used sparingly. Unlimited nested if-statements are supported; however, deeply nested statements are more difficult to read, maintain, and extend and are not considered a best practice.

Related Links

- [Reference Guide](#)
- [Creating Local Variables](#)
- [Creating Patterns](#)
- [Dynamic Objects](#)
- [Creating an Initialization Rule](#)
- [Defining Rules](#)
- [Creating Rule Conditions](#)
- [Creating Rule Actions](#)
- [Assigning an Expression](#)

Triggering Code Completion

When entering SRL syntax in an editor, guidance is provided in the form of code completion lists. Based on the placement of your cursor in the code, the completion list contains the appropriate global and local parameters, classes, object, variables, and other entities. For example, if a class-based entity is selected, typing a dot after the entity triggers a completion list of business terms for that entity. The completion lists may also be populated with keywords and suggested syntax.

Trigger code completion by entering one of the following characters:

- A dot "." after a valid identifier. For example, `input.Applicant.`
- Press **Ctrl + Space**.
- An open parenthesis such as "(" or "[" and quotation marks such as " are auto-completed with the matching closing parenthesis or quotation mark.
- Entering a ' does not automatically add a matching single quotation mark but allows you to use an apostrophe to write human language syntax such as `Applicant's` followed by a space to trigger code completion.

Code completion occurs after a slight delay and only if you have not typed anything after these characters. You can add an item from the completion list to the editor by selecting it. If there is only one item displayed in a completion list, press **Ctrl + Space** or **Enter** to add it to your statement.



Tip To activate code completion in the editors, press **Ctrl + Space**.

Related Links

- [Code Completion for Common Use Cases](#)
- [Code Completion Lists](#)

Code Completion for Common Use Cases

The behavior you see when you trigger code completion depends on the SRL syntax you have already entered.

Table 33: Common Code Completion Behavior

Use Case	Behavior
User presses Ctrl + Space .	The completion list displays with all available global and local parameters, classes, object, variables and other entities.
Completion list contains only one possible choice.	You must explicitly select the choice by pressing Ctrl + Space .
User enters a character while the completion list is open.	The completion list updates to display only those items that begin with that character.
User presses Backspace to remove one or more characters while the completion list is open.	The completion list updates to display those items that match the current text. The list may include different items, new and existing items, or the same items as the previous list.
User enters a character while the completion list is open and no items match it.	Code completion is canceled.
User presses Backspace until the insertion point is positioned before the start of the completion list.	Code completion is canceled.
More than 10 items are displayed in the completion list.	A vertical scrollbar appears.
The width of the list is greater than 500 pixels.	A horizontal scroll bar appears.

Related Links

[Triggering Code Completion](#)

[Code Completion Lists](#)

Code Completion Lists

What you see in the code completion list depends on the current insertion point within the SRL and the syntax you are authoring.

Table 34: Code Completion Lists

Insertion Point Location	Completion List Content
Beginning of a statement.	<ul style="list-style-type: none"> ■ All global variables. ■ All built-in and user defined functionals. ■ All common SRL control constructs such as, if...then, when...do, for each...do, while...do, until...do, apply, create, delete, etc.

Table 34: Code Completion Lists (continued)

Insertion Point Location	Completion List Content
Immediately after a dot preceded by a class or enumeration name.	<ul style="list-style-type: none"> ■ All static properties of that class, or list of enumeration items for an enumeration. ■ All static methods of that class.
Immediately after a dot preceded by an expression evaluating to an object.	<ul style="list-style-type: none"> ■ All non-static properties of that object's class. ■ All non-static methods of that object's class.
An object immediately followed by an apostrophe and the letter "s" and a space. For example, if you enter Applicant 's, you will see a completion list.	<ul style="list-style-type: none"> ■ All non-static properties of that object's class. ■ All non-static methods of that object's class.
After an expression appearing in the middle of a statement	<ul style="list-style-type: none"> ■ Comparison operators that are allowed for that expression type. ■ Boolean operators if the expression type is boolean or numeric.
After a binary boolean operator.	<ul style="list-style-type: none"> ■ The constants True or False. ■ All global variables. ■ All built-in and user defined functionals.
After an arithmetic operator.	<ul style="list-style-type: none"> ■ All global variables. ■ All built-in and user defined functionals.
At the end of a valid SRL identifier without any space between the last character and the cursor.	<p>The same list of all identifiers that would be shown at the beginning of that SRL identifier, except that the list is now filtered to show only those items that begin with the current identifier.</p>

Related Links

[Code Completion for Common Use Cases](#)

[Triggering Code Completion](#)

CHAPTER 15

Controlling the Execution Flow

A decision flow is a graphical presentation of all or a portion of the flow of control in a project. A decision flow defines the sequence in which the entities are executed.

You can do one of the following with a decision flow:

- Use it to start the execution of the project.
- Choose to have a decision flow execute whenever an object of a specified type is posted as an event or when the project is run.
- Invoke a decision flow as a functional.

A decision flow consists of the following:

- A start node that marks the beginning of the decision flow.
- An end node that marks the ending of the decision flow.
- One or more decision flow elements such as tasks and splits that you can insert between the start and end nodes.

The execution of a decision flow proceeds along the assembled flow.



Figure 38: A complete decision flow with a start node, end node, and decision flow elements

If the project was configured using an XML schema, the decision flow is invoked using the **Initialization** entity in the <project name> folder. If the decision service has more than one decision flow, open the Initialization entity and select the one you want to use before compiling the project. Also, if you change the default name of the decision flow, open the Initialization entity and make sure that the correct decision flow name is selected.

Related Links

[Preparing a Project for Testing and Deployment](#)

Decision Flow Editor Interface

The Decision Flow editor consists of several toolbars and panes.

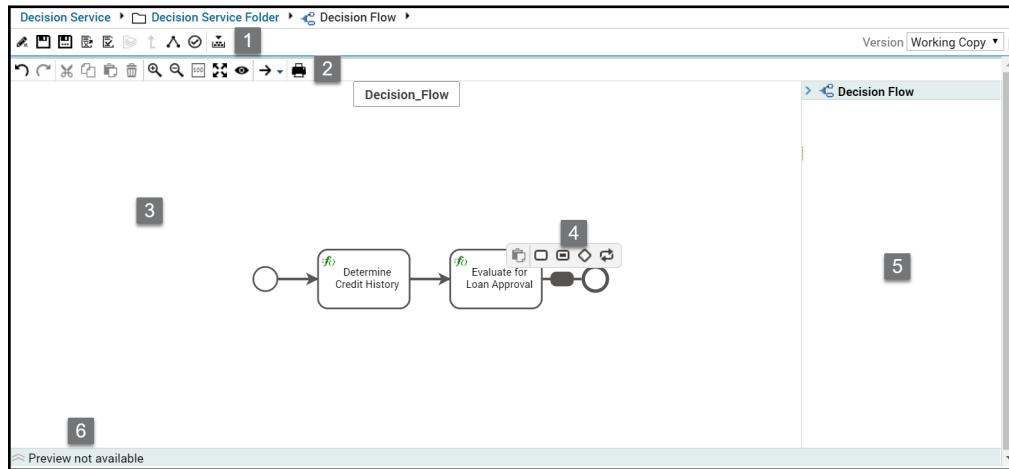


Figure 39: The Decision Flow editor with callouts for important features

Table 35: Description of Decision Flow Editor Toolbars and Panes

Item	Name	Description
1	Editor toolbar	This toolbar contains commands that let you edit and save a decision flow. The Parent View icon is disabled because it is not used in this context.
1	Editor toolbar	This toolbar contains commands that let you edit, save, and perform versioning operations. The Parent View icon is disabled because it is not used in this context.
2	Decision Flow editor toolbar	This toolbar contains commands that let you cut, copy, paste, or delete decision flow entities or change the orientation of the decision flow and magnification of the decision flow diagram.
3	Canvas	The main work area of the editor. Displays the decision flow diagram for editing.
4	Decision Flow Floating toolbar	Paste or insert a task, subflow, split, or loop.
5	Details pane	Defines a decision flow or decision flow elements such as a task, subflow, split, or loop.
6	Preview pane	Displays the entity that has been associated with a decision flow entity such as a task.

Related Links

[Creating Decision Flows](#)

[Inserting Decision Flow Entities](#)

[The Details Pane](#)

Decision Flow Editor Toolbar Commands

The Decision Flow editor toolbar commands are used to perform routine operations on a decision flow.

Table 36: Decision Flow Editor Toolbar Commands

Command	Icon	Description
Undo		Reverts the previous operation.
Redo		Reverts the Undo operation.
Cut		Cuts a decision flow entity. Use the Paste icon on the editor toolbar or the floating toolbar to insert it in another flow location.
Copy		Copies a decision flow entity. Use the Paste icon on the editor toolbar or the floating toolbar to insert it in another flow location.
Paste		Pastes to insert a new decision flow entity or to replace an existing one.
Delete		Deletes a decision flow entity.
Zoom in		Increases the magnification of the decision flow.
Zoom out		Decreases the magnification of the decision flow.
Revert to default size		Reverts to the default magnification of 100% scale.
Fit to current window		Fits the decision flow diagram to the screen and behaves in the following way depending on the size of the diagram and any existing magnification settings. <ul style="list-style-type: none"> ■ If the decision flow diagram is smaller than the width of the canvas area but is magnified more than 100%, use this icon to restore the view to the size of the screen. ■ If the decision flow diagram is larger than the canvas area, use this icon to scale down the diagram to fit the screen. ■ If the magnification is already using the size of the screen and you click this icon, you will see no change.
Show/Hide Overview		Displays the Overview box that you can use to navigate to the area of the decision flow you want to view.
Direction		Displays the decision flow with a horizontal or vertical orientation and a left to right or right to left direction.
Print		Prints the decision flow. Depending on the supported browser you are using and the preferences you have set for your printer, you may see a preview of the decision flow.

Table 36: Decision Flow Editor Toolbar Commands (continued)

Command	Icon	Description
		Note Depending on size of your decision flow and the page orientation setting of your printer, the decision flow will be printed with wrapping.

Decision Flow Floating Toolbar Commands

The Decision Flow Floating toolbar commands are used to create or paste new decision flow elements.

Click anywhere between the start node and the end node of a decision flow to see an insertion point. When you place your cursor over the insertion point, the floating toolbar is displayed.

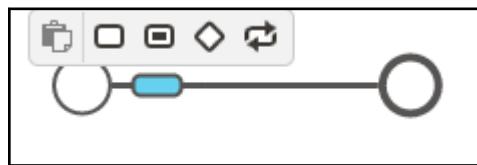


Figure 40: Floating Toolbar

Table 37: Decision Flow Floating Toolbar Commands

Command	Decision Flow Display	Description
Paste icon	None	Pastes a decision flow entity that you cut or copied using the icons on the Decision Flow editor toolbar.
Task icon		Executes an associated entity.
Subflow icon		Invokes another decision flow with its own nodes.
Split icon		Permits processing along each branch based on a boolean condition.

Table 37: Decision Flow Floating Toolbar Commands (continued)

Command	Decision Flow Display	Description
Loop icon		Conditionally repeats a node. There are different control constructs available.

Related Links

- [Defining Tasks](#)
- [Defining Subflows](#)
- [Defining Split Nodes](#)
- [Defining Loops](#)

Creating Decision Flows

Create a decision flow to define the sequence in which the decision entities are executed when data is passed into the project.

- 1 In the **Project Explorer**, select the folder where you want the decision flow created.
- 2 Click the **New** drop-down menu.
- 3 Select **Decision Flow**.
- 4 Enter a unique name in the **Name** field. Every decision entity must have a unique name in the project.
- 5 Click **Create**.
- 6 Click on the start node or end node or Decision Flow name in the canvas to display the details pane to the right side of the canvas.
- 7 Select an **Input type** from the drop-down list. By default, the Input type is the class you selected when you configured the decision service.
If there is a global variable of the same class type as the Input type, you see the variable name displayed in the **Assign input to** field.
- 8 (Optional) You can use the **Decision flow variables** table to add flow variables to your decision flow.
- 9 (Optional) Enter the purpose of the decision flow in the **Comments** field.
- 10 Click **Apply**.

Related Links

[Naming Decision Entities and Folders](#)

[Inserting Decision Flow Entities](#)

[The Details Pane](#)

The Details Pane

The details pane contains the fields and drop-down lists where you can define the properties for a given decision flow or decision flow element.

To edit decision flow properties in the details pane, select the start node. To edit the properties of a decision element, such as a task or loop, select the element in the canvas. What you see in the details pane is specific to the decision flow entity that is selected in the decision flow. However, there are areas of the details pane that are common to most decision flow elements.

Table 38: Features in the Details Pane

Area	Description
Reference Name field	A unique name for the decision entity in the project. A valid reference name consists of one or more letters (A-Z, a-z), digits (0-9), and underscores (_). It cannot contain spaces, punctuation marks, or special characters, such as \$ or %.
Open Editor icon	Opens the Rule Builder editor where you can create a condition. You see fields in the details pane for split and loop decision flow entities.
Comments text area	Enter information about the decision flow entity for other users who may access it.
Navigate to the Entity icon	Closes the Decision Flow editor and opens an editor where you can view the decision entity.
Preview the Entity icon	Opens the Preview pane located near the bottom of the Decision Flow editor to display the entity associated with the decision flow entity.
Apply button	Registers the changes you have made to the canvas.

Decision Flow Variables

Decision flow variables can be used to define conditions or pass values from one task to another task in a flow. The decision flow variables can only be used within a flow.

In Edit mode, you can create a new decision flow variable by clicking on the start node of a decision flow to display the Details pane.

Creating a Flow Variable

You create a flow variable when you want to pass values from one task to another task in the flow or you want to define branch conditions.

- 1 Click **Add** on the context toolbar to add a flow variable row to the **Decision flow variables** table.
- 2 Click **Apply** before clicking on the default flow variable name.
- 3 Enter a name for the flow variable in the **Name** field.
- 4 Select a type from the **Type** drop-down list.
You can select types including, boolean, date, duration, integer, money, real, string, time, timestamp, object, and any classes you have in your Business Object Model. Depending on the type you selected, the Initial Value field will display an appropriate default value.

 **Note** If you have a scorecard in your project, you can use the NdScoreModelReturnInfo support class as the type.

Setting a Decision Flow Variable, Parameter, or Return Type

When creating a decision flow variable or parameter, or return type, you can select a type and set an initial value.

Value Types

There are three different value types:

- **Value (default)**
This is a fixed value that is automatically added by default.
 **Note** This option is not available for return types.
- **Reference**
The reference is to an existing variable (local or global) or objects that are of matching type.
- **Expression**
You can use the Rule Builder to set an object property or use a built-in function.

Table 39: Description of Value Types

Type	Default Value	Reference	Expression
boolean	Default initial value is true.	If there is a variable of boolean type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type boolean.

Table 39: Description of Value Types (continued)

Type	Default Value	Reference	Expression
date	The default value is the current month, date, year displayed as mmm, dd, yyyy. Use the calendar widget to select a different date value.	If there is a variable of date type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type date.
duration	The default value is 1 second.	If there is a variable of duration type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type duration.
integer	The default value is 0.	If there is a variable of integer type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type integer.
money	The default value is USD 1.0.	If there is a variable of money type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type money.
real	The default value is 0.	If there is a variable of real type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type real.
string	The default value is 0.0.	If there is a variable of string type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type string.
time	The default value is the current time displayed as hh:mm:ss AM/PM.	If there is a variable of time type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type time.
timestamp	The default value is the current date and time displayed as mmm dd, yyyy hh:mm:ss AM/PM.	If there is a variable of timestamp type, this variable will be available in the drop-down list.	Create an expression that evaluates to a value of type timestamp.
object	The default value is null.	Select another object of the same object type.	Create an expression that evaluates to a value of type object.
Class in the Business Object Model	The default value is null.	Select a class or variable of the same Class type.	Create an expression that evaluates to a value of the class you selected as your type.

Table 39: Description of Value Types (continued)

Type	Default Value	Reference	Expression
NdScoreModelReturnInfo (Support class for the Scorecard.)	The default value is null.	Select a class or variable of the same Class type.	Create an expression that evaluates to a value of the class you selected as your type.

Inserting Decision Flow Entities

Insert or paste a task, subflow, split, or loop to the decision flow using the Decision Flow Floating toolbar.

- 1 Click anywhere between the start node and the end node to add an insertion point. Place your cursor over the insertion point to display the Decision Flow Floating toolbar.
- 2 Select the desired command from the toolbar.
- 3 In the details pane on the right side of the canvas, define the decision flow entity.
The fields that are displayed depend on the entity you inserted.

Related Links

- [Decision Flow Editor Toolbar Commands](#)
- [Decision Flow Floating Toolbar Commands](#)
- [Defining Tasks](#)
- [Defining Subflows](#)
- [Defining Split Nodes](#)
- [Defining Loops](#)

Defining Tasks

A task represents a unit of work that is completed by a decision entity such as a function or a ruleset. Define a task by entering a label name, selecting a decision entity and, if applicable, assigning a value to one or more parameters and a return type value.

- 1 Enter a name in the **Label** field. This name displays in the task node in the decision flow.
- 2 Select a decision entity to associate with the task.
The drop-down list shows the full signature for each available decision entity.
- 3 If the decision entity has one or more parameters in its signature, assign values to the parameters. Parameters can be set as a value, reference, or an expression.
- 4 If the decision entity has a return value, assign a return value of a matching type. The return value can be set as a reference (default) or as an expression.

- 5 (Optional) Enter information in the **Comments** field for other users who may need to maintain the decision flow.
- 6 Click **Apply** to register the changes.

Related Links

[The Details Pane](#)

Defining Subflows

Define a subflow that represents a segment of the execution flow. A subflow is a decision flow that represents a well-defined unit of work within a larger execution flow. From a design point of view, using a subflow allows you to partition the tasks in your execution flow to create a set of smaller, reusable decision flows.

As you analyze the execution flow of your business process, you may choose to either:

- Break down the high-level tasks into many smaller tasks visible in the decision flow diagram.
- Create a set of smaller decision flows to represent each high-level task and insert them in a super decision flow as subflow nodes.

If applicable, parameters and return values can be assigned to the subflow. A value, a reference to another entity, or an expression can be assigned to a parameter or return value.

Related Links

[The Details Pane](#)

[Setting a Decision Flow Variable, Parameter, or Return Type](#)

Defining Split Nodes

Insert a split node when you need to add conditional branches to your decision flow.

Define a split by entering a label name and then creating a condition expression for each branch using the **Rule Builder**. For example, this is a condition:

```
customer.accountType is "standard"
```

If the data that is passed into split node matches one of the branch conditions, the data is evaluated by the decision flow entities on that branch.

You can use special value keywords when creating condition expressions to test for the presence of a value not specified in the XML payload, and the property is used in the decision logic. The keywords are **available**, **unavailable**, **known**, **unknown**, and **null**.

Related Links

[The Details Pane](#)

Defining Loops

Define a loop when you want to repeat the execution of one or more tasks a specified number of times, for each instance in a collection, or until a certain condition has been satisfied.

You define a loop by selecting a loop type and entering a label name in the details pane. The loop type determines the fields you see in the details pane.

Table 40: Types of Loops

Loop Type	Execution of one or more tasks
Repeat	Iterates specified number of times or while a condition you specify is satisfied.
For each	Iterates for each instance of a given collection. To use a For each loop you can have a top-level global variable that is an array of complex type or a collection property of complex type. Variables of collection type are not currently supported.

Related Links

[The Details Pane](#)

Looping Through a Complex Type Collection Property

The use of a complex type collection property in a Decision Flow is not supported in *for each* loops. If you need to use a complex type collection property, use a *repeat* loop.

- 1 In the Decision Flow editor, select the start node and create a flow variable to hold the total count/size of the collection as an integer type with an initial value of 0.
For example, `totalCount=0;`
- 2 Create another flow variable to hold the current position in the collection as an integer type and give it an initial value of 0.
For example, `currentPosition=0;`
- 3 Insert a **Task**, enter a label and select the function you created to get the total count/size as your **Implementation** and assign the flow variable you created to hold the total count/size to the **Return value** field.
- 4 Insert a **Loop** with a **Loop type** of **Repeat**.
- 5 Assign the flow variable you created to hold the current position to the **Initial assignment** field in the Rule Builder.
For example, `currentPosition=0;`
- 6 In the **Repeat While** field, create a condition using the current position and the total count/size flow variables in the Rule Builder.
For example, `currentPosition is smaller or equal to totalCount;`
- 7 Increment the current position flow variable in the **After each iteration do** field.
For example, `currentPosition=currentPosition+1;`

8 Save your changes.

CHAPTER 16

Searching for Entities Using a Query

Use a query to search for entities in your project using multiple criterion. With a query you can specify detailed criterion, run the query, and then save it for reuse.

Using a query, you can search for entities based on the following criterion:

- Name of an entity as listed in the Project Explorer.
- Template on which an entity was based.
- Display text and values on a web page in any of the entities.
- Location of the entity.
- Date the entity was created.
- Author of the entity.
- Historical information.

The search fields in a query can be used individually or with other search terms. If more than one search term is used, all search terms must be satisfied before any results are returned. Any fields where a search term has not been defined are ignored or default values are used.

 **Note** If you just need to quickly search for an entity using a text string, use the Search field in the Global Command Bar.

Creating and Running Standard Business Queries

Use the Query editor to create and run a query to locate specific entities.

- 1 In the **Search** pane, select **New Search > New Search Query**.
- 2 Under **Search for any policy**, use one or more of the following fields to build the search:
 - a **whose name** field: Retain the `contains` match text operator or select another text operator and enter a string or a regular expression for an entity name. For example, to search for the PricingRuleset, you could enter `Pricing*` or `^Pricing` to locate the entity. The text operator drop-down contains a list you can use to constrain the query. If you are using the `exactly matches` or `is equal to` operators, enter the full, case-sensitive name.

- b **whose type** field: Retain the contains match text operator or select another text operator and enter a string or a regular expression to locate all of the entities based on a specific template name. For example, Loan Approval*. In this case, any entities based on a template with the words Loan Approval would be returned.
 - c **whose display text** field: Retain the matches all words in option or select another option and enter a case-sensitive string.
 - d **located in** field: Click the down arrow to display a choice of locations. Choose one of the following locations:
 - **Anywhere** (default)—The project will be searched for matching entities.
 - **Folder**—A specific folder will be searched for matching entities. Enter a path from the root folder of the decision service to the subdirectory that you want searched.
For example, folder/subfolder/subfolder.
If you use the exactly matches or is equal to text operators, enter the full, case-sensitive path name for the folder.
 - e In the **last modified** field, click the down arrow to display the following duration options.
 - **Anytime** (default)
 - **After a date**—Queries for entities modified after the date you enter.
 - **Before a date**—Queries for entities modified before the date you enter.
 - **Between 2 Dates**—Queries for entities modified between the two dates.
 - f In the **by** field, click the down arrow to display possible ways to search according to user name:
 - **Anytime** (default)
 - **One of the following Users**—If this option is selected, you can use a text operator from the drop-down list and enter a case-sensitive text string or regular expression to constrain the query.
- 3 Click the **Search** icon on the Query editor toolbar to start the search. The results are also saved in the Search pane so you can refer to them at anytime during the current session.
- 4 (Optional) Click the **Save** icon or click the **Check in** icon.

Query Field Definitions

Supplemental information is available about the query fields.

Table 41: Supplemental Information About Query Fields

Search Field	Behavior
whose name	When used on its own, a query based on this field returns any top-level entity whose name matches the string or regular expression entered in the field.

Table 41: Supplemental Information About Query Fields (continued)

Search Field	Behavior
whose type	When used on its own, a query based on this field returns any top-level entity based on a template that matches the string or regular expression entered in the field.
display text field	When used on its own, a query based on this field returns any entity that contains the display text (string) or regular expression entered in the field. You can refine a display text query by restricting the search to the entities matching the string entered in the whose name field or the string entered in the whose type field.

Additional Search Criteria

There are two additional query conditions: Management Property and Historical Information.

- Management Property—This condition requires that the top-level entities contain a management property that matches the specified criteria. More than one management property condition can be used.

 **Note** This condition is applicable only if the project was created in Blaze Advisor and management properties were used with the decision entities.
- Historical Information—This condition searches for the latest version of a decision entity or a comment in the latest version.

These conditions can be used separately or in conjunction with each other. All of the conditions used in a query must be satisfied before a result can be returned.

Search conditions are added using the toolbar shown in this figure.

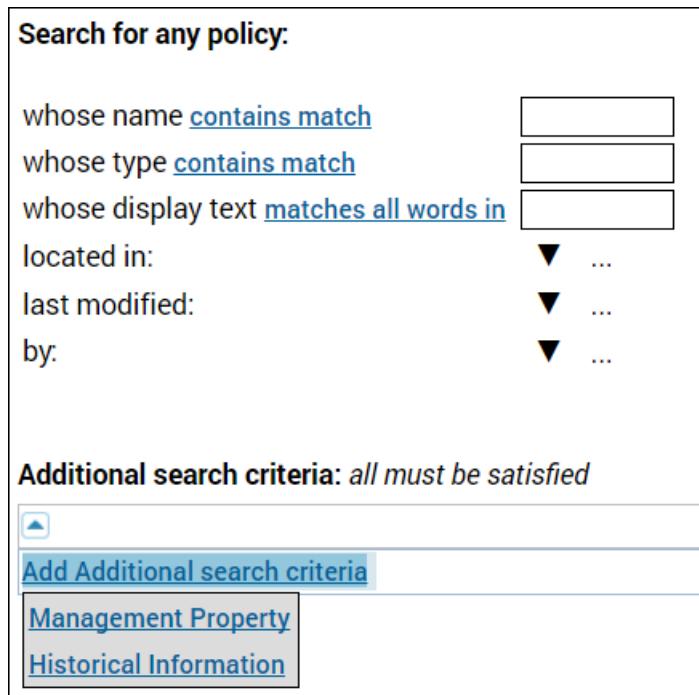


Figure 41: Toolbar for adding additional search criteria

CHAPTER 17

Comparing Differences Between Entities

Compare the differences between entities using a comparison query. You can compare a prior version of an entity to another prior version of an entity or to the current version.

You can use one or more filters to narrow the query. For example, you could use a filter to compare only the rulesets authored by a specific user. Additionally, you can view comparisons between two different entities in a separate editor. Like other queries, these queries can be saved, edited, and run again. After running the query, the results appear in the Comparison Criteria Results Merge editor.

Creating and Running Comparison Queries

Use the Comparison Query editor to create and run comparison queries.



Note

- To see the latest content in your workspace, click **Update** on the Project Explorer toolbar before creating or running a query.
- For consistent results, add the decision entity on the left side of the comparison and the decision entity with the potentially different values on the right side.
- The placement of the entity is significant if you anticipate merging the comparison results from the **Source** (left column) to the **Target** (right-hand column). The entity referenced on the right side should be the one you want to change if differences are detected.



Tip For best results, when creating entities that you may want to use for later comparisons, use distinct entity names so that you can easily differentiate between the entities in your comparison results.

1 In the Navigation pane, click the **Compare** pane to expand it.

2 Click **New Comparison**.

3 From the **Compare** menu, select the scope for the comparison.

You can choose the **Projects**, **Project folders**, or **Project entities** level scope for your comparison.

- If you select **Projects**, you can select the **Compare instances only** check box to limit the comparison to instances in your project.
- If you select **Project folders**, you can select the **Compare instances only** check box and/or the **Compare subfolders** check box to compare the contents in folders and subfolders.

- If you select **Project entities**, you can compare different versions of the same entity. This option is not applicable for the decision trees.
- 4 Select one of these options from the **Version** drop-down menu.
Because folders are not explicitly versioned, there are no previous versions to compare.
- **Local version**
The current edited version.
 - **Base version**
The working version of the last update.
You can compare the Local version to the Base version to track changes since the last update.
 - **Latest in repository**
The latest version in the repository.
You can compare the Latest in repository version against the Base version to see if your entities are out-of-date and your workspace needs to be updated.
 - The number representing a prior version such as 1, 2, 3....
Any version numbers in the list match the ones displayed in the History tab for the entity.
- 5 (Optional) Configure filter criteria.
- a Add one or more filter conditions to narrow the comparison criteria.
 - b Select **View** to make the filter criteria read-only.
- 6 Click the **Compare** icon to run the query.
- 7 (Optional) Click the **Check In** icon to store the query in the versioning service.

Related Links

[Comparison Query Results](#) [Merge Actions](#)

Comparison Query Results

When a comparison query is run, the results are displayed in the **Comparison Criteria Results Merge** editor. The editor allows you to merge differences detected during the comparison process.

The results are displayed in a table with three columns. The **Source** column is located on the left side and the **Target** column is located on the right side. The **Source** and **Target** columns are divided by a narrower **Merge** column that displays the merge actions that are available for the entities.

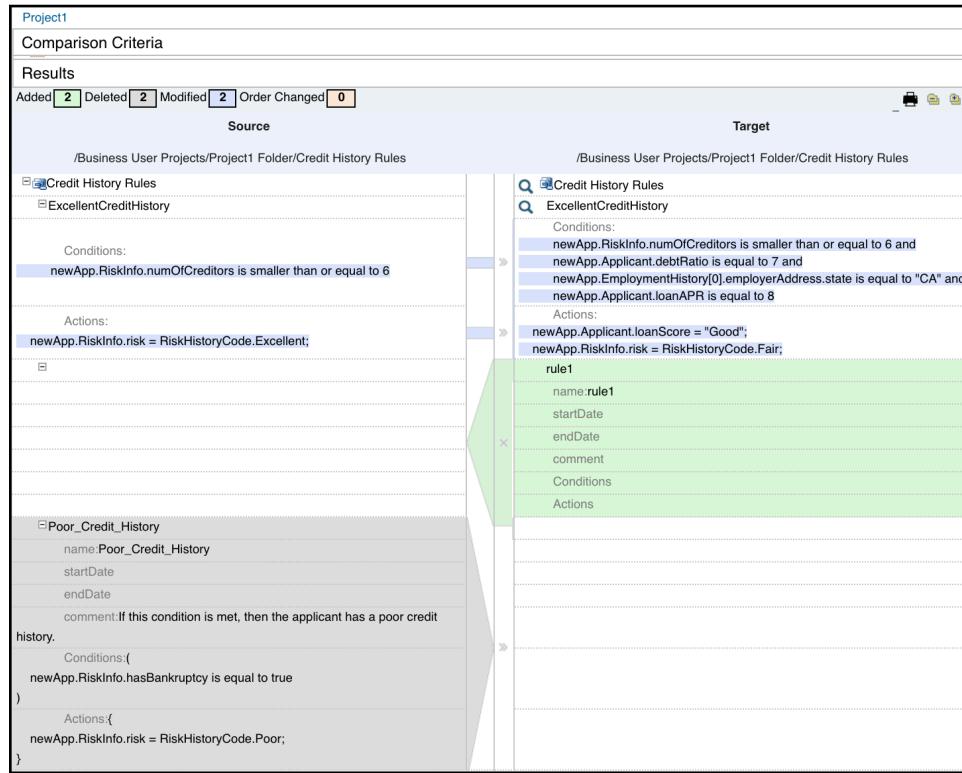


Figure 42: Example of a Query Performed on Two Different Versions of a Ruleset

To see the rule contents of an individual rule, open the Visual Difference dialog by clicking the **Visual Differences** icon in the **Target** column.

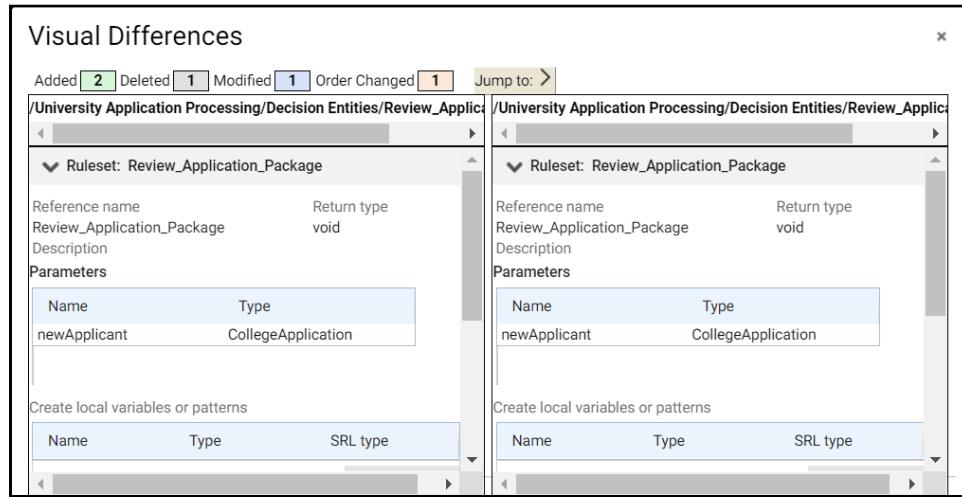


Figure 43: Visual Differences Dialog

In the **Visual Comparison** editor, if there are additional differences in the values within an entity that can be viewed, a **Jump to** drop-down menu displays in the upper right corner of the editor. Select a rule in the drop-down menu to open a separate **Visual Comparison** editor that displays just that portion of the difference. For example, if you run a comparison query to compare versions of a ruleset and

differences are detected in a rule, you can open the **Visual Comparison** editor window to see the differences between the rule and view the modified values.

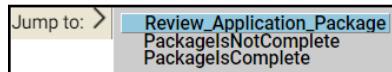


Figure 44: Example of Jump to Menu

Note The **Visual Comparison** editor is not available for decision trees.

Related Links

[Creating and Running Comparison Queries](#)

Comparison Query Results Merge Actions

In the **Comparison Criteria Results Merge** editor, you can merge one or more differences using the commands in the **Merge** column.

The commands in the **Merge** column allow you to selectively incorporate differences detected in a comparison query. If a merge icon is displayed but disabled, place your cursor over the icon to obtain more information about the reason why merge action is restricted or disallowed. There are several reasons why an entity cannot be merged including:

- The target entity may be locked by another user.
- The target entity needs to be checked out before it can be merged.
Open the entity and check it out or check it out from the Project Explorer.

If a merge icon is displayed, place your cursor over the icon to see the tooltip name. Some icons look similar but may have a different meaning depending on the color associated with the row in which they are displayed.

Table 42: Descriptions of Merge Icons

Comparison Summary Category	Merge Icon	Description
Deleted	Add »	If you have entities that are present in the Source but not in the Target, you can use the Add icon in the Merge column to add those entities to the Target.
Modified	Replace »	If a difference between the value or type in the Source and the value or type in the Target is detected, you can use the Replace icon in the Merge column to overwrite the value or type in the Target with the value or type from the Source.
Added	Delete ✘	If you have entities that are present in the Target but not in the Source, you can use the Delete icon in the Merge column to remove those entities from the Target.

Table 42: Descriptions of Merge Icons (continued)

Comparison Summary Category	Merge Icon	Description
Order Changed	Reorder 	If you have entity contents in the Source and the Target where the order of the contents have changed, you can use the Reorder icon to overwrite the order in the Target with the order from the Source.

Limitations in Comparison Queries

These limitations exist for comparison queries in this release.

- Comparison queries are not supported for comparing decision trees.
- The **Visual Difference** editor is not available for decision trees.

CHAPTER 18

Detecting Anomalies in Entities

Create and run verification queries to detect anomalies in entities such as rulesets and decision tables. An anomaly is a logical problem that is evaluated by using one or more algorithms to highlight areas of potential conflict, inefficiencies in your rule architecture, and semantic errors. You can run verification tests on one entity or configure a verification query to run the tests on all or a subset of entities in the project.

Verification queries are supported for the following entities:

- rulesets
- functions
- decision tables
- decision flows
- scorecards
- decision trees (partial support only)



Note When the Verifier is run on a decision tree, the Usage Tests are not run even if the check boxes for those tests are selected in a verification instance.

There is a toolbar icon in most editors that you can use to run a default set of verification tests for that entity type. If you want the decision logic in more than one entity to be evaluated or you want to select which verification tests to run, you must create and configure a verification query and run it in the Verify pane.

The Verifier runs all tests that are appropriate for each entity type or just the tests you select. It automatically disables any tests that are not applicable or that provide unnecessary feedback. For example, decision tables allow the creation of a large number of rules with identical conditions and actions. As a result, the Verifier automatically disables any test that generates warnings about these types of anomalies for decision tables. The Verifier runs only those tests that provide meaningful feedback.

After the Verifier is run, if any anomalies are found, a message is printed for each one along with the name of the entity where the problem was detected. The Verifier reports errors and it also generates warning and informational messages that provide you with an opportunity to correct any potential problems. While you do not have to address an anomaly generated by a warning or informational message, it is recommended that you review the entity where the anomaly was diagnosed to see if it is a potential problem or it is a result of your decision logic.

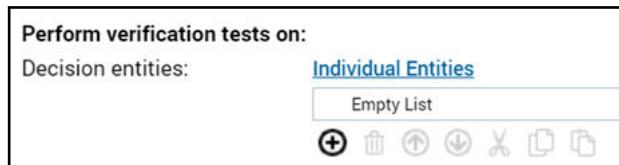
Creating and Running Verification Queries

When you create a verification query, you can select which verification tests to run and specify whether the tests should be performed on all entities in the project or just a subset.

There are some types of SRL expressions that are not supported for verification testing. By default, the Verifier does not print messages for unsupported expressions when it encounters them. However, you can select the **Conditions not evaluated by the verifier** option in the verification query and the Verifier will print messages for any expressions it does not evaluate.

To create and run a verification query:

- 1 Expand the **Verify** pane.
- 2 Click **New Verification Query** to create a new Verification Query.
- 3 Using the **Decision Entities** menu, retain **All** to run the applicable tests on all decision entities or click **All** and select **Individual Entities**.
If **Individual Entities** is selected, the **Empty List** is displayed.
 - a Click in the row with the words **Empty List** to enable the toolbar.
 - b Click the **Add** icon to enable the drop-down menu where you can select the entity name.



- c Select the entity and click **Add**.
- d Repeat Steps 3b and 3c for each entity that you want tested.
- 4 (Optional) Disable any tests that you do not want run.
- 5 (Optional) Scroll down the list of tests and select **Conditions not evaluated by verifier** to display messages about the condition expressions that are not evaluated.
- 6 Click the **Verify** icon on the toolbar.
- 7 (Optional) Click **Verification Criteria** located at the top of the page to configure the verification query again to run other tests.
- 8 (Optional) Click the **Save As** icon or click the **Check In** icon

Verification Query Results

After the Verifier is run, it prints a message about each anomaly in the Results pane or it generates a message saying that no verification results were found. Each type of anomaly is associated with an icon, a message, and the name of the decision entity where the problem was encountered.

This table shows the three levels of severity and what each level means for the discovered anomalies:

Table 43: Levels of Severity

Symbol	Type of Message	Description
	Error message	A serious error has been detected and it will probably affect the efficiency and outcome of the decision logic.
	Warning message	An anomaly has been detected; however, it may or may not affect the efficiency and the outcome of the decision logic.
	Information message	The information is provided as a convenience, but there is no anomaly associated with the message that will prevent the decision logic from executing as expected.

The Verifier diagnoses an anomaly as erroneous if there is a combination of input parameters that could lead to a compilation error or unpredictable behavior. If the anomaly does not generate erroneous conditions, but leads to wasted resources, such as initialized variables that are not used, then a warning is generated.

CHAPTER 19

Preparing a Project for Testing and Deployment

This information applies only if you created new projects in your Decision Modeler workspace. It does not apply if your workspace contains projects imported from Blaze Advisor.

To successfully deploy a project, any business term sets referenced in the decision entities must be initialized. When a business term set is created, a global variable of the class type selected in the Business Term editor is automatically generated. This variable includes an `input` property. This `input` property must be assigned to the `input` property of the global variable that represents your imported object model.

For example, if you create a business term set called `AccountTermSet`, a global variable is automatically created called `varAccountTermSet`. If the business term set extends the `LoanApplication` class, you must assign `varAccountTermSet.input` property to the `LoanApplication.input` property.

The **Initialization entity** provides a mechanism for doing this without having to write any code. The business term set fields in the **Initialization entity** are only available after a business term set is created in the project.

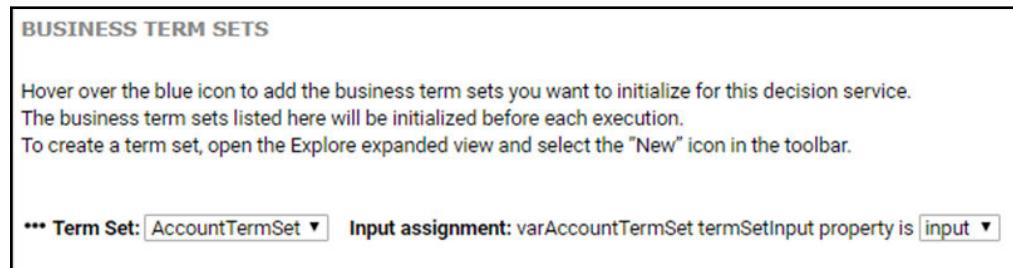


Figure 45: An example of how the initialization appears in the editor

If the business term set is deleted after it has been initialized and used in the decision logic, modify the **Initialization entity** and remove any references to the business terms set from the rules.

If a project has more than one decision flow or the default decision flow has been renamed, use the **Initialization entity** to select a decision flow to invoke your project. Click **Edit in Rule Builder** and use the **Item Selector** to select a decision flow for the invocation or click **Edit in Advanced Builder** to edit the decision flow name and signature.

CHAPTER 20

Deploying Decision Services

The projects in a workspace move through three environment in the product lifecycle: Development, Staging, and Production.

- **Development**

When a project is created in a workspace or a Blaze Advisor workspace is imported into a workspace, the projects are in the Development environment. This is primarily where the decision logic is authored and unit testing occurs.

- **Staging**

Projects in this environment undergo more thorough testing, which may include batch processing to simulate the volume of data that may be run through the decision service in Production.

Any projects that pass the testing phase in the Staging environment are marked as Approved. Any projects that do not pass the testing phase are marked as Rejected and return to the Development environment. After the required changes have been made, the project can be resubmitted from the Development environment to Staging. Projects that are Approved in the Staging environment are ready to be deployed to the Production environment.

- **Production**

Projects in this environment have successfully passed testing and are ready to be used in an enterprise-wide application to return a decision.

The first time projects in a workspace are deployed to any environment, the environment is being created so the process takes a few minutes. Creating or deploying additional projects or subsequent versions of a project in the workspace will be faster because the environment already exists.



Note The default execution mode for deployed decision services is compiled sequential. For projects imported from Blaze Advisor, during decision service deployment Decision Modeler honors the execution mode that is specified in the project.

This is the process for deploying decision services:

- 1 Select the row in the table containing the workspace.
- 2 Click the **Manage deployment** drop-down menu on the Projects toolbar and select **Submit to staging**.

- 3 After the project or projects have been tested. Select **Change status** from the **Manage deployment** drop-down menu on the Projects toolbar.
If you have approval rights, you can approve the project for deployment to Production or reject the project and return it to the Development environment.
 - a Select **Approved** for projects that have passed your testing criteria.
 - b Select **Deploy to production**, if you are authorized to deploy the projects into Production.

Obtain the URLs for the decision service endpoints for each environment in the project lifecycle using the **Project links** icon on the Projects toolbar.

Batch Processing

You can create a batch job to configure, schedule, and run batch processing. A batch job reads in a set of data records and the data are evaluated by a decision service.

A batch job can be run in the Development, Staging or Production environment as long as the project has been deployed to that environment. You must upload data separately for each environment.

To run a batch job, select a workspace, click the **View batch jobs** command on the Projects toolbar. The Batch window opens in a separate tab. The first time you click **View batch jobs**, a service for configuring and running batch jobs is created for that workspace. After the service is created, the user interface opens where you can configure and run batch processing.

Detailed information about using batch processing is available in the Decision Management Platform Help system. To access the DMP Help system, select **View Solutions** on the Decision Modeler Projects page. Another browser tab opens the DMP. Click the **Help** link in the banner and search for Batch Processing for a Solution.



Note The term component in the DMP Help system is represented as a workspace in Decision Modeler. Any information about running Spark jobs is not applicable to Decision Modeler workspaces.

Requesting Access to a Decision Service

OAuth 2.0 bearer tokens authorize secure access to decision services. You can use OAuth authorization to specify access based on roles and types of data. The Decision Management Platform provides a REST API for obtaining an OAuth bearer token for a service.

For more information about OAuth 2.0, see the Decision Management Platform documentation.

Obtaining the Client ID and Secret

When you are ready to deploy a project as a decision service, you must obtain the client ID and secret to request a bearer token.

The Decision Management Platform has a REST API that can be used to obtain the bearer token by passing the client ID and secret as parameters.

- 1 In the Projects page, expand the workspace and select the project.
- 2 Click **Project Links** command on the Projects toolbar.
- 3 Click the **Client ID** tab to obtain the client ID and secret.

Related Links

[Viewing Project Links and the Client ID](#)

Obtaining the Bearer Token

The Decision Management Platform (DMP) has a REST API that can be used to request an OAuth bearer token for decision service requests. The bearer token is valid for 60 minutes.

You need two pieces of information to obtain the bearer token: the client ID and secret, and the Request URL. The Request URL is the server environment name for your DMP installation as well as the path to the API. The URL takes this form: `https://<environment>/registration/rest/client/token` and can be supplied by your DMP administrator.

You can use any browser REST client to send your request. For example you can use FireFox RESTClient or the Google Chrome Advanced REST Client. The steps for entering a request for a bearer token are similar for each client.

- 1 Enter the Request URL.
- 2 Use the POST request type.
- 3 Specify the Header content type as `application/json`.
- 4 Supply the `client ID` and `secret` as parameters in the JSON object.
For example:

```
{  
  "clientId": "4xxb75k1vx",  
  "secret": "amBrou4R79q0ryr2tnRJ9T0*qBiJ629980M8"  
}
```

- 5 View the response that should contain the bearer token.

Using the Bearer Token for Testing

You can use the bearer token to send requests to your decision service during the testing phase.

To test your decision service, download or install an interface that will allow you to enter your bearer token, connect to your decision service, send request data, and

retrieve a response. To successfully send data to your decision service, you must have an active bearer token.

This example uses the SOAPUI to show you how to use your bearer token to send requests and view the response.

- 1 Create a project and add your WSDL that contains your decision service endpoint URL.
The endpoints for your environment are available in the **Project Links and Client ID** dialog.
- 2 Add a Request Header that contains your active bearer token.
 - a Enter Authorization for the HTTP Header name.
 - b Enter the active bearer token and takes the form:
Bearer<space><active bearer token>.
- 3 You need to enter values for the request.
- 4 Submit the request.
- 5 View the response after the request is processed by the decision service.

SOAP and REST Endpoints in Decision Services

When configuring a project with Java POJO inputs/output, REST endpoints are generated along with the SOAP endpoint.

The main difference between SOAP and REST endpoints is the way that data is marshaled. In the case of a SOAP endpoint, the data is marshaled between XML and Java POJO. For REST endpoints, the data is marshaled between JSON and Java POJO. JSON is supported only for REST deployments.

For a SOAP endpoint, the .wsdl describes each of the available operations and the operations are specified with a single request. For a REST endpoint in a project, the last part of each path is the operation.

For example, a project with entry points is defined as follows:

```
processWithDecisionFlow(AutoPolicy)  
processBatchWithDecisionFlow(fixed array of AutoPolicy)
```

In a SOAP deployment, these entry points will result in a single URL endpoint represented by a WSDL.

In a REST deployment, these entry points will result in two URL endpoints.

 **Note** When using an entry point functional with a French accent character in the name, use URL encoding.

```
/rest/service/processWithDecisionFlow  
/rest/service/processBatchWithDecisionFlow
```

Another significant difference between SOAP and REST endpoints is that in a SOAP deployment, multiple inputs can be specified. With REST only one JSON input file is allowed, although the input could have multiple parameters. For this

reason, when a project entry point with multiple inputs has been selected, a wrapping input object is constructed as the input to the REST service. When marshaling data with a single parameter or multiple inputs, the parameter must be specified in JSON.

 **Note** You can use a tools such as SOAP UI or Postman to test your REST deployment locally.

The JSON can include:

- Double quotes can be used with numeric and boolean values as well as string values. (Postman allows numeric and boolean value to be specified without quotes.)
- Arrays need to be specified with square brackets.

For example, for a given entry point, `processWithDecisionFlow(AutoPolicy)` where `AutoPolicy` only has one property, `policyID`, the request payload would be:

```
{
  "autoPolicy" : {
    "address": {
      "addrType": "BillingAddress",
      "city": "Denver",
      "country": "America",
      "zip": "98374"
    },
    "customer": {
      "communication": {
        "email": "ssirmons@fico.com",
        "phone": "(286) 784-9224"
      },
      "creditScore": "664",
      "drivingRecord": {
        "accidentInfo": [
          {
            "accidentDescription": "9iaMy4K",
            "accidentTypeCd": "Property_Damage",
            "atFaultInd": "false",
            "bodilyInjuriesInd": "false",
            "dateOfAccident": "1989-09-16T00:00:00Z"
          }
        ],
        "anyAccidentInLast3Years": "false",
        "anyMovingViolationsInLast3Years": "false",
        "licNumber": "H 3",
      }
    }
}
```

Whereas, for a given entry point, `processWithDecisionFlow(AutoPolicy, AutoPolicy)` where there are two `AutoPolicy` inputs, the request payload would be:

```
{
  "autoPolicy" : {
    "address": {
      "addrType": "BillingAddress",
      "city": "Denver",
      "country": "USA",
      "zip": "98374"
    },
    "customer": {
      "communication": {
```

```

        "email": "ssirmons@fico.com",
        "phone": "(286) 784-9224"
    },
    "creditScore": "664",
    "drivingRecord": {
        "accidentInfo": [
            {
                "accidentDescription": "9iaMy4K",
                "accidentTypeCd": "Property_Damage",
                "atFaultInd": "false",
                "bodilyInjuriesInd": "false",
                "dateOfAccident": "1989-09-16T00:00:00Z"
            }],
        "anyAccidentInLast3Years": "false",
        "anyMovingViolationsInLast3Years": "false",
        "licNumber": "H 3",
    },
    {
        "autoPolicy" : {
            "address": {
                "addrType": "BillingAddress",
                "city": "California",
                "country": "USA",
                "zip": "95110"
            },
            "customer": {
                "communication": {
                    "email": "jdoe@fico.com",
                    "phone": "(408) 583-4783"
                },
                "creditScore": "750",
                "drivingRecord": {
                    "accidentInfo": [
                        {
                            "accidentDescription": "8hyNz8P",
                            "accidentTypeCd": "Personal_Injury",
                            "atFaultInd": "true",
                            "bodilyInjuriesInd": "true",
                            "dateOfAccident": "2019-10-09T00:00:00Z"
                        }],
                    "anyAccidentInLast3Years": "false",
                    "anyMovingViolationsInLast3Years": "true",
                    "licNumber": "D 1",
                }
            }
        }
    }
}

```

Finally, because there is no formally required way to describe a REST schema as there is with SOAP (.wsdl), we are following Decision Management Platform (DMP) conventions by providing a `swagger.json` to describe the REST endpoint schema. The path to the `swagger.json` is:

`<serverName>/DecisionExecutor/rest/swagger.json`

 **Tip** You can copy and paste the `swagger.json` contents into a browser to see the REST endpoint schema or you can copy and paste it to a Swagger online editor to see the JSON file in more a readable format.

 **Note** Only POST methods are supported for REST service requests.

Date, Time, and Time Zones for Decision Services

Date, time, and time zone formats for service requests are the same for REST and SOAP in Decision Modeler projects.

Web services are potentially used worldwide where the client can be located in a time zone in the Pacific United States and the server can be located on the other side of the world in Europe or Asia. This has an impact on how time and date information is passed to and from the web service. The problem is that the local time on the client side might differ by several hours from the local time on the server side or daylight savings time might be observed on the client but not the server.

If you need to supply the time in another time zone as part of an input, you must calculate the time in your time zone supplying the offset in hours relative to GMT (Greenwich Mean time) making the time information absolute to GMT.

In Decision Modeler, REST web services need to use the same date time format as SOAP web services.

For example: YYYY-MM-DDT:HH:MM:SS-GMT would be written as:
`<licensedSince>2020-09-16T00:00:00-07:00</licensedSince>`.

As a best practice, any dates in the input should always specify a time zone. If the time zone is not specified, the server time zone is the default. The response will always have a date based on the server time zone.

Visualizing Data

If you are using the cloud version of Decision Modeler and you have a license for Tableau Server, you can create reports based on the execution data for your projects.

The Decision Modeler Projects Page provides a convenient way to deploy your projects to the Decision Management Platform (DMP). The DMP includes the Analytic Data Mart (ADM) for capturing and retrieving solution event data. ADM captures data from the projects in your workspace and makes it available for reporting and analysis.

ADM consists of two data stores-- the event store and the report store. The event store is a log of past decisions and the report store is an analytical data store. Together, these data enable you to improve decision models and to track progress against key strategic and operational goals.

Each time a decision service is invoked, event data about the invocation and the environment is captured and persisted in the Analytic Data Mart. Depending on the settings you chose for the workspace, the input and output payloads of the decision service requests may also be captured. The data can be accessed from a dedicated Tableau Server site, where you can develop the reports. The site contains a separate Tableau project for each workspace environment: Development, Staging, and Production.

More information about using Tableau is available in the Decision Management Platform Help system. To access the DMP Help system, select **View Solutions** on

the Decision Modeler Projects page. Another browser tab opens the DMP. Click the **Help** link in the banner and search for Visualizing Solution Data.

-  **Note** In the DMP documentation, the term component refers to any component on the DMP Platform. In Decision Modeler, a component is represented as a workspace.

CHAPTER 21

Decision Testing and Analysis

Use Decision Testing to upload, import or generate a dataset to test whether or not your decision logic is generating the desired results. If you upload your own dataset, you can include expected data that will be compared to the result values.

During testing the data is evaluated by invoking an entry point or a decision entity. Usually an entry point is selected when you want to test all or most of your decision logic. To just evaluate a subset of the decision logic, you select a specific decision entity. In all cases, the selected entry point or decision entity must have a return type and at least one parameter. Decision entities are defined as rulesets, functions, and decision flows to name a few examples. When a dataset is run, the data is evaluated by the decision logic in the entry point or selected decision entity and by all referenced decision entities.

There are several ways to provide a dataset:

- Upload a CSV file or upload one from FICO Drive that conforms to the specific requirements for Decision Testing.
- Import data from the Analytic Data Mart in the Decision Management Platform. The data is generated from batch processing or web service deployments. When the data is imported into Decision Testing, it is transformed into a CSV file.
- Specify the result properties that you want tested in a wizard and let Decision Modeler generate a CSV dataset for you.

After a dataset is evaluated, you can view reports that show the distribution of values. If you uploaded a dataset and expected data was included, the reports provide graphical comparisons of the differences between the expected and actual results. This reporting capability lets you easily see the impact of changes to the decision logic so you can assess any shifts in the values and the likely impact on the decision outcome.

If one or more tests do not generate the anticipated results, you can trace the execution to determine if the problem lies with your decision logic or your data. While tracing the execution, the result state at various points is shown in the execution flow so you know which conditions in a decision entity were met and the actions that modified any object properties. The ability to test and analyze the results helps you to debug issues with your decision logic before deploying your changes.

Related Links

- [Running the Tests](#)
- [Configuring Datasets in CSV Files](#)
- [Importing Data from the Analytic Datamart](#)
- [Generating Datasets](#)
- [Understanding the Results](#)
- [Viewing Impact Analysis Reports](#)
- [Analyzing the Results](#)

Entry Point or Decision Entity Requirements

To run Decision Testing, you must select an entry point or a decision entity in the Test page. An entry point is also a decision entity, but it is called an entry point because the signature of the decision entity defines one of the possible input and output types for the project.

Regardless of whether you select an entry point or a decision entity to invoke Decision Testing, the following requirements must be met.

- The decision entity must have a return type and at least one parameter. Note that a scorecard created by importing a PMML scorecard uses void as the return type so it cannot be used as the decision entity.
- The decision entity must include a return statement in the decision logic.
- If the decision entity or any of the referenced decision entities use business terms in the decision logic, the business term sets must be initialized.
- All classes in a project must be initialized. In the entry point or a referenced decision entity, using a class property in a condition initializes the class. If a class property from an uninitialized class is used in an action statement and no properties from the same class are used in the conditions, an error may be thrown during Decision Testing.



Note To use Decision Testing, the project must compile successfully.

Global Variable Usage

Decision testing is designed to run records as a batch. If your project is using global variables, it is important to ensure that the global variables are initialized or reset appropriately so that they do not have any unwanted impact on the results.

Configuring Datasets in CSV Files

Use the **Configure Data** window when you want to create your own CSV file that will be uploaded directly to the **Test** page or uploaded to FICO Drive where the dataset can be shared with other workspace users. Workspace users can then upload the dataset from FICO Drive when they are ready to run the data.

The **Configure Data** window generates a CSV file with the correct structure that you download and populate with input data and any expected data. In the Configure Data window, the parameters in the selected entry point or decision entity appear as inputs. You can select any primitive type parameters or properties from any class-type parameters needed for the input data. The expected and result values are selected from the return type parameter.

Before using the Configure Data window, it is recommend that you do the following:

- Determine the entry point or the decision entity that will be used to invoke the project.
 - Review the decision logic in any decision entities called by the selected entry point or decision entity. During testing, any decision logic in the entry point or decision entity and all referenced decision entities will be evaluated.
 - Make a list of the input and result properties or primitive type parameters that are needed to test the decision logic.
- 1 Expand the **Test** pane and select **Decision Testing**.
 - 2 Click the drop-down menu and select an entry point or a decision entity. (Only decision entities that have at least one parameter and a return type appear in the list.)
 - 3 Click **Configure Data** to open a window where you can select the properties on which to base the input values.

 **Note** Do not select read-only properties as input properties.
 - 4 Select the properties for which you want the results displayed. The maximum depth at which you can select a property is 20. (You can select a class and all properties in the class will be selected.)
 - 5 (Optional) Select the check boxes next to any result values where expected values will be supplied. The expected values will be compared with the result values during the decision testing run. Click **Done**.
 - 6 Click the **Download** icon to download the file.
 - 7 (Optional) In the **Data-ID** column, enter a label for each test. Each row is a separate test.
 - 8 Leave the **Result-State** and **Result-Detail** columns empty. Any data entered in these columns is ignored because these columns are reserved for feedback from an output run. You can remove these columns in dataset without any impact to a Decision Testing run; however, the columns are added back automatically after each run.
 - 9 Populate the input columns with the input data.

- 10 Populate any expected columns with the expected data. Any cells in the expected columns without values are ignored for validation purposes.

Dataset Requirements

If the **Configure Data** window is used to create the file structure for the dataset, all you need to do is to populate the file with the input data and, if applicable, the expected data. Review the input and expected data requirements to understand how to prepare the data.

If you create your own CSV file, review the file structure requirements to avoid any problems when uploading the file. This information is also useful if you make modifications to a CSV file and you encounter errors when uploading the file.

Related Links

[Input and Expected Data Requirements](#)

[File Structure Requirements](#)

Input and Expected Data Requirements

The input and expected data must meet these requirements.

- The input data values must be listed in the columns with the headers displaying the parameter property names, such as `accountBalance`.
- If expected columns are included in the dataset, the expected values must be listed in the columns with the headers displaying the word `expected` prefixed before the property name. The word `expected` should always be used regardless of the locale, such as `expected:accountBalance`.
- Each column must contain data that converts to the correct type for the property listed in the column header. For example, if `accountBalance` is a real property, the input or expected data must be of type `real`.
- Values do not require SRL punctuation. For example, date and time-related values do not require single quotes.
- For timestamp values, use a four-digit year in the values.
- If there is no value in an input data cell, the special value `unavailable` is automatically used.
- Do not include values for read-only properties as input data.
- If a test contains an empty cell in an expected data column, validation does not occur for the associated result value.
- The special values `unknown`, `unavailable`, or `null` can be entered as input and expected values. These values must be entered as `unknown`, `unavailable`, or `null` regardless of the locale. The special values `known` and `available` are not applicable.
- For enumeration values in a dataset, you can use the full name such as `enumerationName.enumerationItem` or just `enumerationItem`. For example, if `Color` is the name of the enumeration, you can enter `Color.green` or `green`.

- If your Java object model contains display labels for enumerations, do not use display labels as values. Use the full name of the enumeration item or the enumeration item name.
- When supplying input or expected data for business terms of type string that have an associated value list, use the **Value** names and not the **Display Value** names.
- In a decision table, if a condition cell uses the **Not Applicable** cell format, the condition will always resolve to **true**, which means that any value of the correct data type is ignored. To avoid errors in your dataset, do not use the words **Not Applicable** or **N/A** as input values to test these condition cells because they will be considered string values and may generate errors in Decision Testing. When testing **Not Applicable** conditions, leave the cells empty or use a value of the correct data type for that condition column or row. Alternatively, you can use the special values **unknown** or **unavailable**.
- If an expected value is entered for a real value, the precision of the result value will be made to match the precision of the expected value. If no expected value is entered for a real value, the full precision of the result value is displayed.

For information about the formats for the various data type values, see the FICO Structured Rule Language Syntax chapter.

 **Note** When using Microsoft Excel, ensure that the data in the cells use "text" formatting. Sometimes the format in the cells changes without warning so it is important to verify that your data is in the correct format before uploading the file to the Test page.

Collection-Type Properties

The data for collection-type properties must meet specific requirements.

A collection-type property is defined as a property that is one of the following:

- Fixed or dynamic array of a primitive type.
- Member of a fixed or dynamic array of a class type.

When a collection-type property is selected as an input, result, or expected property, the generated configuration file includes columns for two members. This is an example of how a configured dataset appears by default when collection-type properties are selected as input and result properties. In this case, two properties from `VehicleInfo[]`, which is a collection of class `VehicleInfo`, have been selected as input properties and one property was selected as a result property.

A	B	C	D	E	F	G
Data-ID	car#1.Make	car#1.VIN	car#2.Make	car#2.VIN	result:discount#1#	result:discount#2

Figure 46: Configuration File Example

Add columns for each additional member that is required as an input, result, or expected value or remove any unwanted columns.

While it is expected that each member in a collection will have a full set of data, there are some cases where the data might not be available. Gaps in data are permitted only if the gaps occur in cells that are in columns where the member number is greater than the member number of the last contiguous set of values. For example in the first row, there is a gap in the third member, but that is permitted because data are available for the first and second members in the collection.

A	B	C	D	E	F	G
Data-ID	car#1.Make	car#1.VIN	car#2.Make	car#2.VIN	car#3.Make	car#3.VIN
1	Honda	ABC123	Tesla	JKL123		
2	Ford	DEF456	Toyota	MNO456	Cadillac	STU482
3	Mercury	GHI789	Honda	PQR789	BMW	XYZ137

Figure 47: Gaps in Data that Do Not Result in Errors

This is an example of non-contiguous data that results in errors. In each case, the first two members are missing data. If the second member in row 2 or the first member in row 3 had at least one value in a column, an error would not be thrown.

A	B	C	D	E	F	G
Data-ID	car#1.Make	car#1.VIN	car#2.Make	car#2.VIN	car#3.Make	car#3.VIN
1	Honda	ABC123	Tesla	JKL123		
2	Ford	DEF456			Cadillac	STU482
3			Honda	PQR789	BMW	XYZ137

Figure 48: Gaps in Data that Result in Errors

 **Note** Properties of multi-dimensional arrays are not supported.

File Structure Requirements

If the **Configure Data** window is used to generate a dataset in a CSV file, the structure is created for you. However, if you modify the structure of the file or create your own file, you need to be aware of the file structure requirements or you may encounter problems when uploading the file.

- The first row in the file must contain only column headers.
- The Data-ID column is an optional column that you can use to label each test. Each row of data is a separate test.
- If the Data-ID field is used, the column header must be **Data-ID**. The header is case-sensitive.
- If the Data-ID field is used, it must be the first column.
- For each input property, the column header must be the case-sensitive name of a property passed in from your object model or a primitive type parameter.
- For each result property, the column header must be the case-sensitive name of a property from your object model or a primitive type parameter with the

word **result**: prepended to the name. This is an example:
result:accountBalance.

- For each expected property, the column header must be the case-sensitive name of a property from your object model or a primitive type parameter with the word **expected**: prepended to the name. This is an example:
expected:accountBalance.
- For each expected property column, there must be a matching result value column.
- The columns of data must be contiguous. If an empty column without a column header appears in between two columns of data, an error is thrown.
- Input columns must appear before result and expected data columns or an error is thrown.
- The order of the expected and result columns does not matter as long as they appear after the input columns.
- For duplicate property names at different hierarchies in the object model, use the shortest path that includes the name of the parent classes until the property names are unique. This is an example of two identically named properties in the `LoanApplication` class: `Employer.address.zipCode` and `Applicant.address.zipCode`. For best results, use the **Configure Data** window to generate a sample file to see how the duplicate property names must appear.
- If a functional takes a parameter and the input file does not have an input column for that parameter, the values passed to the functional for the missing parameter are unknown.
- The encoding for the CSV file must be Unicode (UTF-8).

Dataset Sizing Guidelines

Decision Testing is a validation tool to test decision logic. The tests in a dataset should be based on what is required to test all aspects of your decision logic. Decision testing should not be used to test the volume of test cases that can be run through your decision logic.

There is no limit on the number of test cases that can be evaluated in a Decision Testing run; however, the number that you can use may depend on your server and browser capability. While there is no specific limit to the number of test cases in an uploaded dataset, a conservative guideline is to use a maximum of 200 columns and 3000 rows. This is only a guideline and your environment may support larger datasets.

Importing Data from the Analytic Datamart

If you are using the cloud version of Decision Modeler, you can use data generated from executing a decision service in the Decision Management Platform (DMP) as test data in Decision Testing. Data generated while executing a decision service via a batch job, SOAP or REST web services are stored in the Analytic Datamart and can be imported into Decision Testing.

The entry point or the decision entity you select in the Test page determines the data that is imported from the Analytic Datamart. If a solution contains more than one component, data in the Analytic Datamart that is not relevant to the entry point or decision entity is ignored.

By default, the results data from the decision service execution is used as expected data in Decision Testing. This allows you to see how additional changes made to the decision logic in Decision Modeler affect the decision service outcome. If you do not want the result values used as expected data, you can disable this option before importing the data.

When data is imported from the Analytic Datamart, the data is transformed into a CSV dataset. Just like with uploaded CSV datasets, the datasets can be downloaded, modified, and uploaded again for additional test runs.



Note Do not use values for read-only properties in the input data.

Enabling Data Capturing in the Decision Management Platform

By default when a project is created, data capturing in the Analytic Datamart (ADM) is enabled and the environment is set to Design. If you have modified the settings or do not know the environment where your data is being stored, open the Data Capture Settings page in the Decision Management Platform (DMP). Data must be available in the ADM and you must know the environment in which it is stored before importing data into Decision Testing.



Note The Design environment in the Decision Management Platform is known as the Development environment in Decision Modeler.

- 1 On the Decision Modeler Projects page, select the project and click **View Solutions**.
- 2 Locate the solution, which should have the same name as the project.
- 3 Click on the solution name.
- 4 In the **Solution** panel on the left, click **Settings**.
- 5 Select the environment where you want the generated data stored.
Note the environment where data is being stored.
- 6 Ensure that the check boxes for **Input** and **Output** are selected.
The check box for **Internal** is not applicable.

7 If you made changes in this page, click **Save**.

Generating Datasets

You can use a wizard to generate a dataset to test the coverage of your decision logic or to test for unexpected business scenarios. The wizard allows you to select the type of dataset and the result properties that you want tested. The generated dataset is a CSV file that becomes automatically available in the table and is ready to run.

Please review this information about dataset generation before using the wizard.

- The input properties are the properties used in the conditions in the decision logic. These properties do not appear in the wizard, but you will see them listed in the table in the **Tests** tab when you select the generated dataset.
- The result properties that you have to choose from in the wizard depend upon the selected entry point or decision entity. The properties that are modified by the decision logic are automatically selected as result properties. You can clear the check box next to any of those properties and select other properties if any are available.
- In the case of arrays, only one instance of an array object is available in the wizard.
- Test data generation works well with standard data types. Using properties that are user-defined data types or generic data types may generate unexpected results.
- The number of test cases (rows) in the dataset is determined by how many properties are being tested. The number of input columns is multiplied by three to obtain the number of test cases. If the dataset has 10 input columns, the dataset will have 30 rows of test data. The calculation is fixed and cannot be changed.



Note It is expected that running a generated dataset will result in failures during a test run if the decision logic does not contain robust checks such as checking for null or unknown values. Resolving these types of failures will result in more robust decision logic.

There are two options for generating test data. The type of dataset that you choose depends upon your testing goal.

Naïve

Goal: Coverage analysis for unusual scenarios

The dataset will contain realistic values for each input property but the values will not be based on the decision paths. The dataset may also include a combination of unrealistic values that are unlikely to be passed into your decision logic during execution. This dataset type lets you see how well the decision logic handles unusual data.

In an application that evaluates applicants for car insurance, you might see these types of values:

```
age=104.  
product_type=car_insurance.  
yearsOfDrivingHistory=2.
```

Random

Goal: Robustness testing for unexpected or missing values

The dataset will contain values for the input properties based on their data type. The values may or may not be realistic. The dataset can include missing values or highly unlikely values.

In an application that evaluates applicants for car insurance, you might see these types of values:

```
age=-40.  
product_type=car_insurance.  
yearsOfDrivingHistory=732.
```

 **Note** While you can choose to test the decision logic for just one decision entity, the dataset will include properties based all decision logic in the project. This has no affect on the outcome of the decision testing run for the selected decision entity.

Running the Tests

Upload, import or generate a dataset for evaluation. After the decision testing run, the results are displayed in a table in the **Results** page. If a test does not generate the expected results, open the test in the **Analyze** page to step through the decision logic to see if the problem is with the decision logic or your test data. To view reports that show the distribution of the results values or comparisons between the expected and actual results click the **Impact** tab.

 **Note** To run the evaluation, the project must compile successfully.

Decision Testing is run in the Development environment. If the project has a Java object model, you can make changes to the working copy in the Development environment and run the dataset against another version of the project in the Staging or Production environment. Additionally, you can compare a version of the project in a committed solution to a version in the Staging or Production environment.

To run a dataset against another version of the project in the Staging or Production environment, these conditions must be met:

- The project must contain a Java object model. If both a Java and an XML object model are being used, the entry point or the selected decision entity must have an input parameter and a return type based on Java object properties.
- The entry point or entry points must be the same as the ones in the project in the Staging or Production environments.
- The project must have been submitted to the Staging or Production environment using the Decision Modeler Projects page.

 **Note** This functionality is not supported if the project is submitted to the Staging or Production environment using the Decision Management Platform (DMP) component menu commands.

- 1 Expand the **Test** pane and select **Decision Testing**.
- 2 Click the drop-down menu and select an entry point or a decision entity. (Only decision entities that have at least one parameter and a return type appear in the list.)
- 3 Select a dataset using one of these options:
 - Click **Upload data file** to select a CSV file, and click **Open**.
 - Click **Upload data file from FICO Drive** to select a file from your FICO Drive and click **Select**.
 - Click **Import from Analytic Datamart**.
 - 1 Select the environment where the data is stored. (If the data is stored in the Design environment in DMP, retain Development.)
 - 2 Set the date range for the dataset using the drop-down menus or the calendar icons.
 - 3 (Optional) Clear the **Use result data as expected data** check box if you do not want the result data from the Analytic Datamart used as expected data.
 - 4 Click **Preview** to view the incoming data.
 - 5 Click **Create** to transform the data into a dataset.
 - Click **Generate data**.
 - 1 Select the type of dataset.
 - 2 Clear the check box next to any result property that you do not want to generate the results for.
 - 3 Review the dataset.
 - 4 Click **Create** to generate the dataset. The dataset appears in the table and is ready for use.
- 4 In the **Compare with** menu, retain **Expected data** to run the data in the selected dataset. The default value (**Expected data**) is always applicable to the Development environment regardless of the object model type. It is also applicable even if the dataset does not contain expected data.

If the project has a Java object model, you can compare the working version of the project with a version in the Staging or Production environment. Alternatively, you can compare a version of the project in a committed solution with a version in the Staging or Production environment.
- 5 Click **Run Test**.
- 6 (Optional) Click the **Filter columns** icon to hide columns from the display.
- 7 (Optional) Click the **Download** icon to download a CSV file to your desktop. The file contains any input data, expected data, and the result values. There is an additional column that appears after the Data-ID column called **Result-State** that lists the status of each test. If the file is uploaded for another test

run, the results in the **Result-State** column are overwritten with the results of the new run.

Related Links

- [Sharing Files Using FICO Drive](#)
- [Entry Point or Decision Entity Requirements](#)
- [Configuring Datasets in CSV Files](#)
- [Importing Data from the Analytic Datamart](#)
- [Generating Datasets](#)
- [Understanding the Results](#)
- [Viewing Impact Analysis Reports](#)
- [Analyzing the Results](#)

Understanding the Results

Each row in a dataset is one test. After each test in the dataset is evaluated, the result values are populated in the results columns. There are three possible outcomes for each test: passed, validation failure, or runtime error.

After the evaluation, the number of any tests that passed (green), generated validation failures (yellow), or runtime errors (red) are shown in the bar above the table.

Runtime errors are usually generated for one or more of the following reasons:

- A cell contains a value that cannot be converted to the correct data type.
- Empty values or missing input columns may lead to evaluation errors in statements that expect data to be set.

If a test generates unavailable or unknown in a result column and you were not expecting that value, review your decision logic and the associated input values. These are a few of the reasons why a special value may occur in a result column:

- An unknown value is one that has not been initialized. This typically happens when a rule or statement setting the value is not executed. Properties that are not present in the input data set are not initialized and, as a result, are unknown by the entities.
- A value is unavailable when there is no data. Any empty input cells are considered unavailable, and may lead to unavailable results.
- The special value unavailable or unknown was entered as an input for one or more values in a test.

Any cells in a row that are highlighted in yellow means that a validation failure occurred because a result value in the test did not match the corresponding expected value. The expected and result values must not only match, but they must be of the same data type.

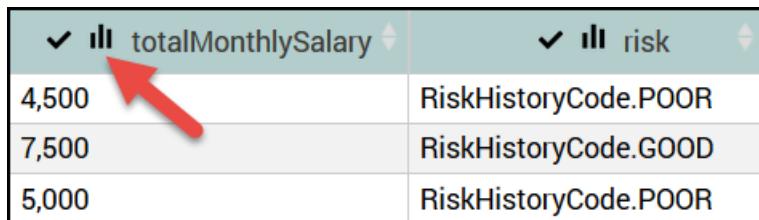
 **Note** The result for a real value may be rounded to match the desired precision. The precision is set by the expected value, if one has been entered.

If a test did not generate the expected results, click the **Analyze Decision** icon next to the test to open the Analyze page where you can step through the execution to see how the decision logic determined the outcome.

Viewing Impact Analysis Reports

After a Decision Testing run, you can view reports that graphically show the distribution of the result values. If expected data was included in the dataset, the reports show the differences between the results and the expected data. The reports make it easy to assess the impact of any changes made to the decision logic in the Development environment. Having this information enables you to promote only those changes that have a positive business impact.

The reports are accessed by clicking the **Impact** tab and selecting a result property or by clicking the report icon next to a result property in the **Results** tab, as shown in this figure:



✓ totalMonthlySalary	✓ risk
4,500	RiskHistoryCode.POOR
7,500	RiskHistoryCode.GOOD
5,000	RiskHistoryCode.POOR

Figure 49: Icon that Opens a Report for a Result Property

The reports are based on the selection of a result property. Reports are available only for result properties of type real, integer, boolean, string, enumeration, and money.

 **Note** Reports are not available for result properties of type date, duration, time, or timestamp.

When expected data is included in a dataset, you have the choice of viewing the expected and actual results separately or viewing the combined results. The type of graphical chart displayed depends upon the result property type. For example, if the result property is a boolean, string, or enumeration type, you can view the expected and actual results as separate pie or bar charts or the combined results as a bar chart. Underneath the chart, a swap set analysis is displayed for boolean, string, or enumeration result properties.

In this figure, the swap set analysis shows that the expected results match the actual results of the Decision Testing run. You can tell because the distribution between true and false values is the same on the x-axis in the **Actual Total** row and on the y-axis in the **Expected Total** column.

		Actual		Expected Total
Expected		false	true	Expected Total
	false	13	0	13
	true	0	7	7
Actual Total		13	7	20

Figure 50: Swap Set Analysis Before a Change to the Decision Logic

Suppose a change was made to the decision logic that resulted in a shift in the distribution of values, as shown in this figure:

		Actual		Expected Total
Expected		false	true	Expected Total
	false	11	2	13
	true	1	6	7
Actual Total		12	8	20

Figure 51: Swap Set Analysis After a Change to the Decision Logic

You can see the distribution change in the cells with the white background. In this case, a change to a model resulted in swapping out two true values and swapping in one additional false value.

For result properties of integer, real, or Advisor money types, you can view the expected and actual results separately in a bar chart or the combined results in a bar or box chart. Underneath the chart, a table shows the statistical max, mean, and min for the results. If you choose to view the combined results, the max, mean, and min are based on the distribution of the combined data.

When you are viewing a bar chart for a real or integer property, you can select either coarse or fine binning. Selecting the coarse binning option will result in a smaller number of bins with a larger bin size, while selecting the fine binning option will result in a larger number of bins with a smaller bin size. The actual size of the bins is determined implicitly depending on the data distribution.

Analyzing the Results

After the tests are run, you can select a particular test and step through the decision logic to see how the results were generated. If the results for the test were not what you expected, running an analysis on the test can help you to determine whether the problem lies with your decision logic or your data. During the analysis, you will see the result state at various points in the execution flow so you know which conditions in a decision entity were met and the actions that modified any object properties.

To analyze a test, you must first run the tests in the **Test** page. After the run, click the **Analyze Decision** icon next to the test that you want to analyze; otherwise, the first test in the list is selected if you click the **Analyze** tab.

Once a test is selected, the trace graph provides an overview of the execution flow so you can see which decision entities impacted the outcome for the test. You can step through the decision logic manually by clicking on each decision entity or you can click the **Replay execution** command on the toolbar to automatically run the execution. Regardless of which method you choose, the decision logic that was executed is displayed in the **Rule Details** pane and any properties that were modified are displayed in the **Result State** pane. Alternatively, if you click on a property in the **Result State** pane, the decision entity where the value was last set is highlighted in the trace graph. You can then manually step through to other decision entities in the trace graph or replay the execution.

For those decision entities where the decision logic iterates over values such as a pattern in a rule or a loop in a decision flow, you can manually step through each iteration. Underneath the decision entity in the trace graph, there are angle brackets around the iteration number. When one or more of the brackets are highlighted, you can step forward or backwards through the iterations. If a bracket is disabled, this means that the minimum or maximum number of iterations has been reached.

If a test fails because of the decision logic in a decision entity, that entity is highlighted in the trace graph. You can hover over the node to see the name of the decision entity or click the node to display the name in the Rule Details pane. If a test fails because of an error in a dataset, no decision entities are displayed in the trace graph. Instead a message appears in the Rule Details pane noting the input value for the property that caused the error.

After an analysis is run, you can click the **Download execution trace** icon on the toolbar to download a spreadsheet with a list of the executed decision entities. In addition to the type and name of each decision entity, the information in the **Rule Id** field helps you to locate the decision logic that was executed in the corresponding decision entity.

To analyze another test, click the **Results** tab and click the **Analyze Decision** icon next to the test you want to analyze.

CHAPTER 22

Unit Testing

If a project was configured by uploading an exported Blaze Advisor project and the project contains unit tests, you can run them using the brUnit testing framework, which is accessible from the Test pane.

Unit tests perform an action that compares an expected result with the actual result. Use the testing framework to run test cases to verify that your entities are generating the expected results.

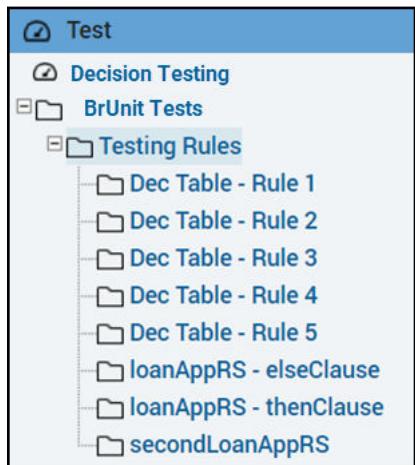


Figure 52: brUnit Test Cases

To open the testing framework, click on the project name or on a test case folder. In the preceding figure, the project name is **Testing Rules** and each of the folders represents a test case. A test case is a group of related tests. Grouping tests into test cases is usually used when the tests need to run in a specified order.

To run the tests, select the project name or a test case folder. The test page changes so that you can run the test. Click **Start new run** at the top of the interface. After the tests have been evaluated, the results appear in the Status column. Any error messages are displayed in the Test Result column.

If you decide to stop the evaluation while testing framework is processing the tests, click **Stop current run**. After you click the icon, the entire test run is stopped after the test that is currently being evaluated is finished. Note that you can start a new test run, but that you cannot resume the one that has been stopped.

- Note** You can run the testing framework if the project has a compilation error, but you see errors after the tests are evaluated. For best results, compile the project first and then run the testing framework.

Running the Testing Framework

In the Test pane, select the project name or a test case to run in the testing framework.

The project used to generate the RMA must contain test cases.

- 1 In the Navigator, expand the **Test** pane.
- 2 Perform either one of the following steps:
 - Select the project name to run all of the tests in the project.
 - Select a test case to run a particular group of tests.
- 3 Click **Start new run**.
- 4 View the results.

Viewing the Testing Results

After the testing framework evaluates the tests, it displays several visual cues that let you quickly scan the results. There are three possible outcomes that are displayed - Success, Failed, or Error

- Success—A test passes successfully when the test evaluates as expected.
- Failed—A test fails when the test does not evaluate as expected.
- Error—A test generates an error message, when an error is encountered while evaluating the test.

An error in the testing framework is different from a compilation error. The testing framework informs you about problems with the way tests are structured, but any errors with your decision entities are reported as compilation errors.

Click the link to the entity, make the necessary changes and then rerun the testing framework. If the entity that failed is not accessible in the application, contact your system administrator to have the entity updated.

CHAPTER 23

XML Object Model Reference

While almost all XML constructs are supported in an XML schema, there are a few that are not supported, and there are some limitations in this release.

This is the list of limitations:

- The <xsd:choice> element is ignored. The contained elements are mapped as object properties. There is no mechanism to verify that only one value is set.
- Schema (.xsd) files that include other schemas are not supported.
- Keys are not supported.
- Lists are mapped to properties of type array and they are supported in an XML business object model. However, properties of type array are not displayed in the **Insert New Column** dialog in the Decision Table editor, **Add Variable** dialog in the Scorecard editor, or in the **Create Decision Tree** wizard.
- Multi-occurring elements are mapped to properties of type array and they are supported in an XML BOM. However, properties of type array are not displayed in the following windows: **Insert New Column** dialog in the Decision Table editor, **Add Variable** dialog in the Scorecard editor, or in the **Create Decision Tree** wizard.

Data Types

The vast majority of the types available through XML schemas are supported. Types that are not currently supported are mapped to a string type.

This table describes the mapping between the XML Schema types and Decision Modeler types:

Table 44: XML Schema and Decision Modeler Types

XML Schema Type	Decision Modeler Type
boolean	boolean
integer	integer
positiveInteger	integer
nonPositiveInteger	integer
negativeInteger	integer
nonNegativeInteger	integer
long	integer

Table 44: XML Schema and Decision Modeler Types (continued)

XML Schema Type	Decision Modeler Type
short	integer
int	integer
decimal	real
float	real
double	real
string	string
QName	string
NMTOKEN	string
NMTOKENS	array of string
ID	string
IDREF	XmlObjectHandle
IDREFS	array of XmlObjectHandle
date	date
time	time
dateTime	timestamp
duration	duration
anyURI	string
multiple occurrence of <x>	array of <x>

Enumerations

When importing enumerations, any spaces or hyphens used in the enumeration values will be replaced with underscores. For example, an enumeration value such as `low - income housing` will be rendered as `low__income_housing`.

Related Links

[Handling FICO® Application Studio Issues Related to Enumeration Types](#)

Unsupported Blaze Advisor Types

While most Blaze Advisor types are supported in FICO® Decision Modeler, there are some limitations in this release.

These Blaze Advisor types are not supported:

- Dynamic arrays
- Blaze Advisor Associations

CHAPTER 24

Troubleshooting

These topics provide troubleshooting assistance.

Issues with Multiple Components in a Solution

This is an advanced topic and knowledge of the Decision Management Platform (DMP) is required.

A workspace in the Decision Modeler Projects page represents a Decision Modeler component in the DMP. When you create a workspace, one Decision Modeler component is created in a solution. If you add other Decision Modeler components to the solution using the DMP, some functionality will not be available when the projects are submitted to the Staging environment. When you submit projects to Staging, if there are two components in the solution, an information message appears in the dialog informing you that some functionality will not be available.

There are two areas where the projects in the Staging environment will not be recognized.

Decision Testing

If a project has a Java object model, you can make changes to a version of the project in the Development environment and run the dataset against another version of it in the Staging or Production environment. However, if the solution contains more than one component, you cannot run this type of comparison.

Processes in the Solution Editor

A Decision Modeler component can be connected to a task element within a process in the Solution editor. When a solution has more than one component, this functionality is not available.

Decision Table Profiling

Review these common profiling problems that can easily be addressed.

Rule profiling problems fall into several categories:

- Issues with attaching and mapping data files.
- Data files that contain missing or incorrect data.

 **Tip** In the dataset (CSV file), line 1 contains the headers (condition labels) so the first line containing the values is line 2. An error reported at line 44 may actually be an error on line 45.

Issue Attaching and Mapping Dataset Files

Errors may occur when you attach a dataset file to a decision table.

The following messages appear at the top of the **Run Sample Data** dialog when a problem occurs while attaching a dataset file:

- Unsupported file format. Please upload a .csv file.
Use a CSV file for rule profiling. Selecting any other type of file results in this message.
- Incorrect header bindings: Supplied number of headers in dataset is less than the number of condition rows and columns.
The rule profiler must be able to match the headers in the dataset file with the condition labels in the decision table. This message indicates that the dataset does not contain sufficient matching headers and that the mapping is incomplete.

Dataset Files Contain Missing or Incorrect Data

Errors occur when the dataset contains missing or incorrect data.

When you run the profiler, error messages may indicate problems with the content of the dataset.

In order for rule profiling to run successfully, the data must be organized in columns and each set of data values must be complete. The only exception is for string values, which can have an empty values.

These messages occur when the dataset contains an empty data value of the specified type:

- Invalid money value: Cannot parse “ as a money amount.
- Invalid boolean value: Cannot parse “ as a boolean amount.
- Invalid integer value: Cannot parse “ as a integer amount.

Each column of values must contain data of the correct type for the corresponding condition expressions.

These messages appear when the dataset contains a value of the wrong data type:

- The row '42' of data set contains '2' columns, '4' were expected.
In the dataset, each row of condition values must contain the exact same number of values. This error message was generated because the values in two columns were missing.
- Invalid boolean value: YES
- Invalid integer value: blue
- Invalid money value: 2000 Cannot parse '2000' as a money amount as it does not specify a currency

Changes to the Dataset Do Not Appear in Rule Profiling

When a dataset is modified, you need to reattach it to the decision table before running rule profiling. The modified file does not get upload automatically.

Decision Table Export and Import

Export and import problems or failures can occur for a variety of reasons. For example, mismatched formatting or values formatted with non-allowed expressions can cause errors.

Cell Coordinate Messages

A cell coordinate message indicates a problem with the import operation and identifies the location where it occurs in the CSV file. The location is an x, y coordinate. X is for the row and y is for the column.

The following are the cell coordinate messages:

- The cell at coordinates [22, 1], has value value "", which does not match any of the available formats for the cell.
This message appears because the cell is empty.
- The cell at coordinates [2, 4], has value value "90", which does not match any of the available formats for the cell.
This message may appear for one of the following reasons:
 - A disallowed cell format has been applied to a value in the CSV file.
 - The column order has changed

 **Note** Error messages that display the coordinates for a CSV file table cell count the header row as row 1. In this CSV file example, the coordinates for the Ford Thunderbird Rating value are (4, 4), even though in a decision table they might appear to be (3, 4).

	A	B	C	D
1	Make	Model	Year	Rating
2	Dodge	Charger	1,968	4
3	Dodge	Charger	< 1,968	0
4	Ford	Thunderbird	≥ 1,958	1

Figure 53: A CSV file example with headers in row 1

"Import Cannot Proceed" Messages

Import cannot proceed messages specify the reason why the import stopped.

Usually the import operation stops for one of the following reasons:

- Header column or row order has changed.
- One or more header columns or rows have been deleted.
- One or more header columns or rows have been added.
- One or more dimensions of a Double-axis decision table have changed: the number of columns or the number of rows.

Layout of Double-axis Decision Tables Misaligned When Printing

This problem occurs when using the **Print** icon on the editor toolbar. That icon is used to print the contents of the entire editor. Use the **Print table data** icon on the Decision Table toolbar to print the decision table.

org.xml.sax.SAXException:No TargetNamespace

Problem

The `org.xml.sax.SAXException:No TargetNamespace` message appears when you upload an XML schema or an exported Blaze Advisor project to Decision Modeler or when you deploy your decision service.

Solution

If you are using an XML business object model, you must set the `targetNamespace`. For example: `<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://www.blazesoft.com/OES" targetNamespace="http://www.blazesoft.com/OES" version="1.0">`

Verifying the Business Object Model in your Decision Service

If you want to verify the business object model in your project, use the Project Configuration wizard.

To open the window, click the **Project Configuration** icon in the top corner of the Decision Modeler application. You can use this wizard to view, remove, or replace your XML or Java business object model. If you remove or replace your object model, you must redeploy the project using the **Configure Wizard**.

Error Displays When Importing a Business Term Set as a Java BOM

If you have created a Business Term Set in Decision Management Suite, you can export it as a JAR file and then import the JAR file into a Decision Modeler project as a Java BOM using the Project Configuration dialog in Project Explorer.

The JAR file contains a class for each Business Term and a class for the Business Term Library. If you import the Business Term Library class from the JAR file, an error is thrown. An error is thrown because the Business Term Library class has a dependency on a resource that is not available and not needed in Decision Modeler because it does not contain any actual business term content.

You can safely ignore this error and successfully use the Business Term Set in your project.

Uploaded Blaze Advisor Project Fails Deployment

Uploaded Blaze Advisor projects containing Java BOMs may fail to deploy if the Java packages are unnamed.

According to the Java specification, unnamed packages are allowable only for learning and demo purposes. See <https://docs.oracle.com/javase/specs/jls/se11/html/index.html>.

When projects containing unnamed Java packages are deployed to Production, the deployment fails because these unnamed Java package cannot support subpackages.

Handling FICO® Application Studio Issues Related to Enumeration Types

If your FICO® Application Studio (FASt) application is passing XML documents containing elements that are XML enumeration types as input to invocations of a Decision Modeler web service, you may encounter a known problem with FASt's handling of these types.

Problem

The problem occurs if the value of an XML enumeration element or attribute in the input document is not one of the legal values for that element as defined in the schema. This can easily occur if these elements are actually only intended to hold return results from the execution of the web service and the input document is also used as the return type.

The recommended best practice to avoid this problem in general is to cleanly partition your input and output elements into different document types in the schema and use a different input type and output type for the web service so that the input document does not contain any XML enumeration elements that correspond to result values. However, this may not always be possible so alternative solutions for this issue are described below.

When this issue occurs, you do not get any error displayed in the FASt client application. The typical symptom of this problem is an apparent successful execution of the web service but no results are returned to the FASt application. If you do not examine the FASt Web Reference Log as described in the following section, you may just think that your component's web service is not working properly.

Suppose that your input contains the following Applicant XML Schema with the NextActionCode enumeration element.

```
<xs:element name="Applicant" >
  <xs:complexType>
    <xs:all>
      <xs:element name = "firstName" type = "xs:string"/>
      <xs:element name = "lastName" type = "xs:string"/>
      <xs:element name = "middleName" type = "xs:string"/>
      <xs:element name = "telephoneNumber" type = "xs:string"/>
      <xs:element name = "ownPrimaryResidence" type =
                    "xs:boolean"/>
      <xs:element name = "ownRentalProperty" type =
                    "xs:boolean"/>
      <xs:element name = "numOfRentals" type = "xs:int"/>
      <xs:element name = "married" type = "xs:boolean"/>
      <xs:element name = "monthlyHousingPayment" type =
                    "xs:float"/>
      <xs:element name = "monthlyCarPayment" type =
                    "xs:float"/>
      <xs:element name = "monthlyCreditCardPayment" type =
                    "xs:float"/>
      <xs:element name = "totalMonthlyDebtPayment" type =
                    "xs:float"/>
```

```

<xs:element name = "housingRatio" type = "xs:float"/>
<xs:element name = "debtRatio" type = "xs:float"/>
<xs:element name = "loanAmount" type = "xs:float"/>
<xs:element name = "loanTerm" type = "xs:int"/>
<xs:element name = "loanScore" type = "xs:int"/>
<xs:element name = "status" type = "xs:string"/>
<xs:element name = "loanAPR" type = "xs:float"/>
<xs:element name = "needs2ndMtge" type = "xs:boolean"/>
<xs:element name = "qualifiesForLoan" type =
                           "xs:boolean"/>
<xs:element name = "nextActionCode" type = "NextActionCode"/>
</xs:all>
</xs:complexType>
</xs:element>

```

Here the NextActionCode enumeration element is defined as:

```

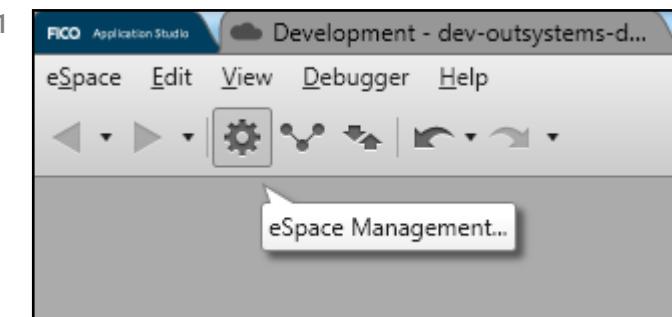
<xs:simpleType name="NextActionCode">
    <xs:restriction base = "xs:string">
        <xs:enumeration value=
            "Send application for a credit card"/>
        <xs:enumeration value=
            "Send rejection letter for the first mortgage"/>
        <xs:enumeration value=
            "Send application for a second mortgage"/>
        <xs:enumeration value=
            "Contact customer with Silver rate for second mortgage"/>
        <xs:enumeration value=
            "Contact customer with Gold rate for second mortgage"/>
        <xs:enumeration value=
            "Contact customer with Premier rate for second mortgage"/>
        <xs:enumeration value=
            "Invite customer to special seminar for investors"/>
        <xs:enumeration value=
            "Call for appointment to discuss other financial products"/>
    </xs:restriction>
</xs:simpleType>

```

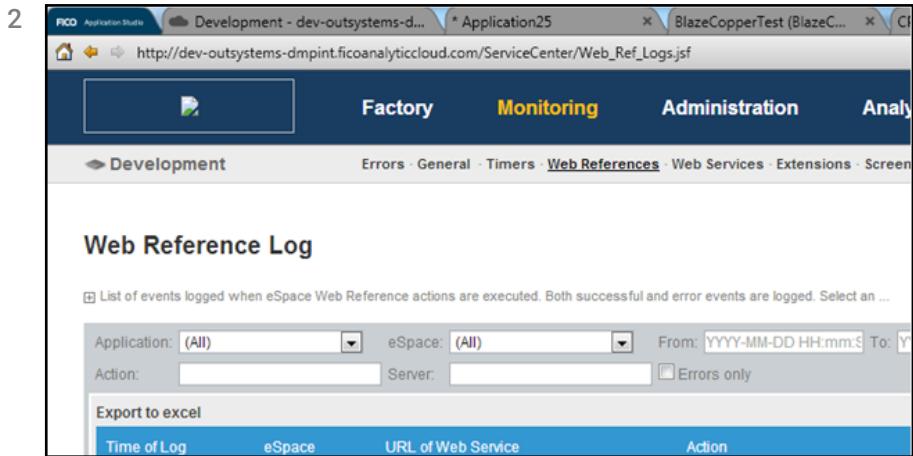
When sending a Request from the FASt Client to the Server, the following situations may occur:

- The value for NextActionCode is not filled in, for example, it is Empty.
- The value for NextActionCode is set incorrectly.
- The value for NextActionCode is set correctly.

In the case of 1 and 2, when NextActionCode is either empty or set to an invalid value, FASt will be unable to proceed and will fail, but no error message will be displayed in the FASt client application. In order to locate the error message (in case it is not visible on the browser screen) you will need to do the following:



Click on eSpace Management icon.



Click on **Monitoring > Web References**.

This screen will show you a list of all errors captured while processing calls to Web References. In the Case of Errors resulting from empty or incorrect values in NextActionCode, the errors will look similar to this: ...at osmortgageappclient.webreferences.rmadynamicwebservice.NextActionCode.fromValue(Unknown Source)...

This error is an indication that FAST was unable to resolve the value (missing or incorrect) presented in the input to those specified in the schema.

Solution #1 Do not let the user submit the Input Form without filling in each value

Variable	EditRecord1.Record.Applicant.nextAction
Validation Parent	EditRecord1
Mandatory	True
Null Value	

Figure 54: FAST Client with the form field Mandatory set to True

The best way to ensure that the user always fills in one of the values for an enumeration is by making the form field Mandatory=True in the FAST client at design time. To make the enumeration type field easy to use, you can add a Combo Box. Although the details of creating a Combo Box and its values are beyond the scope of this topic, you can get these details in FAST by dragging the Combo Box widget onto the Form and pressing the F1 key.

Solution #2 Encode a Null Value into the field

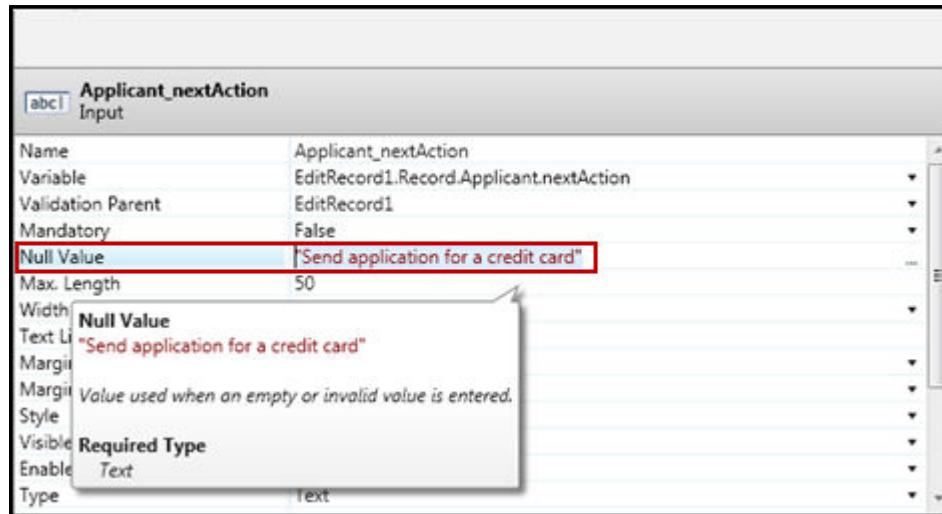


Figure 55: FASt with a Null Value encoded

If you think it is possible that the user may not know what value to enter, then forcing the user to fill in each field (Solution #1) may not be the best solution. Instead, you can specify a Null Value in FASt at design time so that the Null Value will be sent if the user leaves the field empty.

Note To use this solution, the Null Value should be one of the approved values in the enumeration.

Solution #3 Make the referenced enumeration element optional and set the Don't Send value in the FASt Client

You can use this solution when the result of the web services call involves returning the request data structure with some of the same elements filled in. If possible, the practice of having the same data structure for input and output to a web service should be avoided. The input request and the output response should ideally be separate elements that can be independently varied.

In cases when separate elements for the input and output are not feasible, the elements in the structure that only correspond to the output response should be marked as optional. In this example, the nextActionCode enumeration element is specified only on the output and does not have to be included in the input and this is indicated by adding the `minOccurs="0"` attribute to the Schema element declaration as shown below:

Before: Recall the Applicant XML Schema with the NextActionCode enumeration element.

```
<xss:element name="Applicant" >
  <xss:complexType>
    <xss:all>
      <xss:element name = "firstName" type = "xs:string"/>
      <xss:element name = "lastName" type = "xs:string"/>
      <xss:element name = "middleName" type = "xs:string"/>
      <xss:element name = "telephoneNumber" type = "xs:string"/>
      <xss:element name = "ownPrimaryResidence" type =
```

```

                "xs:boolean"/>
<xs:element name = "ownRentalProperty" type =
                "xs:boolean"/>
<xs:element name = "numOfRentals" type = "xs:int"/>
<xs:element name = "married" type = "xs:boolean"/>
<xs:element name = "monthlyHousingPayment" type =
                "xs:float"/>
<xs:element name = "monthlyCarPayment" type =
                "xs:float"/>
<xs:element name = "monthlyCreditCardPayment" type =
                "xs:float"/>
<xs:element name = "totalMonthlyDebtPayment" type =
                "xs:float"/>
<xs:element name = "housingRatio" type = "xs:float"/>
<xs:element name = "debtRatio" type = "xs:float"/>
<xs:element name = "loanAmount" type = "xs:float"/>
<xs:element name = "loanTerm" type = "xs:int"/>
<xs:element name = "loanScore" type = "xs:int"/>
<xs:element name = "status" type = "xs:string"/>
<xs:element name = "loanAPR" type = "xs:float"/>
<xs:element name = "needs2ndMtge" type = "xs:boolean"/>
<xs:element name = "qualifiesForLoan" type =
                "xs:boolean"/>
<xs:element name = "nextAction" type = "NextActionCode"/>
</xs:all>
</xs:complexType>
</xs:element>
```

After: Make the following change to the NextActionCode enumeration element.

```

<xs:element name="Applicant" >
    <xs:complexType>
        <xs:all>
            <xs:element name = "firstName" type = "xs:string"/>
            <xs:element name = "lastName" type = "xs:string"/>
            <xs:element name = "middleName" type = "xs:string"/>
            <xs:element name = "telephoneNumber" type = "xs:string"/>
            <xs:element name = "ownPrimaryResidence" type =
                "xs:boolean"/>
            <xs:element name = "ownRentalProperty" type =
                "xs:boolean"/>
            <xs:element name = "numOfRentals" type = "xs:int"/>
            <xs:element name = "married" type = "xs:boolean"/>
            <xs:element name = "monthlyHousingPayment" type =
                "xs:float"/>
            <xs:element name = "monthlyCarPayment" type =
                "xs:float"/>
            <xs:element name = "monthlyCreditCardPayment" type =
                "xs:float"/>
            <xs:element name = "totalMonthlyDebtPayment" type =
                "xs:float"/>
            <xs:element name = "housingRatio" type = "xs:float"/>
            <xs:element name = "debtRatio" type = "xs:float"/>
            <xs:element name = "loanAmount" type = "xs:float"/>
            <xs:element name = "loanTerm" type = "xs:int"/>
            <xs:element name = "loanScore" type = "xs:int"/>
            <xs:element name = "status" type = "xs:string"/>
            <xs:element name = "loanAPR" type = "xs:float"/>
            <xs:element name = "needs2ndMtge" type = "xs:boolean"/>
            <xs:element name = "qualifiesForLoan" type =
                "xs:boolean"/>
            <xs:element minOccurs="0" name = "nextAction" type =
```

```
"NextActionCode"/>
  </xs:all>
</xs:complexType>
</xs:element>
```

On the FASt Client:

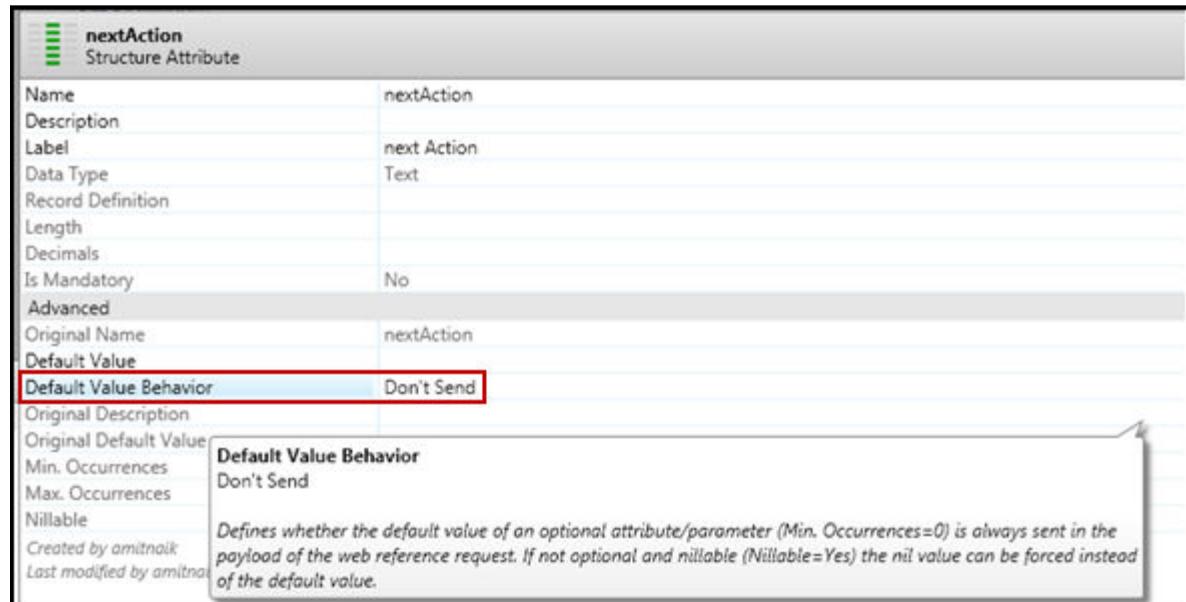


Figure 56: FASt Client with the Default Value Behavior set to Don't Send

- 1 Import this new XML schema and to the **Data Panel**.
- 2 Locate the `nextActionCode` element under the schema name in the list of Structures.
- 3 Set the **Value of the Default Value Behavior** tab to **Don't Send**. FASt will not send the `nextAction` tag in the input request at all.
- 4 Optionally, you can prevent the `nextActionCode` field from appearing in the input request by setting the **Visible** attribute to **false** because the user is not expected to provide this value as input.

Note If the element in the schema is not specified as optional (meaning it does not have the `minOccurs="0"` attribute), you will not be able to set the **Default Value Behavior** in the FASt to **Don't Send**. In that case, you will either have to modify the schema or use one of the other solutions described earlier.

Handling FICO® Application Studio Issues Related to Duration Types

If your FICO® Application Studio (FASt) application is passing XML documents containing elements that are XML duration types as input to invocations of a Decision Modeler web service, you may encounter a known problem with FASt's handling of these types.

Problem

The problem occurs if the value of an XML duration element in the input document is not one of the legal values for that element as defined in the schema. This can easily occur if these elements are actually only intended to hold return results from the execution of the web service and the input document is also used as the return type.

The recommended best practice to avoid this problem in general is to cleanly partition your input and output elements into different document types in the schema and use a different input type and output type for the web service so that the input document does not contain any XML duration elements that correspond to result values. However, this may not always be possible so alternative solutions for this issue are described below.

When this issue occurs, you do not get any error displayed in the FASt client application. The typical symptom of this problem is an apparent successful execution of the web service but no results are returned to the FASt application. If you do not examine the FASt Web Reference Log, you may just think that your component's web service is not working properly.

Solutions

For instructions on how to view the FASt Web Reference Log and resolve this problem, you can use the solutions outlined in the following topic [Handling FICO® Application Studio Issues Related to Enumeration Types](#) on page 262.



Note Any fields with the duration data type require input values that are formatted according to ISO 8601 guidelines. To see information about the ISO guidelines for duration, copy and paste the following URL into the address field of your browser.

Web Service Request 503 and 502 Errors

It is possible to execute single object requests many times successfully. However, if a component instance is not configured to scale properly based on the expected load, this could result in the component instance crashing.

To avoid a crash due to too many concurrent requests, there should be a one-to-one correspondence between the number of clients and scaled up gears. If you experience a 502 or 503 error, restart the component instance. For more

information, see the Batch Processing Data Through a Solution chapter in the DMP documentation.

WSDL No Longer Works for Decision Modeler

If you are unable to send a request for a decision service via the SOAP UI, you may need to update your SOAP UI startup parameters.

Problem

Due to an update to TLS v1.2, WSDL appears to no longer work for Decision Modeler services.

Solution

Update the SOAP UI startup parameters to enable the TLS 1.2 protocol.

- 1 Locate the `soapui.bat` file in `<SMARTBEAR_HOME>/bin` and open it in a text editor.
- 2 At approximately line 32, locate `JAVA_OPTS=-Xms128m`.
- 3 Add `-Dsoapui.https.protocols=TLSv1.2` to this line.

For example:

```
set JAVA_OPTS=-Xms128m -Xmx1024m  
-XX:MinHeapFreeRatio=20 -XX:MaxHeapFreeRatio=40  
-Dsoapui.properties=soapui.properties -Dsoapui.https.protocols=TLSv1.2 "-  
Dsoapui.home=%SOAPUI_HOME%"  
-splash:SoapUI-Splashscreen.png
```

- 4 In `<SMARTBEAR_HOME>/bin` locate the `SoapUI-x.x.x.vmoptions` file and add the following entry `-Dsoapui.https.protocols=TLSv1.2`. For example:

```
-XX:MinHeapFreeRatio=20  
-XX:MaxHeapFreeRatio=40  
-Xms128m  
-Xmx1000m  
-Dsoapui.https.protocols=TLSv1.2  
-Dsoapui.properties=soapui.properties  
...
```

Decision Executor Appears in SOAP Links

The words `Decision Executor` appear in the soap links. This is only cosmetic and does not affect the use of the links.

Decision Testing: "No Records Were Found"

If this message appears in the **Import Data from Analytic Datamart** wizard and you know that payload data has been successfully generated to the ADM within the last few minutes, it could be that the resources are busy. The ADM logs and stores event data as an offline process. Depending upon how busy the resources are, it may take several minutes to store the data after decision service execution. After the data is stored, you can access it from Decision Testing.

CHAPTER 25

Reference Guide

You can add built-in functions, support classes methods and Structured Rule Language (SRL) syntax when writing rules and functions, creating conditions for splits in a decision flow or defining a business term.

FICO Structured Rule Language Syntax

Structured Rule Language (SRL) is an easy-to-understand, English-like language. Alternative forms of the SRL syntax can be used to write rules. For example, you can refer to a property of an object by using any of the following:

- objectName.propertyName
- the propertyName of objectName
- objectName's propertyName

Similarly, you can express a comparison operation by using either of the following:

- value1 < value2
- value1 is less than value2

You can choose which forms to use depending on your own preferences, and who will need to review or revise the rules. If you or the reviewers come from a programming background, you may be more comfortable using objectName.propertyName form (sometimes called dot notation) for properties, logical operators for expressions, and so on. However, if you or your reviewers have little programming experience, you may want to use the many English-like alternatives that are available to make your rules easier to read and understand.

 **Note** It is recommended that you adopt one form or another, and use it consistently, to avoid confusion. However, the compiler will accept any form.

Notation Conventions

The grammar for SRL appears in this document as a series of syntax forms. Each syntax form is a combination of SRL keywords, user-supplied names, and punctuation that has a specific meaning in SRL. For example, this syntax form supplies an object definition:

```
objectName is a[n] className [with [Property_Declaration_List]  
[Initialize_Statement]].
```

These types of symbols appear in the notation:

Symbols	Convention
Keywords	Appear in bolded code (a, is a, with in the examples)
Punctuation	Appears in bold (the period [.] in the example)
User-supplied names	Appear in italics (<i>className</i> and <i>parent_ClassName</i> in the example)
Syntax forms	Appear as hypertext links that you can click to get a description (Property_Declaration_List and Initialize_Statement in the example)

Keywords and punctuation are the smallest, indivisible pieces of the grammar. They represent sequences of characters that must be present in order for the compiler to correctly recognize the syntax form in which they appear.

 **Note** SRL is case-sensitive. For example, **is** is a reserved keyword in SRL, but **Is** is not. For more information, see [Keywords](#) on page 312.

The notation uses certain metasymbols. These have special meanings and are intended to simplify the description of the grammar:

Metasymbols	Descriptions
[]	Indicate that the enclosed symbols (or sequence of symbols if parentheses enclose the sequence) can occur no more than once and are optional.
...	Indicates that you can repeat the preceding sequence of symbols enclosed within parentheses an arbitrary number of times.
	Indicates a choice between individual symbols or between syntax forms. Either the symbol to the left of the metasymbol or the symbol to the right of the metasymbol are valid, but you cannot use both. Note: When more than one symbol is valid within the syntax form, the symbols appear in separately listed syntax forms.

The metasymbols [], . . . , and | that appear in a syntax form are not bolded to better distinguish them from the required SRL punctuation. Other symbols, as described in [Punctuation](#) on page 322, are part of the SRL syntax.

Related Links

[Initialize_Statement](#)

[Property_Declaration_List](#)

Language Contents

This section contains the syntax for the FICO Structured Rule Language (SRL).

Object_Model

You can declare objects, properties, enumerations, and variables in SRL. Object properties and variables can also be initialized in SRL.

Object_Declaration

Syntax

```
objectName is a[n] className [with Property_Declaration_List]
[Initialize_Statement].
```

or

```
objectName is an association from className | interfaceName to className |
interfaceName
[with Property_Declaration_List]
[initially { [it[key] = value][ ,it[key] = value] ... }].
```

or

```
objectName is a[n] array of className | interfaceName [with
Property_Declaration_List]
[initially { it.append(value)| it.insert(value),
it.append(value) | it.insert value, ... }].
```

or

```
objectName is a[n] fixedarray of className | interfaceName
[ with [Property_Declaration_List]
initially { it[key] = value, it[key] = value, ... }].
```

Description

The declaration of an object. The absence of the leading article a or an distinguishes it from a class definition. SRL syntax accepts objects with a property list. In most cases, however, your object will derive from classes you import. Therefore, it will omit the property list:

```
myCar is a vehicle initially { brand = "Ford" }.
airports is an association from string to string
initially { it["New York"] = "JFK",
            it["San Francisco"] = "SFO",
            it["Los Angeles"] = "LAX".
}.
itinerary is an array of city
initially { it.append(city2),
            it.append(city3),
            it.append(city4).
}.
```

Example of creating an object using SRL

- 1 Enter an object definition with this syntax:

```
objectName is a ParentClassName.
```

Each statement must end with a period (if it is the last statement), or a comma. For example:

```
InvestmentPromotion is a PromotionalOffering.
```

```
// PromotionalOffering is a class with three properties:  
// promotedProduct, promotionalPrice, and  
// promotionExpirationDate.
```

- 2 (Optional) Add one or more properties specific to your object after the parent class name, using this syntax:

```
...with  
{  
    a newPropertyName1 : a type,  
    a newPropertyName2 : some objectType  
}
```

Many object-oriented methodologies do not let you add additional properties to objects; however, SRL permits this. For example:

```
InvestmentPromotion is a PromotionalOffering with {  
    a currentStatus : a boolean  
}
```

- 3 Provide initial values for the properties using the keyword **initially**.

You do not have to provide initial values for all properties. You can provide values for only one or some of them.

- a For objects with only inherited properties, use this syntax:

```
initially  
{  
    inheritedPropertyName1 = initialValue.  
    inheritedPropertyName2 = initialValue.  
}
```

For example:

```
InvestmentPromotion is a PromotionalOffering initially  
{  
    promotedProduct = InvestmentServices,  
    promotionalPrice = $225.00.  
    promotionExpirationDate = '12/31/2020'.  
}
```

- b For objects with new properties specific to them, use this syntax:

```
...with  
{  
    a newPropertyName1 : a type.  
    a newPropertyName2 : some objectType  
}  
initially  
{  
    inheritedPropertyName1 = initialValue.  
    inheritedPropertyName2 = initialValue.  
    newPropertyName1 = initialValue.  
    newPropertyName2 = initialValue.  
}
```

For example:

```
InvestmentPromotion is a PromotionalOffering with  
{  
    a currentStatus : a boolean  
}
```

```

initially
{
promotedProduct = InvestmentServices.
promotionalPrice = $225.00.
promotionExpirationDate = '12/31/2020'.
currentStatus = true.
}

```

Related Links

[Initialize_Statement](#)

[Property_Declaration_List](#)

Property_Declaration_List

Syntax

```
{ Property_Declaration [, Property_Declaration] ... }
```

Description

The definition of a single property or set of properties within the definition of an object. For example, a single property:

```
{ a firstName : a string }
```

or, as a list:

```
{ a lastName : a string,
an age : an integer,
a temperature : a real }
```

Related Links

[Property_Declaration](#)

Property_Declaration

Syntax

```
a[n] propertyName : a[n]
a[n] propertyName : a[n] enumerationName
```

or

```
a[n] enumerationName
a[n] propertyName : some className | interfaceName
```

or

```
some className | interfaceName
a[n] propertyName : some association from className | interfaceName
  to className | interfaceName
a[n] propertyName : some [fixed] array of className | interfaceName
```

Description

The declaration of an object property. The property can be of these types:

- Primitive data type

```
a lastName : a string  
an age : an integer  
a temperature : a real
```

- Enumeration

```
an applicationStatus : a status  
a status
```

Where *status* is an enumeration:

```
a status is one of { accepted, rejected }.
```

- Class-instance reference

```
an employee : some person  
some person
```

Where *person* is a class as follows:

```
a person is an object with {  
    a firstName : a string  
    a lastName : a string }.
```

- Collection

```
an accountSet : some association from string to account  
an address : some array of string
```

You do not have to use any specific order for property declarations.

Initialize_Statement

Syntax

```
initially { Assignment_Statement [, Assignment_Statement]... }
```

or

```
initially Create_Statement | Assignment_Statement
```

Description

A statement that introduces an initializer or an initial value for:

- A property for an Object_Declaration through:

- An assignment statement

```
initially { temperature = 98.6, firstName = "Bill" }
```

- A method call

```
initially { it.setTemp(98.6), firstName = "Bill" }
```

In an assignment statement or a method call, the compiler interprets the properties or methods as applying to the object you are initializing. When calling a method for objects and classes, use the keyword `it`. Otherwise, the engine will try to find the method in the rule-agent class.

You cannot use the `it` keyword with a method call to initialize a variable.

- A variable definition

A function or method must return a value of the corresponding type.

If the initializer consists of more than one statement, enclose the statements within curly braces (`{}`).

Related Links

[Assignment_Statement](#)

[Object_Declaration](#)

[Create_Statement](#)

Enumeration_Declaration

Syntax

```
a[n] enumerationName is one of { itemName , itemName [, itemName ] ... }.
```

Description

The declaration of an enumerated type that specifies a set of possible valid values. For example:

```
a color is one of { red, green, blue }.
```

You can use the enumeration values in a rule condition:

```
if car.color is (red or green or blue)
```

Constants defined by enumerations must be unique. Constants defined by enumerated types cannot have any values in common.

Variable_Declaration

Syntax

```
variableName is a[n]
    [initially true | false | unavailable | null | unknown | value].  
  
variableName is a[n] enumerationName
    [initially enumerationItemName].  
or  
variableName is some className | interfaceName
    [initially objectExpression].  
  
variableName is some association from className | interfaceName
| to className | interfaceName |
    [initially objectExpression].  
  
variableName is some [fixed] array of className | interfaceName
| [initially objectExpression].  
  
variableName is a[n] [initially true | false | unavailable | null | unknown | value].  
  
variableName is a[n] enumerationName
    [initially enumerationItemName].  
or  
variableName is some className | interfaceName
    [initially objectExpression].  
  
variableName is some association from className | interfaceName to className | interfaceName initially objectExpression].  
  
variableName is some [fixed] array of className | interfaceName [initially objectExpression].
```

Description

The declaration of a variable can be

- A primitive data type:

```
i is an integer.
```

- An enumeration:

```
sample is a color.
```

where *color* is an enumeration:

```
a color is one of { red, green, blue }.
```

- An object variable that belongs to a class:

```
the_boss is some Person.
```

where *Person* is a class:

```
a Person is an object with {
```

```
a firstName : a string
a lastName : a string }.
```

- An object variable that refers to a collection of objects:

```
ListofAccidents is some array of accident.
```

You can assign an initial value to a variable through the keyword **initially**. With variables of primitive data types or of enumeration types, the **initially** clause includes a literal value or an expression that produces one:

```
i is an integer initially 0.
sample is a color initially blue.
```

In the case of the object variable (**is some**), you need to specify an object name or an expression that produces an object:

```
the_boss is some Person initially Ron.
the_boss is some Person initially Joanna.manager.
the_boss is some Person initially a Person initially
    { firstName = "Bill" }
```

where:

```
Ron is a Person.
Joanna is a Person with { a manager : some Person }.
```

If a variable of type...	User the keywords...	Examples
boolean, integer, real, or string	a or an variableName is a dataType.	currentPayLevel is an integer.
Class or interface	some variableName is some className_or_interfaceName.	boss is some Employee.
Collection	some array of variableName is some array of className.	technicians is some array of SkilledWorkers.
Association	some association from Xxx to Yyy variableName is some association from key className_or_key_interfaceName to associated className_or_associated_interfaceName.	airports is some association from string to Airport.

Expressions

SRL provides several types of expressions that can be used to evaluate your object model.

Primary_Expression

Syntax

```
Boolean_Expression | Comparison_Expression | Numeric_Expression |
Quantified_Expression |
Temporal_Expression | Literal_Value
```

Description

An expression that evaluates to either of these:

- A value of a primitive type
- An object

Related Links

[Boolean_Expression](#)
[Comparison_Expression](#)
[Numeric_Expression](#)
[Temporal_Expression](#)
[Literal_Value](#)

Boolean_Expression

Syntax

```
true | false
Comparison_Expression
not Boolean_Expression
Boolean_Expression and Boolean_Expression
Boolean_Expression or Boolean_Expression
```

Description

An expression that evaluates to true, false, or a Blaze Advisor special value (unavailable, unknown, or null). It can contain:

- A Comparison_Expression
- A Quantified_Expression
- An object property of boolean type
- A boolean expression that a function, method, or ruleset call returns

You can use the keywords and and or to combine boolean expressions. You can use the keyword not to **negate** a boolean expression.

Boolean expressions can appear in statements that require a condition or constraint, including the condition of:

- An [If_Then_Else](#) construct
- A [While_Do](#) construct
- An [Until_Do](#) construct

Examples

```
if x.value is 100
if car.red and (car.sporty is true)
if (blood.pressure > 120) or (temp.value > 98.6)
if car.red and car.sporty and (car.price > 40000)
while (bankAccount.balance > 0) and (credit.balance < 500)
until i < 3000
```

Related Links

- [Comparison_Expression](#)
[Quantified_Expression](#)

Comparison_Expression

Syntax

`Primary_Expression, Comparison_Operator, Literal_Value`

where the two primary expressions are of compatible data types.

Description

An expression that uses a `Comparison_Operator` to compare two values or expressions. It evaluates to a boolean value.

Examples

```
account.balance < 1000
car.price >= 40000
product.quantity > 100
credit.balance <> 500
order.accepted is true
payment is equal to amountDue
payment is different from amountDue
payment is not equal to amountDue
payment is less than amountDue
payment is smaller than amountDue
today's date is earlier than theApplication's date
today's date precedes theApplication's date
today's date comes before theApplication's date
payment is more than amountDue
payment is larger than amountDue
today's date is later than theApplication's date
today's date follows theApplication's date
today's date comes after theApplication's date
payment exceeds amountDue
```

```
payment is less than or equal to amountDue
payment is smaller than or equal to amountDue
today's date is earlier than or equal to theApplication's date
retailPrice is up to suggestedPrice
retailPrice is at most $500.00
payment is more than or equal to amountDue
payment is larger than or equal to amountDue
today's date is later than or equal to theApplication's date
retailPrice is at least $500.00
```

Related Links

[Primary_Expression](#)

[Comparison_Operator](#)

[Literal_Value](#)

Numeric_Expression

Syntax

```
Property_Value | value
```

where both Property_Value and value evaluate to numeric values.

Description

An expression that evaluates to a numeric value. It can use a Numeric Operator to perform a numeric operation on compatible numeric values or expressions.

Examples

```
product.quantity - 600
account.balance + account.deposit
```

Related Links

[Property_Value](#)

[Numeric_Operator](#)

Quantified_Expression

Syntax

```
at least n className [such that Boolean_Expression] satisfy Boolean_Expression
at most n className [such that Boolean_Expression] satisfy Boolean_Expression
every className [such that Boolean_Expression] satisfies Boolean_Expression
exactly n className [such that Boolean_Expression] satisfies Boolean_Expression
zero className [such that Boolean_Expression] satisfies Boolean_Expression
or
at least n elementType in collectionName [such that Boolean_Expression]
    satisfy Boolean_Expression
at most n elementType in collectionName [such that Boolean_Expression]
    satisfy Boolean_Expression
every elementType in collectionName [such that Boolean_Expression]
    satisfies Boolean_Expression
```

```
exactly n elementType in collectionName [such that Boolean_Expression]
    satisfies Boolean_Expression
zero className in collectionName [such that Boolean_Expression] satisfies
Boolean_Expression
```

Description

A quantified expression performs a test that returns a boolean result:

- You specify the number of objects needed to fulfill the test, *n*, as a positive integer value. (You **omit** this with the keyword **every**.) The strings “one” “two” or “three” may also be used as the *n* value.
- The optional **such that** clause specifies a constraint that restricts the set of instances of either:
 - The class or interface
 - The elements of a collection

This expression is evaluated first in order to limit the **satisfies** test to a subset of objects. If an expression that combines several tests (with the **and** or **or** operators) follows the **such that** clause, you must **delimit** the expression following **such that** with parentheses.
- The **satisfy/satisfies** clause is a required expression that defines the test to apply to every object (or to a subset of objects when you specify a constraint) of either:
 - The class or interface

Note that you cannot cast a class within the **satisfy** predicate clause if the ruleset is in default execution mode. If you do this it will cause a runtime error.
 - The elements of a collection

You use the keyword **it[n]** within the **such that** and **satisfy/satisfies** clauses to refer to a value at index *n* of a collection.

You can use a Quantified_Expression in:

- Rule conditions
- Functions

Example

This rule uses a quantified expression to test if no product in the inventory satisfies an order:

```
if zero product satisfies
  ( name is currentOrder.product.name and
    color is currentOrder.product.color and
    quantity < currentOrder.product.quantity )
then { currentOrder.accepted = false,
      print("The order for " currentOrder.product.name
            " cannot be accepted at this time.")
}
```

Related Links

[Boolean_Expression](#)

[Functions](#)

Range_Comparison_Expression

Syntax

```
Primary_Expression is between Primary_Expression | Literal_Value and  
Primary_Expression | Literal_Value  
or  
Primary_Expression is not between Primary_Expression | Literal_Value and  
Primary_Expression | Literal_Value
```

where the primary expressions are of compatible data types.

Description

An expression that compares an expression to see if it is within a range defined by a value to two values or expressions. Each range endpoint must appear in an increasingly sequential order. In other words, the number with the lowest value appears first and is the starting point for the range.

The expression evaluates to a boolean value.

Example

```
theAccount's balance is between $00.00 and $500.00  
(You cannot use: theAccount's balance is between $500.00 and $0.00.)
```

Example 2

```
if newApp's mosInCurrentJob is between 6 and 13  
then set newApp's score to 1000.
```

In this rule, if the `mosInCurrentJob` property is any of these values (6, 7, 8, 9, 10, 11, 12, 13), then the rule fires.

Related Links

[Primary_Expression](#)

[Literal_Value](#)

SRL-Regular_Expressions

Syntax

```
stringExpression contains match regularExpression  
stringExpression contains match (ignoring case)regularExpressionString  
stringExpression does not contain match regularExpression  
stringExpression does not contain match (ignoring case)regularExpressionString
```

```

stringExpression exactly matches regularExpression
stringExpression exactly matches (ignoring case) regularExpressionString
stringExpression does not exactly match regularExpression
stringExpression does not exactly match (ignoring case) regularExpressionString

```

Description

SRL Regular Expression keywords are used to analyze the relationship between a stringExpression and one of the two regular expression forms: regularExpression and regularExpressionString. The keyword checks whether a match between the string and the regular expression can be calculated and then determines whether this resulting match is equivalent to the entire string.

The calculation takes the form of a boolean result.

There two types of keyword reporting can be performed:

- **contains match** If a match is found and that match is located within the string (inclusive), then the expression will evaluate to true. A false result is returned otherwise.
- **exactly matches** If a match is found and that match is equal to the string itself, then the expression will evaluate to true. A false result is returned otherwise.

The two regular expression keyword forms are:

- **regularExpression**
This can be either an instance of the support class RegularExpression or a regularExpressionString.
- **regularExpressionString**
The regularExpressionString is considered a more restricted form because it can only be expressed as a string data type. To specify case sensitivity with this form, you use the ignoring case string qualifier.

 **Note** The ignoring case qualifier can only be used with regularExpressionString.

The SRL Regular Expression support classes provide a property to specify case sensitivity.

You can use SRL Regular Expressions keywords in rules and functions.

Examples

These examples show the difference between contains match and exactly matches:

This SRL Regular Expression comparison evaluates to true because a letter in the character class a-z, defined in the regular expression is contained in “qwerty”

```
“qwerty” contains match “[a-z]”
```

This SRL Regular Expression comparison evaluates to false because a letter in the character class a-z, defined in the regular expression does not equal “qwerty”

```
“qwerty” exactly matches “[a-z]”
```

This SRL Regular Expression comparison evaluates to true because a letter in the character class a-z, defined in the regular expression is contained in “QWERTY” because the keyword uses the ignoring case string qualifier.

```
“QWERTY” contains match(ignoring case) “[a-z]”
```

This SRL Regular Expression comparison evaluates to false because a letter in the character class a-z, defined in the regular expression does not equal “QWERTY” even though the keyword uses the ignoring case string qualifier.

```
“QWERTY” exactly matches(ignoring case) “[a-z]”
```

Explanation of the regular expression syntax used in these examples:

The [a-z]character class matches any lowercase, alphabetic character from a to z inclusive.

Temporal_Expression

Syntax

```
Property_Value | value  
dateValue + durationValue  
dateValue - durationValue  
timeValue + durationValue  
timeValue - durationValue  
timestampValue + durationValue  
timestampValue - durationValue
```

or

```
durationValue after dateValue  
durationValue before dateValue  
durationValue after timeValue  
durationValue before timeValue  
durationValue after timestampValue  
durationValue before timestampValue
```

where:

- Property_Value and *value* evaluate to date, timestamp, time, or duration values
- *dateValue* evaluates to date
- *timeValue* evaluates to time
- *timestampValue* evaluates to timestamp
- *durationValue* evaluates to duration

Description

An expression that evaluates to a date, timestamp, time, or duration value.

Example

```
30 days after theAccount’s openingDate
```

```
30 days before theCustomer's birthDate
1 hour after theApplication's requestTime
1 hour before theApplication's requestTime
30 days after theSaleRequest's requestDate
30 days before theSaleRequest's requestDate
```

Related Links

[Property_Value](#)

Literal_Value

Syntax

```
true | false | unavailable | unknown | available | known | null | value
```

Description

All of the literal values that can be part of an expression. To be more precise, available and known do not correspond to a single value. Instead, they let you test a state that covers many values.

The special value keywords (unavailable, unknown, available, null and known) are valid on the right side of a comparison operator (is or <>). However, they are invalid in **most** expressions (such as `x + unknown`).

Values include string, integer, and real constants.

Property_Value

Syntax

```
variableName | objectExpression.propertyName
or
the propertyName of objectExpression | variableName
objectExpression's | variableName's propertyName
or
the propertyName [of the objectPropertyName] of objectExpression |
variableName
```

[Indexed_Property_Value](#)

Description

A value of a property, where `objectExpression` is any expression that evaluates to an object. There are two forms:

- The short form (with the property access operator, ".") represents the name of the object (or variable) and its property.
- The long form (English notation) represents the property of a particular object (or variable).

These forms have identical meanings; you can use them interchangeably. For example, to access a customer's account, you can use either:

```
customer.account  
the account of customer  
customer's account
```

You can nest property access using either form. For example, to access the location name of a particular destination (an object property defined as a type **Location**) of a shipment, you can use:

```
aShipment.destination.locationName  
the locationName of the destination of aShipment  
aShipment's destination's locationName
```

If you work with external objects from object models such as Java, you may need to use the syntax forms for default properties and indexed properties.

Related Links

[Indexed_Property_Value](#)

[Default_Property_Value](#)

Default_Property_Value

Syntax

```
objectExpression.$value  
or  
objectExpression.$object
```

Description

Additional syntax forms for the value of an object or the object itself when the object is defined outside of Decision Modeler and has a default value. The object's default value is accessed when you refer to *objectExpression*. These additional syntax forms are provided for situations where there is ambiguity between accessing the default value and the object itself.

You use these forms **only** when the value property is an object or a collection.

Usually, you use a default value property in objects you map from Java. The Business Object Model Wizard displays a Value Property node for classes and objects that specify a default property.

You need **\$value** and **\$object** only if there is a potential conflict when the value property is an object in:

- A test against a special value
- An assignment

Example

If you have an object product with a default value property price, you usually access the default value price by specifying product:

```
product = newPrice
```

This is equivalent to:

```
product.price = newPrice
```

However, if there is ambiguity between accessing the default value and the object itself, you can use the additional syntax forms to distinguish between the two:

- To refer to the object:

```
product.$object
```

- To refer to the default value:

```
product.$value
```

Indexed_Property_Value

Syntax

```
objectExpression.propertyName[index]
```

Description

A value of an indexed property. You use this syntax for:

- Indexed-value properties
- Collection-value properties

The index can be:

- An integer for indexed-value properties
- An integer, string, or an object for collection-value properties

Example

If you have a collection ListofAccidents:

```
ListofAccidents is an array of accident.
```

you can use the *Indexed_Property_Value* syntax to access individual property values within the collection:

```
ListofAccidents[6] = newAccident.  
print("Amount of claim = " ListofAccidents[6].claimAmount).
```

Statements

SRL provides several types of statements that represent actions.

Statement_Block

Syntax

```
Statement on page 290
```

or

```
{ Statement [, Statement ]... }
```

Description

A set of one or more statements, where each statement represents an action. You can omit the curly braces in a single statement. For example:

```
then BankAccount.withdraw = 100
```

When there are two or more statements, you must use the curly braces:

```
then { BankAccount.withdraw = 100, BankAccount.update() }
```

An individual Statement_Block defines one name space. An individual block can be a:

- For_Each construct
- While_Do construct

Related Links

[Statement](#)

[For_Each](#)

[While_Do](#)

Statement

Syntax

```
Assignment_Statement | Compound_Assignment_Statement |  
Create_Statement | Delete_Statement | Execute_Function |  
Method_Call | Case_Selection | If_Then_Else | For_Each |  
While_Do | Until_Do
```

Description

The statements that can appear in:

- The actions of a rule
- A function

Related Links

[Assignment_Statement](#)
[Compound_Assignment_Statement](#)
[Create_Statement](#)
[Delete_Statement](#)
[Execute_Function](#)
[Method_Call](#)
[Case_Selection](#)
[For_Each](#)
[Until_Do](#)
[If_Then_Else](#)
[While_Do](#)

Apply_Ruleset

Syntax

```
apply rulesetName [with { Parameter_Bindings }]
or
apply rulesetName( Argument_List )
or
rulesetName( Argument_List )
```

Description

A statement that applies the rules you define in a ruleset. There are three forms:

- With the first form, you use Parameter_Bindings to assign expressions to the ruleset parameters. You use the keyword apply. For example:

```
apply creditApprovalRuleSet with {
    customer = currentCustomer,
    rating = creditBureauReportRating
}.
```

The parameter bindings are *optional*. You can omit any parameter binding. In that case, the parameter is set to the special value unknown.

- The second form is more compact. You use the keyword apply and an Argument_List to pass parameter values. For example:

```
apply creditApprovalRuleSet (currentCustomer,
    creditBureauReportRating).
```

You use positional notation in passing parameter values. You separate multiple parameter values with commas (,). You can omit any parameter. (In other

words, a comma can immediately follow an open parenthesis or another comma.) In that case, the parameter is set to the special value unknown.

- The third form is more even more compact. You omit the apply keyword and simply use an Argument_List to pass parameter values. For example:

```
creditApprovalRuleSet (currentCustomer,  
                      creditBureauReportRating).
```

```
customer.approved = apply creditApprovalRuleSet with {  
    customer = currentCustomer,  
    rating = creditBureauReportRating  
}.  
customer.approved = creditApprovalRuleSet (currentCustomer,  
                                         creditBureauReportRating).
```

Apply_DecisionTable

Syntax

```
apply decisionTableName [with { Parameter_Bindings }]  
or  
apply decisionTableName( Argument_List )  
or  
decisionTableName( Argument_List )
```

Description

A statement that applies the rules you define in a decision table. There are three forms:

- With the first form, you use Parameter_Bindings to assign expressions to the decision table ruleset parameters. You use the keyword apply. For example:

```
apply creditApprovalDecisionTable with {  
    customer = currentCustomer,  
    rating = creditBureauReportRating  
}.
```

The parameter bindings are **optional**. You can omit any parameter binding. In that case, the parameter is set to the special value unknown.

- The second form is more compact. You use the keyword apply and an Argument_List to pass parameter values. For example:

```
apply creditApprovalDecisionTable (currentCustomer,  
                                  creditBureauReportRating).
```

You use positional notation in passing parameter values. You separate multiple parameter values with commas (,). You can omit any parameter. (In

other words, a comma can immediately follow an open parenthesis or another comma.) In that case, the parameter is set to the special value unknown.

- The third form is more even more compact. You omit the apply keyword and simply use an Argument_List to pass parameter values. For example:

```
creditApprovalDecisionTable (currentCustomer,
    creditBureauReportRating).
```

```
customer.approved = apply creditApprovalDecisionTable with {
    customer = currentCustomer,
    rating = creditBureauReportRating
}.
customer.approved = creditApprovalDecisionTable (currentCustomer,
    creditBureauReportRating).
```

Apply_DecisionTree

Syntax

```
apply decisionTreeName [with { Parameter_Bindings }]
or
apply decisionTreeName( Argument_List )
or
decisionTreeName( Argument_List )
```

Description

A statement that applies the rules you define in a decision tree. There are three forms:

- With the first form, you use Parameter_Bindings to assign expressions to the decision tree ruleset parameters. You use the keyword apply. For example:

```
apply creditApprovalDecisionTree with {
    customer = currentCustomer,
    rating = creditBureauReportRating
}.
```

The parameter bindings are *optional*. You can omit any parameter binding. In that case, the parameter is set to the special value unknown.

- The second form is more compact. You use the keyword apply and an Argument_List to pass parameter values. For example:

```
apply creditApprovalDecisionTree (currentCustomer,
    creditBureauReportRating).
```

You use positional notation in passing parameter values. You separate multiple parameter values with commas (,). You can omit any parameter. (In other

words, a comma can immediately follow an open parenthesis or another comma.) In that case, the parameter is set to the special value unknown.

- The third form is more even more compact. You omit the apply keyword and simply use an Argument_List to pass parameter values. For example:

```
creditApprovalDecisionTree (currentCustomer,  
                           creditBureauReportRating).
```

```
customer.approved = apply creditApprovalDecisionTree with {  
    customer = currentCustomer,  
    rating = creditBureauReportRating  
}.  
customer.approved = creditApprovalDecisionTree (currentCustomer,  
                                               creditBureauReportRating).
```

Apply_ScoreModel

Syntax

```
apply scoremodelName [with { Parameter_Bindings }]  
or  
apply scoremodelName( Argument_List )  
or  
scoremodelName( Argument_List )
```

Description

A statement that applies the rules you define in a score model. There are three forms:

- With the first form, you use Parameter_Bindings to assign expressions to the decision tree ruleset parameters. You use the keyword apply. For example:

```
apply creditApprovalScoreModel with {  
    customer = currentCustomer,  
    rating = creditBureauReportRating  
}.
```

The parameter bindings are **optional**. You can omit any parameter binding. In that case, the parameter is set to the special value unknown.

- The second form is more compact. You use the keyword apply and an Argument_List to pass parameter values. For example:

```
apply creditApprovalScoreModel (currentCustomer,  
                               creditBureauReportRating).
```

You use positional notation in passing parameter values. You separate multiple parameter values with commas (,). You can omit any parameter. (In other words, a

comma can immediately follow an open parenthesis or another comma.) In that case, the parameter is set to the special value unknown.

- The third form is more even more compact. You omit the apply keyword and simply use an Argument_List to pass parameter values. For example:

```
creditApprovalScoreModel (currentCustomer,
    creditBureauReportRating).
```

```
customer.approved = apply creditApprovalScoreModel with {
    customer = currentCustomer,
    rating = creditBureauReportRating
}.
customer.approved = creditApprovalScoreModel (currentCustomer,
    creditBureauReportRating).
```

Apply_Ruleflow

Syntax

```
ruleflowName( )
```

Description

A statement that invokes a ruleflow.

Optionally, the apply or execute keywords may be used with this statement, although the behavior of the statement is the same without using the keywords.

- A ruleflow that is set to start automatically or a ruleflow does not define a start event is invoked without an argument:

```
ruleflowName();
```

Assignment_Statement

Syntax

```
Property_Value = Primary_Expression | true | false | unavailable | unknown |
null | value
or
set Property_Value to Primary_Expression | true | false | unavailable | unknown |
null | value
```

Description

A statement that assigns the value of the Primary_Expression to either:

- The property the Property_Value represents
- A variable that you declare

Assignment statements cannot appear in rule conditions, only actions.

Related Links

[Property_Value](#)

[Primary_Expression](#)

Compound_Assignment_Statement

Syntax

```
Property_Value += Primary_Expression | value  
Property_Value -= Primary_Expression | value  
Property_Value *= Primary_Expression | value  
Property_Value /= Primary_Expression | value  
or  
increment Property_Value by Primary_Expression | value  
decrement Property_Value by Primary_Expression | value  
multiplyProperty_Value by Primary_Expression | value  
divideProperty_Value by Primary_Expression | value
```

Description

A compound assignment statement:

- Performs a numeric operation on a property or variable
- Assigns the result back to the property or variable

These symbols...	Are the same as...
+= increment	Property_Value = Property_Value + Primary_Expression
-= decrement	Property_Value = Property_Value - Primary_Expression
*= multiply	Property_Value = Property_Value * Primary_Expression
/= divide	Property_Value = Property_Value / Primary_Expression

These operators can appear in:

- Rule actions
- Function statements
- Initialization statements

The data type-compatibility conditions for [Numeric_Operators](#) also apply to [Collection_Comparison_Operators](#).

Related Links

[Property_Value](#)

[Primary_Expression](#)

Create_Statement

Syntax

```
a[n] className [ Initialize_Statement ]
```

Description

The statement to create an object and initialize it implicitly:

- On the right side of an assignment

```
order.newproduct = a Product initially {
    name = "Sweater",
    color = red }.
```

- In the declaration of an object variable

```
instockProduct is some Product initially a Product
    initially { name = "Shirt", color = blue }.
```

where Product is a class with two properties, name (a string) and color (an enumeration). For example:

```
a Product is an object with { name : a string,
    color : a Color }.
```

To access the object, you must declare it as either:

- An object property
- An object variable

For example, if you declare Product as an object property of the class Order:

```
an Order is an object with { productOrdered : some Product,
    quantityOrdered : an integer }.
```

You can then use it in a rule action:

```
newOrder is an Order.
if orderRequested
then newOrder.productOrdered is a Product initially
    { name = "Shirt", color = blue }
```

Related Links

[Initialize_Statement](#)

Delete_Statement

Syntax

```
delete objectExpression
delete objectExpression.propertyName
delete the propertyName of objectExpression
```

Description

A statement that deletes an object or object property during a rule session. An expression that produces an object follows the delete operator. For example:

```
delete BankAccount
```

The Delete_Statement can delete *only* objects created using the create operator during rule processing.

Return_Statement

Syntax

```
return [ Primary_Expression ]
```

Description

A statement that returns the value of the Primary_Expression from:

- A rule action within a ruleset
- A function

In order to return values from rulesets or functions, you must first declare the type of the returned value in the ruleset or function.

Return statements cannot appear in rule conditions, **only** in actions. If a return type is set for a ruleset, but no return statement is encountered, the ruleset returns the special value unavailable.

If a return statement is not encountered in the body of a function, the function will not compile.

Related Links

[Primary_Expression](#)

Execute_Function

Syntax

```
execute functionName [with { Parameter_Bindings }]
or
execute functionName( Argument_List )
or
functionName( Argument_List )
```

Description

A statement that specifies a call to a function. There are the following forms:

- You use Parameter_Bindings to assign expressions to the function's parameters. You use the keyword execute. For example:

```
execute Withdrawal with { acct = currentAccount,
                           amt = withdrawalAmount }.
```

Parameter bindings are optional. You can omit any parameter binding. If you do not assign values to parameters, they are set to the special value unknown.

- You use an Argument_List to pass parameter values. For example:

```
execute Withdrawal (currentAccount, withdrawalAmount).
```

You use positional notation in passing parameter values. You separate multiple parameter values with commas (,). You can omit any parameter. (In other words, a comma can immediately follow an open parenthesis or another comma.) In that case, the parameter is assigned the special value unknown.

- This form is much like the form above, except that you simply omit the keyword execute:

```
Withdrawal (currentAccount, withdrawalAmount).
```

If the function returns a value, the Execute_Function statement evaluates to a value of the specified return type. There are two examples:

```
amount = execute Withdrawal with { acct = currentAccount,
                                   amt = withdrawalAmount }.
```

```
amount = Withdrawal (currentAccount, withdrawalAmount).
```

You can overload a function name with several argument types. The compiler selects the appropriate function by comparing the argument types in the argument list to the function definitions. The argument list must exactly match the parameter declaration list.

Related Links

- [Argument_List](#)
- [Parameter_Bindings](#)

Method_Call

Syntax

```
className.methodName ( Argument_List )
or
objectExpression.methodName ( Argument_List )
or
call methodName on objectExpression using Argument_List
call objectExpression's methodName using Argument_List
or
set Property_Value to the outcome of methodName on objectExpression using
```

```
Argument_List  
set Property_Value to the outcome of objectExpression's methodName using  
Argument_List
```

Description

Specifies a call to an external method from:

- Rule actions
- Function statements
- Rule conditions, if the method returns a value and has no side effects.
Methods used in conditions must:
 - Not create or delete an object
 - Not modify a property of a Blaze Advisor object
 - Always return the same result when called with the same arguments

A method has a fixed number of arguments that its signature defines. You use these forms:

- *className.methodName* is a call to a static method of the class that maps to *className*.
- *objectExpression.methodName* is a call to an instance method. The method must be an instance method inherited from the mapped parent class of the object.

Use the **call** keyword when the method does not have a return value. Use the **set** keyword when the method does have a return value. Use the **set** keyword when the method does have a return value. Use the **call** keyword when the method does not have a return value.

This is an example of such a call, where the method is `withdraw()`. It takes one float argument and returns a void value.

```
BankAccount.withdraw (1000.0)  
call withdraw on BankAccount using 1000.0
```

Related Links

[Argument_List](#)

[Property_Value](#)

Functions

You can define a function body including parameter bindings and argument lists.

Function_Body

Syntax

```
{ Statement_Block }
```

Description

You can include a Statement_Block that contains one or more action statements to execute when calling the function. These can include any "", such as:

- Control_Constructs

You can use Control_Constructs in the function body to control the execution of action statements.

Example

This function applies three rulesets for a customer promotion. It also prints the names of customers who accepted the offered promotion.

As you can see from this example, you can within a function:

- Use control constructs

Any variables and objects you declare within a function are local to that function. You cannot access them outside the function. Once the function returns, any local variables or objects are no longer available.

Related Links

[Control Constructs](#)

[Statement_Block](#)

Parameter_Bindings

Syntax

```
parameterName = Primary_Expression [, parameterName = Primary_Expression ]...
```

Related Links

[Primary_Expression](#)

Argument_List

Syntax

```
[ Primary_Expression [, Primary_Expression ]...]
```

Description

An argument list that you pass when invoking:

- A function
- A ruleset

- A decision metaphor
- A method

The list can be empty if you do not need to pass any arguments.

The expressions in the argument list *must* evaluate to values that match the parameter types you declare in the function, ruleset, decision metaphor or method definition. For example, if you define a function as having these three parameters:

```
{  
    amount: a real,  
    personalIdentityNumber: a string,  
    approved: a boolean } . . .
```

then your argument list must pass expressions to that function that match the parameter types:

```
CreditCheck(5000.00, 211031976, true).  
CreditCheck(2000 * 0.25, customer.socialSecurityNumber, false).
```

Related Links

[Primary_Expression](#)

Control Constructs

There are several types of constructs that can be used in SRL.

Case_Selection

Syntax

```
select Primary_Expression  
case Primary_Expression : Statement_Block  
case Primary_Expression : Statement_Block  
...  
otherwise : Statement_Block
```

Description

This is similar to a C/C++/switch construct. The compiler compares the first expression (which select precedes) to several possible expressions (which case precedes). The matching expression then determines the action the engine performs.

Example

```
select countryofOrigin of product  
case China : apply ChinaTariffRules  
case Japan : apply JapanTariffRules  
case Thailand : apply ThailandTariffRules  
otherwise : print("No tariffs apply.").
```

SRL does not restrict the expressions to constants. As a result, several cases may be true at the same time. When this happens, only the statement of the first true case executes.

Related Links

[Primary_Expression](#)

[Statement_Block](#)

For_Each

Syntax

```
for each className | interfaceName [such that Boolean_Expression ]
    do Statement_Block
or
for each eElementTypeName in collectionName [such that Boolean_Expression ]
    do Statement_Block
```

Description

The `for each` construct performs the same action on all instances of a class or interface that satisfy a given expression. The constraint `such that Boolean_Expression` is optional. If you omit it, the statement iterates over all the instances of the class.

You can use the `for each` construct (above) to iterate through all the elements in a collection that satisfy a given expression:

Example

```
for each customer such that (it.balance > 10000)
do {
    print(it.name " qualifies for the promotion.").
    apply PromotionRuleset.
}
for each accident in ListOfAccidents
do {
    print("Amount of claim = $" it.claimAmount)
}
```

You access the current object in the `for each` loop by using the keyword `it`.

Related Links

[Boolean_Expression](#)

[Statement_Block](#)

If_Then_Else

Syntax

```
if Boolean_Expression then Statement_Block [else Statement_Block ]
```

Description

The syntax of the `if...then...else` construct is similar to that of the `if...then...else` structure of a rule. However, it does not use the Agenda. Therefore, the rule engine does not reevaluate it when data changes. You can distinguish a control construct from a business rule by:

- The context
In a ruleset, the construct is a rule, unless it occurs as an action statement of another rule. In a function body, it is always a control construct.
- The presence of a rule name

The `else` clause is optional.

If the boolean expression contains several clauses that boolean operators (`and` or `or`) separate, place the entire expression within parentheses `(())`. If the `else` clause follows two overlapping `if...then` clauses, it associates with the last one.

Example

```
if (customer.purchase > 5000 and customer.credit is "good")
then apply HighDiscountRuleset
else if (customer.purchase > 5000 and
        customer.credit <> "good")
then apply LowDiscountRuleset
else if (customer.purchase > 2500 and
        customer.credit is "good")
then apply LowDiscountRuleset
else print("Purchase not eligible for discount.").
```

Related Links

[Boolean_Expression](#)

[Statement_Block](#)

While_Do

Related Links

[Until_Do](#)

Until_Do

Syntax

```
while Boolean_Expression do
until Boolean_Expression do Statement_Block
```

Description

A statement in which the rule engine repeatedly performs an action:

- While an expression is true, or
- Until an expression becomes true

If the expression contains several clauses that and or or separate, place the entire expression within parentheses (()).

Examples

When you use the keyword **while**, the engine repeatedly performs the action while the expression is true. It stops when the expression becomes false. For example:

```
while (productsRemaining > 0) do
{
    apply CheckPriceRuleset(product).
    productsRemaining -= 1.
}
```

If the expression is initially **false**, the engine does not perform the actions in the **Statement_Block**. If the expression evaluates to **unknown** or **unavailable**, the construct throws an exception.

When you use the keyword **until**, the engine repeats the action while the expression is false. It stops when the expression becomes true. For example:

```
until (productsRemaining = 0) do
{
    apply CheckPriceRuleset(product).
    productsRemaining -= 1.
}
```

until is equivalent to **while not**.

Related Links

[Boolean_Expression](#)

[Statement_Block](#)

Exceptions

SRL includes statements for handling exceptions.

Try_Catch_Finally_Statement

Syntax

```
try Statement_Block  
catch a[n] exceptionClassName with Statement_Block  
[finally Statement_Block ]
```

Description

A statement that provides the actions for handling an exception. It has three parts:

- **try Statement_Block**
This clause specifies the block of statements (for example, a control construct) that might generate an exception.
- **catch a[n] exceptionClassName with Statement_Block**
This clause specifies the actions to perform when an exception occurs. *exceptionClassName* specifies the name of the exception. The *Statement_Block* contains the actions to perform. Use the keyword *it* to refer to the current exception.
You can include several *catch* clauses if you want to catch several exceptions.
You can catch any object that is an instance of the `java.lang.Throwable` class.
- **finally Statement_Block**
This clause contains a block of statements that are always executed, regardless of how the statements in the *try* clause execute. The *finally* clause generally performs any necessary cleanup for the statements in the *try* clause. It is optional.

You must follow the *try* clause with at least one *catch* or *finally* clause.

Related Links

[Statement_Block](#)

Throw_Statement

Syntax

```
throw exceptionExpression
```

Description

A statement to indicate that an exception has occurred. *exceptionExpression* is an expression that evaluates to an object that extends the class.

SRL Operators

SRL includes several types of operators.

Assignment_Operator

Syntax

```
=
```

or

```
set objectExpression | variableName to objectExpression | variableName
```

Description

The assignment operator that appears in an [Assignment_Statement](#) on page 295. It assigns a value:

- To a property of an object or
- To a variable that you declare

Assignment statements cannot appear in rule conditions, only in actions.

Boolean_Operator

Syntax

```
not | or | and
```

Description

The boolean operators that can appear in:

- Rule condition expressions
- Action assignments
- Initialization statements
- Initial values

The operands of these operators must be booleans.

Collection_Comparison_Operator

Syntax

```
is empty | is not empty
```

Description

The collection comparison operators that appear in a comparison statement. You can use them to determine whether or not a collection is empty.

Comparison_Operator

The following table provides usage information for comparison operators:

Short form	Long form	Negated long form	Example
=	is equal to	is not equal to	if payment is equal to amountDue then ...
	is		if todaysDate is paymentDate then ...
<>	is different from	is not different from	if payment is different from amountDue then ...
	is not equal to		if payment is not equal to amountDue then ...
<	is less than	is not less than	if payment is less than amountDue then ...
	is smaller than	is not smaller than	if payment is smaller than amountDue then ...
	is earlier than	is not earlier than	if today's date is earlier than theApplication's date then ...
	precedes	does not precede	if today's date precedes theApplication's date then ...
	comes before	does not come before	if today's date comes before theApplication's date then ...
>	is more than	is not more than	if payment is more than amountDue then ...
	is larger than	is not larger than	if payment is larger than amountDue then ...
	is later than	is not later than	if today's date is later than theApplication's date then ...
	follows	does not follow	if today's date follows theApplication's date then ...
	comes after	does not come after	if today's date comes after theApplication's date then ...

Short form	Long form	Negated long form	Example
	exceeds	does not exceed	if payment exceeds amountDue then ...
<=	is less than or equal to	is not less than ...	if payment is less than or equal to amountDue then ...
	is smaller than or equal to	is not smaller than ...	if payment is smaller than or equal to amountDue then ...
	is earlier than or equal to	is not earlier than ...	if today's date is earlier than or equal to theApplication's date then ...
	is up to	is not up to	if retailPrice is up to suggestedPrice then ...
	is at most	is not at most	if retailPrice is at most \$500.00 then ...
=>	is more than or equal to	is not more than ...	if payment is more than or equal to amountDue then ...
	is larger than or equal to	is not larger than ...	if payment is larger than or equal to amountDue then ...
	is later than or equal to	is not later than ...	if today's date is later than or equal to theApplication's date then ...
	is at least	is not at least	if retailPrice is at least \$500.00 then ...
	is between n1 and n2 (where n1 is less than n2)	is not between n1 and n2	if creditScore is between 120 and 150 then ... (Each value is an end point for a range and must appear in an increasingly sequential order. So, you cannot use something like this: if creditScore is between 150 and 120.)

Description

The comparison operators that can appear in:

- Condition expressions of rules
- Boolean expressions
- Condition expressions of control constructs

The two operands must both be either numeric or string data types, unless you are using the `is` (is equal to) and `<>` (is not equal to) operators, or their alternate forms. These can compare all data types (including boolean operands, classes, and enumerations).

Compound_Assignment_Operator

Description

The compound assignment operators that appear in a compound assignment statement. You can use them to:

```
+ = | - = | * = | / =
```

- Perform a numeric operation on a property or variable
- Assign the result back to the property or variable

The data type compatibility conditions for a `Numeric_Operator` also applies to `Collection_Comparison_Operator`.

Related Links

[Numeric_Operator](#)

[Collection_Comparison_Operator](#)

Numeric_Operator

Description

The numerical operators that can appear in:

- Rule condition expressions
- Action assignments
- Initialization statements
- Initial values

The two operands must be of compatible numeric data types:

- If both operands of a numeric operator are of the same type (integer or real), then the result is also of that type. However, this is not true when you divide an integer by another integer using the `/` operator. This produces a real result.
- If the operands are of different types, the integer operand is converted to a real number. The operation then produces a result that is also a real.

There are two options for division:

- Use the / operator if you want a real quotient. For example:

```
19 / 5 = 3.8
```

- Use the div operator if you want an integer quotient. For example:

```
19 div 5 = 3
```

String_Comparison_Operator

Syntax

```
starts with | does not start with | ends with | does not end with |
    is blank | is not blank | Comparison_Operator
or
Comparison_Operator ( ignoring case ) | String_Comparison_Operator ( ignoring case )
```

where the **ignoring case** keyword causes uppercase characters to be evaluated at the same ASCII value as the corresponding lowercase character.

Description

The string comparison operators that can appear in:

- Condition expressions of rules
- Boolean expressions
- Condition expressions of control constructs

String comparison operations are based on the ASCII values of the characters.

Examples

```
if theForm's name starts with "req" then ...
if theForm's name ends with "req" then ...
if theApplicant's lastName is blank then...
```

Related Links

[Comparison_Operator](#)

Operator Precedence

This is the precedence order of the operators, from highest to lowest:

Table 45: Operator Precedence

Highest to Lowest level		
1	. (dot)	Property access
2	as, []	Type conversion, index

Table 45: Operator Precedence (continued)

Highest to Lowest level		
3	+, - ,not	Unary prefix (arithmetic, boolean)
4	*, /, mod, div	Multiplication, division
5	+, -	Addition, subtraction
6	" "	String concatenation
7	<, <=, >, >=	Comparison
8	=, is, <>, *=, /=, +=, -=	Assignment, declaration, comparison, compound assignment
9	and, or	Boolean

The above operators associate to the left at a given precedence level.

Keywords

The keywords outlined in this section are the reserved words of the SRL syntax, and they are **case-sensitive**.

Object-specific Keywords

The following table provides usage information for object-specific keywords:

Keyword	Description
a, an	Introduces a property or data type in the declaration of an object, enumeration, variable, or property. You can also use it in statements with a class name following to indicate an instantiation of the class. You can use the article a or an to form English-like sentences.
array	Introduces a collection that holds a number of objects or values.
association from Xxx to Xxx	Introduces a collection where objects or values of one type are associated with those of another type.
fixed array	Introduces a collection that holds a fixed number of objects or values.
initially	Introduces an initializer or initial value. When it appears in an object declaration, or a create-object statement, an initializer follows it. This can be a statement or block of statements. When it appears in a variable declaration, an expression (usually a constant) follows. It specifies the initial value of the variable.

Keyword	Description
is a, is an	This keyword combination is characteristic of an object declaration. It introduces the class from which the object inherits. You can also use this keyword combination with a class name following to check the type of an object.
is any	This keyword combination is characteristic of a pattern declaration. A class name follows it. The condition that refers to the pattern will bind to all the instances of the class. This causes the then actions or else actions to execute for every object that satisfies the condition. Note Pattern usage is available only if your Ruleset editor supports local patterns.
is one of	This keyword combination declares an enumeration and introduces the values that the enumeration may take.
is some	This keyword combination specifies the type of the variable.
it, he, she	These keywords designate the current object you are initializing or testing. You can use them to access the properties of the referenced object. The forms he and she are suitable for use with objects that represent people.
some	Declares a property or variable that refers to objects. The name of the class from which the referred object derives follows it. The referred object can change during execution.
such that	This keyword combination introduces a constraint that objects that a pattern or quantified condition designate must satisfy. Note Pattern usage is available only if your Ruleset editor supports local patterns.
the ... of's ...	These keyword combinations access the property of an object. You can specify an object name or expression that produces an object. The alternate form of accessing properties uses the property-access operator through this notation: <i>object.property</i> .
with	Introduces the properties of a subclass or object.

Ruleset and Rule-specific Keywords

The following table provides usage information for ruleset and rule specific keywords:

Keyword	Description
apply	Begins a statement that applies a ruleset or decision metaphor.
at immediate priority	This keyword combination defines a rule-execution priority. It specifies that the rule is to execute during the next action phase.
at least... satisfy...	This keyword combination defines a quantified condition. It tests whether at least a specified number of objects of a particular class exist that can satisfy a condition (which satisfies introduces). Optionally, you can apply a constraint to a class using a such that clause.

Keyword	Description
at most... satisfy...	This keyword combination defines a quantified condition. It tests whether no more than a specified number of objects of a particular class exist that can satisfy a condition (which satisfies introduces). Optionally, you can apply a constraint to a class using a such that clause.
at priority	This keyword combination defines a rule-execution priority. A numeric value that specifies the rule ordering on the rule queue follows it.
effective from ...	Adds a conditional expression to a rule that only fires the rule after a specified date.
effective to...	Adds a conditional expression to a rule that only fires the rule before a specified date.
effective from ... to ...	Adds a conditional expression to a rule that only fires the rule if the current date is between the specified dates.
effective everyday from ...	Adds a conditional expression to a rule that only fires the rule after a specified time.
effective everyday to ...	Adds a conditional expression to a rule that only fires the rule before a specified time.
effective everyday from ... to ...	Adds a conditional expression to a rule that only fires the rule if the current time is between the specified times.
else	Introduces the set of actions to execute when the negation of a rule's condition is satisfied.
every... satisfy...	This keyword combination defines a quantified condition. It tests whether every object of a particular class can satisfy a condition (which satisfies introduces). Optionally, you can apply a constraint to a class using a such that clause.
every... (set operations properties only)	This keyword specifies a class name used in a set operations properties expression. This is an example: <code>arrayName[every className]</code> . In this case, every must be contained in brackets along with the class name.
exactly... satisfy....	This keyword combination defines a quantified condition. It tests whether a specified number of objects of a particular class exist that can satisfy a condition (which satisfies introduces). Optionally, you can apply a constraint to a class using a such that clause.
for	Introduces the parameter declarations in a ruleset definition.
if	Introduces the set of conditions within a rule.
is	Introduces the rule body.
is changed	This keyword combination represents the built-in value- or object-changed event that appears only in when statements. It tests a property value or pattern. It then indicates that the rule engine will re-evaluate the rule every time the property value or pattern changes.
is needed	This keyword combination represents the built-in value-needed event that appears only in when statements.

Keyword	Description
	It tests a property value, variable, or pattern. It then indicates that the rule engine will evaluate the rule when the tested property value, variable, or pattern is currently unknown and when it could evaluate other rules if known. Rules the needed property value, variable, or pattern control are value-lookup rules.
return	Specifies the value or expression a function or ruleset returns.
returning	Specifies the type, class, or interface of the value a function or ruleset returns.
rule	Begins the definition of a rule (optional).
then	Introduces the set of actions to execute if a rule's condition statement is satisfied.

Event-specific Keywords

The following table provides usage information for event-specific keywords:

Keyword	Description
do	Precedes one or more statements that the rule engine executes when an event occurs.
is changed	This keyword combination appears in an event rule that handles an event generated by a change in the value of a property.
is created	This keyword combination appears in an event rule that handles an event you generate when you create (or map) an object, but before you initialize it.
is deleted	This keyword combination appears in an event rule that handles an event you generate when you delete (or unmap) an object, just before its properties and methods become inaccessible.
is initialized	This keyword combination appears in an event rule that handles an event you generate just after you create (or map) and initialize an object.
is needed	This keyword combination appears in an event rule that handles events generated when the rule engine needs the value of a property or variable.
new	Operator that provides access to the new value of the changed property to which it is prefixed. You can only use it in an <code>is changed</code> event rule.
occurs	This keyword appears in an event rule that handles external events. It follows the name of the external event class.
old	Operator that provides access to the old value of the changed property to which it is prefixed. You can only use it in an <code>is changed</code> event rule.
raw	Operator that inhibits the <code>is needed</code> event rule for a specific expression.
whenever	Precedes the name of a property, class, or external event in an event rule.

Control Construct-specific Keywords

The following table provides usage information for control construct-specific keywords:

Keyword	Description
case	Introduces each expression (and its corresponding statement) in a case-selection construct.
do	Introduces a statement block for loop constructs (for each...do, while...do, and until...do).
for each	This keyword combination begins a for each...do construct.
if ... then ... else ...	In a ruleset, this combination is interpreted as a rule unless it appears in the action block of another rule. In a function body, this acts as a control construct.
in	Precedes a collection name in a for each...do construct or a pattern.
otherwise	In a case-selection construct, introduces the statement to execute when none of the case expressions are satisfied.
select	Begins a case-selection construct.
until	Begins an until...do construct. until is equivalent to while not.
while	Begins a while...do construct.

Function-specific Keywords

The following table provides usage information for function-specific keywords:

Keyword	Description
execute	Begins a function call.
return	Specifies the value a function or a ruleset returns.
returning	Specifies the type, class, or interface of the value a function or ruleset returns.

Exception-specific Keywords

The following table provides usage information for exception-specific keywords:

Keyword	Description
catch	This keyword introduces the catch clause in a Try_Catch_Finally_Statement. It precedes the name of the exception and the actions to perform when that exception occurs.
finally	This keyword introduces the finally clause in a Try_Catch_Finally_Statement. The finally clause generally performs any necessary cleanup for the statements in the try clause. It is optional.

Keyword	Description
throw	This keyword introduces a <code>Throw_Statement</code> , which indicates that an exception has occurred.
try	This keyword introduces the <code>try</code> clause in a <code>Try_Catch_Finally_Statement</code> . It precedes the block of statements that might generate an exception.

String-specific Keywords

The following table provides usage information for string-specific keywords:

Keyword	Description
starts with	Tests two strings to see if the first string starts with the second string.
does not start with	Tests two strings to see if the first string does not start with the second string.
ends with	Tests two strings to see if the first string ends with the second string.
does not end with	Tests two strings to see if the first string does not end with the second string.
is blank	Tests a string to see if it is blank.
is not blank	Tests a string to see if it is not blank.
contains text	Tests two strings to see if the first string contains any occurrences of the second string.
is contained in text	Tests two strings to see if the first string occurs in the second string.
is not contained in text	Tests two strings to see if the first string does not occur in the second string.
(ignoring case)	Used in conjunction with any other string comparison operator to cause uppercase characters to be evaluated at the same ASCII value as the corresponding lowercase character.
contains match	Tests to see whether a regular expression match is contained within a string.
contains match (ignoring case)	Tests to see whether a case insensitive regular expression match is contained within a string. See (ignoring case)
does not contain match	Tests to see if a regular expression does not contain a match within a string.
does not contain match (ignoring case)	Tests to see if a case insensitive regular expression does not contain a match within a string. See (ignoring case)
exactly matches	Tests to see if a regular expression completely describes a string.
exactly matches (ignoring case)	Tests to see if a case insensitive regular expression completely describes a string. See (ignoring case)

Keyword	Description
does not exactly match	Tests to see if a regular expression does not completely describe a string.
does not exactly match (ignoring case)	Tests to see if a case insensitive regular expression does not completely describe a string. See (ignoring case)

Operators

The following table provides usage information for operators used to test values:

Keyword	Description
and	Combines boolean expressions using a logical AND within a rule condition. (Also known as a logical conjunction.)
as	The type-conversion operator.
delete	The delete-object built-in operator.
div	The integer-quotient operator for numeric expressions.
mod	The remainder operator for numeric expressions.
not	The boolean-negation operator for boolean expressions.
or	Combines boolean expressions using a logical OR within a rule condition. (Also known as a logical disjunction.)
is	In declarations, the is operator separates what you are declaring (to the left of the operator) from what you are declaring it to be (the remainder of the statement). In expressions, SRL interprets is as an equality-test operator.
= set ... to ...	The assignment operators. In statements, SRL interprets = as the assignment operator. In expressions, SRL interprets = as an equality-test operator.
= is is equal to is not different from	The equality-test operators. In statements, SRL interprets = as the assignment operator. In expressions, SRL interprets = as an equality-test operator. Another version is "set x to y".
<> is different from is not equal to	The inequality-test operators.
< is less than is smaller than is earlier than	The less than test operators.

Keyword	Description
precedes comes before is not at least	
> is greater than is more than is larger than follows comes after exceeds is not at most	The greater than test operators.
<= is less than or equal to is smaller than or equal to is earlier than or equal to is up to is at most is not more than is not larger than is not later than does not follow does not come after does not exceed	The less than or equal test operators.
>= is greater than or equal to is more than or equal to is larger than or equal to is later than or equal to is at least is not less than is not smaller than is not earlier than does not precede does not come before	The greater than or equal test operators.
is between n1 and n2	The range comparison operator.
" "	Concatenate by juxtaposing expressions and string constants ("..."). If an expression appears next to a string constant (immediately before or immediately after), its value is converted to a string and it is concatenated with the string constant. If neither of the expressions that you want to concatenate is a string constant, you can insert an empty string constant ("") between the expressions. This is true whether the expressions result in an actual value or a special value such as unknown or unavailable.

Keyword	Description
+ plus	The addition operators for numeric expressions.
- minus	The subtraction operator for numeric expressions.
* times	The multiplication operators for numeric expressions.
/ divided by	The division operators for numeric expressions. The resulting quotient is a real.
+= increment	The compound assignment operator for incrementing numeric expressions.
-- decrement	The compound assignment operator for decrementing numeric expressions.
*=	The compound assignment operator for multiplying numeric expressions.
/=	The compound assignment operator for dividing numeric expressions.
.	The property-access operator. Note: SRL distinguishes <code>o.p</code> and <code>o. p</code> . In the first case, it interprets the dot (.) as a property-access operator. In the second case, it interprets it as a statement terminator.

Data Types

The following table contains information about primitive data types keywords that are supported.

Keyword	Description
boolean	The boolean data type.
date	The date data type.
duration	The duration data type.
integer	The integer data type. An integer is implemented as a long .
money	The money data type.
object	The object class.
real	The real data type. A real is implemented as a double .
string	The string data type.
time	The time data type.
timestamp	The timestamp data type.

Special Values

Use special values keywords when you want to test for the presence of a value not specified in the XML payload, and the property is used in the decision logic.

Keyword	Description
available	Use available to test if a computed value was successfully obtained.
known	Use known to test if a value has been determined.
null	Use null to test if an object, a variable or object property of a complex type has been initialized.
unavailable	Use unavailable to test if a value cannot be obtained or computed.
unknown	Use unknown to test if a value has not yet been determined.
needed	Denotes a property of an object whose value should be obtained because it is particularly relevant to rule processing. This value cannot be assigned or tested in a rule condition. This special value is supported only in the Advanced Builder editor when authoring in FICO's Structured Rule Language (SRL).

Special values are available for all primitive types, Advisor primitives types, enumerations, and business terms. When using the drop-down menus in the **Rule Builder** for Boolean values, enumeration values, or business terms of type string associated with a value list, the special values always appear at the bottom of the list in bold font. When entering a value for other primitive types and Advisor primitive types, enter the special value that you want to use in the field. In most cases, when building condition expressions, the only operator that is available for special values is the **is equal to** operator.

This is an example of a special value used to create a condition:

```
customer.accountType is unavailable or customer.accountType is "Standard"
```

The special value keywords are valid on the right side of a comparison operation; however, their use is invalid in most expressions such as `x + unknown`.

Special values can be included in statements for initialization rules and variable initializations. For example, you can create a statement such as this one:

```
customer.accountType = unknown
```

Special values can be passed as parameters to functionals or used as a return type. However, `null` is applicable only if the parameter or return type is an object or a variable of a complex type. Special values are not supported with Built-in functions or methods.

 **Note** To use a special value keyword as a string literal, enter the keyword in the field, and then enclose it in double quotes.

Grammar

Like any other language, there are grammatical rules in SRL.

Naming Requirements and Conventions

Use the following requirements and conventions when naming decision entities:

- Display names viewed in the Project Explorer must contain only letters, digits, or underscores (no spaces, punctuation marks or special characters).
- Either ASCII or Unicode letters and digits are allowed. For example, a name can include an accented or Japanese character.
- Names are case-sensitive. Upper and lowercase names are treated differently.
- The first character of any name cannot be a digit.
- Certain words (such as the names of types, operators, functions, and special values) are reserved by the engine. For example, using "function" as a name throws an error because function is a reserved keyword.
- All reference names for decision entities must be unique in the project.
- Parameter names must be unique in the decision entity in which they are created.
- Variable and pattern names must be unique in the ruleset in which they are created.



Note Pattern usage is available only if your Ruleset editor supports local patterns.

Punctuation

Use the following punctuation rules:

- Inside a block, statements can end with a semicolon (;), comma (,) or a period (.). FICO recommends using the period.
- If a statement or a declaration ends on a curly brace, SRL does not require punctuation just after the closing curly brace (}).
- You can add a period (.) or comma (,) just before a closing curly brace (}), but SRL does not require it.
- You must enclose block statements, such as a list of properties, in curly braces ({{}}) unless they contain a single statement.
- SRL allows a comma (,) at the end of a declaration list.
- You must separate top-level declarations (class, object, rule, and so on) with a period (.). SRL does not allow a comma here. The period is optional after a closing curly brace (}).
- You must separate the name of a property and its type with a colon (:).
- You must separate arguments in a method with commas (,).

Reserved Words

A reserved word is a word used for a special purpose (such as the name of a type of operator). These words cannot be used as an ordinary identifier, such as a ruleset name. Because case-sensitivity is observed, reserved words are reserved only when the letters are all in lowercase. For example, `array` is a reserved word but `aRay` is not.

This is the list of reserved words:

a
an
and
any
any other value
apply
array
as
ask
association
at
available
be
becomes
boolean
by
case
catch
changed
created
count
date
decreases
deleted
do
does
duration
each
else
event
execute

false
finally
fixed
for
from
function
has
have
if
ignore
immediate
import
in
increases
initialized
initially
integer
is
known
least
less
money
more
must
needed
new
no
not
null
object
occurs
of
otherwise
outcome
otherwise
package
pattern
priority
raw

real
return
returning
rule
ruleset
satisfy
is
select
set
some
string
such
times
than
that
the
then
throw
time
timestamp
to
true
try
unavailable
unknown
until
value
when
whenever
while
with

Writing Comments in SRL

The “//” is often used during the development phase to temporarily “comment out” certain lines of SRL so that the rest of the project can compile. If you choose to include comments in the SRL, you must use “//” before:

- Each line of your descriptive sentences
- Any line spaces

Date, Time, Timestamp, and Duration Data Types

Your application includes date, time, timestamp, and duration data types and support classes that let you easily and efficiently use dates and times in FICO's Structured Rule Language (SRL).

This is an overview of each data type:

- **date**
Indicates a particular date. It does not include any time information. These are valid SRL examples:
 - November 19, 1999
 - January 1, 2020
- **time**
Indicates a particular time in a given day. These are valid SRL examples:
 - '10:30 am' or '10:30 AM'
 - '6:25 pm' or '6:25 PM'
- **timestamp**
Indicates a particular date and time. These are valid SRL examples:
 - 'November 19, 1998, 10:30 am' or 'November 19, 1998, 10:30 AM'
 - 'January 1, 2020, 6:25 pm' or 'January 1, 2020, 6:25 PM'

Use the time data type instead of the timestamp data type when the date information is irrelevant and when you are only interested in time relative to the beginning of the current day.
- **duration**
Indicates an amount of time. These are valid SRL examples:
 - 3 years 1 year = 365 days (366 days if the duration is not tied to a specific date)
 - 5 months 1 month = 31 days

Single quotation marks are not needed for duration values.

You can add or subtract a duration value from a timestamp, date, or time value to yield another timestamp, date, or time value. The difference between two timestamp, date, or time values is a duration. However, the duration value itself is not a timestamp, date, or time, it is an amount of time.

These data types have the following ranges and precision:

Data Type	Range	Precision
timestamp	-292,471,208 years to +292,471,208 years	Nanosecond precision
time	-292,471,208 years to +292,471,208 years	Nanosecond precision
date	-2,147,483,648 years to +2,147,483,647 years	Day precision
duration	-5,883,516 years to + 5,883,516 years	Nanosecond precision

Calculating and Comparing Date, Time, Timestamp, and Duration Values

You perform calculations and comparisons on date, time, timestamp, and duration values by using:

- Structured Rule Language (SRL) operators
See [Built-in Operators](#) on page 327
- Methods in the support classes

The values computed by these operations and methods take into account:

- Differences in the number of days in different months
- Time zone changes

If you add a one-month duration to a date, the duration is interpreted as 28, 29, 30, or 31 days, depending on the date to which it is added. Here are some examples:

SRL Statement	Result
'15-Oct-2020' + 1 month	November 15, 2020
'15-Nov-2020' + 1 month	December 15, 2020
'15-Oct-2020' + 31 days	November 15, 2020
'15-Nov-2020' + 31 days	December 16, 2020

If a timestamp or a time value includes time zone information, the time zone offsets are taken into account when comparing two values or when computing the difference between two values.

If the time zone is not specified, the default time zone set by the environment is used. For example:

SRL Statement	Result
'14:08:30 EST' - '14:08:30 GMT'	5 hours

Localization is facilitated by using the default `java.util.Locale` object associated with your project to provide default input and output string formats for creating and printing date, time, timestamp, and duration values. The support classes also provide methods for customizing your input and output strings.

Built-in Operators

You can perform various operations with SRL expressions to manipulate date, time, timestamp, and duration values. SRL enforces strict typing rules and supports these operations:

Operation	Type of Result
timestamp + duration	timestamp
timestamp - duration	timestamp

Operation	Type of Result
date + time	timestamp
date + duration	date
date - duration	date
time + duration	time
time - duration	time
time - time	duration
duration + duration	duration
duration - duration	duration
- duration	duration
integer * duration	duration
timestamp comparison_operator timestamp	boolean
date comparison_operator date	boolean
time comparison_operator time	boolean
duration comparison_operator duration	boolean

where `comparison_operator` can be any of these:

```
< <= > >= is <>
```

The `NdDate`, `NdTime`, `NdTStamp`, and `NdDuration` support classes provide various methods for performing additional operations on these data types.

Creating a Date, Time, Timestamp, or Duration

This section contains information on how to create Blaze Advisor primitive data types.

Declaring a Date, Time, Timestamp, or Duration Variable

You declare a date, time, timestamp, or duration as a variable the same way you declare an integer, real, or string variable:

```
startingDateTime is a timestamp initially  
'03-Nov-2020 06:43:24 PM'.  
dueDate is a date initially '12/30/2020'.  
dailyStartTime is a time initially '09:00:00 AM'.  
daysPastDue is a duration initially 1 year.
```

 **Note** You use `a` instead of `some` in declaring these variables. This is because timestamp, date, time, and duration are primitive data types, and not classes.

Declaring a Date, Time, Timestamp, or Duration Property

You can declare a date, time, timestamp, or duration using Structured Rule Language (SRL).

You declare a date, time, timestamp, or duration as a property the same way you declare an integer, real, or string property:

```
a Policy is an object with {
    an effectiveDate: a date,
    an createdTimestamp: a timestamp,
    an expirationDate: a date.
    an effectiveDuration: a duration.
} initially {
    effectiveDate = '1/01/2020',
    createdTimestamp = calendar().currentTimestamp(true),
    expirationDate = '03/31/2020',
}
```

You can initialize a date, time, timestamp, or duration using a string literal, enclosed by single quotation marks, as in this example:

```
startingDateTime is a timestamp initially
'03-Mar-2020 06:43:24 PDT'.
dueDate is a date initially '10/30/2020'.
dailyStartTime is a time initially '09:00:00 AM'.
daysPastDue is a duration initially 1 year.
```

You can use various formats for the string literal initializer.

Blaze Advisor date, time, timestamp, and duration data types have properties that you can access using the usual property notation. For example, you access the property `year` of a date property called `birthday` in this way:

```
birthday.year
```

Date Properties

The properties in the Blaze Advisor date data type are read-only. They behave like the other built-in data types (`boolean`, `integer`, `real`, `string`). You can use the operators to produce new values, but not to modify values. The date data type has these properties:

Property	Description
<code>era</code>	The era, either <code>java.util.GregorianCalendar.BC</code> or <code>java.util.GregorianCalendar.AD</code> .
<code>yearInEra</code>	The year in the era as a positive number; for example, 52 for 52 BC.
<code>year</code>	The year, as a positive or negative number. Note: To avoid any discontinuity when computing differences, 1 BC is returned as 0 and 2 BC as -1.
<code>month</code>	The month, as one of the <code>java.util.Calendar</code> constants (<code>Calendar.JANUARY</code> , ...).

Property	Description
monthDay	The day in the month in the [1,31] range.
yearDay	The day in the year in the [1,366] range.
weekDay	The weekday as one of the java.util.Calendar constants (Calendar.MONDAY, ...).

Time Properties

The properties in the Blaze Advisor `time` data type are read-only. They behave like the other built-in data types (`boolean`, `integer`, `real`, `string`). You can use the operators to produce new values, but not to modify values. The `time` data type has these properties:

Property	Description
dayOffset	The day offset number of days the time value is offset. It is usually zero. It becomes nonzero when the addition or subtraction of a duration value produces a time value that logically belongs to a different day.
amPm	The a.m. or p.m. indicator, either <code>java.util.Calendar.AM</code> or <code>java.util.Calendar.PM</code> .
amPmHour	The hour for the a.m. or p.m. system in the [1,12] range.
hour	The hour in the day in the [0,23] range.
minute	The minute in the hour in the [0,59] range.
second	The second in the minute in the [0,59] range.
milliseconds	The millisecond in the second in the [0,999] range.
microseconds	The microsecond in the second in the [0,999999] range.
nanoseconds	The nanosecond in the second in the [0,999999999] range.
timeZone	The time zone as an instance of the <code>java.util.TimeZone</code> class. It is <code>null</code> if you have not set the time zone.
daylightOffset	The duration value the time is offset because of daylight-savings time.

Timestamp Properties

The properties in the Blaze Advisor `timestamp` data type are **read-only**. They behave like the other built-in data types (`boolean`, `integer`, `real`, `string`). You can use the operators to produce new values, but not to modify values. The `timestamp` data type has these properties:

Property	Description
era, yearInEra, year, month, monthDay, yearDay, weekDay	See Declaring a Date, Time, Timestamp, or Duration Property on page 329.
amPm, amPmHour, hour, minute, second, milliseconds, microseconds, nanoseconds, timeZone, daylightOffset	See Declaring a Date, Time, Timestamp, or Duration Property on page 329.

Property	Description
millisecondsSinceOrigin	The number of milliseconds since January 1, 1970, 00:00:00 GMT.
extraNanoseconds	To get a timestamp with full precision, the number of nanoseconds (in the [0,999999] range) to add to millisecondsSinceOrigin.
date	The date part of the timestamp as a Blaze Advisor built-in date value.
time	The time part of the timestamp as a Blaze Advisor built-in time value. The time-zone and daylight-savings-time information is preserved.

Duration Properties

The properties in the Blaze Advisor duration data type are **read-only**. They behave like the other built-in data types (`boolean`, `integer`, `real`, `string`). You can use the operators to produce new values, but not to modify values. The duration data type has these properties:

Property	Description
months	This property combines the number of month units and the number of year units (1 year = 12 months).
days	This property is the number of day units without any attempt to convert month or year units into days.
milliseconds	The number of milliseconds obtained by combining the number of hour, minute, second, and millisecond units (1 hour = 60 minutes, 1 minute = 60 seconds, 1 second = 1000 milliseconds), without any attempt to convert day, month or year units into milliseconds.
extraNanoseconds	To get the duration with full precision, the number of nanoseconds (in the [0,999999] range) to add to milliseconds.

Initializing a Date, Time, Timestamp, or Duration

You initialize a date, time, timestamp with:

- A string literal, enclosed by single quotation marks
- A return value from one of the factory methods of the `NdCalendar` support class
- An assignment of an existing value of a compatible type

You initialize a duration with:

- An SRL expression that includes an integer value and a unit of time
`daysPastDue` is a duration initially 3 days
 You do not need single quotation marks to initialize a duration variable
- A return value from a method that returns a duration.

Input Formats for Date, Time, Timestamp, and Duration

This section describes the input formats for the various data types. You can use these input formats to initialize a data type or use it as a constant.

Date

You initialize a date with a string literal using one of these forms:

```
'dd-MMM-yyyy'  
'default date format for the locale'
```

The first date format does not depend on the current locale setting and assumes the US locale. The second date format depends on the current locale setting. If you use it, your project may not compile everywhere.

Time

You initialize a time with a string literal using one of these forms:

```
'HH:mm:ss z' // 24-hour clock  
'HH:mm:ss aa z' // 12-hour clock. Specify am/pm.  
'HH:mm:ss'  
'default time format for the locale'
```

where:

- z is the three-character code for a time zone
- default time format for the locale depends on the current locale setting

The first two time formats do not depend on the current locale setting and assume the US locale. The third time format depends on the current locale setting. If you use the third format, your project may not compile everywhere. If you do not specify the time zone, the default time zone of the computer is used.

Timestamp

You initialize a timestamp with a string literal using one of these forms:

```
'dd-MMM-yyyy HH:mm:ss z'  
'dd-MMM-yyyy HH:mm:ss'  
'default timestamp format for the locale'
```

where:

- z is the three-character code for a time zone
- default timestamp format for the locale depends on the current locale setting

The first two timestamp formats do not depend on the current locale setting and assume the US locale. The third timestamp format depends on the current locale setting. If you use the third format, your project may not compile everywhere. If you do not specify the time zone, the default time zone of the computer is used.

 **Note** The default formats are defined by the `java.text.DateFormat` class. See your JDK documentation for additional details.

Duration

You initialize a duration by using this SRL syntax:

```
(integer_expression) unit
```

where these are the valid unit names:

- year[s]
- month[s]
- day[s]
- hour[s]
- minute[s]
- second[s]
- millisecond[s]
- microsecond[s]
- nanosecond[s]

For example:

```
effectiveDuration is a duration
initially 4678 milliseconds.
```

Data Type Conversions for Date, Time, Timestamp, and Duration Values

Various type conversions are implicitly performed to give you maximum flexibility in writing SRL statements. Additionally, you can convert a date, time, timestamp, or duration value to a string by simple concatenation.

You can use the types shown in the table interchangeably in assignment statements, and method calls.

Implicit casts for calculations and comparisons are not performed. For example, you cannot add a duration to a Date property value without explicitly casting the Date to a Blaze Advisor date.

You perform computations or comparisons by including the cast. For example:

```
myObject.javaDateProperty as a date + 2 days
```

These conversions are performed implicitly:

From Type	To Type
java.util.Date java.sql.Date	Blaze Advisor date
Blaze Advisor date	java.util.Date java.sql.Date

From Type	To Type
java.sql.Time	Blaze Advisor time
Blaze Advisor time	java.sql.Time
java.util.Date java.sql.Timestamp	Blaze Advisor timestamp
Blaze Advisor timestamp	java.util.Date java.sql.Timestamp

Formatting Output Strings for Date, Time, Timestamp, and Duration

You convert a date, time, timestamp, or duration value to a string by simple concatenation. The value is formatted using the default locale format.

If you want to customize how the string is formatted, you can set the values of the `defaultTimestampFormat`, `defaultDateFormat`, and `defaultTimeFormat` properties of the default NdCalendar object for the project.

Money Data Type

The money data type is considered a primitive type. However, there are some additional considerations you should be aware of when working with it.

Money values include:

- The amount of money
- The currency (for example, dollar, euro, yen)
- The number of decimal digits

The currency included in a money value contains this information:

- The currency name as ISO 4217 defines it. You use a three-letter code to identify the currency. For example:
 - USD for the United States dollar
 - EUR for the European Union euro
 - JPY for the Japanese yenUse uppercase letters for the three-letter currency codes.
- The number of decimal digits to get the smallest currency unit (usually two).
- Euro conversion rates for currencies that belong to the euro area. These rates were fixed on January 1, 1999.
- Global conversion rates. These rates are variable. You must set them before attempting any conversion.

The support class `NdCurrencyManager`, keeps track of all the currencies for your project and provides methods for specifying formats and controlling computations.

Creating a Money Type

This section describes how to declare and initialize a money type.

Declaring a Money Type Variable

You declare a money variable the same way you declare an integer, real, or string variable:

```
variableName is a money.
```

For example:

```
purchasePrice is a money initially $5000.00.
```

 **Note** You use a instead of some in declaring money variables. This is because the money type is a primitive data type, and not a class.

You declare a currency variable the same way you declare variables of classes:

```
variableName is some NdCurrency.
```

For example:

```
someCurrency is some NdCurrency.
```

Declaring a Money Property

You can declare a money property using Structured Rule Language (SRL).

```
my_account is a Account with {
    a acctNumber : an integer,
    a acctName : a string,
    a balance: a money.
```

Properties

You can access the properties of money with the usual property notation. For example, you access the property currency of a money called price in this way:

```
price.currency
```

You can obtain these properties:

Property	Description
value	The amount of money as a real
decimalPrecision	The number of decimal digits as an integer
currency	The currency, an instance of the NdCurrency class
currencyManager	The currency manager, an instance of the NdCurrencyManager class

All these properties are **read-only**. They behave like the other built-in data types (`boolean`, `integer`, `real`, `string`). You can use the built-in operators to produce new values, but not to modify values. For example, if you declare and initialize the variable `price`:

```
price is a money initially 2.50 USD.
```

then it will have these properties:

- `price.value` is 2.5.
- `price.decimalPrecision` is 2.
- `price.currency` is USD.

Initializing a Money Value

You initialize a money type or use a money type as a constant using one of these formats:

amount currency_name

Where:

- `amount` is either a decimal number or an integer.
- `currency_name` is a three-letter currency code defined by the ISO 4217 standard (USD, EUR, JPY). The three-letter currency code must be all uppercase.

Example of initializations:

```
purchasePrice is a money initially 432.57 USD.  
basePrice is a money initially 1200 JPY.
```

Example of constants:

```
1 USD  
2.57 EUR
```

currency_sign amount

Where:

- `currency_sign` is one of these currency signs:
 - `$` for USD
 - `£` for GBP
 - `¥` for JPY
 - `€` for EUR
- `amount` is either a decimal number or an integer.

Example of initializations:

```
purchasePrice is a money initially $432.57.  
basePrice is a money initially ¥1200.
```

Example of constants:

```
$1  
£23
```

amount currency_sign

Where:

amount is either a decimal number or an integer.

currency_sign is one of these currency signs:

- \$ for USD
- £ for GBP
- ¥ for JPY
- € for EUR

Example of initializations:

```
purchasePrice is a money initially 432.57 $.  
basePrice is a money initially 1200 ¥.
```

Example of constants:

```
1 $  
23 £
```

Converting Money Values

A built-in function is provided to return specific money formats and to perform various type conversions. Additionally, you can convert a money value to a string by using simple concatenation.

Currency Conversions

You can convert money values to a different currency by using the built-in function `convertedTo()` along with support class functions for setting exchange rates. `convertedTo()` is a function of `NdMoneyAmount`, the implementation class for the money type.

convertedTo(currency, precision)

Where:

- currency is the destination currency (an instance of `NdCurrency`).
- precision is the decimal precision of the resulting value (an integer).

This form returns the money value obtained by converting to the specified currency.

convertedTo(currency)

Where currency is the destination currency (an instance of `NdCurrency`).

This form also returns the money value obtained by converting to the specified currency. However, it uses the default decimal precision for the destination currency.

 **Note** Before attempting any currency conversions, be sure to set the exchangeRate properties of both the original NdCurrency and the target NdCurrency to the current exchange rates.

Various type conversions are implicitly performed to give you maximum flexibility in writing your SRL statements. The types shown in the table can be used interchangeably in assignment statements, method calls, or other SRL expressions.

The defaultMappingCurrency property of your project's default NdCurrencyManager instance is used when converting from a `java.math.BigDecimal` or `java.math.BigInteger` to a money.

The default mapping currency is USD. You can assign any currency to be the default mapping currency. For example, this code snippet sets the default mapping currency to be the Japanese yen:

```
currencies().defaultMappingCurrency = currencies().yen.
```

These conversions are implicitly performed:

From Type	To Type
Advisor money	<code>java.math.BigDecimal</code>
<code>java.math.BigDecimal</code>	Advisor money
<code>java.math.BigDecimal</code>	Advisor real
<code>java.math.BigInteger</code>	Advisor integer

Setting the Rounding Behavior for Money Values

Each money value has a specified precision. Under certain circumstances, such as when you perform calculations with money values, the resulting money value will be rounded if it has a precision greater than any one of the values used in the calculation.

For example depending on the rounding mode, the result of an expression such as 1.33 USD multiplied by 1.33, would be either 1.77 USD or 1.76 USD.

You can set the rounding mode in your rule project like this:

```
cur is some NdCurrencyManager initially currencies( ).  
cur.roundingMode = NdCurrencyManager.ROUND_HALF_UP.
```

The default setting of the roundingMode property for the NdCurrencyManager class is ROUND_HALF_EVEN, otherwise known as "Banker's rounding." By using this rounding mode, .5 is rounded either up or down to the nearest even number. For example, the money value 4.455 USD would be rounded up to 4.46 USD and the value 3.425 USD would be rounded down to 3.42 USD. By using this rounding mode the amount lost or gained from rounding is cancelled out on average.

NdCurrencyManager defines seven rounding modes:

Rounding Mode	Behavior
ROUND_UP	Rounds the discarded amount "up", or away from zero. If the calculated value is positive, the value will be increased after rounding; if the calculated value is negative, the value will be decreased after rounding.
ROUND_DOWN	Rounds the discarded amount "down", or towards zero. If the calculated value is positive, the value will be decreased after rounding; if the calculated value is negative, the value will be increased after rounding.
ROUND_CEILING	If the value is positive, it behaves the same as ROUND_UP; if the value is negative, it behaves the same as ROUND_DOWN. Note that this rounding mode never decreases the calculated value.
ROUND_FLOOR	If the value is positive, it behaves the same as ROUND_DOWN; if the value is negative, it behaves the same as ROUND_UP. Note that this rounding mode never increases the calculated value.
ROUND_HALF_EVEN	(Default rounding mode) Rounds up if the amount to be rounded is greater than .5, rounds down if the amount to be rounded is less than .5, and rounds up or down depending on which is even if the amount to be rounded is equal to .5
ROUND_HALF_UP	Rounds up if the amount to be rounded is greater than or equal to .5 and rounds down if the amount to be rounded is less than .5
ROUND_HALF_DOWN	Rounds up if the amount to be rounded is greater than .5 and rounds down if the amount to be rounded is less than or equal to .5

Setting Decimal Precision for Money Values

The number of decimal digits that you use to initialize a money type is used to set the decimal precision. However, you can explicitly set the precision of a money value using the `preciseAt()` method:

```
preciseAt(precision)
```

where `precision` is the decimal precision of the resulting value.

For example:

```
(1 USD).preciseAt(3) results in 1.000 USD.  
(1.64 USD).preciseAt(0) results in 2 USD.
```

When you increase the precision of a money value, the value does not change. However, the results of future computations may change. For example, if you later divide or multiply the value, the final result may be different.

If the specified precision is less than the original precision, the value is rounded using the `roundingMode` of the `NdCurrencyManager` class.

Decimal precision does **not** affect comparison and equality tests. For example, 1 USD is equal to 1.00 USD.

Formatting Output Strings for Money Values

You can convert a money value to a string by simple concatenation. The money value is formatted with the number of decimal digits its decimal precision specifies. For example:

```
price is a money initially 2.50 USD.  
print("price: " price).
```

This is the output:

```
price: 2.50 USD
```

You can change the output format by setting the `formatMode` property of the default `NdCurrencyManager` for your project. For example:

```
// Change the default output format.  
currencies().formatMode = NdCurrencyManager.FORMAT_SYMBOL_RIGHT.
```

This table shows acceptable values for `formatMode` property and the output.

Defined Constant	Output Format	Example
FORMAT_LEFT	locale_code amount	USD 432.57
FORMAT_ISO_LEFT		
FORMAT_RIGHT	amount locale_code	432.57 USD
FORMAT_ISO_RIGHT		
FORMAT_SYMBOL	currency_symbol amount	\$432.57
FORMAT_SYMBOL_LEFT		
FORMAT_SYMBOL_RIGHT	amount currency_symbol	432.57\$
FORMAT_PARENTHESSES	(locale_code amount) Negative values are shown in parentheses.	(USD 432.57)
FORMAT_MINUS	locale_code -amount Negative values are shown with a minus (-) sign.	USD-432.57

Calculating and Comparing Money Values

You perform calculations and comparisons on money values by using:

- Structured Rule Language (SRL) operators
See [Built-in Operators](#) on page 327
- Methods in the support classes

You can use the built-in operators to manipulate money amounts. Note that strict typing rules are enforced.

This table lists the supported operations:

Operation	Result
money + money	money

Operation	Result
money - money	money
- money	money
integer * money	money
real * money	money
money / integer	money
money / real	money
money / money	real
money comparison_operator money	boolean

where comparison_operator can be any of these:

< <= > >= is <>

Collections

Creating a collection simplifies operations that involve a one-to-many relationship.

Suppose that you have a bank-customer business object. This object might have several different account objects (a checking account, a car-loan account, a credit-card account, and so on). In this instance, you could create an Accounts collection that contains all of the customer's accounts. You could then use the collection in your decision logic to do the following:

- Access or modify elements in the collection. For example:
Change the status of each account to "inactive."
- Access or modify individual elements in the collection. For example:
Change checking account to "preferred."
- Search through each element in the collection. For example:
If any account balance is negative, send a warning notice.

Two types of collections are supported:

- Arrays, which allow access to each element in the collection through an integer index value.

The index value refers to the position of the element in the array. The first element in any array is at index 0, the second is at index 1, the third at index 2, and so on.

Both fixed and dynamic arrays are supported. Fixed arrays have a specified number of elements, and dynamic arrays have any number of elements that can be added to or deleted from during processing.

- Associations, which allow access to each element in the collection through an associated key object or value.

The key is an identifier assigned directly to the element. It lets you access and modify individual elements through the key, without knowing their position in the collection. For example, you could associate a Stock Keeping Unit (SKU)

number with a product and then access the product in the collection using the SKU.

You can:

- Create collection objects as you would for objects of any other type.
- Declare class properties to be collections of any type.
- Declare variables to be collections of any type.
- Assign collection objects to properties or variables.

Collections are type-safe. This means that the compiler checks whether the elements put into, or received from, the collection are of a type compatible with the definition of the collection.

 **Note** When using the **Rule Builder** to author decision logic, you can use arrays, but not associations. To work with associations, use the **Advanced Builder** editor.

Arrays

An array can be of any primitive type or imported Java or XML class type. An array is created in an imported class and then it is assigned to a variable or a property of another object, or it is accessed through a pattern in a ruleset. Access an element in an array by referencing its position in the array with an index value. The index values for an array are always zero through n, with n being one less than the total number of elements in the array. Arrays are used the same way Java arrays are used.

Create a fixed array when there is a fixed number of elements in the array. Create a dynamic array when you want to increase or decrease the number of elements in the array by inserting, appending, or deleting them.

Properties and Methods for Arrays	Description
(Read-only) integer count	Number of elements in the array.
boolean contains(element type)	Returns true if the passed element is in the array.
boolean isContained[element type]	Indexed property of boolean values. The elements in the collection serve as the index value. If the element is contained in the collection, the value at index element is true. Note This method is available only when authoring decision logic in Structured Rule Language (SRL) in the Advanced Builder editor.
integer getFirstElementIndex(element type)	Returns the index of the first occurrence of the passed element in the array. Note This method is available only when authoring decision logic in SRL in the Advanced Builder editor.

Properties and Methods for Arrays	Description
integer getLastElementIndex(element type)	Returns the index of the last occurrence of the passed element in the array. Note This method is available only when authoring decision logic in SRL in the Advanced Builder editor.

Additional Methods for Dynamic Arrays	Description
void insert(integer index, element type)	Inserts the passed element in the array at the specified index. The index can correspond to the last known index + 1.
void append(element type)	Appends the passed element at the end of the array.
void remove(integer index)	Removes the element at the specified index.
void clear()	Removes all elements from the array.

Set operations properties (language constructs) are available for fixed or dynamic arrays so you can find the minimum, maximum, average, or sum of a set of values.

Arrays in the Rule Builder

Variables, objects, and object properties of type fixed and dynamic arrays are available from the Item Selector in the **Rule Builder**. Note that support for multi-dimensional and jagged arrays is not available. To write decision logic using those types of arrays, use the **Advanced Builder** editor.

You can perform these types of operations on object that is an array of primitive type or a complex type:

- Drag and drop a dynamic array object and use it to create a new dynamic array.

```
vocationalSchoolApplication.TranscriptDetails = a new array of  
TranscriptDetails
```

- Drag and drop a fixed array object and use it to create a fixed array. The default number is 1 so replace the number with the desired array size. Note that there is no validation in the Rule Builder to ensure that the number of elements in the array does not exceed the specified size.

```
vocationalSchoolApplication.TranscriptDetails = a new fixed array of 1  
TranscriptDetails
```

- Drag and drop an array object and assign a special value, such as unknown or unavailable, or perform a value comparison.

```
vocationalSchoolApplication.TranscriptDetails[] = unknown
```

To perform an operation on an array property, expand the array object, expand **\$element**, drag and drop an array property on to the canvas. These are some of the operations that you can perform:

- Assign a value for the array property or compare it to a value.
`communityCollegeApplication.TranscriptDetails[0].gradePointAverage = 3.96.`
- Assign the array property equal to another array property or compare them:
`communityCollegeApplication.TranscriptDetails[0].gradePointAverage = vocationalSchoolApplication.TranscriptDetails[1].gradePointAverage`
- Assign the array property to a special value or compare it to a special value.
`communityCollegeApplication.TranscriptDetails[0].gradePointAverage = unavailable`
- Assign the array property to the return value of a method or compare it to the return value of a method.
`communityCollegeApplication.TranscriptDetails[0].gradePointAverage = computeGradePointAverage(real)`

Implicit conversions are supported when assigning an array to a subclass of the same array. For example, `array of account = array of bank account` where `array of bank account` is a subclass of an `array of bank account`.

Array Statements

If there is at least one element in the array, you can use built-in properties and methods to perform operations on the array using the **Rule Builder** or the **Advanced Builder** editor.

Examples of Array Statements

- Change the value of elements in the array.
`ListOfAccidents[0] = FirstCrash.`
- Insert elements into the array.
`ListOfAccidents.insert(0, NewCrash).`
- Append elements to the end of the array.
`ListOfAccidents.append(NewCrash).`
- Access individual elements directly through their integer index.
`TruckCrash = ListOfAccidents[0].`
- Determine the number of elements the array contains.
`Crashes = ListOfAccidents.count.`
- Determine if the array contains a specific element.
`if ListOfAccidents.contains(CarCrash) then...`
- Find the index of the first occurrence of an element in the array.
`FirstAccidentIndex =
ListOfAccidents.getFirstElementIndex(CarCrash).`
- Find the index of the last occurrence of an element in the array.
`LastAccidentIndex =
ListOfAccidents.getLastElementIndex(CarCrash).`

- Remove elements from the array.
`ListOfAccidents.remove(0).`
- Remove **all** elements from the array.
`ListOfAccidents.clear().`
- Iterate, or loop, through all of the elements of the array. (To iterate through the elements in array, use the **Advanced Builder** editor.)

```
for each accident in ListOfAccidents do {
    print("Amount of claim = " it.claimAmount) }
```

Set Operations

You can find the minimum, maximum, average, or sum of a set of values or objects using set operations properties. These (read-only) properties are language constructs that operate on a fixed or dynamic array of primitives or primitive properties of objects. The class type of the array can be a Java or XML imported class type.

Set operations are used to do the following:

- Operate dynamically against defined sets of data.
This can be a fixed or dynamic array of primitives or objects.
- Allow more sophisticated operational systems that reason over arrays.
- Trigger rules based on collective results.

You can compute the following:

- A set of values
`someArray.intProp.sum`
- An attribute of some set of objects
`someArray[every Class].intProp.sum`
- An attribute of some attribute of some set of objects
`someArray[every Class].subObject.intProp.sum`
- An attribute of some attribute of one or more sets of objects
`someArray[every Class].subObject1.subObject2.subObject3.intProp.max`

Set operations properties become available when the type of an object is set as a fixed or dynamic array of objects or primitives.

For an array of primitives, there are four set operations properties that are available, depending upon the data type of a property:

- min (minimum)
- max (maximum)
- avg (average)
- sum

This table shows the set properties that are available for each data type.

Data Type	Min	Max	Avg	Sum
boolean	N/A	N/A	N/A	N/A
date	Yes	Yes	Yes	N/A
duration	Yes	Yes	Yes	Yes
integer	Yes	Yes	Yes	Yes
money	Yes	Yes	Yes	Yes
real	Yes	Yes	Yes	Yes
string	Yes	Yes	N/A	N/A
time	Yes	Yes	Yes	N/A
timestamp	Yes	Yes	Yes	N/A

For an array of objects, there are set operations properties associated with each supported type of property in the array. For example, there are set operations properties for string properties, but not for boolean properties.

This is the format that is used for set operations properties in an array of objects:

[every Item].property1.avg: real

This is a breakdown of each part of the format:

- every is an SRL keyword that is used with set operations properties.
- Item is the class name.
- property1 is the name of the property that you want to operate on.
- avg is the set operation property (it could also be min, max, or sum depending upon the data type).
- real is the data type in this example.

All set operations properties start with the keyword “every”. Value properties are supported, which can be used in set operation properties references, such as array.\$value.max and array[every class].\$value.property1.max.

You can write logic with set operations properties in the **Rule Builder** or in the **Advanced Builder** using Structured Rule Language (SRL).

Set Operations Properties in the Rule Builder

In the Rule Builder, you can drag and drop set operations properties to use in expressions, which avoids the need to learn the semantic requirements for using set operations properties in Structured Rule Language (SRL).

Set operations properties are available in the Item Selector in the **Rule Builder** under the array nodes. To see the set operations properties, expand the array node, and then expand \$element, as shown in this figure:

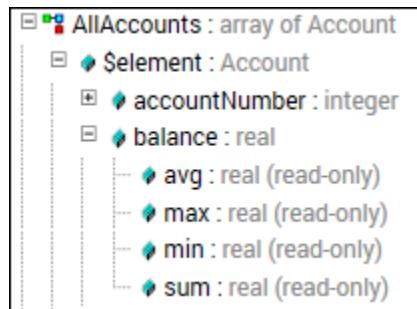


Figure 57: Accessing Set Operations Properties

To drag and drop set operations properties to use in rule conditions, locate the array object property, and then drag and drop the set operations property. The syntax for the expression appears in the canvas. All you need to do is to select an operator and enter the value to be compared.

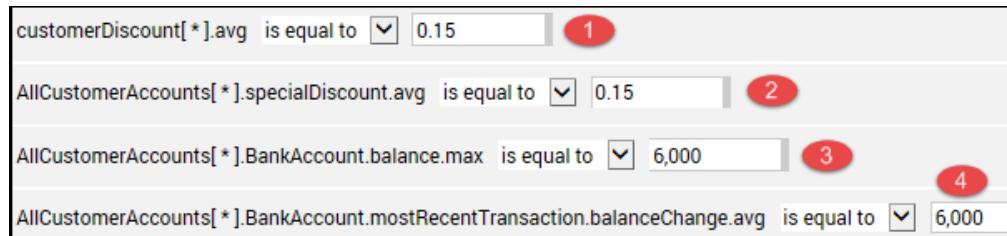


Figure 58: Examples of using set operations properties in condition expressions

Example	Description
1	This expression computes the average for the array elements and checks to see if the average is equal to .15.
2	This expression computes the average for the specialDiscount property in a set of objects to see if the average is equal to .15.
3	This expression computes the maximum value for the balance property in a set of objects to see if the maximum is equal to 6000. In this case, BankAccount is a property of type BankAccount and it has a property called balance.
4	This expression computes the average for the balanceChange property in a set of objects to see if the average is equal to 6000. In this case, BankAccount has a property called mostRecentTransaction that is of type Transaction and it has a property called balanceChange.

In statements, set operations properties can be used on the right-hand side of equations and as arguments in methods and functions.

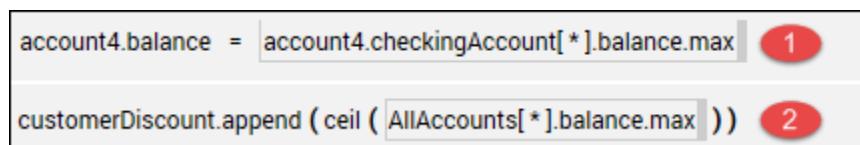


Figure 59: Examples of using set operations properties in statements

Example	Description
1	This statement shows a set operations property used on the right-hand side of an equation.
2	This statement shows a set operations property used in a built-in function.

Set Operations Properties in Structured Rule Language

In Structured Rule Language (SRL), there are several forms of expressions that are used to operate on a array of primitives or an array of objects.

Additionally, there are three expression formats that you can use to operate on an array of primitives:

- arrayExpression.max
- the max of arrayExpression
- arrayExpression's max

While you can mix formats, FICO recommends that you choose one as a standard for your project.

This is an example of rule written using each format:

- if airlineTickets.max is greater than 2350
then set the status of applicant to "Good customer".
- if the max of airlineTickets is greater than 2350
then set the status of applicant to "Good customer".
- if airlineTickets's max is greater than 2350
then set the status of applicant to "Good customer".

There are three expression formats that you can use to operate on an array of objects:

- arrayExpression[every class].property.max
- the max of the property of every class in arrayExpression
- arrayExpression's every class's property's max

This is an example of rule written using each format:

- if AllAccounts[every Account].balance.max > 5000
then set the status of customerAccount to "Offer Card Promotion".
- if the max of the balance of every Account in AllAccounts > 5000
then set the status of customerAccount to "Offer Card Promotion".
- if AllAccounts's every Account's balance's max > 5000
then set the status of customerAccount to "Offer Card Promotion".

Additionally, you can write rules on a property that is referenced from an array of objects. Here are two examples:

- if the sum of the balance of the savAcct of every account in AllAccounts > 20000
then print ("Sum is " the sum of the balance of the savAcct of every account in AllAccounts).
- if AllAccounts[every account]savAcct.balance.sum >20000
then print ("Sum is " AllAccounts[every Account]savAcct.balance.sum).

You can use set operations properties to write rules on specific objects. This is an example:

```
if theCarpool[3].dailyMileage.avg > 21
then print ("The sum of the daily mileages for RentalCars is "
theCarpool[3].dailyMileage.avg).
```

Semantic Requirements for Using Set Operations Properties in Structured Rule Language (SRL)

There is a list of semantic requirements for using set operations properties in Structured Rule Language (SRL).

- A class name or interface must follow the keyword every. If .dot notation is used in the SRL, the keyword and class name must appear in brackets using this format: [every className]
- There can only be one [every className] in a set operation property reference.
- There must be an [every className] clause in a set operation property reference or the result property must be a direct member of an array of a primitive type. This is an example of using a direct member of an array of a primitive type: airlineTickets.max.
- A functional expression or an indexed expression cannot be used in the path of a set operation property reference. This is an example of a reference that is not supported: AllAccounts[every Account].accountOwners[1].max

For best results when using set operations properties, do not create properties in an array object that are called max, min, avg, or sum.

Associations

Associations are collections that map objects or values of one type to objects or values of another type. You access each element in the collection through an associated key. A key can be any primitive or imported class type. To author decision logic with an association, use Structured Rule Language (SRL) in the **Advanced Builder** editor.

In SRL, you declare associations using this syntax:

```
an objectName is an association from typeName to typeName
```

where typeName can be any primitive or imported class type.

You can store associations:

- In variables:
variableName is some association from typeName to typeName
- In properties of other objects:
propertyName : some association from typeName to typeName

Association Property	Description
count	A read-only integer that indicates the number of elements in the association.

Association Methods	Description
contains(element)	Returns true if the passed element is in the association.
remove(key)	Removes the element at the specified key.
clear()	Removes all elements from the association.

Unlike arrays, associations do not support the insert() or append() methods. As a result, you can only populate an association with elements by using direct assignment.

Association Statements

Use associations when you need to provide a convenient, direct connection between a known value in one group of elements and the corresponding element in another group.

For example, a homeowner's insurance policy may include zero or more riders that extend the insurance coverage for special items such as jewelry or computer equipment. You can use an association to represent the collection of policy riders indexed by the type of items they cover.

There are built-in properties and methods that enable you to do the following operations:

- Declare an association from a primitive type to a class type (such as `rider` in this code sample).
`ListOfRiders` is some association from `string` to `rider`.
- Specify the value of elements in the association.
`ListOfRiders["Jewelry"] = a rider.`
- Access individual elements directly through their key.
`JewelryRider = ListOfRiders["Jewelry"].`
- Determine the number of elements the association contains.
`NumberOfRiders = ListOfRiders.count.`
- Determine if the association contains a specific element.
`if ListOfRiders.contains(JewelryRider) then...`
- Remove elements from the association.
`ListOfRiders.remove("Jewelry").`
- Remove all elements from the association.
`ListOfRiders.clear().`

Example of creating an association:

```
associationName is an association from type to type initially {  
    it[keyValue] = initialValue.  
}
```

For example:

```
Products is an association from integer to Product initially {
    it[094] = someProduct.
}
```

Dynamic Objects

Dynamic objects are instances of external classes defined in functions, rule actions, and initialization statements. Create object instances to hold known values, intermediate results of rule processing, or as test objects while authoring rules.

Dynamic objects are created or mapped into the project during rule processing. Create a dynamic object by using a variable or property assignment statement using the assignment (=) operator or by calling a `newInstance()` method of a class.

You can map dynamic objects as a result of calling a method that returns an object or as property values of objects that you explicitly map. Dynamic objects are available to the rules as long as there is a reference to them.

An object has a reference to it if one of the following applies:

- It is assigned to a variable.
- It is assigned to the property of another object.
- It is stored in a collection.

Use dynamic objects when one of the following applies:

- You do not know how many objects to create until rule processing occurs, and you do not have to save the objects after processing. For example, you might need to create a temporary account object on the basis of information that a user provides.
- You need to create an object to process information gathered during rule processing. For example, you might need to create an object if the user chooses an out-of-stock item and you want the rules to suggest another alternative to the customer.

Create a dynamic object using one of the following options:

- In the Rule Builder for an initialization rule or rule action, drag and drop a class-type property, variable, or pattern to the canvas. When an assignment statement is created, a new dynamic object is created and assigned to the class-type property, variable or pattern. A dynamic object of an external class type can be created only if the external class has a zero argument constructor.
- In the Advanced Builder editor, instantiate the object and then assign it to an *object property* using Structured Rule Language (SRL).

```
objectName.objectPropertyName = a className initially
    { property1=value, property2=value, ... }
```

- In the Advanced Builder editor, instantiate the object and then assign it to a *variable* using Structured Rule Language (SRL).

```
variableName = a className initially  
    { property1=value, property2=value, ... }
```

- In the Advanced Builder editor, you can use the `newInstance()` function to call a constructor to create an *object of an external class using arguments* that you specify. Then assign it to an object property or variable where the arguments must be of the same type the external class constructor specifies.

```
objectName.objectPropertyName = a className initially  
    extClassName.newInstance(argument1, argument2, ...)
```

The `newInstance()` function lets you instantiate objects with arguments if the class has no zero-argument constructors. You can call this function with the appropriate arguments as part of an initialization rule or action statement. For example:

```
newProduct=product.newInstance(size, name)
```

When a local object is declared in a ruleset or a function, only statements and expressions within that decision entity can access or change the value of that object. In a ruleset, to create a reference to an object without an object reference, declare a pattern or variable of its type, and use it to write your rules.

Related Links

[Rule Builder Interface](#)

[Authoring Decision Logic Using SRL](#)

Math Built-ins (NdMathBuiltIns)

The math built-in functions allow you to perform common numeric operations in the SRL.

 **Note** Due to differences in the ways that various hardware and software environments process floating point numbers, the exact result returned by some of the functions may differ slightly from platform to platform.

This section lists all of the math built-in functions. For a specific function description and implementation, click the Function name.

Function	Description
math().abs() on page 356	Returns absolute value of a number.
math().arctan() on page 357	Returns the arc tangent of an angle.
math().ceil() on page 353	Returns an integer that is closest or greater than a real number.
math().cos() on page 357	Returns the cosine of a real number.
math().exp() on page 358	Returns e raised to a number.
math().floor() on page 353	Returns an integer that is closest or smaller than a real number.
math().log() on page 358	Returns the natural logarithm of a number.

Function	Description
math().max() on page 360	Compares two numbers and returns the greater value of the two.
math().min() on page 361	Compares two numbers and returns the lesser value of the two.
math().mod() on page 359	Returns the remainder of an integer division.
math().power() on page 359	Returns the value of the first number raised to the power of the second number.
math().round() on page 354	Reduces the digits in a number while trying to keep its value similar.
math().sin() on page 359	Returns the trigonometric sine of a number.
math().tanh() on page 360	Returns the hyperbolic tangent of a number.
math().truncate() on page 355	Reduces the number of digits to the right of the decimal point in a number.

Converting Values

These built-in functions can be used to convert values.

math().ceil()

Description

This function converts a real number to the closest greater or equal integer.

Implementation

```
math().ceil(real):integer
```

SRL Example

```
real1 is a real initially 89.75.  
integer1 is an integer.  
integer1 = math().ceil(real1).  
print("This is the answer: "integer1).
```

The result is that integer 1 is 90.

math().floor()

Description

This function converts a real to the closest smaller or equal integer.

Implementation

```
math().floor(real):integer
```

SRL Example

```
real1 is a real.  
real1 is 89.75.  
integer1 is an integer.  
integer1 = math().floor(real1).  
print("This is the answer: "integer1).
```

The result is that integer1 is 89.

math().round()

Description

This function converts a real or integer value to the closest number.

Implementation

There are three signatures:

```
math().round(real):integer
```

It rounds a number to the closest integer value.

```
math().round(real, integer):real
```

It rounds the first number to the closest integer value based on the value of the second integer. The second integer must be in the range [0, 9] or an exception is thrown.

```
math().round(integer, integer):integer
```

It rounds the first number to the closest real value based on the value of the integer. The integer must be in the range [-14, +14] or an exception is thrown.

SRL Examples

```
real1 is a real initially 48243.23.  
integer1 is an integer.  
integer1 = math().round(real1).  
print("This is the answer: "integer1).
```

The result is 48243.

```
real1 is a real initially 104.3000090.  
integer1 is an integer initially -5.  
integer2 is a real.  
integer2 = math().round(real1, integer1).  
print("This is the answer: "integer2).
```

The result is 104.30001.

```
integer1 is an integer initially 104.
integer2 is an integer initially 2.
integer3 is an integer.
integer3 = math().round(integer1, integer2).
print("This is the answer: "integer3).
```

The result is 100.

Value to be rounded (First Parameter)	Position (Second parameter)	Return Value
3.14159	-5	3.14159
3.14159	-4	3.14160
3.14159	-3	3.14200
3.14159	-2	3.14000
3.14159	-1	3.10000
3.14159	0	3.00000
31415.9	1	31420.0
31415.9	2	31400.0
31415.9	3	31000.0

math().truncate()

Description

This function truncates a number by reducing the number of digits to the right of the decimal point.

Implementation

There are three signatures:

```
math().truncate(real):integer
```

It truncates a number to the closest integer value.

```
math().truncate(integer, integer): integer
```

It rounds the first number to the closest integer value based on the value of the second integer. The second integer must be in the range [0, 9] or an exception is thrown.

```
math().truncate(real, integer): real
```

It rounds the first number to the closest real value based on the value of the integer. The integer must be in the range [-14, +14] or an exception is thrown.

SRL Example

```
real1 is a real.  
real1 = math().truncate(3.1459).  
print("This is the answer: "real1).
```

The result is that real1 is 3.0.

```
integer1 is an integer.  
integer1 = math().truncate(314159, 3).  
print("This is the answer: "integer1).
```

The result is that real1 is 314000 because 314159 was truncated by 3 positions.

```
real1 is a real.  
real1 = math().truncate(3.14159, 3).  
print("This is the answer: "real1).
```

The result is that real1 is 0.0 because 3.14159 was truncated by 3 positions.

Number to Truncate	Position to Truncate	Result
3.14159	-5	3.14159
3.14159	-4	3.1415
3.14159	-3	3.141
3.14159	-2	3.14
3.14159	-1	3.1
3.14159	0	3
314159	1	314150
314159	2	314100
314159	3	314000
314159	4	31000
314159	5	300000
314159	6	0

Calculating and Returning Values

These built-in functions can be used to calculate and return values.

math().abs()

Description

This function calculates and returns the absolute value of an integer or real number.

Implementation

```
math().abs(integer):integer
math().abs(real):real
```

SRL Example

```
string1 is a string initially "frequent".
real1 is a real.
if string1 = "frequent"
then real1 = math().abs(-1000.45).
print("This is the answer: "real1).
```

The result is that real1 equals 1000.45.

math().arctan()

Description

This function returns the arc tangent of an angle. It is the inverse of [math\(\).tanh\(\)](#) on page 360.

Implementation

```
math().arctan(real):integer
```

SRL Example

```
real1 is a real initially 89.75.
real2 is a real.
real2 = math().arctan(real1).
print("This is the answer: "real2).
```

The result is that real2 equals 1.559654726558251.

math().cos()

Description

This function returns the cosine of a real number.

Implementation

```
math().cos(real):real
```

SRL Example

```
real1 is a real initially 0.10500.
real2 is a real.
real2 = math().cos(real1).
print("This is the answer: "real2).
```

The result is that real2 is 0.9944925627484974

math().exp()

Description

This function calculates and returns the mathematical constant e raised to a power. The constant value of e is 2.7182818 which is the base of the natural logarithm.

Implementation

```
math().exp(int):real  
math().exp(real):real
```

SRL Example

```
real1 is a real.  
real1 = math().exp(12.56).  
print("This is the answer: "real1).
```

The result is that real1 is 284930.3376286781

math().log()

Description

This function calculates the natural logarithm of a positive number using a base of e (2.7182818). If the parameter to this function is zero or a negative number, the following results occur:

- When the parameter is zero, *-infinity* is returned
- When the parameter is a negative number, *NaN* (Not a Number) is returned

Implementation

```
math().log(integer): real  
math().log(real):real
```

SRL Example

```
integer1 is an integer initially 11.  
real1 is a real.  
real1 = math().log(integer1).  
print("This is the answer: "real1).
```

The result is that real1 is 2.3978952727983707

math().mod()

Description

This function calculates and returns the remainder from an integer division. (Mod is short for Modulo.)

Implementation

`math().mod(integer, integer):integer`

If the divisor (second integer) is set to zero, a runtime error occurs.

SRL Example

```
integer1 is an integer initially 21.
integer2 is an integer initially 4.
integer3 is an integer.
integer3 = math().mod(integer1, integer2).
print("This is the answer: "integer3).
```

The result is that integer3 contains 1 which is the remainder after the calculation of 21/4.

math().power()

Description

This function returns the value of the first number raised to the power of the second number.

Implementation

`math().power(real, real):real`

SRL Example

```
real1 is a real initially 4.
real2 is a real initially 3.
real3 is a real.
real3 = math().power(real1, real2).
print("This is the answer: "real3).
```

The result is that real3 is 64.0.

math().sin()

Description

This function computes and returns the trigonometric sine of a real number.

Implementation

`math().sin(real):real`

SRL Example

```
real1 is a real initially 0.10500.  
real2 is a real.  
real2 = math().sin(real1).  
print("This is the answer: "real2).
```

The result is 0.10480716882888248.

math().tanh()

Description

This function computes and returns the hyperbolic tangent of a real number.

Implementation

`math().tanh(real):real`

The parameter is the real number for which you want to compute the hyperbolic tangent.

SRL Example

```
real1 is a real initially 2.5.  
real2 is a real.  
real2 = math().tanh(real1).  
print("This is the answer: "real2).
```

The result is that real2 is 0.9866142981514304.

Comparing Values

These built-in functions can be used to compare values.

math().max()

Description

This function compares two numbers and returns the value of the one that is greater.

Implementation

`math().max(integer, integer):integer`

`math().max(real, real):real`

SRL Example

```
integer1 is an integer initially 125.
integer2 is an integer initially 123.
integer3 is an integer.
integer3 = math().max(integer1, integer2).
print("This is the answer: "integer3).
```

The result is that integer3 is 125 because it is the greater of the two values.

 **Note** If you want to do a comparison on strings, use portable().max().

Related Links

[portable\(\).max\(\)](#)

math().min()

Description

This function compares two numbers and returns the value of the one that is smaller.

Implementation

```
math().min(integer, integer):integer
math().min(real, real):integer
```

SRL Example

```
integer1 is an integer initially 125.
integer2 is an integer initially 123.
integer3 is an integer.
integer3 = math().min(integer1, integer2).
print("This is the answer: "integer3).
```

The result is that integer3 is 123 because it is the smaller of the two values.

 **Note** If you want to do a comparison on strings, use portable().min().

Related Links

[portable\(\).min\(\)](#)

Date and Time Data Type Values

These data types and support classes are provided to let you efficiently use dates and times in your rules.

Date and Time Data Types

You can format the output for Date and Time data types using support classes.

Returning the Default Calendar

You obtain the default calendar for a project by calling the built-in `calendar()` function and accessing its properties and methods; for example:

```
//Initialize a date variable to the current date.  
today is a date initially calendar().currentDate().
```

Or as another example:

```
//Initialize an NdCalendar variable and then access the methods through the  
variable.  
cal is some NdCalendar initially calendar().  
//isLeap is a method of NdCalendar that returns a boolean.  
if cal.isLeap(2020) then  
print("This is a leap year."cal).
```

Setting Default Date, Time, and Timestamp Formats

[NdCalendar](#) on page 364 in contains the following three properties that let you set default formats for date, time, and timestamp data types:

- [defaultDateFormat](#)
- [defaultTimeFormat](#)
- [defaultTimestampFormat](#)

You can specify the output format for values by using a simple `java.text.SimpleDateFormat` pattern. The `SimpleDateFormat` class both formats and parses dates in a locale-sensitive manner; for example `dd-MM-yyyy`, which would return a date as `15-01-2020`.

Format = date to text, and *parse* = text to date.

By default, these properties are set to `null`, and the MEDIUM format for the current locale (short month, day, year) is used; for example, Jan 15, 2020.

See [Valid Default Format Patterns](#) on page 362 for a list of pattern string components.

Valid Default Format Patterns

`SimpleDateFormat` lets you specify date and time formats by using letters from A to Z and from a to z to form date and time pattern strings.

For more information on format strings, see [SimpleDateFormat](#)

defaultDateFormat

To set the default date format for a project, use the `defaultDateFormat` property.

SRL Example

```
//date variable initialized to the current date.  
currentDate is a date initially calendar().currentDate().  
  
//Print the variable using the default format.  
print("Original format: " currentDate).  
  
//Explicitly set the default date format.  
calendar().defaultDateFormat = "yyyy G, MMM dd".  
  
//Print the variable using the new format.  
print("New format: " currentDate).
```

Result

If the current date is April 9, 2020 and you are using the default US locale, then the result is (the original format depends on your current setting):

```
Original format: 09-Apr-2020  
New format: 2020 AD, Apr 09
```

defaultTimeFormat

To set the default time format for a project, use the `defaultTimeFormat` property.

SRL Example

```
//time variable initialized to the current time.  
currentTime is a time initially calendar().currentTime(true).  
  
//Print the variable using the default format.  
print("Original format: " currentTime).  
  
//Explicitly set the default time format.  
calendar().defaultTimeFormat = "HH:MM:ss:SSS ZZ".  
  
//Print the variable using the new format.  
print("New format: " defaultTime).
```

Result

If the current time is 2:30 p.m. and you are using the default US locale, then the result is (the original format depends on your current setting):

```
Original format: 2:30:03.832 PM PDT  
New format: 14:30:03.832 PDT -0700
```

 **Note** If you include seconds (ss) in the format string, milliseconds (SSS) are appended automatically.

defaultTimestampFormat

To set the default timestamp format for a project, use the `defaultTimestampFormat` property.

SRL Example

```
//timestamp variable initialized to the current timestamp.  
currentTimestamp is a timestamp initially calendar().currentTimestamp(true).  
  
//Print the variable using the default format.  
print("Original format: "currentTimestamp).  
  
//Explicitly set the default timestamp format.  
calendar().defaultTimeFormat = "MM.dd.yyyy G 'at' HH:mm:ss:SSS ZZ".  
  
//Print the variable using the new format.  
print("New format: "currentTimestamp).
```

Result

If the current date and time is April 16, 2020, 2:30 p.m. and you are using the default US locale, then the result is (the original format depends on your current setting):

```
Original format: Apr 16, 2020 5:28:36.799 PM PDT  
New format: 04.16.2020 AD at 17:28:36.799 PDT
```

 **Note** If you include seconds (ss) in the format string, milliseconds (SSS) are appended automatically.

Calendar, Date, Time, Timestamp and Duration Operations

These support classes provide useful properties and methods for handling date, time, timestamp duration, and calendar operations.

NdCalendar

NdCalendar provides properties and methods to support the Blaze Advisor date, duration, time, and timestamp data types. There are factory methods for generating new values for date and time types.

currentDate()

Description

The `currentDate()` method returns the current date of the host system.

Syntax

```
currentDate() : date
```

SRL Example

```
//Initialize a date variable to the current date.  
theDate is a date initially calendar().currentDate().  
print("This is the current date: "theDate).
```

Result

The current date on your machine.

currentTime()

Description

The `currentTime()` method returns the current time of the host system.

Syntax

```
currentTime(boolean withTimezone) : time
```

The argument `withTimezone` indicates whether or not to explicitly set the timezone of the returned time to the current default time zone for your locale. If this value is false, then returned time has no explicit timezone. It will always be interpreted as a time in the current default timezone even if that timezone changes after the time is created.

SRL Example

```
//Initialize a time variable to the current time.  
theTime is a time initially calendar().currentTime(true).  
print("This is the current time: "theTime).
```

Result

The current time on your machine.

currentTimestamp()

Description

The `currentTimestamp()` method returns the current timestamp of the host system.

Syntax

```
currentTimestamp(boolean withTimezone) : timestamp
```

The argument *withTimezone* indicates whether or not to explicitly set the timezone of the returned time to the current default time zone for your locale. If this value is true, then the time zone for the returned timestamp will be displayed if the timezone is converted to a string or printed.

SRL Example

```
//Initialize a timestamp variable to the current timestamp.  
theCurrentTS is a timestamp initially calendar().currentTimestamp(true).  
print("This is the current timestamp: "theCurrentTS).
```

Result

The result is that the current timestamp is printed.

date()

Description

The `date()` method returns a date with values obtained by parsing the `dateString`.

Syntax

```
date(string dateString) : date  
date(string dateString, string fmtString) : date  
date(string dateString, string fmtString, Locale Locale) : date
```

- *dateString* is the string to parse for day, month, and year values
The default format is dd-MMM-yyyy; for example, 15-Jan-2020.
- *fmtString* is the format string based on `java.text.SimpleDateFormat` directives
- *Locale* is an instance of `java.util.Locale` where `locale` specifies the location of the initial date value

If the format is not specified, the default formats are tried in sequence. An exception occurs if parsing fails with each format. If you do not specify a locale, the default becomes the locale specified in the project settings.

For information on valid default format patterns, see [Valid Default Format Patterns](#) on page 362.

Using Factory Methods

You do not need to call a factory method to make simple variable or property initializations. These two statements are equivalent:

```
effectiveDate is a date initially '20-mar-2020'.  
effectiveDate is a date initially calendar().date("20-mar-2020").
```

The first statement is more efficient because it is evaluated at compile time. The second statement is evaluated at runtime.

SRL Example

```
//calendar() is a builtin that returns the instance of NdCalendar associated
with the
session.
cal is some NdCalendar initially calendar().
d1 is a date initially cal.date("5 März 2020", "dd MMM yyyy", Locale.GERMAN).
print("d1 = "d1).
```

Result

```
d1 = 05-Mar-2020
```

indexedMonthWeekDay()

Description

The `indexedMonthWeekDay()` method returns a date for a specified year, month in the year, week in the month, and the day of the week.

Syntax

```
indexedMonthWeekDay(integer year, integer month, integer week,
integer day) : date
```

- The argument *year* is the year as an integer with 4 digits for years after 1900 and < 0 for BC years
- The argument *month* is the month in the year as an integer in the range 1->12
- The argument *week* is the week in the month (with 1 representing the first full week in the month) as an integer in the range 1->53
- *day* is the day in the week (locale specific) as an integer in the range 1->7

In this method, the first week of the month is the first *full* week that includes the day specified in the *day* argument.

For example, the following statement returns the date January 5, 2020:

```
indexedMonthWeekDay(2020, 1, 1, 1)
```

January 5 is the first day of the first full week of the month.

To return the week that contains the exact day specified in by the *day* argument, no matter which month contains the week, use [monthWeekDay\(\)](#) on page 369. In the example above, using `monthWeekDay()` would return December 29, 2019 instead of January 5, 2020.

SRL Example

```
//Initialize a date variable to the specified year, month, week, and day.
theDate is a date initially calendar().indexedMonthWeekDay(2020, 3, 4, 2).
print("theDate").
```

Result

The date is printed in your default format.

```
This is the date: 23-Mar-2020
```

indexedYearWeekDay()

Description

The `indexedYearWeekDay()` method returns a date for the specified year, week in the year, and the day in the week.

Syntax

```
indexedYearWeekDay(integer year, integer week, integer day) : date
```

- The argument `year` is the year as an integer with 4 digits for years after 1900 and < 0 for BC years
- The argument `week` is the week in the year (with 1 representing the first full week in the year) as an integer
- The argument `day` is the day in the week (locale specific) as an integer in the range 1->7

In this method, the first week of the month is the first *full* week that includes the day specified in the `day` argument.

For example, the following statement returns the date January 5, 2020:

```
indexedYearWeekDay(2020, 1, 1)
```

January 5 is the first day of the first full week of the month.

To return the week that contains the exact day specified in by the `day` argument, no matter which month contains the week, use [yearWeekDay\(\)](#) on page 371. In the example above, `yearWeekDay()` would return December 29, 2019 instead of January 5, 2020.

SRL Example

```
//Initialize a date variable to the specified year, month, and day.  
theDate is a date initially calendar().indexedYearWeekDay(2020, 12, 3).  
print("This is the date: "theDate).
```

Result

The date is printed in your default format:

```
This is the date: 24-Mar-2020
```

monthWeekDay()

Description

The `monthWeekDay()` method returns the date for a specified year, month in the year, week in the month, and day in the week.

Syntax

```
monthWeekDay(integer year, integer month, integer week, integer day) : date
```

- The argument *year* is the year as an integer with 4 digits for years after 1900 and < 0 for BC years
- The argument *month* is the month in the year as an integer in the range 1->12
- The argument *week* is the week in the month (with 1 representing the week in which the first of the month occurs) an integer
- The argument *day* is the day in the week (locale specific) as an integer in the range 1->7

In this method, the first week of the month is the week that includes the day specified in the *day* argument, even if it is a fragment of a week.

For example, the following statement returns the date December 29, 2019:

```
monthWeekDay(2020, 1, 1, 1)
```

December 29 is the first day of the week that contains the specified day, January 1.

To return the first week in the month that includes the weekday specified by the *day* argument, use [indexedMonthWeekDay\(\)](#) on page 367. In the example above, `indexedMonthWeekDay()` would return January 5, 2020 instead of December 29, 2019 because that is the first day of the first full week in the month.

SRL Example

```
//Initialize a date variable to the specified year, month, week, and day.  
theMWD is a date initially calendar().monthWeekDay(2020, 3, 4, 2).  
print("This is the date: "theMWD).
```

Result

```
This is the date: 23-Mar-2020
```

time()

Description

The `time()` method returns a time with values obtained by parsing the `timeString`.

Syntax

```
time(string timeString) : time
time(string timeString, string fmtString) : time
time(string timeString, string fmtString, Locale Locale) : time
```

- *timeString* is the string to parse for hour, minute, second, millisecond, and time zone values.
The default format is HH:mm:ss; for example, 02:30:03.
- *fmtString* is the format string based on `java.text.SimpleDateFormat` directives
- *Locale* is an instance of `java.util.Locale` where locale specifies the location of the initial time value

If the format is not specified, the default formats are tried in sequence. An exception occurs if parsing fails with every format.

For information on valid default format patterns, see [Valid Default Format Patterns](#) on page 362.

Using Factory Methods

You do not need to call a factory method to make simple variable or property initializations. These two statements are equivalent:

```
effectiveTime is a time initially '10:34:16 am GMT'
effectiveTime is a time initially calendar().time("10:34:16 am GMT")
```

The first statement is more efficient because it is evaluated at compile time. The second statement is evaluated at runtime.

SRL Examples

```
//calendar() is a builtin that returns the instance of NdCalendar associated
with the
session.
    cal is some NdCalendar initially calendar().
    t1 is a time initially cal.time("14:30:03", "HH:mm:ss", Locale.US).
    print("t1 = "t1).
```

Result

```
t1 = 2:30:03 PM PST
```

timestamp()

Description

The `timestamp()` method returns a timestamp with values obtained by parsing the `tsString`.

Syntax

```
timestamp(string tsString) : timestamp
timestamp(string tsString, string fmtString) : timestamp
timestamp(string tsString, string fmtString, Locale Locale) : timestamp
```

- *tsString* is the string to parse for day, month, year, hour, minute, second, millisecond, microsecond, nanosecond, and time zone values.
The default format is dd-MMM-yyyy HH:mm:ss aa zz; for example, 15-Jan-2020 02:30:03 PM PST
- *fmtString* is the format string based on `java.text.SimpleDateFormat` directives
- *Locale* is an instance of `java.util.Locale` where *locale* specifies the location of the initial timestamp value

If the format is not specified, the default formats are tried in sequence. An exception occurs if parsing fails with every format.

For information on valid default format patterns, see [Valid Default Format Patterns](#) on page 362.

Using Factory Methods

You do not need to call a factory method to make simple variable or property initializations. These two statements are equivalent:

```
effectiveTimestamp is a timestamp initially '20-mar-2020 10:34:16 am GMT'.
effectiveTimestamp is a timestamp initially calendar().date("20-mar-2020
10:34:16 am
GMT").
```

The first statement is more efficient because it is evaluated at compile time. The second statement is evaluated at runtime.

SRL Example

```
//calendar() is a builtin that returns the instance of NdCalendar associated
with the session.
cal is some NdCalendar initially calendar().
ts1 is a timestamp initially cal.timestamp("05-März-2020 10:52:15 pm PST",
"dd-MMM-yyyy HH:mm:ss a z", Locale.GERMAN).
print("ts1 = " ts1).
```

Result

```
ts1 = Mar 5, 2020 10:52:15 AM PST
```

`yearWeekDay()`

Description

The `yearWeekDay()` method returns a date for a specified year, week in the year, and day in the week.

Syntax

```
yearWeekDay(integer year, integer week, integer day) : date
```

- The argument *year* is the year as an integer with 4 digits for years after 1900 and < 0 for BC years
- The argument *week* is the week in the year as an integer in the range 1->53
- The argument *day* is the day in the week (locale specific) as an integer in the range 1->7

In this method, the first week of the month is the week that includes the day specified in the *day* argument, even if it is a fragment of a week.

For example, the following statement returns the date December 29, 2019:

```
yearWeekDay(2020, 1, 1)
```

December 29 is the first day of the week that contains the specified day, January 1.

To return the first week in the year that includes the weekday specified by the *day* argument, use [indexedYearWeekDay\(\)](#) on page 368. In the example above, indexedYearWeekDay() would return January 5, 2020 instead of December 29, 2019 because that is the first day of the first full week in the month.

SRL Example

```
//Initialize a date variable to the specified year, month, and day.  
theYWD is a date initially calendar().yearWeekDay(2020, 12, 3).  
print("This is the date: "theYWD).
```

Result

```
This is the date: 17-Mar-2020
```

Comparison and Calculation Methods

Calculation, comparison, and value accessor methods for date and time types.

isLeap()

Description

The `isLeap()` method tests if an integer is a leap year and returns a boolean value.

Syntax

```
isLeap(integer year) : boolean
```

The argument *year* is the year as an integer. If *year* is both a valid year and a leap year, the result returned is true; otherwise it is false.

The integer is a leap year if *year* is either:

- Divisible by 4 but not by 100; for example, 2020
- Divisible by both 100 and 400; for example, 2000

SRL Example

```
//Initialize an NdCalendar variable and then access the methods through the
variable.
cal is some NdCalendar initially calendar().
print("2020 is a leap year: " cal.isLeap(2020)).
```

Result

```
2020 is a leap year: true
```

month()

Description

The `month()` method returns the month of a year as an integer from 1–12, with January being 1 and December being 12.

Syntax

```
month(string name) : integer
```

The argument *name* is the name of the month as a string. This method tries to match both the full and the abbreviated month names.

SRL Example

```
//Initialize an integer variable to the specified month.
theMonth is an integer initially calendar().month("July").
print("This is the month number: "theMonth).
```

Result

```
This is the month number: 7
```

monthLength()

Description

The `monthLength()` method returns the length of the specified month in number of days.

Syntax

```
monthLength(integer year, integer month) : integer
```

- The argument *year* is the year as an integer
- The argument *month* is the month of the year as an integer

SRL Example

```
//Initialize integer variables to the specified year and month.  
theMonthLength is an integer initially calendar().monthLength(2020, 3).  
print("This month contains "theMonthLength " days.").
```

Result

```
This month contains 31 days.
```

monthName()

Description

The `monthName()` method converts an integer to its full corresponding month name.

Syntax

```
monthName(integer month) : string
```

The argument *month* is the month of the year as an integer from 1–12, with January being 1 and December being 12.

SRL Example

```
//Initialize a string variable to the specified month.  
theMonthName is a string initially calendar().monthName(11).  
print("The month is: "theMonthName).
```

Result

```
The month is: November
```

shortMonthName()

Description

The `shortMonthName()` method converts an integer to its corresponding abbreviated month name.

Syntax

```
shortMonthName(integer month) : string
```

The argument *month* is the month of the year as an integer from 1–12, with January being 1 and December being 12.

SRL Example

```
//Initialize a string variable to the specified month.
theshortMonthName is a string initially calendar().shortMonthName(11).
print("The abbreviated month name is: "theshortMonthName).
```

Result

```
The abbreviated month name is: Nov
```

shortWeekDayName()

Description

The *shortWeekDayName()* method converts an integer to its corresponding and abbreviated week name.

Syntax

```
shortWeekDayName(integer weekDay) : string
```

The argument *weekDay* is the day of the week as an integer from 1–7, with Sunday being 1 and Saturday being 7.

SRL Example

```
//Initialize a string variable to the specified month.
theshortWeekDayName is a string initially calendar().shortWeekDayName(3).
print("The abbreviated day name is: "theshortWeekDayName).
```

Result

```
The abbreviated day name is: Tue
```

weekDay()

Description

The *weekDayName()* method returns the specified weekday as an integer from 1–7 with Sunday being 1 and Saturday being 7.

Syntax

```
weekDay(string name) : integer
```

The argument *name* is the name of the day as a string.

This method tries to match both the full and the abbreviated three-letter weekday names.

SRL Example

```
//Initialize an integer variable to the specified weekday.  
theweekDay is an integer initially calendar().weekDay("Tuesday").  
print("The number of the weekday is: "theweekDay).
```

Result

```
The number of the weekday is: 3
```

weekDayName()

Description

The `weekDayName()` method converts an integer to its corresponding weekday name.

Syntax

```
weekDayName(integer weekDay) : string
```

The argument *weekDay* is the day of the week as an integer from 1–7, with Sunday being 1 and Saturday being 7.

SRL Example

```
//Initialize a string variable to the specified weekday.  
theweekDayName is a string initially calendar().weekDayName(5).  
print("The weekday is: "theweekDayName).
```

Result

```
The weekday is: Thursday
```

NdDate

The class `NdDate` provides a set of properties and methods that support the Blaze Advisor date data type; for example:

```
currentTs is a timestamp initially today.add(calendar().currentTime)).
```

add()

Description

The `add()` method has two forms:

- Form 1 returns a timestamp consisting of the `time` argument appended to the date
See [Syntax, Form 1](#) on page 377
- Form 2 returns a date consisting of the `duration` argument added to the date
See [Syntax, Form 2](#) on page 377

Syntax, Form 1

```
add(time t) : timestamp
```

The argument `t` is the time value to append to the date: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the `add()` method of a date produces the same result as using the plus (+) operator with a date and a time. For example, if `d` is a date and `t` is a time, then this statement:

```
d + t
```

returns the same result as this statement:

```
d.add(t)
```

SRL Examples

```
//Initialize a timestamp variable to the current timestamp.
theTime is a timestamp initially 'April 18, 2020 10:13:10 am PDT'.
newTime is a timestamp.
newTime = theTime + (2 hours).
print("The timestamp is: "newTime).

//Initialize a timestamp variable to the current timestamp.
theTime is a timestamp initially 'April 18, 2020 10:13:10 am PDT'.
newTime is a timestamp.
newTime = theTime.add(2 hours).
print("The timestamp is: "newTime).
```

Results

```
The timestamp is: Apr 18, 2020 12:13:10 PM PDT
```

Syntax, Form 2

```
add(duration d) : date
```

The argument `d` is the duration value to add to the date: years, months, weeks, or days. Other duration units are not valid.

Calling the `add()` method of a date produces the same result as using the plus (+) operator with a date and a duration. For example, if `d` is a date and `dur` is a duration, then this statement:

```
d + dur
```

returns the same result as this statement:

```
d.add(dur)
```

SRL Examples

```
//Initialize a date variable to the current date.  
theDate is a date initially 'April 18, 2020'.  
newDate is a date.  
newDate = theDate + (2 days).  
print("This is the date: "newDate).  
  
//Initialize a date variable to the current date.  
theDate is a date initially 'April 18, 2020'.  
newDate is a date.  
newDate = theDate.add(2 days).  
print("This is the date: "newDate).
```

Results

```
This is the date: 20-Apr-2020
```

format()

Description

The `format()` method returns a string with the year, month, and day values of the date in the format defined by the argument. The method can take either of two arguments:

- `fmtString`
See [Syntax, fmtString](#) on page 378
- `fmtConstant`
See [Syntax, fmtConstant](#) on page 379

Syntax, `fmtString`

```
format(string fmtString) : string
```

The argument `fmtString` is a formatting string that uses the directives defined by the `java.text.SimpleDateFormat` class. See [Valid Default Format Patterns](#) on page 362.

SRL Example

```
//theDate is an initialized date value
```

```
theDate is a date initially calendar().currentDate().
print(theDate.format("yyyy, MMM, dd")).
```

Result

The current date in the specified format; for example:

```
2020, Dec, 04
```

Syntax, *fmtConstant*

```
format(integer fmtConstant) : string
```

The argument *fmtConstant* is an integer value from 0 to 3, corresponding to NdDate static integer properties as listed in the table below. You can also use the properties FULL, LONG, MEDIUM, or SHORT.

SRL Examples

```
//theDate is an initialized date value
theDate is a date initially calendar().currentDate().
print("This is the answer: "theDate.format(0)).

//theDate is an initialized date value
theDate is a date initially calendar().currentDate().
print("This is the answer: "theDate.format(NdDate.FULL)).
```

Results

```
This is the answer: Saturday, December 12, 2020
```

See the following table for other available formats.

NdDate Property	Integer Value	Output Format
FULL	0	<p>Weekday, month, day, year. For example:</p> <pre>print(theDate.format(NdDate.FULL)).</pre> <p>returns this result:</p> <pre>Saturday, December 12, 2020</pre>
LONG	1	<p>Month, day, year. For example:</p> <pre>print(theDate.format(NdDate.LONG)).</pre> <p>returns this result:</p> <pre>December 12, 2020</pre>
MEDIUM	2	<p>Short month, day, year. For example:</p> <pre>print(theDate.format(NdDate.MEDIUM)).</pre>

NdDate Property	Integer Value	Output Format
		<p>returns this result:</p> <pre>Dec 12, 2020</pre>
SHORT	3	<p>Month/day/year. For example:</p> <pre>print(theDate.format(NdDate.SHORT)).</pre> <p>returns this result:</p> <pre>12/12/20</pre>

stampAt()

Description

The `stampAt()` method returns the timestamp obtained by combining the date and time values while also preserving the daylight-saving offset contained in the time value.

Syntax

```
stampAt(time t) : timestamp
```

The argument `t` is the time to append to the date.

This method produces the same result as the plus (+) operator except when the daylight-saving offset recorded in time does not correspond to the normal offset for the date. In this case, this method preserves the time's offset.

SRL Examples

```
ts1 is a timestamp initially '1-apr-2020'.stampAt('09:30:00 PDT').
print("This is the answer: "ts1).

ts1 is a timestamp initially '1-apr-2020' + '09:30:00 PDT'.
print("This is the answer: "ts1).
```

Results

```
This is the answer: Apr 1, 2020 8:30:00 AM PST
```

 **Note** Jan 1, 2020 10:30 **PST** and Jan 1, 2020 09:30 **PDT** are equal in the `stampAt()` method.

subtract()

Description

The `subtract()` method has two forms:

- Form 1 returns a timestamp consisting of the `time` argument appended to the date
See [Syntax, Form 1](#) on page 381
- Form 2 returns a date consisting of the `duration` argument subtracted from the date
See [Syntax, Form 2](#) on page 381

Syntax, Form 1

```
subtract(time t) : timestamp
```

The argument `t` is the time value to subtract from the date: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the `subtract()` method of a timestamp produces the same result as using the minus (-) operator with a date and a time. For example, if `ts` is a timestamp and `t` is a time, then this statement:

```
ts - t
```

returns the same result as this statement:

```
ts.subtract(t)
```

SRL Examples

```
//Initialize a timestamp variable to the current timestamp.
theTime is a timestamp initially calendar().currentTimestamp(true).
newTime is a timestamp.
newTime = theTime - (2 hours).
print("This is the answer: "newTime).

//Initialize a timestamp variable to the current timestamp.
theTime is a timestamp initially calendar().currentTimestamp(true).
newTime is a timestamp.
newTime = theTime.subtract(2 hours).
print("This is the answer: "newTime).
```

Results

The current timestamp minus 2 hours; for example:

```
This is the answer: Apr 18, 2020 9:20:43.404 AM PDT
```

Syntax, Form 2

```
subtract(duration d) : date
```

The argument *d* is the duration value to subtract from the date: years, months, weeks, or days. Other duration units are not valid.

Calling the `subtract()` method of a date produces the same result as using the minus (-) operator with a date and a duration. For example, if *d* is a date and *dur* is a duration, then this statement:

```
d - dur
```

returns the same result as this statement:

```
d.subtract(dur)
```

SRL Examples

```
//Initialize a date variable to the current date.  
theDate is a date initially calendar().currentDate().  
newDate is a date.  
newDate = theDate - (2 days).  
print("This is the date: "newDate).  
  
//Initialize a date variable to the current date.  
theDate is a date initially calendar().currentDate().  
newDate is a date.  
newDate = theDate.subtract(2 days).  
print("This is the date: "newDate).
```

Results

The current date minus 2 days; for example:

```
This is the date: 20-Apr-2020
```

`subtractInDays()`

Description

The `subtractInDays()` method returns the difference between the date value and the argument, expressed as a number of days.

Syntax

```
subtractInDays(date d2) : duration
```

The argument *d2* is the date value to subtract from *d1*.

SRL Example

```
//Initialize a date variable and specify a date to subtract to determine the  
duration.  
theDate is a duration initially '1-jan-2020'.subtractInDays('1-nov-2016').  
print("This is the answer: "theDate).
```

Result

This is the answer: 1156 days

subtractInMonths()

Description

The `subtractInMonths()` method returns the difference in years, months, and days between the date value and the argument.

Syntax

```
subtractInMonths(date d2) : duration
```

The `d2` argument is the date value to subtract from `d1`.

SRL Example

```
//Initialize a date variable and specify a date to subtract to determine the
duration.
theDate is a duration initially '1-jan-2020'.subtractInMonths('20-oct-2014').
print("This is the answer: "theDate).
```

Result

This is the answer: 5 years 2 months 12 days

NdDuration

The class `NdDuration` provides a set of properties and methods that support the Blaze Advisor duration data type.

These are the read-only properties of the duration data type.

The methods are called using standard dot notation on a Blaze Advisor duration; for example:

```
theDuration is a duration initially 2 hours.
newDuration is a duration initially theDuration.add(5 minutes).
```

Use the calculation and comparison methods that are defined in the following sections.

add()

Description

The `add()` method returns a duration obtained by adding the argument to the duration value.

Syntax

```
add(duration d) : duration
```

The argument *d* is the duration value to add to the initial duration: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the `add()` method of a duration produces the same result as using the plus (+) operator with a duration and a duration. For example, if *d1* is a duration and *d2* is a duration, then this statement:

```
d1 + d2
```

returns the same result as this statement:

```
d1.add(d2)
```

SRL Examples

```
//Initialize a duration variable to a duration.  
theDur is a duration initially 2 days.  
newDur is a duration.  
newDur = theDur + (3 days).  
print("The duration is: "newDur).  
  
//Initialize a duration variable to a duration.  
theDur is a duration initially 2 days.  
newDur is a duration.  
newDur = theDur.add(3 days).  
print("The duration is: "newDur).
```

Results

```
The duration is: 5 days
```

compare()

Description

The `compare()` method returns:

- Positive 1 (1) if the original duration is greater than the argument value
- Negative 1 (-1) if the original duration is less than the argument value

Syntax

```
compare(duration d2, integer monthLen, integer yearLen) : integer
```

- The argument *d2* is the duration value to compare to the duration
- The argument *monthLen* is the number of days to equal 1 month
- The argument *yearLen* is the number of days to equal 1 year

The built-in operators let you compare duration values. However, because a duration does not have a context for distinguishing between months, the built-in operators rely on these default values:

- 1 month = 31 days
- 1 year = 365 days (366 days if the duration is not tied to a specific date)

This method lets you explicitly set the number of days in a month and in a year that you want to use in your comparison.

Use this support method instead of the built-in comparison operators when the default number of days in a month and in a year are not appropriate for the comparison.

SRL Example

```
d1 is a duration initially 10 months.
d2 is a duration initially 7 months.
int1 is an integer initially 0.
int1 = d1.compare(d2, 31, 365).
print("dur1 is greater than dur2: " int1).
```

Result

```
dur1 is greater than dur2: 1
```

invert()

Description

The `invert()` method returns the inverse of the original duration value.

Syntax

```
invert() : duration
```

This method takes no arguments.

SRL Example

```
//Initialize a duration variable to a specified duration.
d is a duration initially 3 days.
d2 is a duration initially d.invert().
print("The duration is: "d2).
```

Result

```
The duration is: -3 days
```

multiply()

Description

The `multiply()` method returns a duration obtained by multiplying the original duration value by the argument.

Syntax

```
multiply(integer m) : duration
```

The argument `m` is the integer by which you will multiply the duration.

SRL Example

```
//Initialize a duration variable to a specified duration.  
d is a duration initially 3 days.  
d2 is a duration initially d.multiply(3).  
print("The duration is: "d2).
```

Result

```
The duration is: 9 days
```

subtract()

Description

The `subtract()` method returns a duration obtained by subtracting the argument from the original duration.

Syntax

```
subtract(duration d2) : duration
```

The argument `d2` is the duration value to subtract from the duration: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds.

Calling the `subtract()` method of a duration produces the same result as using the minus (-) operator with two duration values. For example, if `d1` is a duration and `d2` is a duration, then this statement:

```
d1 - d2
```

returns the same result as this statement:

```
d1.subtract(d2)
```

SRL Examples

```
//Initialize a duration variable to a specified duration.
theDur is a duration initially 3 days.
newDur is a duration.
newDur = theDur - (2 days).
print("The duration is: "newDur).
//Initialize a duration variable to a specified duration.

theDur is a duration initially 3 days.
newDur is a duration.
newDur = theDur.subtract(2 days).
print("The duration is: "newDur).
```

Results

```
The duration is: 1 day
```

NdTime

The class NdTime provides a set of properties and methods that support the Blaze Advisor time data type.

These are the read-only properties of the time data type.

These methods are called using standard dot notation on a Blaze Advisor time value; for example:

```
theTime is a time initially '17:42:08 pm'.
theTime is a time initially calendar().currentTime(true).
```

add()

Description

The add() method returns a time obtained by adding the argument to the time value.

Syntax

```
add(duration d) : time
```

The argument *d* is the duration value to add to the time: hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the add() method of a time produces the same result as using the plus (+) operator with a time and a duration. For example, if *t* is a time and *dur* is a duration, then this statement:

```
t + dur
```

returns the same result as this statement:

```
t.add(dur)
```

SRL Examples

```
//Initialize a time variable to the current time.  
theTime is a time initially calendar().currentTime(true).  
newTime is a time.  
newTime = theTime + (2 hours).  
print("The time is: "newTime).  
  
//Initialize a time variable to the current time.  
theTime is a time initially calendar().currentTime(true).  
newTime is a time.  
newTime = theTime.add(2 hours).  
print("The time is: "newTime).
```

Results

The current time plus 2 hours; for example:

```
The time is: 2:07:48 PM PDT
```

inDaylightSavings()

Description

The `inDaylightSavings()` method returns `true` if the time zone of the time is a daylight-saving time zone; for example, Pacific Daylight Time.

Syntax

```
inDaylightSavings() : boolean
```

SRL Example

```
//Initialize a time variable to the current time.  
theStatus is a time initially calendar().currentTime(true).  
print("Is it Daylight Savings Time?: " theStatus.inDaylightSavings()).
```

Result

```
Is it Daylight Savings Time?: true OR Is it Daylight Savings Time?: false
```

The answer returned depends on the time on your machine.

format()

The `format()` method returns a string with the hour, minute, second, millisecond, and time-zone values of the time in the format defined by the argument. The method can take either of two arguments:

- `fmtString`
See [Syntax, fmtString](#) on page 389
- `fmtConstant`
See [Syntax, fmtConstant](#) on page 389

Syntax, `fmtString`

```
format(string fmtString) : string
```

The argument `fmtString` is a formatting string that uses the directives defined by the `java.text.SimpleDateFormat` class. See [Valid Default Format Patterns](#) on page 362.

SRL Example

```
//theTime is an initialized time value.  
theTime is a time initially calendar().currentTime(true).  
print("This is the time: " theTime.format("HH:mm:ss:SS aa zz")).
```

Result

The current time; for example:

```
This is the time: 14:58:46:735 PM PDT
```

Syntax, `fmtConstant`

```
format(integer fmtConstant) : string
```

The argument `fmtConstant` is a integer from 0 to 3, corresponding to `NdTime` static integer properties. You can also use the properties FULL, LONG, MEDIUM, or SHORT.

SRL Examples

```
//theTime is an initialized time value.  
theTime is a time initially calendar().currentTime(true).  
print("This is the time: " theTime.format(0)).  
  
//theTime is an initialized time value.  
theTime is a time initially calendar().currentTime(true).  
print("This is the time: " theTime.format(NdTime.FULL)).
```

Results

The current time; for example:

```
This is the time: 3:00:14 PM PDT
```

See the following table for other available formats.

NdTime Property	Integer Value	Output Format
FULL LONG	0 1	Hour:minutes:seconds AM/PM time zone. For example: <pre>print(theTime.format(NdTime.FULL)). returns this output: 9:35:34 AM PST</pre>
MEDIUM	2	Hour:minutes:seconds AM/PM. For example: <pre>print(theTime.format(NdTime.MEDIUM)). returns this output: 9:35:34 AM</pre>
SHORT	3	Hour:minutes AM/PM. For example: <pre>print(theTime.format(NdTime.SHORT)). returns this output: 9:35 AM</pre>

inTimeZone()

Description

The `inTimeZone()` method returns a time with the specified time zone. The returned time value will correspond to the same Coordinated Universal Time (UTC) as the original time, but with the appropriate value for the target time zone. (Zero hours UTC is midnight in Greenwich, England.)

Syntax

```
inTimeZone(TimeZone destTimezone) : time
```

The `destTimezone` argument is the target time zone that the adjusted time will use.

SRL Example

```
//Initialize a TimeZone variable to the specified time zone.  
theTimeZone is some TimeZone initially TimeZone.getTimeZone("GMT").
```

```

theTime is a time initially calendar().currentTime(true).
print("The local time is: "theTime).
adjTime is a time.
adjTime = theTime.inTimeZone(theTimeZone).
print("The adjusted time is: " adjTime)

```

Result

The initial and adjusted times and time zones; for example:

```

The local time is: 3:01:56.257 PM PST
The adjusted time is: 11:01:56.257 PM GMT

```

shiftToTimeZone()

Description

The *shiftToTimeZone()* method returns a time with the same time value as the original, but with the specified time zone.

Syntax

```
shiftToTimeZone(TimeZone destTimezone) : time
```

The *destTimezone* argument is the target time zone that the adjusted time will use.

SRL Example

```

//Initialize a TimeZone variable to a specified time zone.
theTimeZone is some TimeZone initially TimeZone.getTimeZone("GMT").
theTime is a time initially '2:24:08 PM PST'.
print("The local time is: " theTime).
newTime is a time.
newTime = theTime.shiftToTimeZone(theTimeZone).
print("The new time is: " newTime).

```

Result

The time value with the initial and adjusted time zones; for example:

```

The local time is: 2:24:08 PM PST
The new time is: 2:24:08 PM GMT

```

stripTimeZone()

Description

The *stripTimeZone()* method returns a time that is equivalent to the original time except for the time zone, which is set to the default time zone.

Syntax

```
stripTimeZone() : time
```

SRL Example

```
//Initialize a time variable to a time and a time zone.  
theTS is a time initially '18:34:16 pm GMT'.  
print("The time is: " theTS.stripTimeZone()).
```

Result

```
The time is: 6:34:16 AM
```

subtract()

Description

The subtract() method has two forms:

- Form 1 returns a time obtained by subtracting the argument from the time value
See [Syntax, Form 1](#) on page 392
- Form 2 returns a duration obtained by subtracting the argument from the time value
See [Syntax, Form 2](#) on page 393

Syntax, Form 1

```
subtract(duration d) : time
```

The argument *d* is the duration value to subtract from the time: hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the subtract() method of a time produces the same result as using the minus (-) operator with a time and a duration. For example, if *t* is a time and *dur* is a duration, then this statement:

```
t - dur
```

returns the same result as this statement:

```
t.subtract(dur)
```

SRL Examples

```
//Initialize a times variable to the current time.  
theTime is a time initially calendar().currentTime(true).  
newTime is a time.  
newTime = theTime - (2 hours).  
print("The time is: "newTime).  
  
//Initialize a times variable to the current time.  
theTime is a time initially calendar().currentTime(true).  
newTime is a time.  
newTime = theTime.subtract(2 hours).  
print("The time is: "newTime).
```

Result

The current time and time zone minus 2 hours; for example:

```
The time is: 10:20:56.686 AM PDT
```

Syntax, Form 2

```
subtract(time t) : duration
```

The argument *t* is the time value to subtract from the time: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other units are not valid.

Calling the `subtract()` method for a time produces the same result as using the minus (-) operator with two time values. For example, if *t* is a time and *t2* is a time, then this statement:

```
t - t2
```

returns the same result as this statement:

```
t.subtract(t2)
```

SRL Examples

```
//Initialize a time variable to a time.
theTime is a time initially '3:14 PM'.
theDur is a duration.
theDur = theTime - ('14:54:00 GMT').
print("The duration is: "theDur).

//Initialize a time variable to a time.
theTime is a time initially '3:14 PM'.
theDur is a duration.
theDur = theTime.subtract('14:54:00 GMT').
print("The duration is: "theDur).
```

Result

```
The duration is: 500 minutes
```

NdTimestamp

The class NdTimestamp provides a set of properties and methods that support the Blaze Advisor timestamp data type.

The methods are called using standard dot notation on a Blaze Advisor timestamp value; for example:

```
theTimestamp is a timestamp initially calendar().currentTimestamp(true).
newTimestamp is a timestamp initially theTimestamp.add(2 hours).
```

add()

Description

The `add()` method returns a timestamp obtained by adding the argument to the timestamp value.

Syntax

```
add(duration d) : timestamp
```

The argument `d` is the duration value to add to the timestamp: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other duration units are not valid.

Calling the `add()` method of a timestamp produces the same result as using the plus (+) operator with a timestamp and a duration. For example, if `ts` is a timestamp and `dur` is a duration, then this statement:

```
ts + dur
```

returns the same result as this statement:

```
ts.add(dur)
```

SRL Examples

```
//Initialize a timestamp variable to the current timestamp.  
theTS is a timestamp initially 'Feb 3, 2020 4:54:15 pm PST'.  
newTS is a timestamp.  
newTS = theTS + (2 weeks).  
print("The timestamp is: "newTS).  
  
//Initialize a timestamp variable to the current timestamp.  
theTS is a timestamp initially 'Feb 3, 2020 4:54:15 pm PST'.  
newTS is a timestamp.  
newTS = theTS.add(2 weeks).  
print("The timestamp is: "newTS).
```

Results

```
The timestamp is: Feb 17, 2020 4:54:15 PM PST
```

format()

Description

The `format()` method returns a string with the year, month, day, hour, minute, second, millisecond, and time-zone values of the timestamp in the format defined by the argument.

The method can take either of two arguments:

- `fmtString`
See [Syntax, fmtString](#) on page 395
- `fmtConstant`
See [Syntax, fmtConstant](#) on page 395

Syntax, *fmtString*

```
format(string fmtString) : string
```

The argument *fmtString* is a formatting string that uses the directives defined by the `java.text.SimpleDateFormat` class. See [Valid Default Format Patterns](#) on page 362.

SRL Example

```
//theTimestamp is an initialized timestamp value.  
theTimestamp is a timestamp initially 'December 4, 2020 6:34:16'.  
print("The answer is: "theTimestamp.format("yyyy, MMM, dd HH:mm:ss:SS a Z")).
```

Result

The timestamp in the specified format and your GMT offset; for example:

```
The answer is: 2020, Dec, 04 06:34:16:00 AM -0800
```

Syntax, *fmtConstant*

```
format(integer fmtConstant) : string
```

The argument *fmtConstant* is a n integer from 0 to 3, corresponding to `NdTime` static integer properties. You can also use the properties FULL, LONG, MEDIUM, or SHORT.

SRL Examples

```
//theTimestamp is an initialized timestamp value.  
theTimestamp is a timestamp initially 'December 4, 2020 6:34:16'.  
print("The answer is: " theTimestamp.format(0)).  
  
//theTimestamp is an initialized timestamp value.  
theTimestamp is a timestamp initially 'December 4, 2020 6:34:16'.  
print("The answer is: " theTimestamp.format(NdTimestamp.FULL)).
```

Results

The timestamp in FULL format and your time zone; for example:

```
The answer is: Friday, December 4, 2020 6:34:16 AM PST
```

See the following table for other available formats.

NdTimestamp Property	Integer Value	Output Format
FULL	0	<p>Weekday, month, day, year hour:minutes:seconds AM/PM time zone. For example:</p> <pre>print(theTimestamp.format(NdTimestamp.FULL)).</pre> <p>returns this result:</p> <pre>Friday, December 4, 2020 10:34:16 AM PST</pre>
LONG	1	<p>Month, day, year hour:minutes:seconds AM/PM time zone. For example:</p> <pre>print(theTimestamp.format(NdTimestamp.LONG)).</pre> <p>returns this result:</p> <pre>December 4, 2020 10:34:16 AM PST</pre>
MEDIUM	2	<p>Short month, day, year hour:minutes:seconds AM/PM. For example:</p> <pre>print(theTimestamp.format(NdTimestamp.MEDIUM)).</pre> <p>returns this result:</p> <pre>Dec 4, 2020 10:34:16 AM</pre>
SHORT	3	<p>Month/day/ year hour:minutes:seconds. For example:</p> <pre>print(theTimestamp.format(NdTimestamp.SHORT)).</pre> <p>returns this result:</p> <pre>12/4/20 10:34 AM</pre>

inDaylightSavings()

Description

The `inDaylightSavings()` method returns `true` if the time zone of the timestamp is a daylight-saving time zone; for example, PDT (Pacific Daylight Time).

Syntax

```
inDaylightSavings() : boolean
```

SRL Example

```
//Initialize a timestamp to the specified time.
newTS is a timestamp initially '05-Feb-2020 10:00:03 PDT'.
x is a boolean initially false.
```

```
x = newTS.inDaylightSavings().
print("This is a daylight-saving time zone: " x).
```

Result

```
This is a daylight-saving time zone: false
```

inTimeZone()

Description

The `inTimeZone()` method returns a timestamp with the specified time zone. The returned timestamp value will correspond to the same Coordinated Universal Time (UTC) as the original timestamp, but with the appropriate values for the target time zone.

Syntax

```
inTimeZone(TimeZone destTimezone) : timestamp
```

The `destTimezone` argument is the target time zone that the new timestamp will use.

SRL Example

```
//Initialize a TimeZone variable to a a time zone value.
theTimeZone is some TimeZone initially TimeZone.getTimeZone("GMT").
theTS is a timestamp initially calendar().currentTimestamp(true).
print("The local timestamp is: " theTS).
adjTS is a timestamp.
adjTS = theTS.inTimeZone(theTimeZone).
print("The adjusted timestamp is: " adjTS).
```

Result

The specified timestamp and time zone; for example:

```
The local timestamp is: Jun 13, 2020 2:19:06.152 PM PDT
The adjusted timestamp is: Jun 13, 2020 9:19:06.152 PM GMT
```

shiftToTimeZone()

Description

The `shiftToTimeZone()` method returns a timestamp with the same date and time values, but with the specified time zone.

Syntax

```
shiftToTimeZone(TimeZone destTimezone) : timestamp
```

The `destTimezone` argument is the time zone that the new timestamp will use.

SRL Example

```
//Initialize a TimeZone variable to a time zone value.  
theTimeZone is some TimeZone initially TimeZone.getTimeZone("GMT").  
theTS is a timestamp initially 'Feb 20, 2020 2:24:08 PM PST'.  
print("The local timestamp is: "theTS).  
newTime is a timestamp.  
newTime = theTS.shiftToTimeZone(theTimeZone).  
print("The new timestamp is: " newTime).
```

Result

```
The local timestamp is: Feb 20, 2020 2:24:08 PM PST  
The new timestamp is: Feb 20, 2020 2:24:08 PM GMT
```

stripTimeZone()

Description

The `stripTimeZone()` method returns a timestamp that is equivalent to the original timestamp except that the time zone is set to the default time zone.

Syntax

```
stripTimeZone() : timestamp
```

This method takes no arguments.

SRL Example

```
//Initialize a timestamp variable to a timestamp.  
theStatus is a timestamp initially '20-mar-2020 18:34:16 pm GMT'.  
print("The current time is: " theStatus.stripTimeZone()).
```

Result

```
The current time is: Mar 21, 2020 6:34:16 AM
```

subtract()

Description

The `subtract()` method returns a timestamp obtained by subtracting the argument from the timestamp value.

Syntax

```
subtract(duration d) : timestamp
```

The argument *d* is the duration value to subtract from the timestamp: years, months, weeks, days, hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. Other duration units are not valid.

Calling the `subtract()` method of a timestamp produces the same result as using the minus (-) operator with a timestamp and a duration. For example, if `ts` is a timestamp and `dur` is a duration, then this statement:

```
ts - dur
```

returns the same result as this statement:

```
ts.subtract(dur)
```

SRL Examples

```
//Initialize a timestamp variable to the specified timestamp.
theTS is a timestamp initially 'Feb 3, 2020 4:54:15 pm PST'.
newTS is a timestamp.
newTS = theTS - (3 weeks).
print("This is the answer: "newTS).

//Initialize a timestamp variable to the specified timestamp.
theTS is a timestamp initially 'Feb 3, 2020 4:54:15 pm PST'.
newTS is a timestamp.
newTS = theTS.subtract(3 weeks).
print("This is the answer: "newTS).
```

Results

The specified timestamp minus the duration value; for example:

```
This is the answer: Jan 13, 2020 4:54:15 PM PST
```

`subtractInDays()`

Description

The `subtractInDays()` method returns a duration expressed in units of days obtained by subtracting the argument from the original timestamp.

Syntax

```
subtractInDays(timestamp ts2) : duration
```

The `ts2` argument is the timestamp value to subtract from the original timestamp.

SRL Example

```
//subtractInDays(timestamp ts2) : duration
theTS is a timestamp initially 'Mar 3, 2020 14:54:15 pm PST'.
theDur is a duration.
theDur = theTS.subtractInDays('Mar 2, 2013 14:54:15 pm PST').
print("The duration is: "theDur).
```

Result

```
The duration is: 2558 days
```

subtractInMilliseconds()

Description

The `subtractInMilliseconds()` method returns a timestamp expressed in units of seconds and milliseconds, obtained by subtracting the argument from the original timestamp.

Syntax

```
subtractInMilliseconds(timestamp ts2) : duration
```

The `ts2` argument is the timestamp value to subtract from the original timestamp.

SRL Example

```
//Initialize a timestamp variable to the current time and time zone.  
theTS is a timestamp initially 'Mar 3, 2020 14:54:15.45 pm PST'.  
theDur is a duration.  
theDur = theTS.subtractInMilliseconds('Mar 2, 2013 14:54:15.45 pm PST').  
print("The duration in milliseconds is: "theDur.milliseconds).
```

Result

```
The duration in milliseconds is: 221011200000
```

subtractInMonths()

Description

The `subtractInMonths()` method returns a timestamp expressed in units of years, months, days, seconds, and milliseconds, obtained by subtracting the argument from the original timestamp.

Syntax

```
subtractInMonths(timestamp ts2) : duration
```

The `ts2` argument is the timestamp value to subtract from the original timestamp.

SRL Example

```
//Initialize a timestamp to a time value.  
theTS is a timestamp initially 'Mar 3, 2020 14:54:15.45 pm PST'.  
theDur is a duration.
```

```
theDur = theTS.subtractInMonths('Mar 2, 2016 16:51:14.40 pm PST').
print("This is the answer: "theDur).
```

Result

```
This is the answer: 4 years 22 hours 3 minutes 1 second 50 milliseconds
```

brUnit Assertions

The static methods in the Assert support class are used to write assertion expressions that test comparisons between expected and actual values. These assertion expressions are used in tests and the tests are run using the brUnit Module.

There are three categories of Assert support class methods:

- `assert().assert` methods that return void
- `assert().check` methods that take a string message as an argument, and return a boolean value
- `assert().fail` method that return void.

This sample SRL includes a test that uses one of the assertion methods to test the expected return value from a ruleset:

```
account1 is a Account initially {
    type = "checking"
}

function main
is {
    account1.interestRate = apply Account_Ruleset.
    print("The interest rate is " account1.interestRate).
}

function setup_for_Account_Ruleset
is {
    account1.balance = 4500.
}

function test_for_Account_Ruleset
is {
    account1.interestRate = apply Account_Ruleset.
    assert().assertEquals("The interest rate should be: ", 02.75,
    account1.interestRate, 0).
}

ruleset Account_Ruleset
returning a real
is {

    rule interestRateRule
    is
        if account1.balance > 4000
        then {
            set account1.interestRate to 2.75.
            return account1.interestRate.
}
```

```
    }
```

The test passes because the expected value matches the actual return value.

In this case, the `assert().assertEquals(String, double expected, double actual, double delta)` method is used. Other methods that could have been used include:

- `assert().assertEquals(double expected, double actual, double delta)`
This method does the same thing, but it does not include a string message.
- `assert().checkEquals(String, double expected, double actual, double delta)`
This method is similar to the one used in the example, except that it is one of the `assert().check` methods.

brUnit Assertions Methods

Support class for the testing framework to produce assertion exceptions based on comparisons between expected and actual values. Assertion exceptions of the type **NdAssertionFailedError** or **NdComparisonFailure** are thrown from the assertion methods.

Method	Description
<code>assertEquals(boolean, boolean)</code>	Asserts that the two specified booleans are equal. The first boolean is the expected value and the second one is the actual value.
<code>assertEquals(string, boolean, boolean)</code>	Asserts that the two specified booleans are equal. The first value is a string message. The first boolean is the expected value and the second one is the actual value.
<code>assertEquals (byte, byte)</code>	Asserts that two specified bytes are equal. The first byte is the expected value and the second one is the actual value.
<code>assertEquals (string, byte, byte)</code>	Asserts that two specified bytes are equal. The first value is a string message. The first byte is the expected value and the second one is the actual value.
<code>assertEquals (char, char)</code>	Asserts that the two specified characters are equal. The first character is the expected value and the second one is the actual value.
<code>assertEquals (string, char, char)</code>	Asserts that the two specified characters are equal.

Method	Description
	The first value is a string message. The first character is the expected value and the second one is the actual value.
<code>assertEquals (double, double, double)</code>	Asserts that two specified doubles are within the delta. The first double is the expected value, the second double is the actual value, and the third double is the permitted delta between the two.
<code>assertEquals (string, double, double, double)</code>	Asserts that two specified doubles are within the delta. The first value is a string message. The first double is the expected value, the second double is the actual value, and the third double is the permitted delta between the two.
<code>assertEquals (float, float, float)</code>	Asserts that two specified floats are within the delta. The first float is the expected value, the second float is the actual value, and the third float is the permitted delta between the two.
<code>assertEquals (string, float, float, float)</code>	Asserts that two specified floats are within the delta. The first value is a string message. The first float is the expected value, the second float is the actual value, and the third float is the permitted delta between the two.
<code>assertEquals (integer, integer)</code>	Asserts that the two specified integers are equal. The first integer is the expected value and the second one is the actual value.
<code>assertEquals (string, integer, integer)</code>	Asserts that the two specified integers are equal. The first value is a string message. The first integer is the expected value and the second one is the actual value.
<code>assertEquals (long, long)</code>	Asserts that the two specified longs are equal. The first long is the expected value and the second one is the actual value.
<code>assertEquals (string, long, long)</code>	Asserts that the two specified longs are equal. The first value is a string message. The first long is the expected value and the second one is the actual value.
<code>assertEquals (Object, Object)</code>	Asserts that the two specified objects are equal based on how the object model defines equal. Usually this means that the object property values are the same. The first object is the expected object and the second one is the actual object.

Method	Description
<code>assertEquals (string, Object, Object)</code>	<p>Asserts that the two specified objects are equal based on how the object model defines equal. Usually this means that the object property values are the same.</p> <p>The first value is a string message. The first object is the expected object and the second one is the actual object.</p>
<code>assertEquals (real, real, real)</code>	<p>Asserts that the two specified reals are within the delta.</p> <p>The first real is the expected value, the second real is the actual value, and the third real is the permitted delta between the two.</p>
<code>assertEquals (string, real, real, real)</code>	<p>Asserts that the two specified reals are within the delta.</p> <p>The first value is a string message. The first real is the expected value, the second real is the actual value, and the third real is the permitted delta between the two.</p>
<code>assertEquals (short, short)</code>	<p>Asserts that the two specified shorts are equal.</p> <p>The first short is the expected value and the second one is the actual value.</p>
<code>assertEquals (string, short, short)</code>	<p>Asserts that the two specified shorts are equal.</p> <p>The first value is a string message. The first short is the expected value and the second one is the actual value.</p>
<code>assertEquals (string, string)</code>	<p>Asserts that the two specified strings are equal.</p> <p>The first string is the expected value and the second one is the actual value.</p>
<code>assertEquals (string, string, string)</code>	<p>Asserts that the two specified strings are equal.</p> <p>The fist value is a string message, the second string is the expected value and the third string is the actual value.</p>
<code>assertFalse (boolean)</code>	<p>Asserts that the specified boolean condition is false.</p>
<code>assertFalse (string, boolean)</code>	<p>Asserts that the specified boolean condition is false.</p> <p>The first value is a string message.</p>
<code>assertNotNull (Object)</code>	<p>Asserts that the specified object is not null.</p>
<code>assertNotNull (string, Object)</code>	<p>Asserts that the specified object is not null.</p> <p>The first value is a string message.</p>
<code>assertNotSame (Object, Object)</code>	<p>Asserts that the two specified objects are not the same instance.</p>

Method	Description
	The first object is the expected object and the second one is the actual object.
<code>assertNotSame (string, Object, Object)</code>	Asserts that the two specified objects are not the same instance. The first value is a string message. The first object is the expected object and the second one is the actual object.
<code>assertNull (Object)</code>	Asserts that the specified object is null.
<code>assertNull (string, Object)</code>	Asserts that the specified object is null. The first value is a string message.
<code>assertSame (Object, Object)</code>	Asserts that the two specified objects are the same instance. The first object is the expected object and the second one is the actual object.
<code>assertSame (string, Object, Object)</code>	Asserts that the two specified objects are the same instance. The first value is a string message. The first object is the expected object and the second one is the actual object.
<code>assertTrue (boolean)</code>	Asserts that the specified boolean condition is true.
<code>assertTrue (string, boolean)</code>	Asserts that the specified boolean condition is true. The first value is a string message.
<code>checkEquals (string, boolean, boolean)</code>	Asserts that the two specified booleans are equal. The first value is a string message. The first boolean is the expected value and the second boolean is the actual value. This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
<code>checkEquals (string, byte, byte)</code>	Asserts that two specified bytes are equal. The first value is a string message. The first byte is the expected value and the second one is the actual value. This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
<code>checkEquals (string, char, char)</code>	Asserts that two specified chars are equal. The first value is a string message. The first char is the expected value and the second one is the actual value.

Method	Description
	This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
<code>checkEquals (string, double, double)</code>	<p>Asserts that two specified doubles are equal. The first value is a string message. The first double is the expected value and the second one is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
<code>checkEquals (string, float, float)</code>	<p>Asserts that two specified floats are equal. The first value is a string message. The first float is the expected value and the second one is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
<code>checkEquals (string, integer, integer)</code>	<p>Asserts that the two specified integers are equal. The first value is a string message. The first integer is the expected value and the second integer is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
<code>checkEquals (string, long, long)</code>	<p>Asserts that two specified longs are equal. The first value is a string message. The first long is the expected value and the second one is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
<code>checkEquals (string, Object, Object)</code>	<p>Asserts that the two specified objects are equal based on how the object model defines equal. Usually this means that the object property values are the same.</p> <p>The first value is a string message. The first object is the expected object and the second object is the actual object.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>

Method	Description
checkEquals (string, real, real, real)	<p>Asserts that the two specified reals are within the delta.</p> <p>The first value is a string message. The first real is the expected value, the second real is the actual value and the third real is the permitted delta between the two.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
checkEquals (string, short, short)	<p>Asserts that two specified shorts are equal.</p> <p>The first value is a string message. The first short is the expected value and the second one is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
checkEquals (string, string, string)	<p>Asserts that the two specified strings are equal.</p> <p>The fist value is a string message, the second string is the expected value and the third string is the actual value.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
checkFalse (string, boolean)	<p>Asserts that the specified boolean condition is false.</p> <p>The first value is a string message.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
checkNotNull (string, Object)	<p>Asserts that the specified object is not null.</p> <p>The first value is a string message.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>
checkNotSame (string, Object, Object)	<p>Asserts that the two specified objects are not the same instance.</p> <p>The first value is a string message. The first object is the expected object and the second object is the actual object.</p> <p>This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.</p>

Method	Description
checkNull (string, Object)	Asserts that the specified object is null. The first value is a string message. This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
checkSame (string, Object, Object)	Asserts that the two specified objects are the same instance. The first value is a string message. The first object is the expected object and the second object is the actual object. This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
checkTrue (string, boolean)	Asserts that the specified boolean condition is true. The first value is a string message. This method checks for an assertion failure and does not throw an assertion exception unless the check is not handled by the assertion handler.
fail ()	Marks a test as failed.
fail (string)	Marks a test as failed. The first value is a string message.
handleAssertion (NdAssertionFailedError)	Assertion handler.
signal (NdAssertionFailedError)	Signals a failure. This method is called by the check methods and calls the assertion handler. If the assertion is not handled by the assertion handler, then the assertion is thrown.

The Difference Between the assert().assert and assert().check Methods

The difference between the assert().assert methods and the assert().check methods is the behavior that happens after an exception is signaled. When an **assert().assert** method fails, the brUnit Module signals the exception and stops processing any other assertions in the same test. Under the same scenario, if an **assert().check** method is used, brUnit signals the assertion and continues processing the other assertions in the test.

This is an example:

```
function test_for_Account_Ruleset
is {
```

```

account1.interestRate = apply Account_Ruleset.
    assert().checkFalse("This test checks the type property.",
account1.type = "savings").
    //This statement is true, so the assertion fails. Since the assert().checkFalse
is used, brUnit continues processing.
    assert().assertEquals("This tests the delta between the actual and expected
value.", 02.25, account1.interestRate, 0).
    //This test fails because the delta is .50. Since the assert().assertEquals is
used, an exception is signaled and processing stops.
}

```

assert().fail Methods

An assert().fail method is used to signal a test to fail in the brUnit Module. In this example, the assert().fail method is used to signal a failure when an unexpected value is passed to the ruleset.

```

ruleset Test_Ruleset
is {

rule test_for_negative_range_for_balance
is
if account1.balance is less than 0
then {
    account1.interestRate = apply Account_Ruleset.
    assert().assertEquals(0.5, account1.interestRate, 0.0).
}

rule test_for_lower_range_for_balance
is
if account1.balance is greater than or equal to 0
and account1.balance is less than 3500
then {
    account1.interestRate = apply Account_Ruleset.
    assert().assertEquals(1.5, account1.interestRate, 0.0).
}

rule test_for_upper_range_for_balance
is
if account1.balance is greater than or equal to 3500
then {
    account1.interestRate = apply Account_Ruleset.
    assert().assertEquals(2.0, account1.interestRate, 0.0).
}

rule test_for_other_values
is
if true
//which means that other rules didn't fire
then {
    assert().fail("This is a test in case the value passed to the ruleset is
unexpected.").
}

```

Currency Built-in Functions

The currency built-in functions are used to format currency values as strings and strings as currency values.

Currency Built-in Functions

Function	Description
<code>format(BigDecimal):string</code>	Allows the formatting of currency values and returns a string value.
<code>parse(string):money</code>	Allows the formatting of string values and returns a money value.

format(BigDecimal):string

Description

In this case, the `format()` method returns a string value for a numeric `BigDecimal` argument. It can be used along with the `parse()` method which processes string values into currency values.

Syntax

```
format(BigDecimal bigDecimal):string
```

The argument `bigDecimal` is a numeric value that uses the directives as defined by the `java.math.BigDecimal` class.

SRL Example

```
bigDecimal is some BigDecimal initially BigDecimal.newInstance(100.0).
print("Value of bigDecimal is " currencies().format(bigDecimal)).
```

Result

```
Value of bigDecimal is 100.
```

parse()

Description

The `parse()` method parses a string as a money amount.

Syntax

```
parse(string str):money
```

This method accepts all formats that may be produced by applying `toString()` to a money amount. The string must specify a currency either as a symbol or a 3-letter ISO name. The currency indication may be placed before or after the numeric

value. The numeric value should be formatted according to the locale's conventions (decimal separator, the "thousands' separator is optional). A parenthesized string will be interpreted as a negative amount.

SRL Example

```
string1 is a string initially "USD444".
    print("Money Value from String Result = " currencies().parse(string1)).
```

Result

```
Money from String Result = $444.00.
```

Portable Built-ins (NdPortableBuiltins)

The portable built-in functions are platform-independent functions that perform operations on numbers or strings. All of these functions return a boolean, integer, real, or string value.

If you will be using the date related functions, please review [Using Dates with the Portable\(\) Built-in Functions](#) on page 413 before using them.

 **Note** Due to differences in the ways that various hardware and software environments process floating point numbers, the exact result returned by some of the functions may differ slightly from platform to platform.

This section lists all of the portable built-in functions. For a specific function description and implementation, click the Function name.

Function	Description
portable().century() on page 414	Returns the century of a date as an integer.
portable().charToInt() on page 432	Returns the integer value of a single character of a string.
portable().compareDates() on page 415	Compares two dates.
portable().compressString() on page 433	Returns a string with specified characters removed.
portable().concat() on page 433	Joins one string with another.
portable().concatByContent() on page 434	Joins one string with another and allows a specific number of spaces to be placed between the concatenated strings.
portable().date() on page 416	Returns today's date.
portable().dateMinusDays() on page 416	Returns the date resulting from subtracting days from a date.
portable().datePlusDays() on page 417	Returns the date resulting from adding days to a date.
portable().dateToInt() on page 418	Converts a date to an integer in a specified format.
portable().day() on page 418	Returns the day of a date as an integer.

Function	Description
portable().dayOfWeek() on page 419	Returns the day of the week (Tuesday, for example) of a date as an integer.
portable().daysInMonth() on page 420	Returns the number of days in a given month of a given year.
portable().dddyyyy() on page 421	Returns an enumeration of the format that is used in the calling date function in a dddyyyy format.
portable().ddmmmyyy() on page 421	Returns an enumeration of the format that is used in the calling date function in a ddmmmyyy format.
portable().diffInDays() on page 421	Returns the difference in days between two dates.
portable().diffInMonths() on page 422	Returns the difference in months between two dates.
portable().diffInYears() on page 423	Returns the difference in years between two dates.
portable().findChar() on page 442	Searches a string for one of a set of characters and returns its position.
portable().findString() on page 443	Searches a string for a substring and returns its position.
portable().intToDate() on page 424	Converts an integer in a specified format to a date.
portable().is_in() on page 443	Tests if a value is in a specified range.
portable().isFloat() on page 444	Tests if a string forms a valid real constant.
portable().isInteger() on page 445	Tests if a string forms a valid integer constant.
portable().isLeapYear() on page 425	Tests if a year is a leap year.
portable().max() on page 445	Returns the maximum of two numbers.
portable().min() on page 446	Returns the minimum of two numbers.
portable().mmddyyyy() on page 425	Returns an enumeration of the format that is used in the calling date function in a mmddyyyy format.
portable().mmyyyy() on page 426	Returns an enumeration of the format that is used in the calling date function in a mmyyyy format.
portable().month() on page 426	Returns the month of a date as an integer.
portable().not_in() on page 447	Tests if a value is not in a specified range.
portable().selectValue() on page 448	Returns one of two values based on the value of a boolean expression.
portable().subString() on page 434	Returns the substring of a specified string.
portable().time() on page 427	Returns the current time.
portable().toBoolean() on page 435	Casts an integer, real, or string expression to a boolean.

Function	Description
portable().toFloat() on page 436	Casts an integer, boolean, or string expression to a real.
portable().toInteger() on page 438	Casts a real, boolean, or string expression to an integer.
portable().toLowerCase() on page 439	Returns a string with all letters in lowercase.
portable().toString() on page 440	Casts an integer, real, or boolean expression to a string.
portable().toUpperCase() on page 441	Returns a string with all letters in uppercase.
portable().verifyDate() on page 427	Verifies that three integers (year, month, day) form a valid date.
portable().verifyIntDate() on page 428	Verifies that an integer in a specified format is a valid date.
portable().verifyMonth() on page 428	Verifies that an integer is a valid month.
portable().verifyYear() on page 429	Verifies that an integer is a valid year.
portable().year() on page 429	Returns the year of a date as an integer.
portable().ymdToDate() on page 430	Converts three integers (year, month, day) to a date.
portable().yyyyddd() on page 431	Returns an enumeration of the format that is used in the calling date function in a yyyyddd format.
portable().yyymm() on page 431	Returns an enumeration of the format that is used in the calling date function in a yyymm format.
portable().yyymmdd() on page 431	Returns an enumeration of the format that is used in the calling date function in a yyymmdd format.

Using Dates with the Portable() Built-in Functions

When using the portable() built-in functions, there is no “date” type. Dates are internally manipulated as real numbers because they represent the days and the timespan from January 1, 1300 to the present.

Because there is no date type, you must convert any numeric that is meant to represent a date to an integer using [portable\(\).dateToInt\(\) on page 418](#) or to an internal date using [portable\(\).intToDate\(\) on page 424](#). The first step is to convert the numeric into the appropriate format. The next step is to use the appropriate built-in function to perform the required operation.

For example, if you want to compare two dates, you would use [portable\(\).compareDates\(\) on page 415](#). You could convert each numeric to a date using [portable\(\).intToDate\(\)](#) function and then [portable\(\).compareDates\(\)](#) to

compare the dates. Because dates are internally stored as real numbers, the numeric value must be real numbers, as shown in this example:

```
real1 is a real.  
real1 = portable().intToDate(20200104, portable().yyyymmdd()).  
real2 is a real.  
real2 = portable().intToDate(20181003, portable().yyyymmdd()).  
integer1 is an integer.  
integer1 = portable().compareDates (real1, real2).  
print("This is the answer: "integer1).
```

The result is that integer1 is 1. This means that the first date (20200104) is later than the second date (20181003).

For best results, use integer properties for date values because you can nest the portable().intToDate function in the expression that is used for a built-in function such as portable().compareDates(), as shown in this example:

```
integer1 is an integer.  
integer1 = portable().compareDates(portable().intToDate(20200104,  
portable().yyyymmdd()), portable().intToDate(20181003, portable().yyyymmdd())).  
print("This is the answer: "integer1).
```

Again, the result is that integer1 is 1, but this time only one line of SRL was required.

Under some circumstances, you may need to represent a date as a real number in order to perform a mathematical computation, such as adding a certain number of days to a date using the portable().datePlusDays() function. In this case, the function parameters are real numbers which are compared, and an integer value is returned. In order to do the calculation, the real (dates) must be converted to integers using the portable().dateToInt() function. Again, you can nest the conversion function in the mathematical function that you are using, as shown below:

```
integer1 is an integer.  
integer1 = 20200506.  
real1 is a real.  
real1 = portable().datePlusDays(portable().intToDate(integer1,  
portable().yyyymmdd()), 5).  
integer2 is an integer.  
integer2 = portable().dateToInt(real1, portable().yyyymmdd()).  
print("This is the answer: "integer2).
```

The result is that integer2 = 220200511.

The subsequent descriptions for each date-related built-in function includes information as to which type of conversion function (portable().intToDate() or portable().dateToInt()) you need to use.

portable().century()

Description

This function returns the century of a date that is originally of type real.

Implementation

`portable().century(real):integer`

 **Note** Before using this function, you must first convert your date from an integer number to an internal date format using [portable\(\).intToDate\(\)](#) on page 424.

SRL Example

```
integer1 is an integer initially 20200506.
integer2 is an integer.
integer2 =
portable().century(portable().intToDate(integer1,portable().yyyymmdd())).
print("This is the answer: "integer2).
```

The result is that `integer2` is 2000.

portable().compareDates()

Description

This function compares two internal dates (that are real numbers) and returns +1 if the first date is greater than the second date, and -1 if the first date is less than the second date, and 0 if the dates are the same.

Implementation

`portable().compareDates(real, real):integer`

SRL Example

The following example checks the first date against the second date. Note that the “date” values are originally integers that are later converted to a real format using [portable\(\).intToDate\(\)](#) on page 424. The two real values then become the parameters in the `portable().compareDates()` function. (Note that the line that starts with `portable()` wraps into the second line.)

```
integer1 is an integer initially 20200201.
integer2 is an integer initially 20200401.
integer3 is an integer.
integer3 =
portable().compareDates(portable().intToDate(integer1,portable().yyyymmdd()),
portable().intToDate(integer2,portable().yyyymmdd())).
print("This is the answer: "integer3).
```

The result is that `integer3` is -1. The first date is less than the second date.

First date in the function	Second date in the function	Result
20200401	20200201	+1
20200201	20200401	-1
20200201	20200201	0

portable().date()

Description

This function returns the current date (as a real number) from the server where the project is executing.

Implementation

```
portable().date():real
```

For return value to be usable, you must convert it to an integer using [portable\(\).dateToInt\(\)](#) on page 418.

SRL Example

```
integer1 is an integer.  
integer1 = portable().dateToInt(portable().date(),portable().mmddyyyy()).  
print("This is the answer: "integer1).
```

The answer is the current date in this format: mmddyyyy

portable().dateMinusDays()

Description

This function returns the date resulting from subtracting the specified number of days from a date.

Implementation

```
portable().dateMinusDays(real, integer):real
```

For return value to be usable, you must convert it to an integer using [portable\(\).dateToInt\(\)](#) on page 418.

 **Note** The integer parameter may be negative, in which case the specified number of days are added to the date.

SRL Example

```
integer1 is an integer initially 20200331.  
real1 is a real.  
real1 =  
portable().dateMinusDays(portable().intToDate(integer1,portable().yyyymmdd()), 5).  
print("This is the answer: "real1).
```

real1 = 737510.0 which is internal representation and it not usable. To return the date, you must use [portable\(\).dateToInt](#) to specify the desired date format.

```
real1 is a real initially 737510.0.  
integer2 is an integer.
```

```
integer2 = portable().dateToInt(real1,portable().yyyymmdd()).  
print("This is the answer: "integer2).
```

The result is that integer2 = 20200326.

Date parameter (real)	Days subtracted (integer)	New date as integer value using the portable().dateToInt()
20200331	+5	20200326
20200331	-5	20200405

portable().datePlusDays()

Description

This function returns the date resulting from adding the specified number of days from a date.

Implementation

portable().datePlusDays(real, integer):real

For return value to be usable, you must convert it to an integer using [portable\(\).dateToInt\(\) on page 418](#).

 **Note** The integer may be negative, in which case the specified number of days are subtracted from the date.

SRL Example

```
integer1 is an integer initially 20200331.  
real1 is a real.  
real1 =  
portable().datePlusDays(portable().intToDate(integer1,portable().yyyymmdd()), 5).  
print("This is the answer: "real1).
```

real1 is 737520.0 which is internal representation and it not usable. To return the date, you must use portable().dateToInt to specify the desired date format.

```
real1 is a real initially 737520.0.  
integer2 is an integer.  
integer2 = portable().dateToInt(real1,portable().yyyymmdd()).  
print("This is the answer: "integer2).
```

The result is that integer2 = 20200405.

Date parameter (d)	Days added (n)	New date as integer value using portable().dateToInt()
20200331	+5	20200405
20200331	-5	20200326

portable().dateToInt()

Description

This function is almost always used in conjunction with other portable() functions involving dates. It converts a date (internally represented as a real number) to a single integer, in conformance with a specified format.

If your code generates a value that has been cast to the internal date type or has been updated by one of the portable() date functions, your code must include the portable().dateToInt function(), so that the date is usable.

A date is internally stored as a real data type.

Implementation

portable().dateToInt(real, integer):integer

SRL Example

```
integer1 is an integer initially 20200331.  
real1 is a real.  
real1 =  
portable().dateMinusDays(portable().intToDate(integer1,portable().yyyymmdd()), 5).  
print("This is the answer: "real1).
```

The result for real1 is 737510.0, which is internal representation, and it is not usable. To return the date, you must use portable().dateToInt to specify the desired date format.

```
real1 is a real initially 737510.0.  
integer2 is an integer.  
integer2 = portable().dateToInt(real1,portable().yyyymmdd()).  
print("This is the answer: "integer2).
```

The result is that integer2 = 20200326.

 **Note** To see all possible date formats, refer to portable().verifyIntDate().

Related Links

[portable\(\).verifyIntDate\(\)](#)

portable().day()

Description

This function returns an integer which represents the “day” portion of a specific date.

Implementation

portable().day(real):integer

This function extracts the numerical day (represented as a real) from a date. As a result, you must cast the integer to a date using [portable\(\).intToDate\(\)](#) on page 424 before using the portable().day() function.

SRL Example

The following statement nests the portable().intToDate() function within the portable().day() function to achieve the desired results:

```
integer1 is an integer initially 03142020.
integer2 is an integer.
integer2= portable().day(portable().intToDate(integer1,portable().mmddyyyy())).
print("This is the answer: "integer2).
```

The result is that integer2 = 14.

portable().dayOfWeek()

Description

This function examines a date and returns the numerical representation of the day of the week contained in the date.

Implementation

portable().dayOfWeek(real):integer

This chart shows the numerical representation for each day of the week:

Numeric Value	Week Day
1	Sunday
2	Monday
3	Tuesday
4	Wednesday
5	Thursday
6	Friday
7	Saturday

You must cast the integer to a date using the [portable\(\).intToDate\(\)](#) on page 424 function before using the portable().dayOfWeek() function.

SRL Example

The following statement nests the portable().intToDate() function within the portable().dayOfWeek() function to achieve the desired results:

```
integer1 is an integer initially 12312020.
integer2 is an integer.
integer2=
portable().dayOfWeek(portable().intToDate(integer1,portable().mmddyyyy())).
print("This is the answer: "integer2).
```

The result is that integer2 is 5, which represents that December 31, 2020 is a Thursday.

portable().daysInMonth()

Description

This function returns an integer which represents the number of days in a given month of a given year.

Implementation

portable().daysInMonth(integer, integer):integer

If you do not have separate year and month values, you can use [portable\(\).intToDate\(\)](#) on page 424, [portable\(\).month\(\)](#) on page 426, and [portable\(\).year\(\)](#) on page 429 to obtain the year and month before using portable().daysInMonth().

SRL Example

```
integer1 is an integer initially 20200331.  
integer2 is an integer.  
integer3 is an integer.  
integer4 is an integer.  
integer2 = portable().month(portable().intToDate(integer1,portable().yyyymmdd())).  
integer3 = portable().year(portable().intToDate(integer1,portable().yyyymmdd())).  
integer4 = portable().daysInMonth(integer3, integer2).  
print("This is the answer: "integer4).
```

In this example, integer2 = 3, integer3 = 2020 and integer4 = 31 (number of days in March).

Or you can bundle it in one statement using nested functions. (Note that the third line wraps into the next line.)

```
integer1 is an integer initially 20200331.  
integer2 is an integer.  
integer2 =  
portable().daysInMonth(portable().year(portable().intToDate(integer1,portable().yyy  
ymmdd()))),  
portable().month(portable().intToDate(integer1,portable().yyyymmdd()))).  
print("This is the answer: "integer2).
```

In this example, integer2 = 31 (number of days in March).

Or you can pass specific integer values:

```
integer1 is an integer initially 2020.  
integer2 is an integer initially 3.  
integer3 is an integer.  
integer3 = portable().daysInMonth(integer1, integer2).  
print("This is the answer: "integer3).
```

The result is 31 because there are 31 days in March.

portable().dddyyyy()

Description

This function returns an enumeration of the format that is used in the calling date function in a dddyyyy format. "ddd" in this function represents the dddth day of the specified year.

Implementation

portable().dddyyyy():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate (3502020, portable().dddyyyy()).
```

portable().ddmmyyyy()

Description

This function returns an enumeration of the format that is used in the calling date function in a ddmmmyyyy format.

Implementation

portable().ddmmyyyy():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate (14032020, portable().ddmmyyyy()).
```

portable().diffInDays()

Description

This function returns an integer which represents the difference in days between two (real) dates.

Implementation

portable().diffInDays(real, real):integer

The function subtracts the second real (date) from the first real (date). If the first date is greater than the second one, the result is a positive integer. If the first date is less than the second one, the result is a negative integer. For example, if the first date is 20191231 and the second date is 20200101, the difference in days is -1.

You must include code to cast the integer to the internal Blaze Advisor date format by using [portable\(\).intToDate\(\)](#) on page 424. If either one of the specified dates is not a valid date, the function returns zero.

SRL Example

This example sets the value of integer2 to the number of days between integer1 and the current date. The current date is retrieved using [portable\(\).date\(\)](#) on page 416.

This statement nests the portable().intToDate() function within the portable().diffInDays() function to achieve the desired results:

```
integer1 is an integer initially 20200106.  
integer2 is an integer.  
integer2 = portable().diffInDays(portable().intToDate(integer1,  
portable().yyyymmdd()), portable().date()).  
print("This is the answer: "integer2).
```

This example sets the value of integer3 to the number of days between integer1 and integer2. (Note that the fourth line wraps into the next line.)

```
integer1 is an integer initially 20200101.  
integer2 is an integer initially 20181231.  
integer3 is an integer.  
integer3 = portable().diffInDays(portable().intToDate(integer1,  
portable().yyyymmdd()), (portable().intToDate(integer2, portable().yyyymmdd()))).  
print("This is the answer: "integer3).
```

The result is that integer3 is 366 days.

portable().diffInMonths()

Description

This function returns an integer which represents the difference in months between two (real) dates.

Implementation

portable().diffInMonths(real, real):integer

The function subtracts the second (real) date from the first (real) date. If the first one is greater than second one, the result is a positive integer. If first one is less than the second one, the result is a negative integer. For example, if the first one is 20191231 and second one is 20200101, the difference in months is -1.

You must include code to cast the integer to the internal Blaze Advisor date format by using [portable\(\).intToDate\(\)](#) on page 424. If either one of the specified dates is not a valid date, the function returns zero.

 **Note** The number of days within each month are not used in the calculation.

SRL Example

This example sets the value of a property, *integer2*, to the number of months between the *integer1* and the current date. The current date is retrieved using [portable\(\).date\(\)](#) on page 416.

The following statement nests the [portable\(\).intToDate\(\)](#) function within the [portable\(\).diffInMonths\(\)](#) function to achieve the desired results:

```
integer1 is an integer initially 20200201.
integer2 is an integer.
integer2 =
portable().diffInMonths(portable().intToDate(integer1,portable().yyyymmdd())),
portable().date()).
```

This example sets the value of *integer3* to the number of months between *integer1* and *integer2*. (Note that the fourth line wraps into the next line.)

```
integer1 is an integer initially 20200401.
integer2 is an integer initially 20181231.
integer3 is an integer.
integer3 = portable().diffInMonths(portable().intToDate(integer1,
portable().yyyymmdd()), (portable().intToDate(integer2, portable().yyyymmdd()))).
print("This is the answer: "integer3).
```

The result is that *integer3* is 16 (1 year and 4 months).

portable().diffInYears()

Description

This function returns an integer which represents the difference in years between two dates (represented as real numbers).

Implementation

`portable().diffInYears(real, real):integer`

This function subtracts the second date specified from the first date. If the first date is greater than second one, the result is a positive integer. If the first date is less than the second one, the result is a negative integer. For example, if the first one is 20191231 and the second one is 20200101, the difference in years is -1.

You must include code to cast the integer to the internal Blaze Advisor date format by using [portable\(\).intToDate\(\)](#) on page 424. If either one of the specified dates is not a valid date, the function returns a zero.

 **Note** The number of days and months within each year are not used in the calculation.

SRL Examples

This example sets the value of *integer2* to the number of years between *integer1* and today's date. The current date is retrieved using [portable\(\).date\(\)](#) on page 416.

The following statement nests the portable().intToDate() function within the portable().diffInYears() function to achieve the desired results:

```
integer1 is an integer initially 20201231.  
integer2 is an integer.  
integer2 = portable().diffInYears(portable().intToDate(integer1,  
portable().yyyymmdd()), portable().date()).
```

This example sets the value of integer3 to the number of years between the integer1 and integer2.

```
integer1 is an integer initially 20200101.  
integer2 is an integer initially 20151231.  
integer3 is an integer.  
integer3 = portable().diffInYears(portable().intToDate(integer1,  
portable().yyyymmdd()), (portable().intToDate(integer2, portable().yyyymmdd()))).  
print("This is the answer: "integer3).
```

The result is that integer3 is 5.

portable().intToDate()

Description

This function converts a single integer to a internal date (represented as a real number).

Implementation

portable().intToDate(integer, integer):real

When a date is passed as an integer, it is converted to the internal date format (a real number) using the portable().intToDate function. The first parameter is the integer to be converted. The second parameter is the function that formats the data for the first parameter.

These are the valid formatting functions for the second parameter:

- [portable\(\).dddyyyy\(\)](#)
- [portable\(\).ddmmyyyy\(\)](#)
- [portable\(\).mmddyyyy\(\)](#)
- [portable\(\).mmyyyy\(\)](#)
- [portable\(\).yyyyddd\(\)](#)
- [portable\(\).yyyymm\(\)](#)
- [portable\(\).yyyymmdd\(\)](#)

 **Note** “ddd” in the portable().dddyyyy() function represents the dddth day of the specified year.

SRL Example

This example uses the `portable().intToDate()` function nested in `portable().month()` on page 426. If the `portable().month()` was used without the `portable().intToDate()` function, the results would be *incorrect*.

```
integer1 is an integer initially 20200616.
integer2 is an integer.
integer2 =
portable().month(portable().intToDate(integer1,portable().yyyymmdd())).
```

If you just used `portable().month()`, without a parameter of a real type, the result would be 0 because the output is not correctly formatted, which is why the `portable().intToDate()` function is required.

To see other examples that use `portable().intToDate()` function in this document, see [portable\(\).century\(\)](#) on page 414 and [portable\(\).year\(\)](#) on page 429.

portable().isLeapYear()

Description

This function tests if an integer is a leap year.

Implementation

`portable().isLeapYear(integer):boolean`

If the integer is a valid year and it is a leap year, true is returned; otherwise, false is returned. A year is a leap year if the year is divisible by 4 and when the year is divided by 400 the remainder is not 100, 200, or 300.

SRL Example

```
integer1 is an integer initially 2020.
boolean1 is a boolean.
boolean1 = portable().isLeapYear(integer1).
print("This is the answer: "boolean1).
```

The result is that `boolean1` contains true because 2020 is a leap year.

portable().mmddyyyy()

Description

This function returns an enumeration of the format that is used in the calling date function in a `mmddyyyy` format.

Implementation

`portable().mmddyyyy():integer`

There are no parameters defined in this function.

SRL Example

```
portable().intToDate(03142020, portable().mmddyymm()).
```

portable().mmyyyy()

Description

This function returns an enumeration of the format that is used in the calling date function in a mmyyyy format.

Implementation

portable().mmyyyy():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate(032020, portable().mmyyyy()).
```

portable().month()

Description

This function returns the month of a date as an integer.

Implementation

portable().month(real):integer

This function extracts the numerical month from a date. You then need to use [portable\(\).intToDate\(\)](#) on page 424 to return the results of the portable().month() function as an integer.

SRL Example

This statement nests the portable().intToDate() function within the portable().month function to achieve the desired results:

```
integer1 is an integer initially 20200531.  
integer2 is an integer.  
integer2 = portable().month(portable().intToDate(integer1,  
portable().yyyymmdd())).  
print("This is the answer: "integer2).
```

The result is that integer2 contains 5.

portable().time()

Description

This function returns the current time from the server where the project is executing.

Implementation

`portable().time():real`

This function returns the current time as a real number in HHmmssSS format, where:

- HH – hours from 0-23
- mm – minutes
- ss – seconds
- SS – hundredths of a second

SRL Example

```
real1 is a real.  
real1 = portable().time().
```

The result is a date that appears in this format: 9493398.0. In this case, the time translates to 09 hours, 49 minutes, 33 seconds, and 98 hundredths of a second.

portable().verifyDate()

Description

This function verifies that three integers (year, month, day) specify a valid internal date.

Implementation

`portable().verifyDate(integer, integer, integer):boolean`

The first parameter is the year, the second parameter is the month, and the third parameter is the day.

If the specified integers comprise a valid date, the function returns true; otherwise, it returns false.

SRL Example

In this example, the function returns true because all three parameters form a valid date.

```
boolean1 is a boolean.  
if portable().verifyDate(2020, 12, 04) is true  
then boolean1 = true.
```

```
else boolean1 = false.  
print("This is the answer: "boolean1).
```

portable().verifyIntDate()

Description

This function tests if an integer in a specified format is a valid date.

Implementation

portable().verifyIntDate(integer, integer):boolean

The first parameter is the integer that you want to verify and the second parameter is the function that formats the data for the first parameter.

These are the valid date formatting functions for the second parameter:

- [portable\(\).dddyyyy\(\)](#)
- [portable\(\).ddmmyyyy\(\)](#)
- [portable\(\).mmddyyyy\(\)](#)
- [portable\(\).mmyyyy\(\)](#)
- [portable\(\).yyyyddd\(\)](#)
- [portable\(\).yyymmm\(\)](#)
- [portable\(\).yyymmd\(\)](#)

If the specified integer is a valid integer (date), the function returns true; otherwise, it returns false.

 **Important** Leading zeros are required in most cases. For example, 20201203 is a valid date in the yyymmd format, but 2020123 is not.

“ddd” represents the dddth day of the specified year.

SRL Example

In this example, the portable().verifyIntDate() function returns true because 20200109 is a valid integer date:

```
boolean1 is a boolean.  
if portable().verifyIntDate(20200109, portable().yyymmd()) is true  
then boolean1 = true.  
else boolean1 = false.  
print("This is the answer: "boolean1).
```

portable().verifyMonth()

Description

This function tests if an integer is a valid month (1 to 12 inclusive).

Implementation

`portable().verifyMonth(integer):boolean`

If the specified integer is a valid month, the function returns true; otherwise, it returns false.

SRL Example

In this example, the function returns false because 15 is not a value between the inclusive range of 1 and 12:

```
boolean1 is a boolean.  
if portable().verifyMonth(15) = true  
then boolean1 = true.  
else boolean1 = false.  
print("This is the answer: "boolean1).
```

`portable().verifyYear()`

Description

This function tests if an integer is a valid year (1300 to 3300 inclusive).

Implementation

`portable().verifyYear(integer):boolean`

If the specified integer is a valid year, the function returns true; otherwise, it returns false.

SRL Example

In this example, the function returns true because 2020 is a value between the inclusive range of 1300 and 3300:

```
integer1 is an integer initially 2020.  
boolean1 is a boolean.  
if portable().verifyYear(integer1) = true  
then boolean1 = true.  
else boolean1 = false.  
print("This is the answer: "boolean1).
```

`portable().year()`

Description

This function returns the year of a real value which represents a date.

Implementation

`portable().year(real):integer`

Because the portable().year function uses an internal representation for dates, you must cast integers to dates (real values) using [portable\(\).intToDate\(\)](#) on page 424 before using the portable().year() function.

SRL Example

This example nests the portable().intToDate() function within the portable().year() function to achieve the desired results:

```
integer1 is an integer initially 20200507.  
integer2 is an integer.  
integer2 = portable().year(portable().intToDate(integer1,  
portable().yyyymmdd())).  
print("This is the answer: "integer2).
```

The result is that integer 2 = 2020.

portable().ymdToDate()

Description

This function converts three integers (year, month, day) to a date.

Implementation

portable().ymdToDate(integer, integer, integer):real

The first parameter is the year, the second parameter is the month, and the third parameter is the day. If the three specified values comprise a valid date, this function returns a date of type real based on the specified values. Otherwise, it returns 0.

For return value to be usable, you must convert it to an integer using [portable\(\).dateToInt\(\)](#) on page 418.

SRL Example

```
integer1 is an integer initially 2020.  
integer2 is an integer initially 12.  
integer3 is an integer initially 25.  
integer4 is an integer.  
integer4 = portable().dateToInt(portable().ymdToDate(integer1, integer2,  
integer3), portable().yyyymmdd()).  
print("This is the answer: "integer4).
```

The result is that integer4 returns 20201225.

 **Note** This function returns 0 if the year is less than 1300.

portable().yyyyddd()

Description

This function returns a an enumeration of the format that is used in the calling date function in a yyyyddd format. "ddd" in this function represents the dddth day of the specified year.

Implementation

portable().yyyyddd():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate (2020325, portable().yyyyddd())
```

portable().yyyymm()

Description

This function returns a an enumeration of the format that is used in the calling date function in a yyyyymm format.

Implementation

portable().yyyymm():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate (202003, portable().yyyymm())
```

portable().yyymmmdd()

Description

This function returns a an enumeration of the format that is used in the calling date function in a yyymmmdd format.

Implementation

portable().yyymmmdd():integer

There are no parameters defined in this function.

SRL Example

```
portable().intToDate (20200314, portable().yyyymmdd())
```

Converting Values with the Portable() Built-in Functions

Several built-in functions can be used to convert values.

portable().charToInt()

Description

This function returns the integer value of a single specific character of a string.

Implementation

portable().charToInt(string, integer):integer

where:

- *string* is the string to convert.
- *integer* is the position to convert.

This function allows you to use the internal (ASCII, EBCDIC, or Unicode) representation of a character. You can use this function to manipulate strings as if they were integers. It is especially useful for looking for patterns in data.

If a string is thought of as a 1-based array of characters, then the return value is the internal integer code for characters(n). If L is the length of string s, then, if n < 1 or n > L, the return value is zero. This means that if s is a null string, then zero is returned.

 **Note** This is a very technical function and is platform-dependent. It is intended for use by those who have a strong understanding of how strings are represented internally by the computer. For example, portable().charToInt("A", 1) returns 65 if the character set is ASCII, but it will return some other value if the character set is EBCDIC.

SRL Examples

```
integer1 is an integer.  
integer1 = portable().charToInt("Bark", 1).  
print("This is the answer: "integer1).
```

The result is that integer1 is 66 which is ASCII representation of "B".

```
integer1 is an integer.  
integer1 = portable().charToInt("Mac", 2).  
print("This is the answer: "integer1).
```

The result is that integer1 is 97 which is an ASCII representation of "a".

 **Note** To look for a contiguous pattern of characters, see the `portable().findString()` function.

Related Links

[portable\(\).findString\(\)](#)

portable().compressString()

Description

This function returns a string with specified characters removed.

Implementation

`portable().compressString(string, string):string`

Use this function to remove all instances of a specific character from a string of characters. This is especially useful for removing spaces from names.

SRL Example

```
string1 is a string initially "tree top".  
string2 is a string initially "t".  
string3 is a string.  
string3 = portable().compressString(string1, string2).  
print("This is the answer: "string3).
```

The result is that String3 = "ree op".

portable().concat()

Description

This function joins one string with another.

Implementation

`portable().concat(string, string):string`

SRL Example

```
string1 is a string initially "John".  
string2 is a string initially "Smith".  
string3 is a string.  
string3 = portable().concat(string1, string2).  
print("This is the answer: "string3).
```

The result is that string3 contains "JohnSmith".

portable().concatByContent()

Description

This function joins one string with another and allows a specific number of spaces to be placed between the concatenated strings.

Implementation

portable().concatByContent(string, string, integer):string

 **Note** The “contents” of a string are the characters up to the last character that is not a space or null value. In addition, the last parameter, that defines the number of trailing spaces after the first string, needs to be zero or a positive value.

SRL Example

```
string1 is a string initially "John".  
string2 is a string initially "Smith".  
string3 is a string.  
string3 = portable().concatByContent(string1, string2, 1).  
print("This is the answer: "string3).
```

The result is that string3 contains “John Smith” with a space between the two strings.

portable().subString()

Description

This function returns a set of characters specified by the (1-based) first and last index to a given string.

Implementation

portable().subString(string, integer, integer):string

The first parameter is the name of the string from which to extract the characters. The second parameter is the integer number that represents the starting position for the extraction. The third parameter is the integer number that represents the ending position for the extraction.

If the string does not contain any data at runtime, this function returns empty spaces. If the beginning position (specified in the first parameter) is less than 1 at runtime, it is changed to a 1, and if the ending position (specified in the second parameter) is greater than the length of the string, then the value is replaced with the maximum length of the specified string.

SRL Example

```
string1 is a string initially "Sales Department".
```

```

integer1 is an integer initially 2.
integer2 is an integer initially 5.
string2 is a string.
string2 = portable().subString(string1, integer1, integer2).
print("This is the answer: "string2).

```

The result is that string2 contains “ales”.

portable().toBoolean()

Description

This function casts an integer, real, or string expression to a boolean.

Implementation

- portable().toBoolean(string):boolean
- portable().toBoolean(real):boolean
- portable().toBoolean(integer):boolean
- portable().toBoolean(boolean):boolean

Use this function to test whether a value contains a true or false based on the data type. The parameter of the portable().toBoolean() function is converted to a boolean. The exact behavior depends on the parameter type:

Type of Value	Value
string	false if x = "false" true if x = "true"
real	false if x = 0.0 true if x is any other value
integer	false if x = 0 true if x is any other value
boolean	false if x is false true if x is true

When a string is cast to boolean, the string must be either “false” or “true”, or a runtime error occurs.

SRL Examples

- portable().toBoolean(11) = true

```

boolean1 is a boolean.
boolean1 = portable().toBoolean(11).

```

The result is that boolean1 = true.

- portable().toBoolean(0) = false

```

boolean1 is a boolean.
boolean1 = portable().toBoolean(0).

```

The result is that boolean1 = false.

- `portable().toBoolean(10.6) = true`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean(10.6).
```

The result is that `boolean1 = true`.

- `portable().toBoolean(0.0) = false`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean(0.0).
```

The result is that `boolean1 = false`.

- `portable().toBoolean("False") = false`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean("false").
```

The result is that `boolean1 = false`.

- `portable().toBoolean("True") = true`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean("true").
```

The result is that `boolean1 = true`.

- `portable().toBoolean(11 = 11) = true`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean(11 = 11).
```

The result is that `boolean1 = true`.

- `portable().toBoolean(11 = 10) = false`

```
boolean1 is a boolean.  
boolean1 = portable().toBoolean(11 = 10).
```

The result is that `boolean1 = false`.

To add a print statement to any of the code samples above, add this line of SRL:
`print("This is the answer: "boolean1).`

portable().toFloat()

Description

This function casts an integer, boolean, or string expression to a real number.

Implementation

`portable().toFloat(string):real`

`portable().toFloat(real):real`

`portable().toFloat(integer):real`

`portable().toFloat(boolean):real`

The parameter is converted to a real. The exact behavior depends on the parameter type:

Type of value	Value
string	The real value of the string.
real	The value does not change.
integer	The value is converted from an integer to a real .
boolean	0.0 if x is False 1.0 if x is True

When the parameter is a string, the valid syntax for a real number is the same as the syntax for a real in Blaze Advisor (which means, among other things, that exponential notation like 10.6e200 can be used). If the string does not have this syntax, it is a runtime error, and processing terminates.

 **Tip** `portable().toFloat()` can be used to test if a string contains a valid real constant.

 **Important** Due to differences in the ways that various hardware and software environments process floating point numbers, the exact result returned when a value is converted to a real may differ slightly from platform to platform.

SRL Examples

- `portable().toFloat(11) = 11.0`

```
real1 is a real.
real1 = portable().toFloat(11).
```

The result is that `real1 = 11.0`.

- `portable().toFloat(10.6) = 10.6`

```
real1 is a real.
real1 = portable().toFloat(10.6).
```

The result is that `real1 = 10.6`.

- `portable().toFloat(-10.6) = -10.6`

```
real1 is a real.
real1 = portable().toFloat(-10.6).
```

The result is that `real1 = -10.6`.

- `portable().toFloat("-10.6e200") = -1.06E201`

```
real1 is a real.
real1 = portable().toFloat("-10.6e200").
```

The result is that `real1 = -1.06E201`

- `portable().toFloat(11 = 11) = 1.0`

```
real1 is a real.  
real1 = portable().toFloat(11 = 11).
```

The result is that `real1 = 1.0`.

- `portable().toFloat(11 = 10) = 0.0`

```
real1 is a real.  
real1 = portable().toFloat(11 = 10).
```

The result is that `real1 = 0.0`.

To add a print statement to any of the code samples above, add this line of SRL:
`print("This is the answer: "real1).`

Related Links

[portable\(\).isFloat\(\)](#)

portable().toInteger

Description

This function casts a real, boolean, or string expression to an integer.

Implementation

`portable().toInteger(string):integer`

`portable().toInteger(boolean):integer`

`portable().toInteger(integer):integer`

`portable().toInteger(real):integer`

The parameter is converted to an integer. The behavior depends on the parameter type:

Type of value	Value
string	The integer value of the string.
boolean	0 if the value is False 1 if the value is True
integer	The value does not change.
real	The real number is converted to an integer with rounding (loss of significant digits possible).

 **Tip** Use the `portable().isInteger()` function to test if a string contains a valid integer constant.

SRL Examples

- `portable().toInteger(11) = 11`

```
integer1 is an integer.
integer1 = portable().toInteger(11).
```

The result is that `integer1` returns 11.

- `portable().toInteger(10.6) = 11`

```
integer1 is an integer.
integer1 = portable().toInteger(10.6).
```

The result is that `integer1` returns 11.

- `portable().toInteger("+11") = 11`

```
integer1 is an integer.
integer1 = portable().toInteger("+11").
```

The result is that `integer1` returns 11.

- `portable().toInteger(11 = 11) = 1`

```
integer1 is an integer.
integer1 = portable().toInteger(11 = 11).
```

The result is that `integer1` returns 1.

- `portable().toInteger(11 = 10) = 0`

```
integer1 is an integer.
integer1 = portable().toInteger(11 = 10).
```

The result is that `integer1` returns 0.

To add a print statement to any of the code samples above, add this line of SRL:
`print("This is the answer: " + integer1).`

Related Links

[portable\(\).isInteger\(\)](#)

portable().toLowerCase()

Description

The function returns all letters contained within the specified string in their lowercase equivalent.

Implementation

`portable().toLowerCase(string): string`

If the string parameter contains data that does not have a lower case equivalent, such as numbers and special characters, those characters are returned unchanged.

SRL Example

```
string1 is a string initially "SEASONAL VEGETABLES".
string2 is a string.
string2 = portable().toLowerCase(string1).
print("This is the answer: "string2).
```

The result is "seasonal vegetables" in all lowercase letters.

portable().toString()

Description

This function casts an integer, real, or boolean expression to a string.

Implementation

- portable().toString():string
- portable().toString(real):string
- portable().toString(boolean):string
- portable().toString(integer):string
- portable().toString(string):string

The parameter of the function is converted to a string. The behavior depends on the parameter type:

Type of Value	Value
boolean	"false" if x is false, and "true" if x is true
integer	x is converted to a string*.
real	x is converted to a string**.
string	x (no change)

*When x is an integer, it is converted to a string with this format:

```
[-]<digit string>
```

The minus sign only appears if the integer is negative.

**When x is a real, it is converted to a string with this format:

```
[-]<digit string>[.<digit string>]
```

The minus sign only appears if the real value is negative. If an invalid parameter is used when casting a string, a runtime error is issued.

! **Important** The exact format used when a floating value is converted to a string may differ slightly from platform to platform. For example, the exponent 20 may appear in any of these forms:

Language	Format
C	e+020

Language	Format
Runtime Server	E+20
Java	E20

SRL Example

The following rules apply to all formats.

- Trailing zero decimal places are suppressed.
- The "-" character is suppressed whenever the value is positive.
- $\text{String}(11) = "11"$

```
string1 is a string.
string1 = portable().toString(11).
```

The result is that $\text{string1} = "11"$.

- $\text{String}(10.6) = "10.6"$

```
string1 is a string.
string1 = portable().toString("10.6").
```

The result is that $\text{string1} = "10.6"$

- $\text{String("11")} = "11"$

```
string1 is a string.
string1 = portable().toString("11").
```

The result is that $\text{string1} = "11"$

- $\text{String}(11 = 11) = "true"$

```
string1 is a string.
string1 = portable().toString(11 = 11).
```

The result is that $\text{string1} = "true"$.

- $\text{String}(11 = 10) = "false"$

```
string1 is a string.
string1 = portable().toString(11 = 10).
```

The result is that $\text{string1} = "false"$.

To add a print statement to any of the code samples above, add this line of SRL:
`print("This is the answer: " + string1).`

portable().toUpperCase()

Description

This function returns all letters contained within the specified string in their uppercase equivalent.

Implementation

`portable().toUpperCase(string):string`

If the string parameter contains data that does not have an upper case equivalent, such as numbers and special characters, those characters are returned unchanged.

SRL Example

```
string1 is a string initially "my Favorite IS 5".
string2 is a string.
string2 = portable().toUpperCase(string1).
print("This is the answer: "string2).
```

The result is that string2 contains "MY FAVORITE IS 5".

Testing Strings and Numeric Values Using Portable() Built-ins

A variety of built-in functions can be used to test strings and numeric values.

portable().findChar()

Description

This function searches a string for the left-most occurrence of one of a set of specified characters and returns its position.

Implementation

`portable().findChar(string, string):integer`

It locates the left-most position of the first character *that matches any of the characters specified*. The first parameter is the string field that you want searched. The second parameter is the set of values to be located. The `portable().findChar()` function returns the integer position of the first character that matches any character in the specified set. If no match is found, it returns zero. If either of the parameters is an empty string, then zero is returned.

SRL Example

This example looks for the first occurrence of the "@" or "." characters in the email address:

```
string1 is a string initially "1234B67a.FICO@fairisaac.com".
integer1 is an integer.
integer1 = portable().findChar(string1, "@.").
print("This is the answer: "integer1).
```

The result is that integer1. contains the number 9, because the "." character is in position 9.

portable().findString()

Description

This function searches a string for the left-most occurrence of a specified substring and returns its position.

Implementation

`portable().findString(string, string) integer`

The first parameter is the string field to be searched. The second parameter is the substring to be located. The function returns the integer position of the first occurrence of the substring within the string specified by the first parameter. If no match is found, it returns zero. If either of the parameters is an empty string, then zero is returned.

SRL Example

The following example looks for the first occurrence of "FICO" in the email address:

```
Object1 is a class1.
Object1.String1 = "1234B67a.FICO@fairisaac.com".
Object1.Integer1 = portable().findString(Object1.String1, "FICO").
print("This is the answer: "Object1.Integer1).
```

The result is that Object1.Integer1 contains the number 10, because the string "FICO" starts in position 10.

 **Note** To run this code sample, replace "class1" in the first line with the name of a class in your project.

portable().is_in()

Description

This function tests if a value is within a range of values.

Implementation

```
portable().is_in (integer, real, integer):boolean
portable().is_in (real, integer, integer):boolean
portable().is_in (integer, integer, integer):boolean
portable().is_in (string, string, string):boolean
portable().is_in (integer, real, real):boolean
portable().is_in (real, integer, real):boolean
portable().is_in (real, real, real):boolean
portable().is_in (real, real, integer):boolean
```

`portable().is_in (integer, integer, real):boolean`

This function checks the first value against the specified minimum and maximum values. It returns true if the value is within the range; otherwise, it returns false.

The first parameter contains the value to be tested. The second parameter is the minimum value, and the third parameter is the maximum value.

 **Note** Use caution when comparing strings because the answer varies by platform.

SRL Example

The following example checks to see if a certain real value is within the specified minimum and maximum values:

```
real1 is a real initially 100.00.  
integer1 is an integer initially 500.  
integer2 is an integer initially 4000.  
boolean1 is a boolean.  
boolean1 = portable().is_in(real1, integer1, integer2).  
print("This is the answer: "boolean1).
```

The result is that boolean1 returns false because real1 is not in between the range of values as set by integer1 and integer2.

portable().isFloat()

Description

This function tests if a string forms a valid real constant.

Implementation

`portable().isFloat(string):boolean`

The return value is true if the string contains a valid real constant; otherwise, false is returned.

 **Note** A valid real constant includes scientific notation, such as 314159e-5.

SRL Example

- `portable().isfloat("- 11") = true`

```
string1 is a string initially ("- 11").  
boolean1 is a boolean.  
boolean1 = portable().isFloat(string1).
```

The result is that boolean1 returns true.

- `portable().isfloat("- 1.1") = true`

```
string1 is a string initially ("- 1.1").  
boolean1 is a boolean.  
boolean1 = portable().isFloat(string1).
```

The result is that boolean1 returns true.

- `portable().isfloat("- 1x") = false`

```
string1 is a string initially ("- 1x").
boolean1 is a boolean.
boolean1 = portable().isFloat(string1).
```

The result is that boolean1 returns false.

- `portable().isfloat("314159e-5") = true`

```
string1 is a string initially ("314159e-5").
boolean1 is a boolean.
boolean1 = portable().isFloat(string1).
```

The result is that boolean1 returns true.

To add a print statement to any of the code samples above, add this line of SRL:
`print("This is the answer: "boolean1).`

portable().isInteger()

Description

This function tests if a string forms a valid integer constant.

Implementation

`portable().isInteger(string):boolean`

The return value is true if the string contains a valid integer constant; otherwise, false is returned.

SRL Example

- `portable().isInteger("- 11") = true`

```
string1 is a string initially ("-11").
boolean1 is a boolean.
boolean1 = portable().isInteger(string1).
print("This is the answer: "boolean1).
```

The result is that boolean1 contains true.

- `portable().isInteger("- 1.1") = false`

```
string1 is a string initially ("-1.1").
boolean1 is a boolean.
boolean1 = portable().isInteger(string1).
print("This is the answer: "boolean1).
```

The result is that boolean1 contains false.

portable().max()

Description

This function returns the larger of two strings.

Implementation

portable().max(string, string):string

It compares two strings and returns the value of the one that is greater.



Note Use caution when comparing strings because the answer will vary by platform.

SRL Example

```
string1 is a string initially "seasonal vegetables".
string2 is a string initially "seasonal vegetable medley".
string3 is a string.
string3 = portable().max(string1, string2).
print("This is the answer: "string3).
```

The result is that string3 is "seasonal vegetables" because it is the higher of the two values. If string2 was changed to "seasonal vegetables medley" then it would be the higher of the two values.



Note If you want to compare two integers or real numbers, use math().max().

Related Links

[math\(\).max\(\)](#)

portable().min()

Description

This function returns the smaller of two strings.

Implementation

portable().min(string, string):string

It compares two strings and returns the value of the one that is less.



Note Use caution when comparing strings because the answer will vary by platform.

SRL Example

```
string1 is a string initially "seasonal vegetables".
string2 is a string initially "seasonal vegetable medley".
string3 is a string.
string3 = portable().min(string1, string2).
print("This is the answer: "string3).
```

The result is that string3 is "seasonal vegetable medley" because it is the lower of the two values. If string2 was changed to "seasonal vegetables medley" then string1 would be the lower of the two values.

 **Note** If you want to compare two integers or real numbers, use `math().min()`.

Related Links

[math\(\).min\(\)](#)

portable().not_in()

Description

This function tests if a value is not in a range of values.

Implementation

```
portable().not_in(integer, real, integer):boolean
portable().not_in(real, real, real):boolean
portable().not_in(integer, integer, integer):boolean
portable().not_in(real, integer, real):boolean
portable().not_in(real, real, integer):boolean
portable().not_in(string, string, string):boolean
portable().not_in(real, integer, integer):boolean
portable().not_in(integer, real, real):boolean
portable().not_in(integer, integer, real):boolean
```

This function checks the first value against the specified minimum and maximum values. It returns true if the value is not within the range; otherwise, it returns false.

The first parameter contains the value to be tested. The second parameter is the minimum value, and the third parameter is the maximum value.

 **Important** Use caution when comparing strings because the answer will vary by platform.

SRL Example

The following example checks to see if the value in `real1` is outside of the specified minimum and maximum values.

```
integer1 is a real initially 100.
integer2 is an integer initially 500.
integer3 is an integer initially 4000.
boolean1 is a boolean.
boolean1 = portable().not_in(integer1, integer2, integer3).
print("This is the answer: "boolean1).
```

The result is that `boolean1` is true because 100.00 is not between 500 and 4000.

portable().selectValue()

Description

This function returns one of two numbers or strings based on the value of a boolean expression.

Implementation

```
portable().selectValue(boolean, real, real):real  
portable().selectValue(boolean, real, integer):integer  
portable().selectValue(boolean, string, string):string  
portable().selectValue(boolean, integer, integer):integer  
portable().selectValue(boolean, integer, real):real
```

If the Boolean expression is true, then the return value is the second parameter. Otherwise, the return value is third parameter.

SRL Example

```
integer1 is an integer initially 20.  
integer2 is an integer initially 12.  
real1 is a real initially 12.5.  
real2 is a real.  
real2 = portable().selectValue(integer1 > 15, integer2, real1).  
print("This is the answer: "real2).
```

Because the expression is true, the value of integer2, which is 12, is returned as a real value of 12.0.

This is a string example.

```
string1 is a string initially "Bill".  
string2 is a string initially "Smith".  
string3 is a string initially "Jones".  
string4 is a string.  
string4 = portable().selectValue(string1 = "Bill", string2, string3).  
print("This is answer: "string4).
```

Because the expression is true, "Smith" is returned.

Standard Built-in Functions

You can use these built-in functions to perform specific operations.

In the Item Selector, these functions are displayed at the end of the Built-in Functions categories

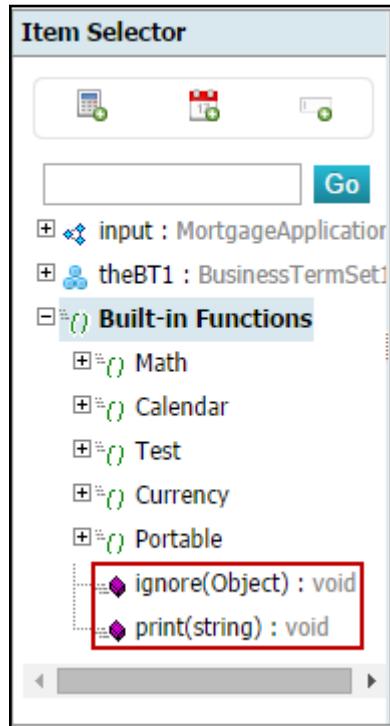


Figure 60: The Item Selector displaying the standard built-in functions

ignore()

Description

By default, you are warned about unused patterns in rule actions. The `ignore()` function can be used to instruct the rule engine to repeat the rule action for each pattern match that satisfies the rule condition even though the pattern match is not referenced in the action. This eliminates the pattern warning.

Implementation

```
ignore(Object)
```

The object is usually a local pattern in a decision entity that supports patterns such as a ruleset.

SRL Example

```
if anyAccount.interestRate > 1.05
then {
  ignore(anyAccount).
  print("Review interest rate.").
}
```

Related Links

- [Unused Pattern Warnings in Rules](#)
- [Creating Patterns](#)

print()

Description

This function prints a string. The processing is handled by a registered execution handler.

Implementation

```
print(string)
```

The string is what will be printed.

SRL Example

```
if newApp.recommendedCard is not null
    then print("You are eligible for a " newApp's recommendedCard " card and you
will receive a " newApp's gift").
    else print("You are not eligible for a card at this time. Please contact one
of our representatives.").
```

Scorecards and Reason Codes

The scorecard and reason code support classes provide a set of properties and methods to support scorecards and the reason codes used by scorecards.

NdReasonCode

The class NdReasonCode provides a set of properties that support reason codes. A reason code provides an explanation as to why a particular score was reached.

An NdReasonCode class includes these properties:

Property Name	Type	Description
code	string	Contains the reason code associated with the score
message	string	Contains the message associated with the reason code
name	string	Name of the reason code. The name property must be alphanumeric and not include any spaces.
rank	integer	Rank associated with the reason code

Example

```

newReason is an NdReasonCode;
newReason.code = "RC12";
newReason.message = "Income is too low";
newReason.name = "MinimumIncome";
newReason.rank = 10;

```

NdScoredCharacteristic

The class NdScoredCharacteristic provides a set of properties that supports each variable in a scorecard.

An NdScoredCharacteristic class includes these properties:

Property Name	Type	Description
baselineScore	real	Baseline score for the variable
binLabel	string	Description of the bin that the value fell into
characteristicName	string	Name of the variable
maxBinScore	real	Maximum score for a bin, if allow expressions is not set
partialScore	real	Partial score for a variable
reasonCode	NdReasonCode	Reason code returned for a variable
unexpected	boolean	Sets flag that indicates whether or not the score was unexpected
weight	real (read-only)	<p>The weight is used in calculating the top reasons. However, the weight means something slightly different based on the reason calculation options as follows:</p> <ul style="list-style-type: none"> ■ If reasons are not calculated, the weight is zero. ■ If reasons are calculated using ranks, the weight is the rank. ■ If reasons are calculated using score distances, the weight is the calculated distance.

Example

```

newCharacteristic is an array of NdScoredCharacteristic;
it[0] = an NdScoredCharacteristic initially {
    newCharacteristic.baselineScore = "15";
    newCharacteristic.binLabel = "Low Income";
    newCharacteristic.maxBinScore = 25;
    newCharacteristic.characteristicName = "Income";
    newCharacteristic.reasonCode = reasoncodelist.name1.
    newCharacteristic.partialScore = 10;
    newCharacteristic.unexpected = true;
}
it[1] = an NdScoredCharacteristic initially {

```

```

newCharacteristic.baselineScore = "10";
newCharacteristic.binLabel = "DebtRatio";
newCharacteristic.maxBinScore = 20;
newCharacteristic.characteristicName = "Debt";
newCharacteristics.reasonCode = reasonodelist.name2;
newCharacteristic.partialScore = 5;
newCharacteristic.unexpected = false;
}

```

NdScoreModelReasonCalculationOptions

The class NdScoreModelReasonCalculationOptions provides a set of properties that support the reason code calculation options in a scorecard.

Property Name	Type	Description
dedupeMethod	integer	Method for resolving duplicate reason codes using score reason differences. The default is DEDUPE_BY_NONE
includeOnlyPositiveDistances	boolean	When set to true, only data with positive distances (distance >=0) are used. When set to false, data with all distances are used. The default is true.
numberOfReasons	integer	Maximum number of reasons to return. This is only applicable when score reasons will be returned. The default is 4.
precalculateScore	boolean	Calculates the score and returns score reasons within the score model ruleset. This option is primarily used with COBOL score models, but can be used with other object model types.
sortOrder	integer	Used to sort reason codes by distance. The default is SORT_ORDER_NONE
allowExpressions	boolean	When set to true, it allows the use of custom SRL for computing the partial score of a bin. The default is false.
distanceComputationMethod	integer	Method for resolving duplicate reason codes by computing score distances. The default is DISTANCE_USE_RANK

An NdScoreModelReasonCalculationOptions class includes these read-only static, integer properties.

Property Name	Used with this property	Description
DEDUPE_BY_MAX	dedupeMethod	Indicates that reason codes are to be returned using the “maximum distance from code”. Duplicate reason codes are based on distance. This is not applicable if allowExpressions is set to true.
DEDUPE_BY_NONE	dedupeMethod	Indicates that reason codes are to be based on rank.

Property Name	Used with this property	Description
DEDUPE_BY_SUM	dedupeMethod	Indicates that reason codes are to be returned using the “summed distance from code”. Duplicate reason codes are summed.
SORT_ORDER_ASCENDING	sortOrder	Indicates that reason codes should be sorted using distance in ascending order.
SORT_ORDER_DESCENDING	sortOrder	Indicates that reason codes should be sorted using distance in descending order.
SORT_NONE	sortOrder	Indicates that reason codes are to be based on rank.
DISTANCE_BASELINE	distanceComputationMethod	Sets the score to be used to compute the distance as the user-defined baseline score
DISTANCE_MAXIMUM	distanceComputationMethod	Sets the score to be used to compute the distance as the maximum score
DISTANCE_USE_RANK	distanceComputationMethod	Indicates that reason codes are to be based on rank.

Example

```

reasonOptions is an NdScoreModelReasonCalculationOptions initially {
    dedupeMethod = NdScoreModelReasonCalculationOptions.DEDUPE_BY_SUM,
    includeOnlyPositiveDistances=true,
    numberofReasons=2,
    sortOrder=NdScoreModelReasonCalculationOptions.SORT_ORDER_ASCENDING,
    allowExpressions = false.

    distanceComputationMethod=NdScoreModelReasonCalculationOptions.DISTANCE_FROM_BAS
ELINE .
}

```

NdScoreModelReturnInfo

The class NdScoreModelReturnInfo provides a set of properties and methods that support the return information from a scorecard.

An NdScoreModelReturnInfo class includes these properties:

Property Name	Type	Description
characteristicCount	integer	The number of variables in the scorecard.
initialScore	real	Initial score, which currently cannot be set in the wizard or template. This can be set to add a constant to the score (sometimes called a “slope”).
options	NdScoreModelReasonCalculationOptions	Contains the specifications for the calculating reason codes in the scorecard

Property Name	Type	Description
score	real	The output of the scorecard. It is the “value property” of the NdScoreModelReturnInfo object.
scoredCharacteristics	fixed array of NdScoredCharacteristic	Array of objects that contains the details for the characteristics.
scoremodelName	string	Name of the scorecard used to produce the score
topReasons	fixed array of NdReasonCode	Contains the top reasons for the score determined by the last call to calculateReasons(). This property can also be used to pre-set an existing array in which to store the top reasons on the next call to calculateReasons(). This can improve performance by avoiding the internal creation of a new array over repeated calls.
unexpectedCount	integer	Number of variables where the unexpected flag is set to true.

addReason(NdReasonCode, real)

This method takes an input of type NdReasonCode and a real number, which either represents rank or distance depending upon whether reason codes are being returned by using score reason ranks or by computing score reason differences.

This method is used to add a reason code and a rank or distance (which is not related to any specific scorecard variable) to the score and reason calculations for a scorecard. The reason code and the rank or distance are added as if they were defined by a bin that was selected in a variable in the scorecard, but they come from some "external source."

```

//query for applicant information
applicant is a CardApplicant initially {
    age = promptInteger("What is your age?").
    income = promptInteger("What is your income?").
}

//call score model based on applicant's age
if( applicant.age < 30)
then {
    scoreResult = apply ScoreModel_Under30( applicant).
    scoreResult.addReason(ReasonCodes.under30,ReasonCodes.under30.rank).
}
else {
    scoreResult = apply ScoreModel_Over29( applicant).
    scoreResult.addReason(ReasonCodes.over29,ReasonCodes.over29.rank).
}

//print out the score
print("Total score = " scoreResult.score);

//get the reasons
reasons is some fixed array of NdReasonCode.

```

```

reasons = scoreResult.calculateReasons().

//print out offer results
if (scoreResult.score = 10)
then print("No offer: score is " reasons[1].message).
else if (scoreResult.score >= 20)
then print ("Offer: " reasons[0].message " special ").
else print ( "No offer: score is suspicious").

//print the reason details
print ("Top Reasons:").
for each NdReasonCode in reasons do {
    print ("Rank is " it.rank " Code is" [ " it.code " ] "
"Message is "
it.message).
}

```

This is input data:

```

Question: What is your age?
Answer: 29
Question: What is your income?
Answer: 40000

```

This is the output:

```

Total score = 30.0
Offer: under 30 special
Top Reasons:
Rank is 1 Code is [ RC1 ] Message is under 30
Rank is 5 Code is [ RC4 ] Message is High

```

If the two scorecards, referenced in this example, were returning score reasons based on computing score reason differences, an appropriate distance value instead of the rank would have been supplied. The distance-specific options (`NdScoreModelReasonCalculationOptions`) that were set for the `scoreResult` would be applied to the distance value.

Related Links

[NdScoreModelReasonCalculationOptions](#)

calculateReasons(NdScoreModelReasonCalculationOptions)

This method takes an input of type `NdScoreModelReasonCalculationOptions` and returns a fixed array of `NdReasonCode`.

```

if true
then {
    print("This is the applicant's score: " scoreResult.score).

    newOptions is an NdScoreModelReasonCalculationOptions.
    newOptions.dedupMethod =
NdScoreModelReasonCalculationOptions.DEDUPE_BY_SUM;
    newOptions.includeOnlyPositiveDistances = true;
    newOptions.numberofReasons = 4;
    newOptions.sortOrder =

```

```
NdScoreModelReasonCalculationOptions.SORT_ORDER_DESCENDING;
    newOptions.allowExpressions = false;
    newOptions.distanceComputationMethod =
NdScoreModelReasonCalculationOptions.DISTANCE_MAXIMUM;
    reasons = scoreResult.calculateReasons(newOptions).
    for each NdReasonCode in reasons do {
        print ("Rank is " it.rank " Code is" [ " it.code " ] "
"Message is "
it.message).
    }
}
```

Related Links

[NdReasonCode](#)

calculateReasons()

This method returns a fixed array of NdReasonCode. You can use this method to print out the scorecard reasons.

```
if theApplicant.creditLimit is greater than 7500
then {

    print("This is the applicant's score: " scoreResult.score).
    reasons = scoreResult.calculateReasons().
for each NdReasonCode in reasons do {
    print ("Rank is " it.rank " Code is" [ " it.code " ] " "Message is "
it.message).

}
```

Related Links

[NdReasonCode](#)

APPENDIX A

Exporting FICO® Blaze Advisor Projects

You can export a Blaze Advisor 7.5, 7.6.x, or 7.7.x project, workspace, or repository as a ZIP file and upload it to a FICO® Decision Modeler workspace. If the ZIP file is exported from Blaze Advisor 7.7, you can select up to a maximum of five projects to import into the Decision Modeler workspace. If the ZIP file is exported from Blaze Advisor 7.5 or 7.6, you can select one project to import into a Decision Modeler workspace. You can edit the projects and deploy them as web services, also known as decision services.

Related Links

- [Deciding Which Projects to Export](#)
- [Preparing Blaze Advisor Projects for Export](#)
- [Exporting Blaze Advisor Projects Using a Wizard](#)
- [Importing Decision Modeler Projects into Blaze Advisor](#)

Deciding Which Projects to Export

The first step is to decide which Blaze Advisor project or projects you want to edit and deploy in Decision Modeler. The next step is to determine if any of those projects include references to other projects or to folders in other projects. If so, the referenced projects will also need to be exported.

When the ZIP file is uploaded to a Decision Modeler workspace, you must explicitly specify the Blaze Advisor projects that you want to make available in Decision Modeler. Those projects become Decision Modeler projects and can be later deployed as decision services.

Just because a Blaze Advisor project is in a Decision Modeler workspace does not mean that you have to make it available. Any project can be hidden in a Decision Modeler workspace, which means that the projects are not editable. Note that hidden projects can still be included in Decision Modeler projects using the **Project Configuration** window. In that case, any hidden project that is included in another Decision Modeler project becomes editable. For best results, export only those Blaze Advisor projects that you want to edit or include in other projects in the Decision Modeler workspace.

If only one Blaze Advisor project will be exported, you can export it as a single project. If more than one project will be exported, you must export the projects in a Blaze Advisor workspace or a repository. The choice as to whether to export the projects in a workspace or repository will depend on the repository type.

You must export the projects in a workspace if the projects are stored in any of the following repository types:

- File repositories without versioning
- File repositories with BVS-shared workspace
- Database repositories
- Subversion repositories

For projects in a BVS-private workspace, you can export the projects in a workspace or a repository.

When projects in a Blaze Advisor versioned repository are exported there are implications regarding the version history.

- When a Blaze Advisor workspace is exported, the version history is not included for the projects, but any local changes are included.
- When a Blaze Advisor repository is exported, the version history is included, but not any local changes. If you need the version history for all projects, check in any local changes.

 **Note** When you upload the ZIP file to Decision Modeler, explicit versioning commands are used for version management.

Related Links

[Exporting FICO Blaze Advisor Projects](#)

Preparing Blaze Advisor Projects for Export

The minimum requirement for exporting a Blaze Advisor project and importing it into Decision Modeler is that the project must have a Rule Service Definition (RSD) with an entry point that returns a primitive or Java class or XML class type. Other object model types such as COBOL or .NET are not supported.

To ensure that your Blaze Advisor projects are imported successfully, become familiar with the requirements for business object models, entry points, and deployment entities. This is also an excellent opportunity to review your content to see if all of it is needed in Decision Modeler and to address any deprecated functionality. Any functionality deprecated in Blaze Advisor is not available in Decision Modeler. Like the RMA, Decision Modeler uses HTML 5 and does not support applets so any instances of Flow, Decision Graph, and Classic Decision Trees Templates are not supported for viewing or editing.

Related Links

[Exporting FICO Blaze Advisor Projects](#)

[Reviewing Your Business Object Models](#)

[Entry Points in Rule Service Definitions](#)

[Excluding Content When Exporting Blaze Advisor Projects](#)

[Addressing Unsupported Functionality](#)

Reviewing Your Business Object Models

If a Blaze Advisor project has an XML or Java Business Object Model (BOM) or both types, the BOMs must meet the requirements and specifications for an object model in a web service. See Deploying Rule Services in the Blaze Advisor Help system or in the `DeployingRuleServices.pdf` file.

There are some requirements that apply to the BOMs in exported Blaze Advisor projects.

XML Business Object Models

If a Blaze Advisor project uses an XML BOM, the schema must include a target namespace or an error occurs in Decision Modeler.

Java Business Object Models

Package all external classes referenced by the projects into one or more JAR files. Use this opportunity to review your external classes and determine which classes are needed and which ones can be removed.

All Java classes must have unique class names even if the Java classes are in different packages. In Blaze Advisor, when a class is added to the Java Business Object Model (BOM), the **SRL Package/Class** name is derived from the name of the Java class without the package. For example, `com.example.package.Variable` is referred to locally as `Variable`. Therefore, you cannot import two classes with the same class name, even if they are in different packages.

If you do not want to change a class name in the source Java files, you can change the **SRL Package/Class** name using the **Java Business Object Model Wizard**. After your changes have been made, update your decision logic to reference the classes with the modified names and compile the project.

Related Links

- [Preparing Blaze Advisor Projects for Export](#)
- [Avoiding Security Manager Exceptions with ObjectMapper](#)
- [Setting the Classpath for Exported Projects](#)

Avoiding Security Manager Exceptions with ObjectMapper

If ObjectMapper is used in a project with a Java Business Object Model (BOM) that you want to export, ensure that ObjectMapper is configured to disable the feature that tries to set accessors on the class.

Due to security manager exceptions, ObjectMapper must be configured *not* to set accessors at the class level by configuring the ObjectMapper as follows:

```
ObjectMapper oMapper = ObjectMapperFactory.getInstance();
oMapper.configure(MapperFeature.CAN_OVERRIDE_ACCESS_MODIFIERS, false);
```

Related Links

[Reviewing Your Business Object Models](#)

Setting the Classpath for Exported Projects

All external Java classes used in Blaze Advisor projects that will be exported to Decision Modeler must be in JAR files. Any JAR files used in a project BOM must be referenced in the project classpath.

Use this information to set the project classpath before starting the export operation.

- Add the JAR file references to the project classpath in the project. (In Blaze Advisor, select **Project > Properties** to open the window.)
- If a JAR file includes a Java BOM, ensure that the classpath settings have been updated in the Java BOM wizard.
- To avoid conflicts, remove references to the classes from the global classpath. (In Blaze Advisor, select **Windows > Preferences > Blaze Advisor** to open the window.)

Related Links

[Reviewing Your Business Object Models](#)

Entry Points in Rule Service Definitions

For a Blaze Advisor project to be available in a Decision Modeler workspace, the project must contain one or more entry points and at least one Rule Service Definition (RSD). For more information about RSDs, see "Deploying Rule Services" in the Blaze Advisor help system or in *DeployingRuleServices.pdf*.

Entry Points

Follow these guidelines to ensure that your project can be used in Decision Modeler.

- The entry point parameters used in the Rule Service Definition must correspond to the Business Object Model (BOM) in the project or be a primitive type.
 - For XML Document Web Service deployments, the type of the object or array of object parameters must be only XML BOM types.
 - For Java POJO Web Service deployments, the object or array of object parameters must be only Java BOM types, such as Java classes or Java enums.
- All entry points must have names that are unique in the Blaze Advisor workspace or repository.

 **Note** The **Export to FICO Analytic Cloud** wizard does not provide validation for all projects in a repository or workspace. An error is displayed only if a Rule

Service Definition does not exist in the selected project, or in at least one project in the selected workspace or repository. For best results, ensure that all projects that you want to use in Decision Modeler meet these requirements.

Related Links

[Preparing Blaze Advisor Projects for Export](#)

Excluding Content When Exporting Blaze Advisor Projects

When exporting a Blaze Advisor project, you can exclude content from the project using a Rule Maintenance Definition (RMAD) and/or a project-level filter.

RMAD

If there is exactly one RMAD in a project, the folders listed in the **Excluded Content** window of the **Navigation** tab are excluded when the project is exported. If a project does not contain an RMAD or contains more than one RMAD, no folders are excluded.

Project-Level Filters

If a project includes a project-level filter when the project is exported, the filter is applied. If the filter is in a folder that will be excluded using the RMAD, then the filter will not be applied. Any filters in a project will be available for editing in Decision Modeler; however, filters cannot be applied or cleared.

If you decide later that you want to include or exclude some of the filtered content in the project, you must download the workspace or repository from Decision Modeler and import the project into Blaze Advisor, clear the filter, and make the necessary changes. You will then need to export the Blaze Advisor project, workspace, or repository again and upload the ZIP file to the Decision Modeler workspace.



Note For best results, review the content in your Blaze Advisor projects before uploading the ZIP file to a Decision Modeler workspace.

Related Links

[Preparing Blaze Advisor Projects for Export](#)

Addressing Unsupported Functionality

While most of the functionality you need to edit and deploy your projects is available in Decision Modeler, some older functionality is not supported. If any of your Blaze Advisor projects contain unsupported functionality, you may want to consider modifying your projects before exporting them. For example, Classic Decision Tree instances are not supported but migrated decision trees are supported in Decision Modeler.

This functionality is not supported in Decision Modeler:

- Entry points of class type with an enumeration property that is used as a parameter or return type are currently not supported for projects with XML BOMs.
- Decision Simulator projects.
- Business User Projects (projects created in the Blaze Advisor RMA).
- Editing Decision Graph, Classic Decision Tree, or Flow Template instances.
- Database Providers.
- Applying filters or creating new ones.
- Editing fixed decision entities, templates, and providers.

Related Links

[Preparing Blaze Advisor Projects for Export](#)

Exporting Blaze Advisor Projects Using a Wizard

Use the **Export to FICO Analytic Cloud** wizard in the Blaze Advisor IDE to export a project, workspace, or a repository as a ZIP file that can be uploaded to a Decision Modeler workspace.



Note If you want the ZIP file automatically uploaded to Decision Modeler, ensure that the URL for Decision Modeler is available in the wizard.

For best results, FICO recommends that you have at least one compilable project containing a RSD with a valid entrypoint in your workspace or repository.

- 1 In Blaze Advisor, open a project from the **Project Explorer**. The project you use to open the export wizard is treated as the host project.
- 2 Select **Project > Export to FICO Analytic Cloud**.
- 3 Select one of these options:
 - **Current project**
Exports the project in the current project context.
 - **Workspace**
This selection is available if the project is stored in a versioned or non-versioned repository. This export does not include version history but

includes local changes. If a workspace contains more than one project, any number of projects can be exported to a ZIP file, however; only those projects containing an RSD with a valid entry point are eligible for import into Decision Modeler.

- **Repository (includes version history but not local changes)**

This selection is enabled if the project is stored in a repository that is a File repository (BVS versioning-private) type. This export includes version history and any changes that have been checked into the repository. If a repository contains more than one project, any number of projects can be exported to a ZIP file, however; only those projects containing an RSD with a valid entry point are eligible for import into Decision Modeler.

4 Select an export target:

- **FICO Analytic Cloud**

Opens a new window with the FICO Analytic Cloud log on page.

- **File system (the .zip file can then be uploaded to the FICO Analytic Cloud)**

Specify a location on your local machine where you want to add the ZIP file.

5 Your export target determines what you need to do next:

- If you selected FICO Analytic Cloud as your export target, click **Next**.

- If you selected File system as your export target:

- a In the ZIP file location field, enter a location or click the ellipsis button to browse to a local file directory where you want your file to be added.

- b (Optional) Modify the name of the ZIP file in the location field.

- c Click **Finish**.

- d Follow the progress of the ZIP file generation in the Console.

If you have more than one RSD, you see a dialog where you can select the RSD that you want to include in your ZIP file.

6 Click **OK.**

Related Links

[Exporting FICO Blaze Advisor Projects](#)

Importing Decision Modeler Projects into Blaze Advisor

If you need to edit a template or a fixed decision entity in your Decision Modeler project, download the workspace or repository from Decision Modeler. You can connect to the workspace or repository and import the project into Blaze Advisor 7.7.x.

If you have multiple projects in a Decision Modeler workspace or repository and you have made any edits to them using Decision Modeler, import all of the projects into a Blaze Advisor workspace or repository. Make your edits to the project that requires editing in Blaze Advisor and then use the wizard to export the Blaze

Advisor repository or workspace in a ZIP file. Uploading the ZIP file will overwrite all projects in the Decision Modeler workspace.

These operations are not supported in Blaze Advisor for projects downloaded from Decision Modeler:

- Compiling or running a project with a decision tree that contains multiple action expressions.
- Compiling or running a project with an imported SAS model.
- Editing template instances in the Blaze Advisor IDE or in the RMA.

If you need assistance, contact FICO Support.

Related Links

[Exporting FICO Blaze Advisor Projects](#)

APPENDIX B

Upgrading Decision Modeler Component Instances

Upgrade a component that you created in a previous version of Decision Modeler using the Decision Management Platform UI.

Upgrading Decision Modeler Component Instances 3.1 or Later

If you are an owner or a system administrator, you can upgrade a component containing a single project or multiple projects. If a component upgrade is available, an Upgrade icon appears in the Name column on the Component page in the DMP.

To upgrade a component that you created using Decision Modeler 3.1 or later to the most recent version of Decision Modeler, you need to use the information in this section.

Preparing Your Component for Upgrade

To prepare a component created using Decision Modeler 3.1 or later for upgrade, ensure that your configured component is submitted to the Staging environment.

Submitting your component to the Staging environment, allows the configured version to be preserved as a revision in the event you need to rollback your upgrade.

Upgrading the Component

Use the **Upgrade** command in the drop-down menu to upgrade the component in the Development environment.

The upgrade process updates the underlying system folder and binary files. Any settings such as those selected in the **Teams** window or the **Togglz Admin Console** are preserved during the upgrade process. For example, if you selected Isolated Environments mode for lifecycle management in the **Teams** window, this setting is preserved during the upgrade.

If you have an instance of the component in Staging or Production, after the upgrade is completed, you can resubmit the component to Staging and if necessary, republish the component to Production.

Rolling Back an Upgraded Component

Use the **Rollback** command in the drop-down menu to revert to the most recent revision in the Development environment.

If you have an instance of the component in Staging or Production, after the rollback is completed, you can resubmit the component to Staging and if necessary, republish the component to Production.



Note Rollback is not recommended if the most recent revision is a upgrade.

Upgrading Decision Modeler Component Instances 3.0 or Earlier

To upgrade a component that you created using Decision Modeler 3.0 or earlier to the latest version of Decision Modeler, you need to use the information in this section:

After the upgrade process, you can edit and manage your Decision Modeler component in the Decision Modeler Projects Page or using the Decision Management Platform UI.

Related Links

[Downloading the Decision Logic and Noting the Component Settings](#)

[Creating a New Decision Modeler Component in Your Solution](#)

[Specifying the Component Settings and Uploading the Decision Logic to the New Component](#)

[Deleting the Older Decision Modeler Component](#)

Downloading the Decision Logic and Noting the Component Settings

Before you create a new Decision Modeler component, FICO recommends that you download the decision logic and note the Togglz and Configure Data Reporting settings for your existing component.

Downloading the Decision Logic in a ZIP File to Your Desktop

Before downloading the decision logic from your existing Decision Modeler component, it is recommended that you compile and test your decision logic to ensure that it is generating the correct results. In the unlikely event that there is an issue with the same decision logic in the new Decision Modeler component, your older Decision Modeler component can serve as baseline for any conversations with FICO Support.

Once you know that your decision logic is correct, download a ZIP file containing the decision logic in a workspace or in a repository. When you download the decision logic in a workspace, the decision logic includes local changes but not the version history. When you download the decision logic in a repository, the decision logic includes the version history but not any local changes. If you need the version history preserved, check in any local changes to the versioning service first and then download the decision logic in a repository.

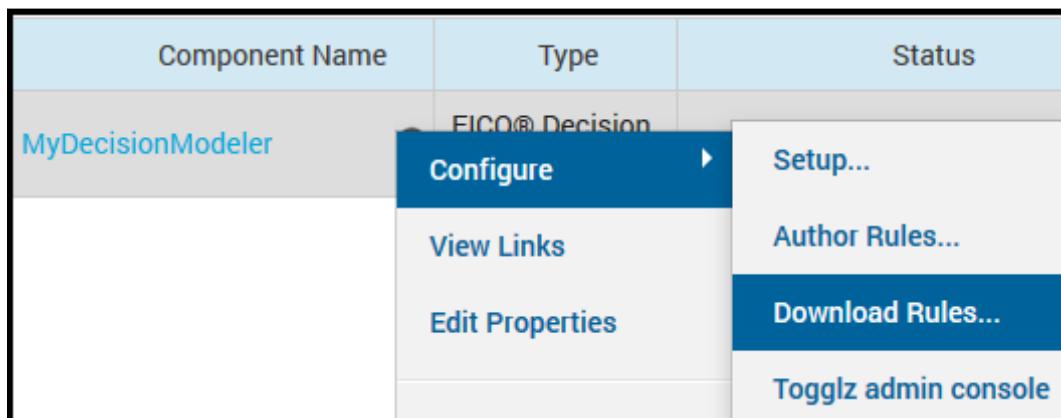


Figure 61: The command used to download the decision logic

By default, when the ZIP file is generated, it always has the name `DecisionExecutor_workspace` or `DecisionExecutor_repository` and the file is saved to your Downloads directory.

- 1 Launch the Decision Management Platform.
- 2 Click the **Solutions** tab.
- 3 Locate your solution and click the link to open it.
- 4 Click **Components** in the left-hand panel.
- 5 Select an existing Decision Modeler component, click the component menu, and select **Configure > Download Rules**.
- 6 Select whether to download the decision logic in a workspace or in a repository.
- 7 Click **Download Rules**.

- 8 Save the file to a folder.

Note the location of the ZIP file because you will need to know where to access it when using the configuration wizard with your new Decision Modeler component.

Noting the Togglz Admin Console Settings

By default, there are settings for logging event data to the FICO Analytic Datamart and for including a wrapper element for REST endpoints. If you want to maintain the same settings in a new Decision Modeler component, open the Togglz admin console and note the settings that are being used for your existing component.

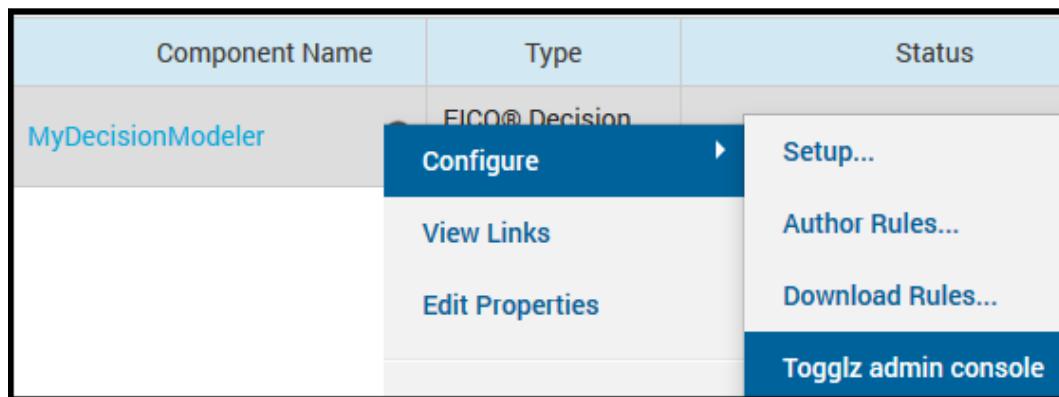


Figure 62: Command to open the Togglz admin console

Some settings in the Togglz admin console that were there in prior releases have been removed because they are no longer needed.

- The **Function Generation on Commit** setting was removed because it is now enabled automatically. Enhanced Batch Mode requires that functions be generated on commit so that they are accessible from the batch component in Staging and Production instances.
- The **Import Project Mode Beta** setting was removed because this functionality is now part of Decision Modeler.

For more information about the settings, see [Togglz Admin Console Settings](#) on page 33.

- 1 Select an existing Decision Modeler component, click the component menu, and select **Configure > Togglz admin console**.
- 2 Note any settings that are disabled.
- 3 Close the console.

Noting the Configure Data Reporting Settings

If the FICO Analytic Datamart Event Logging setting is enabled in the Togglz admin Console, review the settings in the **Configure Reporting Data** window so you can specify these same settings for your new Decision Modeler component.

For more information about the settings, see *Configure Component Reporting Data* in the *Decision Management Platform User Guide*.

Endpoint	Environment		
	DEV	STG	PRD
ALL ENDPOINTS	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
CollegeApplicationProcessing	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 63: Default settings for a new component

- 1 Select an existing Decision Modeler component, click the component menu and select **Configure Reporting Data**.
- 2 Note the settings for each option and close the dialog.

If the menu command is not available, this means that the component does not have valid endpoints or that the FICO Analytic Datamart Event Logging setting is disabled in the Togglz admin console.

Creating a New Decision Modeler Component in Your Solution

Create the new Decision Modeler component in your existing solution using the Decision Management Platform (DMP) Catalog. Note that you cannot have more than one component with the same name in the same solution. When the older component is later deleted, you can select Edit Properties from the component menu and change the name of the new component.

Creating a New Component

Complete the following steps to create a new Decision Modeler component in your solution.

- 1 Click **Add Component**.
- 2 Locate the Decision Modeler component type in the catalog and click **Add**.
- 3 Enter a unique name for the Decision Modeler component.
- 4 Click **Choose your solution** and select the solution name.
- 5 Click **Save**.
You see the Solutions page.
- 6 Open the solution.
- 7 Click **Components** in the left pane. You see that your component is being provisioned. It may take a few minutes for the operation to be completed.

Specifying the Component Settings and Uploading the Decision Logic to the New Component

To upgrade your new component, complete the tasks in this section in sequential order.

Specifying the Togglz Admin Console Settings

If you want to disable any of the Togglz admin console settings, it is important that you do this before uploading your decision logic to the new component; otherwise, this will require additional work for you or another team member.

If you disable any of the Togglz settings after uploading the decision logic, you will need to do the following:

- Open the project configuration wizard, navigate through all wizard pages without changing the settings, and click **Done** and then click **Close** to recreate the decision service.
- Resubmit, redeploy, and restart all component instances so that the changes take effect in the Staging and Production stages of the decision service lifecycle.

Specifying the Togglz Admin Console Settings

- 1 Select the new Decision Modeler Component, click the component menu and select **Configure > Togglz admin console**.
- 2 Use the same settings and the older Decision Modeler component or select new settings.
- 3 Click **Done**.

Uploading the Decision Logic to the New Decision Modeler Component

To use your decision logic in the new Decision Modeler component, upload the ZIP file you downloaded from the older Decision Modeler component to the new component.

- 1 For the new Decision Modeler component, click the component menu and select **Configure > Import Projects**.
- 2 In the wizard, click **Browse** to open a dialog where you can select the ZIP file.
- 3 Select the ZIP file and click **Open**.
- 4 Select the project and click **Next**.
- 5 Review the entry points. The entry points are used to provide data for evaluation. The entry points that are displayed were configured in the decision logic that was downloaded from the older component.
You can change the entry points when you are finished importing the project by selecting **Configure > Configure Project > yourProjectName**.
- 6 Click **Import**.
- 7 Click **Close**.

Specifying the Configure Data Reporting Settings

Complete this procedure if the FICO Analytic Datamart Event Logging setting is enabled in the Togglz admin console. For more information about the settings, see *Configure Component Reporting Data* in the *Decision Management Platform User Guide*.

- 1 Select the new Decision Modeler component, click the component menu, and select **Configure Data Reporting**.
- 2 Use the same settings as the older Decision Modeler component or select new settings.
- 3 Click **Done**.
If the menu command is not available, this means that the component does not have valid endpoints or that the FICO Analytic Datamart Event Logging settings is disabled in the Togglz admin console.

Testing and Managing the New Decision Modeler Component

FICO recommends that you use decision testing to test your decision logic and then publish the Decision Modeler component to the appropriate environment. Do not delete the existing component until you have confirmed that the upgraded component is producing the same results as the original one.

For best results, complete the tasks in this section in sequential order.

Testing the Decision Logic in the New Decision Modeler Component

While upgrading the Decision Modeler component should not affect the contents of your decision logic, it is important to compile and test the decision logic.

To use decision testing, the decision logic must compile successfully. See “Decision Testing and Analysis” in the Decision Modeler User Guide.

- 1 For the new Decision Modeler component, click the component menu and select **Configure > Open Project > yourProjectName**.
- 2 On the **Project Explorer** toolbar, click the **Compile** icon.
- 3 Expand the **Test** pane in the **Project Explorer** to open decision testing.
- 4 Upload a dataset or retrieve data from the Analytic Datamart to test your decision logic.

Related Links

[Decision Testing and Analysis](#)

Publishing the New Decision Modeler Component

After you have tested your new Decision Modeler component, you are ready to promote it to the appropriate lifecycle status.

For more information, see *Publishing a Component Through the Lifecycle* in the *Decision Management Platform User Guide*.

Inserting the New Endpoints URL in the Client Application

Use the **View Links** command on the component menu to obtain the new endpoints URL. Update the endpoints URL in the client application that will call the new Decision Modeler component.

Deleting the Older Decision Modeler Component

Once you have confirmed that the new component is generating the correct results, use the **Delete** command on the component menu to delete the older component.



Note You cannot undo the deletion of a component.

Using the New Component in the Projects Page

If your team is managing their Decision Modeler components using the Decision Modeler Projects page, you can manage your new component on that page. In the Projects page, your new Decision Modeler component is called a workspace and your project is contained in that workspace.

In a Decision Modeler workspace, the maximum number of projects allowed in a workspace is five. You can import projects or create additional projects as long as the total number does not exceed five.

APPENDIX C

Contacting FICO

FICO provides clients with support and services for all our products. Refer to the following sections for more information.

Product Support

FICO offers technical support and services ranging from self-help tools to direct assistance with a FICO technical support engineer. Support is available to all clients who have purchased a FICO product and have an active support or maintenance contract. You can find support contact information and a link to the Customer Self Service Portal (online support) on the Product Support home page (www.fico.com/en/product-support).

The FICO Customer Self Service Portal is a secure web portal that is available 24 hours a day, 7 days a week from the Product Support home page. The portal allows you to open, review, update, and close cases, as well as find solutions to common problems in the FICO Knowledge Base.

Product Education

FICO Product Education is the principal provider of product training for our clients and partners. Product Education offers instructor-led classroom courses, web-based training, seminars, and training tools for both new user enablement and ongoing performance support. For additional information, visit the Product Education home page (www.fico.com/en/product-training) or email producteducation@fico.com.

Product Documentation

FICO continually looks for new ways to improve and enhance the value of the products and services we provide. If you have comments or suggestions regarding how we can improve this documentation, let us know by sending your suggestions to techpubs@fico.com.

Please include your contact information (name, company, email address, and optionally, your phone number) so we may reach you if we have questions.

Related Services

Strategy Consulting: Included in your contract with FICO may be a specified amount of consulting time to assist you in using FICO® Decision Modeler to meet your business needs. Additional consulting time can be arranged by contract.

Conferences and Seminars: FICO offers conferences and seminars on our products and services. For announcements concerning these events, go to www.fico.com or contact your FICO account representative.

FICO Community

The FICO Community is a great resource to find the experts and information you need to collaborate, support your business, and solve common business challenges. You can get informal technical support, build relationships with local and remote professionals, and improve your business practices. For additional information, visit the FICO Community (community.fico.com/welcome).

Index

Symbols

- operator 310
-= operator 310
(ignoring case) operator 311
* operator 310
*= operator 310
/= operator 310
+ operator 310
+= operator 310
= operator 307, 308
=> operator 308
> operator 308
\$object argument 288
\$value argument 288

A

a keyword 312
action statements in rules 88
Action summary 137
actions
 defining 290
 performing repeatedly 305
 without patterns 449
add() method
 NdDate 377
 NdDuration 383
 NdTime 387
 NdTimestamp 394
adding .jar files
 Project Configuration dialog 46
adding Java classes
 Project Configuration dialog 45
addition operators 318
additional search criteria in queries 215
addReason() method 454
advanced builder
 code completion lists 199
 common behavior 199
 in actions 197

 in functions 197
 triggering code completion 198
an keyword 312
Analytic Data Mart
 Decision Modeler 32
analyzing test results 250
and operator 280, 318
 in boolean expressions 280
apply keyword 313
applying
 decision tables 292
 decision trees 293
 ruleflows 295
 rulesets 291
 score models 294
argument lists 301
arithmetic operators 310
array keyword 312
arrays
 creating 342
 example statements 344
 fixed and dynamic 341
 in Rule Builder 343
as operator 318
ASCII characters 322
assert().assert methods 408
assert().check methods 408
assert().fail methods 409
assignment operators 307, 318
 syntax described 307
assignment statements
 compound 296
 syntax 295
associations 312, 349, 350
 creating 350
 from ... to ... keyword 312
 methods and properties 350
at immediate priority keyword 313
at least ... satisfy ... keyword 313
at most ... satisfy ... keyword 313
at priority keyword 313
available keyword 321

available values
as literals 287

B

Batch processing 230
bins in scorecards 148
Blaze Advisor project
 requirements for export 461
Blaze Advisor types that are not supported
 256
blocks 290
boolean expressions
 syntax 280
boolean keyword
boolean operator 307
brUnit assertions 402, 408, 409
 assert().assert and assert().check
 methods 408
 assert().fail methods 409
brUnit test cases
 running unit tests 254
 unit testing 253
built-in functions 271, 410, 448
 currency 410
 standard functions 448
built-in math functions
 math().abs() 356
 math().arctan() 357
 math().ceil() 353
 math().cos() 357
 math().exp() 358
 math().floor() 353
 math().log() 358
 math().max() 360
 math().min() 361
 math().mod() 359
 math().power() 359
 math().round() 354
 math().sin() 359
 math().tanh() 360
 math().truncate() 355
built-in portable functions
 portable().century() 414
 portable().charToInt() 432
 portable().compareDates() 415
 portable().compressString() 433
 portable().concat() 433
 portable().concatByContent() 434
 portable().date() 416
 portable().dateMinusDays() 416
 portable().datePlusDays() 417
 portable().dateToInt() 418
 portable().day() 418
 portable().dayOfWeek() 419
 portable().daysInMonth() 420
 portable().ddyyyy() 421
 portable().ddmmyyyy() 421
 portable().diffInDays() 421
 portable().diffInMonths() 422
 portable().diffInYears() 423
 portable().findChar() 442
 portable().findString() 443
 portable().intToDate() 424
 portable().is_in() 443
 portable().isFloat() 444
 portable().isInteger() 445
 portable().isLeapYear() 425
 portable().max() 445
 portable().min() 446
 portable().mmddyyyy() 425
 portable().mmmyyyy() 426
 portable().month() 426
 portable().not_in() 447
 portable().selectValue() 448
 portable().subString() 434
 portable().time() 427
 portable().toBoolean() 435
 portable().toFloat() 436
 portable().toInteger() 438
 portable().toLowerCase() 439
 portable().toString() 440
 portable().toUpperCase() 441
 portable().verifyDate() 427
 portable().verifyIntDate() 428
 portable().verifyMonth() 428
 portable().verifyYear() 429
 portable().year() 429
 portable().ymdToDate() 430
 portable().yyyddd() 431
 portable().yyymm() 431
 portable().yyymmdd() 431
 using dates 413
built-in standard functions
 ignore() 449
 print() 450
business object model scope
 workspace 44
business term calculations
 advanced builder behavior 199
 advanced builder code completion lists
 199
 triggering code completion in advanced
 builder 198

business term sets
 creating 68
 creating value lists 69
 initializing 71
 mapping business terms to external object model properties 69
 overview 67
 term calculations in business terms 68

business terms
 initialization 227

C

calculateReasons() method
 NdScoreModelReturnInfo 455, 456

calculations 327

case keyword 316

case-selection constructs 302

catch keyword 316

changing
 decision table return type 95
 expression format in decision table cells 100
 order of condition and action expressions 81

classpath issues for exported Blaze Advisor projects 460

clearing filters in decision tables 108

code
 notation conventions 271
 syntax reference 271

code blocks 290

code metasymbols 271

collection comparison operators 307

comes after operator 308

comes before operator 308

comments
 writing in SRL 325

compare() method 384

comparison expressions
 date, time and money values 327
 literal values in 287
 money values 340

comparison operators 308

comparison queries
 creating 217
 limitations 221
 merge editor actions 220
 overview 217
 viewing results 218

compatible versions of Blaze Advisor 17

Compilation Job

Togglz admin console settings 33

compiling decision service 56

compound assignment operators 310, 318

compound assignment statements 296

concatenation operator 318

conditions
 boolean expression syntax 280
 grouping expressions 81
 moving expressions 81
 rules 85
 ungrouping expressions 81

configuring
 project 37

control construct-specific keywords 316

conversions
 currency 337
 convertedTo() method 337
 converting 337

copying and pasting
 decision table cell values 101
 decision table rows and columns 100
 entities 61

create statements 297

creating
 associations 350
 comparison queries 217
 custom entities 53
 decision tables 94
 folders 61
 initialization rules 80
 money properties 335
 new decision entities 53
 objects programmatically 297
 parameters in decision tables 96
 rule conditions 86
 rulesets 73
 sample datasets for rule profiling 103
 scorecards 145
 SRL expressions 279
 standard business queries 213

currency
 conversions 337
 decimal precision 339
 notation 336
 properties of 335
 signs for 336

currentDate() method 364

currentTime() method 365

currentTimestamp() method 365

D

- dataset
 - decision trees 131
- date datatypes
 - converting 333
 - formatting 332, 334
 - initializing 331
 - overview 326
 - range and precision 326
- date operators 327
- date properties 362
- date() method 366
- dates 328, 329, 361
 - properties 329
 - creating properties 329
 - creating variables 328
- decimal precision 339
- decision flow variables
 - about 206
 - creating 207
 - setting a type and initial value 207
- decision flows
 - defining tasks 209
 - creating 205
 - defining 205
 - defining split nodes 210
 - defining subflow nodes 210
 - details pane 206
 - editor 202
 - editor toolbar 203
 - floating toolbar 204
 - inserting a decision flow entity 209
 - loop through a collection property 211
 - loop types 211
 - overview 201
 - repeat loops 211
 - selecting an input type 205
 - selecting one for deployment 227
- Decision Modeler
 - adding an object model 44
 - Analytic Data Mart registration 32
 - authoring rules in a project 51
 - client id and secret 231
 - configuring ADM reporting data 35
 - configuring projects 21
 - creating a project 36
 - creating projects 21
 - date, time, and time zone formats 235
 - decision service life cycle 229
 - enabling rule event logging in the Analytic Data Mart 36
- importing a schema 37
- importing business term sets exported from Decision Management Suite 261
- importing projects 21, 41
- JAR file import errors 47
- limiting data capture in the Analytic Data Mart 34
- manage teams 29
- obtaining a bearer token 231
- obtaining the client ID and secret 39
- PMML mining model example 162
- Project Editor 51
- project permissions 31
- project settings 39
- projects page 22
- replace an object model 44
- requesting access 230
- requests via the SOAPUI do not work 269
- REST endpoints 232
- SOAP endpoints 232
- testing decision services 231
- testing endpoints 40
- update an object model 44
- upgrading a component 3.0 or earlier 466
- uploaded project fails deployment 261
- using data logged in the Analytic Data Mart 32
- viewing links 39
- workspace settings 28
- workspace team roles 30
- workspace with existing projects 42
- workspaces 27
- decision service
 - configure with default settings 38
 - modifying 42
 - verifying an object model 261
- decision tables
 - applying 292
 - changing how profiling results are displayed 104
 - changing the cell format 100
 - changing the return type 95
 - clearing filters 108
 - compiled table optimization 112
 - copying and pasting cell values 101
 - creating 94
 - creating parameters 96
 - creating sample datasets 103
 - cutting, copying, and pasting rows and columns 100
 - defining 94

- disabling rule profiling 107
- editing rule names 98
- editing rules 96
- enabling condition cells for multiple values 99
- exporting data to CSV files 109
- filtering rules 107
- formatting for importing data 109
- generating rule profiling reports 106
- importing data from CSV files 110
- inserting condition and action expressions 97
- keyboard shortcuts 110
- overview 93
- overview of rule profiling 101
- requirements for datasets 102
- rule profiling options 106
- rule profiling report 105
- running rule profiling 103
- showing filters 107
- troubleshooting export and import issues 259
- troubleshooting printing issues 260
- troubleshooting rule profiling issues 258
- decision testing**
 - collection-type properties 241
 - configuring CSV datasets 239
 - dataset generation 245
 - dataset size 243
 - downloading the results in a CSV file 246
 - enabling ADM data capturing 244
 - errors 248
 - filtering the columns to control the display 246
 - generating data 246
 - impact analysis reports 249
 - importing data from Analytic Datamart 244
 - No Records Were Found error 270
 - overview 237
 - precalculate option needed 145
 - preventing variables from accumulating results 238
 - requirements for dataset structure 242
 - requirements for expected data 240
 - requirements for input data 240
 - requirements for the decision entity 238
 - sparse collections 241
 - trying with PMML scorecard example 156
 - unavailable values 248
 - unknown values 248
 - uploading the CSV file with input data 246
- decision trees
 - analyzing profiling results 137
 - any other value node 127
 - applying 293
 - assigning actions 127
 - authoring rules 115
 - changing actions 129
 - collapsing subtrees 123
 - colors for profiled action nodes 136
 - copying and pasting subtrees 125
 - creating 117
 - creating conditions with "or" 126
 - creating local patterns 121
 - creating local variables 120
 - creating parameters 118
 - data type limitations 142
 - dataset requirements 131
 - elements 116
 - expanding subtrees 123
 - export as FSML file 141
 - exporting tree view statistics 135
 - expression actions 128
 - FSML import 169
 - generating a sample dataset for profiling 132
 - inserting levels 124
 - inserting splits 123
 - limitations for decision variables names 142
 - local patterns 120
 - local variables 119
 - manage variables 129
 - merging branches 126
 - node views 135
 - previewing the dataset for profiling 133
 - Profile view 137
 - profiling statistics 134
 - profiling views added to the editor 133
 - recount 140
 - removing duplicate action level 125
 - removing levels 124
 - reordering branches 125
 - repair mode 130
 - replacing a level 125
 - return type 122
 - running data 131
 - running data profiling 133
 - showing proportion of actions 136
 - string type variables 127
 - support for the less than symbol 142
 - tree views 134
- decrement operator 318

decrement statement 296
default mapping conventions
 money 338
defaultDateFormat property 334, 362
defaultMappingCurrency property 338
defaultTimeFormat property 334, 362
defaultTimestamp property 334
defaultTimestampFormat property 362
defining
 actions 290
 initial values in SRL 276, 278
delete operator 297, 318
deleting decision entities 61
deploy to production 229
deployment entities requirements for
 exporting a projects 460
disabling Togglz admin console settings 34
display names
 naming conventions 322
div operator 310, 318
divide statement 296
divided by operator 310
division operators 318
do keyword 315, 316
does not come after operator 308
does not come before operator 308
does not end with operator 311
does not exceed operator 308
does not follow operator 308
does not precede operator 308
does not start with operator 311
downloading
 workspaces 29
duration
 creating properties 329
 creating variables 328
duration data types 361
duration datatypes
 converting 333
 formatting 332, 334
 initializing 331
 overview 326
 range and precision 326
duration operators 327
duration properties 331
dynamic objects 79, 351

E

editing
 decision table rule names 98
editing entities 63
effective everyday from ... keyword 313
effective everyday from ... to ... keyword 313
effective everyday to ... keyword 313
effective from ... keyword 313
effective from ... to ... keyword 313
effective to ... keyword 313
else keyword 313
ends with operator 311
enumerated types
 declaring in SRL 277
enumeration display values 48
equality 308
equality-test operators 318
event rule keyword 315
event-specific keywords 315
every ... satisfy ... keyword 313
exactly ... satisfy ... keyword 313
exceeds operator 308
exception-specific keywords 316
exceptions
 catch statements 306
 finally keyword 306
 throw keyword 306
 try keyword 306
exchange rates 338
execute keyword 298, 316
execution modes
 keywords 313
exporting Blaze Advisor projects
 as a component 462
 business object model requirements 459
 identifying the projects 457
 set objectmapper to avoid security
 manager exceptions 459
 unsupported functionality in Decision
 Modeler 462
expressions
 assigning compound values 310
 assigning to parameters 301
 assigning values 295, 296, 307
 declarations 279, 280, 282
 literal values in 287
 operator evaluation 311
external methods 299

F

false
 as literal 287
FASt issue
 XML duration types 268
 XML enumeration types 262
FICO

community 476
 conferences and seminars 476
 consulting 476
 documentation 475
 education 475
 product support 475
 professional services 476
 support 475
 training 475
FICO drive 60, 61
 add files 61
 filtering rules in decision tables 107
 finally keyword 316
 fixed array keyword 312
 flow-control constructs
 syntax 302
 folder name length limitation 54
 follows operator
 comparison operator 308
 for each keyword 316
 for each...do constructs
 example 303
 syntax 303
 for keyword 313
 format() method
 NdDate 378
 NdTime 389
 NdTimestamp 394
 format()
 BigDecimal types to string 410
 formatMode property 340
FSML
 decision tree limitations 171
 import summary 171
 importing decision trees 169
 importing models 55
 model re-importing 174
 naming decision tree 171
 selecting a file 170
 selecting business terms 170
 FSML Decision Tree Example 172
 function calls 298, 301
 function keyword 300, 316
 function-specific keywords 316
 functions
 about 193
 advanced builder behavior 199
 advanced builder code completion lists
 199
 creating 194
 declaring in SRL 300
 defining 194
 dynamic objects 79, 351
 local variables 195
 overloading 299
 temporary variables 195
 triggering code completion in advanced
 builder 198
 using advanced builder 197

G

generating a report 55
 generating datasets for decision testing 245
 global variables in decision testing 238
 greater than operators 318
 greater than or equal to operators 318

I

if ... then ... else ... keyword 316
 if keyword 313
 if...then...else constructs 304
 ignore() built-in standard function 449
 implicit casts 333
 importing
 decision service 155, 162, 172, 179
 FSML models 55
 PMML models 55
 SAS models 55
 importing projects
 Decision Modeler 41
 in keyword 316
 increment operator 318
 increment statement 296
 inDaylightSavings() method
 NdTime 388
 NdTimestamp 396
 indexed properties
 syntax 289
 indexedMonthWeekDay() method 367
 indexedYearWeekDay() method 368
 inequality-test operators 318
 initializers 276
 initializing
 objects in SRL 297
 initializing business term sets 227
 initially keyword 276, 312
 inserting a decision flow entity 209
 integer keyword
 inTimeZone() method
 NdTime 390
 NdTimestamp 397
 invert() method
 NdDuration 385

is a keyword 312
is an keyword 312
is any keyword 312
is at least operator 308
is at most operator 308
is blank operator 311
is changed keyword 313, 315
is created keyword 315
is deleted keyword 315
is different from operator 308
is earlier than operator 308
is earlier than or equal to operator 308
is empty operator 307
is equal to operator 308
is initialized keyword 315
is keyword 313
is larger than equal to operator 308
is larger than operator 308
is later than operator 308
is later than or equal to operator 308
is less than operator 308
is less than or equal to operator 308
is more than operator 308
is more than or equal to operator 308
is needed keyword 313, 315
is not at least operator 308
is not at most operator 308
is not blank operator 311
is not different from operator 308
is not earlier than operator 308
is not earlier than or equal to operator 308
is not empty operator 307
is not equal to operator 308
is not larger than operator 308
is not larger than or equal to operator 308
is not later than operator 308
is not later than or equal to operator 308
is not less than operator 308
is not less than or equal to operator 308
is not more than operator 308
is not more than or equal to operator 308
is not smaller than operator 308
is not smaller than or equal to operator 308
is not up to operator 308
is one of keyword 312
is operator 308, 318
is smaller than operator 308
is smaller than or equal to operator 308
is some keyword 312
is up to operator 308
isLeap() method 372
isMonth() method 373

issues when more than one component is in a solution 257

K

keyboard shortcuts
 decision tables 110
 Project Explorer 62
keywords
 control construct-specific 316
 event-specific 315
 exception-specific 316
 execution modes 313
 function-specific 316
 listed 312
 notation conventions for 271
 object-specific 312
 ruleset and rule-specific 313
 special values 321
 specifying datatypes 320
 specifying operators 318
known keyword 321
known values
 as literals 287

L

Leaf view 137
less than operators 318
less than or equal to operators 318
literal values 279, 287
literals 329
local patterns in decision trees 120, 121
local patterns in rulesets 76, 77
local variables in decision trees 119, 120
local variables in rulesets 74, 75
locale setting for date and time 332
LOCALE settings 334
localization 327
locating references to decision entities 61
loop types in decision flows 211

M

making entities editable 63
math functions
 See built-in math functions
math operations 310
math().abs() built-in function 356
math().arctan() built-in function 357
math().ceil() built-in function 353
math().cos() built-in function 357
math().exp() built-in function 358
math().floor() built-in function 353

math().log() built-in function 358
 math().max() built-in function 360
 math().min() built-in function 361
 math().mod() built-in function 359
 math().power() built-in function 359
 math().round() built-in function 354
 math().sin() built-in function 359
 math().tanh() built-in function 360
 math().truncate() built-in function 355
 member definitions
 in SRL syntax 272
 merge editor options in comparison queries
 220
 metasymbols
 notation conventions 271
 method calls 299
 minus operator 310
 mod operator 310, 318
 models
 importing PMML files 153
 modifying a decision service 42
 monetary values 337
 money
 comparison expressions for values 340
 converting datatypes 337
 creating datatypes 335
 currency precision for data type 339
 datatypes 334
 datatypes overview 334
 declaring a type variable 335
 declaring properties 335
 default mappings for datatypes 338
 formatting data types 340
 initializing data types 336
 properties 335
 rounding behavior for data type 338
 monthDayName() method 376
 monthLength() method 373
 monthName() method 374
 monthWeekDay() method 369
 multiplication operators 318
 multiply statement 296
 multiply() method
 NdDuration 386

N

naming
 decision entities 54
 naming conventions for decision entities 322
 NdCalendar class 364
 NdCurrencyManager class 334
 NdDate class 376

NdDuration class 383
 NdReasonCode class 450
 NdScoredCharacteristic 451
 NdScoreModelReasonCalculationOptions 452
 NdScoreModelReturnInfo 453, 455
 NdScoreModelReturnInfo methods
 addReason() 454
 calculateReasons() 456
 NdTime class 387
 NdTimestamp class 393
 needed keyword 321
 new keyword 315
 not operator 318
 notation conventions 271
 null keyword 321
 null values
 as literals 287
 numeric expression
 syntax 282
 numeric operator 310
 numeric operators
 arithmetic operators 310
 numeric values
 converting 353

O

object keyword
 object models
 SRL syntax for class and class members
 272
 object-specific keywords 312
 objects
 creating programmatically 297
 declaring in SRL 273
 destroying 297
 initializing 297
 occurs keyword 315
 old keyword 315
 operators
 * 310
 addition 318
 and 318
 as 318
 assignment 307, 310, 318
 comparison 308
 concatenation 318
 date and time values 327
 decrement 318
 delete 318
 div 310, 318
 divided by 310

- division 318
- equality-test 318
- greater than 318
- greater than or equal to 318
- increment 318
- inequality-test 318
- is 318
- keywords 318
- keywords as 318
- less than 318
- less than or equal to 318
- minus 310
- mod 310, 318
- multiplication 318
- not 318
- or 318
- plus 310
- precedence 311
- property-access operator 318
- range comparison operator 318
- set 318
- subtraction 318
- table of 318
- times 310
- or operator 280, 318
 - in boolean expressions 280
- otherwise keyword 316
- output
 - defaults for date, time and timestamp
 - values 362
- overloading functions 299
- overview
 - money datatypes 334
- P**
 - parameter lists 301
 - parameters
 - bindings in SRL 301
 - setting a type and initial value 207
 - parse()
 - string values to money values 410
 - pattern warnings 78, 122
 - patterns
 - ignoring in action statements 449
 - permissions 109
 - plus operator 310
 - PMML
 - mining model import 158
 - built-in functions 166
 - importing models 55, 153
 - importing other PMML models 165
 - importing scorecards 153
 - importing Tree Ensemble models 158
 - mining model re-import 164
 - overwrite entities 166
 - reserved words 166
 - scorecard re-import 157
 - supported models 160, 165
- PMML model
 - invoking from decision flow 159
- PMML Scorecard Example 155
- portable functions
 - See built-in portable functions
- portable().century() built-in function 414
- portable().charToInt() built-in function 432
- portable().compareDates() built-in function 415
- portable().compressString() built-in function 433
- portable().concat() built-in function 433
- portable().concatByContent() built-in function 434
- portable().date() built-in function 416
- portable().dateMinusDays() built-in function 416
- portable().datePlusDays() built-in function 417
- portable().dateToInt() built-in function 418
- portable().day() built-in function 418
- portable().dayOfWeek() built-in function 419
- portable().daysInMonth() built-in function 420
- portable().ddyyyy() built-in function 421
- portable().ddmmyyyy() built-in function 421
- portable().diffInDays() built-in function 421
- portable().diffInMonths() built-in function 422
- portable().diffInYears() built-in function 423
- portable().findChar() built-in function 442
- portable().findString() built-in function 443
- portable().intToDate() built-in function 424
- portable().is_in() built-in function 443
- portable().isFloat() built-in function 444
- portable().isInteger() built-in function 445
- portable().isLeapYear() built-in function 425
- portable().max() built-in function 445
- portable().min() built-in function 446
- portable().mmddyyyy() built-in function 425
- portable().mmmyyyy() built-in function 426
- portable().month() built-in function 426
- portable().not_in() built-in function 447
- portable().selectValue() built-in function 448
- portable().subString() built-in function 434
- portable().time() built-in function 427
- portable().toBoolean() built-in function 435

portable().toFloat() built-in function 436
portable().toInteger() built-in function 438
portable().toLowerCase() built-in function 439
portable().toString() built-in function 440
portable().toUpperCase() built-in function 441
portable().verifyDate() built-in function 427
portable().verifyIntDate() built-in function 428
portable().verifyMonth() built-in function 428
portable().verifyYear() built-in function 429
portable().year() built-in function 429
portable().ymdToDate() built-in function 430
portable().yyyyddd() built-in function 431
portable().yyyymm() built-in function 431
portable().yyyymmdd() built-in function 431
 precedence operators 311
 precedes operator 308
 precision 326, 339
 preventing classpath issues for exported
 Blaze Advisor projects 460
 previewing decision entities 61
 primary expressions 280
 primitive datatypes
 as keywords 320
print() built-in standard function 450
 printing Project Explorer contents 61
 Project Configuration dialog
 adding .jar files 46
 adding Java classes 45
 removing .jar files 46
 removing Java classes 45
 replacing .jar files 46
 replacing an XML schema 45
 replacing Java classes 45
 Project Explorer 51, 61, 62
 commands 61
 keyboard shortcuts 62
 projects
 importing back into Blaze Advisor 463
 maximizing throughput for batch/boxcar
 processing 38
 selecting entry points 38
 promoting an older version to be the latest
 version 65
 properties
 lists of 275
 date and time data types 362
 declaration lists 273
 declarations 275
 declarations in SRL 275, 287
 monetary values 335
 money 335
 primitive datatypes as 329
 using date and time 329
 property-access operator 318
 punctuation
 for SRL 322
 notation convention 271

Q

quantified conditions
 examples 283
 syntax 282
 queries
 comparison 217
 standard business 213
 verification 223

R

range and precision 326
 range comparison expressions 284, 286
 syntax 286
 range comparison operator 318
 raw keyword 315
 real keyword
 reason code lists 144
 adding reason codes 144
 creating 144
 reason codes
 support classes 450
 recreating decision service examples 156,
 163, 173, 180
 reference names)
 naming conventions 322
 references
 to decision entities 60
 refreshing your workspace 61
 Release Notes 17
 releasing locks 65
 removing .jar files
 Project Configuration dialog 46
 removing Java classes
 Project Configuration dialog 45
 renaming decision entities and folders 55, 61
 repairing decision trees 130
 replacing .jar files
 Project Configuration dialog 46
 replacing an XML schema
 Project Configuration dialog 45
 replacing Java classes
 Project Configuration dialog 45
 reports
 decision service contents 55
 decision table rule profiling 105, 106

- impact analysis in decision testing 249
 - requirements for
 - deployment entities needed for exported projects 460
 - exported Blaze Advisor project 461
 - reserved words 323
 - REST payload for single input uses wrapper element 33
 - return keyword 313, 316
 - return statements 298
 - returning keyword 313, 316
 - rule actions
 - using advanced builder 197
 - Rule Builder
 - arrays 343
 - interface 81
 - set operations properties 346
 - rule keyword 313
 - rule-specific keywords 313
 - ruleflows
 - applying 295
 - rules
 - action expressions 87
 - action statements 88
 - allowed values 90
 - condition expressions 85
 - copying and pasting expressions 83
 - copying and pasting statements 83
 - creating condition expressions 86
 - creating in rule builder 83
 - explanation of definition fields 84
 - grouping expressions 87
 - initialization 80
 - validation 89
 - ruleset keyword 313
 - ruleset-specific keywords 313
 - rulesets
 - applying 291
 - creating 73
 - creating local variables 75
 - dynamic objects 79, 351
 - explanation of definition fields 74
 - local patterns 76
 - local variables 74
 - returning values in SRL 298
 - running unit tests 254
- S**
- SAS models
 - file requirements 176
 - data set options and statements 184
 - data step statements 182
 - how to upload license 175
 - import 177
 - importing 55
 - license to use in Decision Modeler 175
 - mapping SAS expressions to SRL 185
 - overview 175
 - re-importing 180
 - SRL mapping 181
 - supported constructs 188
 - supported standard functions 191
 - type mapping 181
 - score models 144, 151, 294
 - applying 294
 - reason code lists 144
 - scorecards
 - adding bins 149
 - baseline score 148
 - bins 148
 - creating 145
 - defining 145
 - deleting bins 150
 - deleting variables 148
 - importing PMML 153
 - introduction 143
 - methods for returning reason codes 152
 - precalculate option 146
 - select variables 148
 - support classes 450
 - validating values in 151
 - variables 147
 - writing rules 147
 - searching for decision entities 57, 59
 - segmentation model size limit 161
 - select keyword 316
 - selecting a decision flow for deployment 227
 - set operations properties
 - in the Rule Builder 346
 - overview 345
 - semantic requirements 349
 - using in SRL 348
 - set operator 318
 - setup wizard
 - default settings 38
 - importing a schema 37
 - shiftToTimeZone() method
 - NdTime 391
 - NdTimestamp 397
 - shortMonthName() method 374
 - shortWeekDayName() method 375
 - showing decision table filters 107
 - special values

keywords for 321
 split nodes in decision flows 210
 SRL language 272
 SRL Regular Expressions
 keywords 284
 contains match 284
 exactly matches 284
 staging environment issues 257
 stampAt() method 380
 standard business queries
 additional search criteria 215
 creating 213
 overview 213
 supplemental information 214
 starts with operator 311
 statements
 assignment 295, 296
 declaring 290, 297, 298
 defined 290
 exception handling 306
 parameter binding in 301
 syntax for multiple 290
 string literals 329
 strings
 comparison operators 311
 money values as 340
 string keyword
 string-specific keywords 317
 stripTimeZone() method
 NdTime 391
 NdTimestamp 398
 Structured Rule Language (SRL)
 control constructs 302
 creating expressions 279
 creating functions 300
 creating object models 272
 keywords 312
 operators 307, 311
 punctuation 322
 statements 290, 295, 297, 298
 syntax reference 271
 writing comments in 325
 subflow nodes in decision flows 210
 submit to staging 229
 subtract() method
 NdDate 381
 NdDuration 386
 NdTime 392
 NdTimestamp 398
 subtractInDays() method
 NdDate 382
 NdTimestamp 399
 subtractInMilliseconds() method 400
 subtractInMonths() method
 NdDate 383
 NdTimestamp 400
 subtraction operators 318
 support classes 271, 361, 450
 primitive data types 361
 score models and reason codes 450
 supported web browsers 18
 symbols 271
 syntax
 boolean expressions 280
 notation conventions 271
 operator precedence 311
 syntax reference 271

T

Tableau Server usage 235
 tasks in decision flows 209
 then keyword 313
 throw keyword 316
 throw statements
 exception handling 306
 time 328, 329, 361
 creating properties 329
 creating variables 328
 time datatypes
 converting 333
 formatting 332, 334
 initializing 331
 overview 326
 range and precision 326
 time operators 327
 time properties 330, 362
 time zones 327
 time() method 369
 times operator 310
 timestamp 326, 328, 329, 332
 creating properties 329
 creating variables 328
 formatting 332
 timestamp datatype 326
 timestamp datatypes
 formatting 334
 timestamp operators 327
 timestamp properties 330, 362
 timestamp() method 370
 timestamps 331, 361
 initializing 331
 Togglz admin console settings 33, 468
 troubleshooting

- 503 and 502 errors 268
- decision table export and import 259
- FASt issue with XML duration types 268
- FASt issue with XML enumeration types 262
- No TargetNamespace Exception 260
- rule profiling 258
- SAXException 260
- SOAP links 269
- verifying an object model 261
- true
 - as literal 287
- try keyword 316
- turning off Analytic Data Mart Event Logging 33
- type conversions
 - date, time and duration 333
 - monetary values 337
- types
 - date and time 326, 327, 361
 - enumerations as 277
 - money 334, 335, 339
 - primitive 320
- U**
 - unavailable keyword 321
 - unavailable values
 - as literals 287
 - Unexpected check box in score model 151
 - Unicode characters 322
 - unit testing
 - brUnit test cases 253
 - viewing the results 254
 - unknown keyword 321
 - unknown values
 - as literals 287
 - unsupported Blaze Advisor types 256
 - until keyword 305, 316
 - until...do statements 305
 - unused pattern warnings 78, 122
 - updating with the latest versions 61
 - user-supplied names 271
- V**
 - validation for rules in rulesets 89
 - validation in scorecards 151
 - value (default) properties
- W**
 - web browser support 18
 - weekDay() method 375
 - whenever keyword 315
 - while keyword 305, 316
 - while statements 305
 - while...do statements 305
 - workspaces
 - downloading 29
- X**
 - XML business object model mapping
 - data types 255
 - enumeration values 256
 - unsupported constructs 255
 - uploading a modified or new XML schema 45
- Y**
 - yearWeekDay() method 371
- syntax 288
- values
 - defining initial 276, 278
 - literal 287
 - range and precision 326, 339
- variables
 - declaring in SRL 278
 - defining initial values for 278
 - in scorecards 147
 - using date and time 328
- verification queries
 - creating and running 224
 - message types 225
 - overview 223
- version management 63
- versioning
 - checking in and checking out 63
 - commands 63
 - promoting an entity 63
 - promoting older version to be the latest one 65
- viewing
 - comparison queries results 218
 - previous versions of entities 64
 - results of running the test framework 254