COGILITY
SOFTWARE

Using Action Semantics with Cogility Studio

Cogility Studio version 7.0

Document version number 7.0

Cogility Software Corporation
111 North Market St., Suite #888
San Jose, CA 95113

support@cogility.com

Printed in the United States of America.

Using Action Semantics

Contents

Chapter 4 Reserved keywords

Chapter 5 Java actions

Chapter 6 Functions

Chapter 7 Queries

Chapter 8 Special

Preface

Cogility Studio provides the tools to create a model for an enterprise information system, and deploy it as a J2EE application. The Cogility Studio documentation provides support for this endeavor.

Cogility Studio documentation

Cogility Studio comes with several volumes of documentation to help you.

- *Installing and Configuring Cogility Studio* describes the installation and configuration of your application server, database and Cogility Studio.
- *Getting Started with Cogility Studio* is a brief overview of Cogility Studio.
- *Modeling with Cogility Studio* tells you how to build a model-driven enterprise application using Cogility Modeler and associated tools.
- *Using Actions in Cogility Studio* provides a reference to modeling action semantics for use with Cogility Studio.
- *Change Management in Cogility Studio* describes the change management system for models and model artifacts.
- *Model Deployment & Execution in Cogility Studio* is a guide to application monitoring, maintenance and migration, and describes the utilities that you can use to test and monitor your model deployed as a enterprise application.

Several white papers on various topics are also available to further your understanding of enterprise application integration, business process management, model driven architecture and other related topics. See the Cogility website:

http://www.cogility.com.

1

Action semantics

The components of a system model designed with Cogility Studio, the information, the interfaces, and the intent are decoupled, allowing any of these to change without requiring the designer to make a complementary change in any of the other components. Action semantics describe the intent of a system model and support a precise executable specification of systems that enables translation of a developer's model into one of several possible designs.

The intent, implemented with action semantics, specifies the business processes and logic to be executed in terms of the information model when an interface is invoked. Thus, action semantics may be described in terms of their use for either information model access or interface access. The intent also describes the control layer made up of control structures and operators used throughout the model. These are described in Figure 1-1 on page 9 and discussed below.

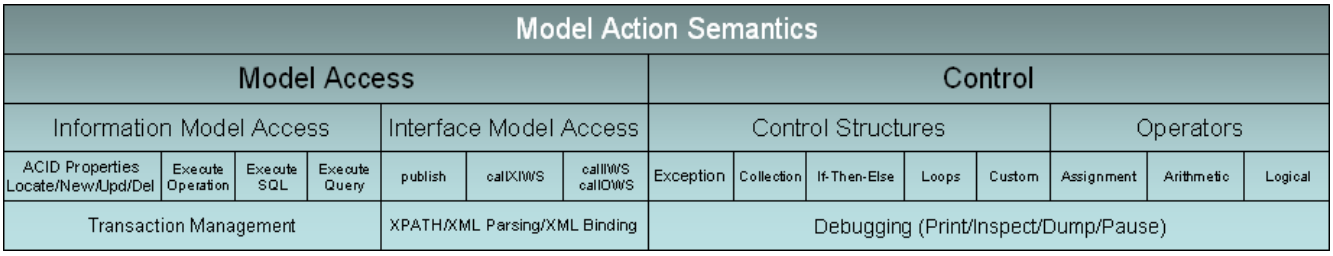

Figure 1-1: Model action semantics

Action semantics provide key database management system features transparently. Transaction processing on the information model with action semantics maintains the database atomicity, consistency, isolation and durability (ACID). You don't have to manage these database properties or worry about transaction semantics; the integrity of your information model objects is managed for you. Cogility Manager will automatically create the proper transaction context for you, and this transaction context will take advantage of the ACID properties that are provided by the underlying database. For more about transactions, see "Transactions" on page 109 of the guide, *Modeling with Cogility Studio*.

Action semantics support the following for the information model:

- Create, read, update and delete (CRUD) functions of the database management system with the new, locate and del keywords and dot-notation-assignment and navigation which allow you to seamlessly explore the data. See the following:
  - "new" on page 43.
  - "locate" on page 41.
  - "del" on page 36.
  - "Class attributes" on page 13.
- Transaction management keywords that override the default transaction handling. See the following:
  - "oclCommit" on page 45.

  - "oclRollback" on page 49.
- Class operations and their execution. See "Class operations" on page 13.
- SQL execution on a legacy database. See "Queries" on page 99.

Action semantics provide access to both the asynchronous, service-oriented interface model and the synchronous, event-driven interface model.

- The keywords that call web services provide transparent XML binding, parsing and XPATH. See the following:
  - "callHTTPIWS DELETE at" on page 31.
  - "callOWS at" on page 34.
  - "callXIWS at" on page 35.
- The publish keyword publishes JMS messages that launch events. See "publish do" on page 50.

Action semantics control structures and operators are available for use throughout the model:

- Exception handling. See "throw" on page 54.
- Working with collections. See "Collection operations" on page 55.
- Conditional statements. See "if elseif else endif" on page 40.
- Loop statements. See "loop do endloop" on page 42.
- Variable assignment. See "Variable assignments" on page 12.
- Arithmetic operators. See "Arithmetic operators" on page 17.
- Logical operators. See "Logical operators" on page 19.

Action semantics as a language

Action semantic keywords are powerful, executable, system logic modeling artifacts. They each have inherent executable capability and do not need to be decomposed further and translated to 3GL code (like Java) to execute. It would take many lines of 3GL code to implement equivalent capability effectively and consistently for every usage. However, action semantics can be safely and easily expanded if required using Java actions. See "Custom Java actions" on page 89.

You will use action semantics throughout Cogility Studio: in Cogility Modeler's action fields and in Cogility Action Pad (see "Cogility Action Pad" on page 79 of the guide *Model Deployment & Execution in Cogility Studio*), and you can run action semantics scripts in .bat files (see "Action semantics batch files" on page 56 of the guide *Model Deployment & Execution in Cogility Studio*).

Action semantics provide a methodology for interpreting execution specifications in a system model. Action semantics resemble a computer language, but unlike such languages as Java or C++, action semantics work with UML concepts, making them more suited to modeling. Also, action semantics do not require the large overhead inherent with formal languages. Yet, action semantics come with many of the facilities of a language like type checking and exception handling.

Action semantics in Cogility Studio work with the Object Constraint Language (OCL), the formal expression language for the Unified Modeling Language (UML). OCL describes constraints about objects in a model, and specifies conditions for changes to a model. However, these specifications alone are not executable. OCL is not an action language. Action semantics provide OCL with the action language functionality and the computational specification in sufficient detail that the specifications can be executed in Cogility Studio.

OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type rules of the language. For example, you cannot compare an integer with a string. Each classifier defined within a UML model represents a distinct OCL type. In

addition, OCL includes a set of supplementary predefined types. OCL compiles dynamically and does not have to be compiled and integrated into the server code base.

For more information on OCL, see http://www.omg.org.

Action semantics is a language framework adopted by the OMG that enables sharing the semantics (or interpretation) of action and operation behavior between UML models and components and between UML tools. Action semantics in Cogility Studio combines OMG action semantics with OCL and provides constructs and operators that you can use to form basic arithmetic, relational, and logical expressions. Action semantics flow control statements enable programmatic choices between actions during execution via "if-else" statements for conditional execution and "loop" constructs for explicit iteration. Exceptions are managed by constructing "try-catch" statements. Action semantics lets non-programmers author expressions, including executable expressions, within a framework similar to that of a formal language.

Action semantics frequently asked questions:

Why don't we just use a programming language?

A programming language such as Java or C++ provides both too much and too little. It provides too much because it allows the modeler access to implementation concepts (pointers or specifically allowable implementations of associ-ations), yet it provides too little because a programming language does not understand UML concepts such as state or attribute typing, nor how the concepts fit together. Programming languages also overspecify sequencing.

Why not just use OCL?

OCL is intended for expressing constraints, that is, relationships within a particular snapshot of a system. While OCL allows a modeler to specify conditions on changes to the instance model, these specifications are not executable, which is what actions are all about. OCL is not an action language.

Basic syntax information

Action semantics use the following basic syntax:

- Statements are separated by semicolons (`;`).
- String constants are surrounded by single quotes (`'`).
- Comments are preceded by two hyphens (`--`).
- Class names or class types must be fully-qualified and qualifiers are separated by double colon (for example, `java::lang::String`).
- Action semantics are case sensitive.

Variable declarations

Action semantics let you define types with any public classes defined in Java and any classes defined in your model. You can integrate Java expressions directly into your expressions. For example, the following expression declares a variable of the Java Collection type:

```
coll : java::util::Collection;
```

You declare variables by naming the variable, followed by a colon (`:`) followed by the fully-qualified classname followed by a semi-colon (`;`). The path qualifiers are separated by double-colons (`::`). Space before or after the double-colons is optional. Paths of two or more qualifiers are allowed.

Multiple variables can be declared on a single line. A comma-separated list of variable names can be in the declaration statement, as demonstrated below:

```
s1,s2,s3 : java::lang::String;
```

Syntax

```
<variable1> , <variable2> , <variable3> :
<qualifier>::<qualifier>::<classname>;
```

In the above, the second and third variables are optional. Any number of variables may be declared according to this syntax.

Example

```
age : java::lang::Integer;

string : java::lang::String;

cust : DOMCE::OrderEntryDIM::Customer;

mimCust : DOMCE::MIM::MIMCustomer;
```

When you declare a variable of any object, you also have access to its attributes implicitly, and you describe them with the same notation as in Java, appending the attribute name to the class name, preceded by a period. For example: `mimCust.firstName`.

Variable assignments

Once you have declared a variable, you assign values using a colon followed by an equals character (`:=`). A single variable can be both declared and initialized by a single statement, as in the following:

```
sum : java::lang::Long := 0;
```

Syntax

```
<variable> : <qualifer>::<qualifier>::<classname> := <expression>;

<variable> := <Expression>;

<variable> := <obj>.<attribute>;

<obj1>.<attribute> := <obj2>.<attribute>;

<obj1>.<attribute> := <obj2>.<methodname>(<parameter>);
```

Example:

```
age := 21;

mimCust := new DOMCE::MIM::MIMCustomer();

mimCust.firstName := 'Gilligan';

mimCust.firstName := cust.firstName;

x := obj1.myMethod( a, b );

Hello_World := 'Hello '.concat('World').concat('!');
```

Class attributes

Classes describe the business objects in your model and may have both attributes and operations. Use Cogility Modeler to create the class and its attributes. See "Classes" on page 33 of the guide, *Modeling with Cogility Studio* for more information about creating classes and attributes.

In action semantics you access class attributes with the same notation as in Java, appending the attribute name to the class name, preceded by a period. For example: `mimcust.firstName`.

Class operations

In addition to assigning attributes, you can define operations for a class. An operation defines an interface to a method defined for that class or any subclass. For each class that has the operation, at least one method implements that interface. An operation has a name, zero or more named and typed input parameters, and zero or one typed return values.

If a return type is defined for the operation, the method must return a value of that type. If a return type is not defined, the return statement with no value (`return;`) must be used. If a return type is defined, `return;` will result in a compilation error. If no return type is defined, `return <expr>;` will result in a compilation error. See "return" on page 52.

Use Cogility Modeler to create class operations. See "Classes" on page 33 of the guide, *Modeling with Cogility Studio*. For each class that implements the operation, you also define a method for that operation. You define the method body with action semantics, as described in the next section.

Method body

Each operation is implemented as a method which consists of action semantics that define the logic for the operation. The class that implements the operation must define a method body. A subclass may implement its own method body to override the superclass. Methods in subclasses can call the methods in the hierarchy using the *self* and *super* keywords. See "self" on page 26 and "super" on page 53.

For example, you might create a web service that performs a price check. You create two classes, Customer and Product.

The Customer has two attributes:

- customerId (type String)
- preferredDiscount (type Double).

The Product has four attributes:

- productId (type String)
- pricePerUnit (type Integer)
- bulkThreshold (type Integer)
- bulkDiscount (type Double)

The Product class also has an operation:

- priceCheck returns a Price (type Double) and has the following parameters.
  - customerIdParam (type (String)
  - quantityParam (type Integer)

The method body for the priceCheck() operation might appear as in the figure below.

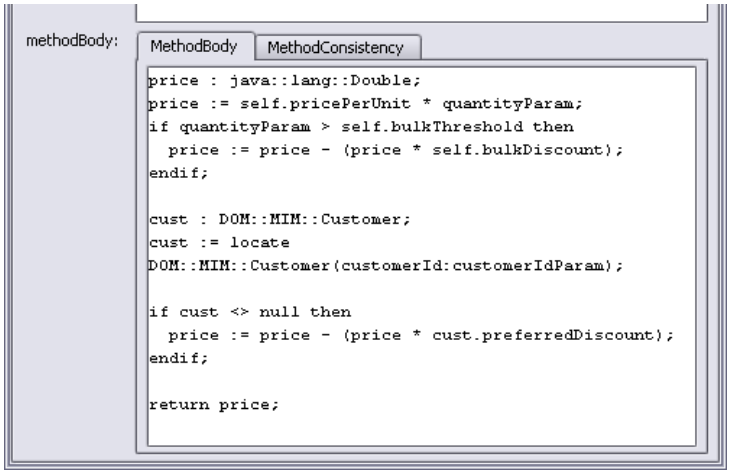

Figure 1-2: Method body

In this method body, the first two lines create a price variable (type Double) and determine the price by multiplying the Product class' (as indicated by the "self" keyword. See "self" on page 26.) pricePerUnit with the quantityParam parameter.

```
price : java::lang::Double;
price := self.pricePerUnit * quantityParam;
```

A conditional clause determines if the quantity is eligible for the Product class' bulkDiscount. If so, the discount is applied.

```
if quantityParam > self.bulkThreshold then
   price := price - (price * self.bulkDiscount);
endif;
```

The next lines define a variable of type Customer, then locates (see "locate" on page 41) the Customer using the value of the customerIdParam parameter.

```
cust : DOM::MIM::Customer;
cust := locate DOM::MIM::Customer(customerId: customerIdParam);
```

Another conditional clause follows which establishes whether the customer exists, and if so, applies any preferredDiscount.

```
if cust <> null then
   price := price - (price * cust.preferredDiscount);
endif;
```

Finally the `return` statement returns a value for the method.

Operation invocation

Once you have defined the operation, you can invoke it in your model. The syntax for invoking an operation is as follows:

```
<variable_of_the_appropriate_type>.<operation>(parameter_name:value,
   parameter_name:value, ...);
```

In the example of the priceCheck() operation (see "Method body" on page 13), you could define Web Service logic as shown in the figure below.

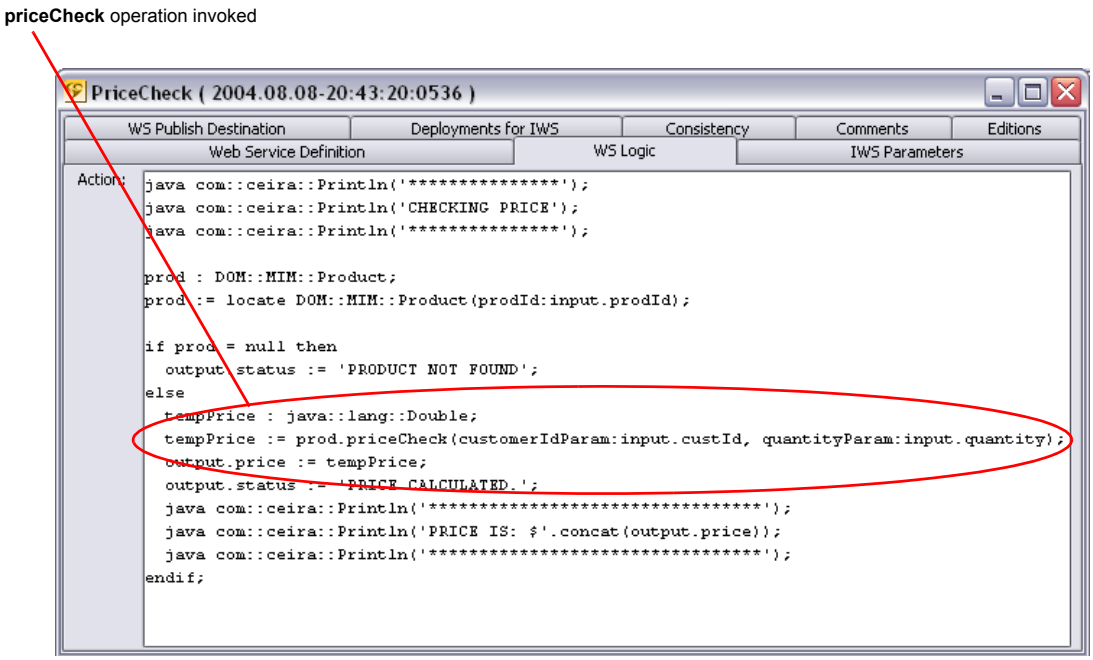

Figure 1-3: Operation invocation

In this logic, a variable called prod is declared of class Product. Then an instance of Product is located with the prodId value input from the Web service. The priceCheck() operation is invoked later in the logic. The customerIdParam and quantityParam parameters are derived from the custId and quantity given during the input.

Actions that require the application server

Unless otherwise noted, action semantics do not require integration with the server code base and may be executed independently of your J2EE application server. However, there are three action semantics keywords that access the J2EE web service and messaging facilities: callXIWS (see "callXIWS at" on page 35), callOWS (see "callOWS at" on page 34) and publish (see "publish do" on page 50).

2 Operators

This section describes arithmetic, relational and logical operators in action semantics.

Precedence

Operators have order of execution precedence between the operator types and within the operator types. Between the types, the operators are evaluated in the following order: arithmetic operators first, relational operators next, and logical operators last. Operators with equal precedence are evaluated from left to right. The following table summarizes the order of precendence.

Operator type	Operator	Description
Arithmetic	**-** (unary)	Negative sign.
	* /	Multiplication and division.
	+ - (binary)	Addition and subtraction.
Relational	= <> < > <= >=	Equal, not equal, less than, greater than, less than or equal to, greater than or equal to.
Logical	**not**	Not.
	and or xor implies	And, or, either not both, implies.

Precedence within the operator types is also described in each section for the type below. You can use parentheses to override the order of execution.

Arithmetic operators

Action semantics support the following arithmetic operators (listed in order of precedence):

Operator	Description	Example	Result
- (unary)	Negative sign	-5	-5
*	Multiplication	1 * 2	2
/	Division	1.0 / 2.0	0.5
+	Addition	1 + 2	3
- (binary)	Subtraction	1 – 2	-1

A compound expression such as `3.0 + 3.0 / 2.0 * 2.0 - 3.0` evaluates to `3.0`. Also, the expression `(1 * (2 + 3) / (4 - 5)) - 7` evaluates to `-12`.

Numeric type promotion and implicit type casting are applied to all arithmetic expressions that have operands of different types, as shown in the following example:

```
int : java::lang::Integer;
flt : java::lang::Float;
flt := 3 * 2.5;
int := flt;
```

In this example, `int` is an Integer and `flt` is a Float, and constants default to Long (3) and Double (2.5). In the first assignment statement, 3 is promoted from Long to Double and multiplied by 2.5. The result is cast to Float and assigned to `flt`. In the second assignment statement, `flt` (7.5) is cast to Integer and assigned to `int(7)`.

Casting from a float or double to an integer or long will truncate, not round. For example, assigning 1.1 or 1.5 or 1.9 to an integer variable will each result in 1.

Relational operators

Action semantics support the following relational operators (listed in order of precedence):

Operator	Description	Example	Result
<	less than	1 < 2	true
>	greater than b	1 > 2	false
=	equal to	1 = 2	false
<=	less than or equal to	1 <= 2	true
>=	greater than or equal to	1 >= 2	false
<>	not equal to	1 <> 2	true

All relational operators are binary operators. They are further categorized as equality operators (`= <>`) and comparable operators (`< > <= >=`). The operands for the equality operators can be of any type, including the null value. The operands for the comparable operators must be of a type that can be compared (e.g., numbers or strings), and, the operands for a single operator must be compatible. It is not valid to ask whether a number is less than a string, for example.

There is no precedence within the group of relational operators and they are evaluated left to right.

As with the Arithmetic Operators, numeric values are promoted such that they can be mixed. For example, it is acceptable to do something like: `anInteger < aDouble`.

```
a : java::lang::Integer;
b := java::lang::Integer;
a := 5;
b := 6;
return (a > b); -- return value is false
```

The keyword **null** can be used with relational = and <> operator.

```
a := new java::lang::Integer(5);
if ( a = null ) then
   return false;
```

```
else
   return true;
endif;
```

Logical operators

Action semantics support the following logical operators (listed in order of precedence):

Operator	Description	Example	Result
not a	Returns true if a is false. Otherwise, returns false.	not true	false
		not 1 < 2	false
		not a = null	true if a is not null
a **or** b	Returns true if a is true, b is true, or both are true. Otherwise returns false.	true or false	true
a **and** b	Returns true if a and b are both true. Otherwise, returns false.	true and false	false
		1 < 2 or 3 > 4 and 5 > 1	true
a **xor** b	Returns true if a is true or b is true, but not both. Otherwise, returns false.	true xor false	true
a **implies** b	Returns b if a is true. Returns true if a is false.	true implies false	false

The **not** logical operator is a unary operator that is evaluated before the other logical operators. Its single operand must be a Boolean expression.

The other logical operators are binary operators. If there are more than one in the expression, they are evaluated from left to right.

3

Bound Variables

Bound variables provide references to information internal to a model. You can use them in any action semantics in Cogility Studio. Bound variables are system-defined variables that are "bound" to a specific context and have an implicit value within that context. They provide you with a way of referencing model objects and their values from within action semantics expressions, without having to reference the model object or value explicitly.

For those bound variables that map to a Java object, you have access to the methods and attributes of that object. For example, the jmsMapMessage bound variable maps to the javax.jms.MapMessage object and allows access to the value of the object's `accountID` attribute as follows:

```
jmsMapMessage.getString('accountID');
```

boPoid

The `boPoid` variable references the persistent object ID for the business object whose state machine is executing the current Action. Compare to "smPoid" on page 27.

Usage

- State machines
  - States
  - Guards
  - Transitions

dos

In a state machine, the `dos` bound variable represents a data object set (DOS) delivered by an event or a DOS defined by embedded transformation units or the diagram object. Since there are potentially many data object sets used in the state machine, you must specify which data object set with the following syntax:

```
dos.<DOS name>
```

In conversions (M2D and D2M), for inbound conversions (from messages), the `dos` bound variable references the DOS that the M2D conversion populates. For outbound conversions (to messages), the `dos` bound variable references the DOS input to the conversions.

In transformation chains, the `dos` bound variable allows access to a data object set that is referenced by the tranformation objects in the chain.

The following syntax applies for all uses:

```
dos.<DOS name>.<object within DOS>.<attribute>
```

Usage

- State machines
  - States
  - Guards
  - Transitions
- Conversions
  - Message to DOS (M2D)
  - DOS to message (D2M)
- Transformations
  - Parallel transformation chains
  - Serial transformation chains

drilledDownFromCat1Name

In a Chart View, the `drilledDownFromCat1Name` bound variable references the name of the Chart Category1 object selected on a parent chart from which a drill down occurs. This offers the "Drills Down To" chart access to context information allowing it to display different views depending on the value of the variable.

Usage

- Chart Views

drilledDownFromCat2Name

In a Chart View, the `drilledDownFromCat2Name` bound variable references the name of the Chart Category2 object selected on a parent chart from which a drill down occurs. This offers the "Drills Down To" chart access to context information allowing it to display different views depending on the value of the variable.

Usage

- Chart Views

drilledDownFromChartName

In a Chart View, the `drilledDownFromChartName` bound variable references the name of the parent chart object from which a drill down occurs. This allows the "Drills Down To" chart to customize its output depending on the name of the parent object.

Usage

- Chart Views

drilledDownFromFullChartName

In a Chart View, the `drilledDownFromFullChartName` bound variable references the name of the parent chart object from which a drill down occurs. This allows the "Drills Down To" chart to customize its output depending on the fully qualified name of the parent object.

Usage

- Chart Views

event

In a state machine guard condition or transition effect, the `event` bound variable references the event that causes the transition. In M2E and E2M conversions, the `event` variable references the event object involved in the conversion.

The following syntax applies for all uses:

```
event.<event attribute>
```

Usage

- State machines
  - Guards
  - Transitions
- Conversions
  - Event to message (E2M)

input

Transformation objects take data object sets (DOS) as input and output. The `input` variable is bound to the input DOS.The `input` variable references the entire set of classes specified in the input DOS. The `attribute` is optional. The following syntax applies for use in transformations:

```
input.<object>[.<attribute>]
```

For inbound web services, the input bound variable references the argument message attributes. The following syntax applies for use in inbound web services:

```
input.<argument attribute>
```

Usage

- Transformations
- HTTP inbound web services
- Inbound web services

Related topics

See also "output" on page 25 and "dos" on page 21.

jmsMapMessage

In M2E or M2D conversions, the `jmsMapMessage` bound variable references information from the JMS message, an object of type javax.jms.MapMessage.

Usage

- Conversions
  - Message to DOS (M2D)
  - Message to Event (M2E)

linkObject

In a postAssociate operation, the `linkObject` bound variable references the AssociationClass ff instance created using a "new to" statement. By default, the `linkObject` is typed as a Relation and must be cast (using oclAsType) to the appropriate class type if you wish to set its attribute values.

Usage

- "Special" Operation postAssociate (for AssociationClasses only)

message

In E2M and D2M conversions, the `message` bound variable references the message object associated with the conversion object.

Usage

- Conversions
  - DOS to message (D2M)
  - Event to message (E2M)

messageContext

In web service Handlers and Handler Chains, the `messageContext` bound variable references the context of a SOAP Message provided to and/or produced by an Inbound web service.

Usage

- Handler
- Handler Chains

newRow

The `newRow` bound variable refers to the record that updates an "oldRow" on a legacy database. It may be used only in the Update Actions field of an activated table or column in a legacy information model.

Usage

- Activated table (Legacy information model)

oclTypeNameForSelf

The `oclTypeNameForSelf` bound variable refers to fully qualified name of the class on which the executing operation is defined.

Usage

- Operation

oldRow

The `oldRow` bound variable refers to the record to be updated on a legacy database. It may be used only in the Update Actions field of an activated table or column in a legacy information model.

Usage

- Activated table (Legacy information model)

Related topics

See also "row" on page 26.

output

Transformation objects take data object sets (DOS) as input and output. The `output` variable is bound to the output DOS and references the entire set of classes specified in the output DOS. The `attribute` is optional. The following syntax applies for use in transformations:

```
output.<object>[.<attribute>]
```

For inbound web services, the `output` bound variable references the result message attributes. The following syntax applies for use in inbound web services:

```
output.<result attribute>
```

Usage

- Transformations
- HTTP inbound web services
- Inbound web services

Related topics

See also "input" on page 23 and "dos" on page 21.

parent

The `parent` bound variable refers to the state which contains the current state, guard condition or transition effect. Compare to the state bound variable (see "state" on page 28).

Usage

- State machines
  - States
  - Guards
  - Transitions

Related topics

See also "state" on page 28.

<*ReplicateName*>_ID

The <*`ReplicateName`*>`_ID` bound variable refers to the identifier of a replicate StateRTInstance and may be used to communicate with a desired replicate StateRTInstance. The number of replicates are are created during runtime is defined in the "Number of Replicates Action" field. This variable is defined as an integer and spans form 1 to the number of replicates.

The name of the variable is dependent upon the name of the Replicate State object which defines the italicized portion of the variable name. This variable is only visible to those behavior artifacts that exist in the replicate region.

Usage

- States
- Guards
- Transitions

row

The `row` bound variable refers to the record inserted or deleted on a legacy database. It may be used only in the Insert Actions or Delete Actions fields of an activated table or column in a legacy information model.

Usage

- Activated table (Legacy information model)

Related topics

self

In a state machine, the `self` bound variable references the business object for which the state machine is executing.

Usage

- State machines
  - States
  - Guards
  - Transitions
- Chart Views
- HTML Views
- Method Bodies

smPoid

The `smPoid` bound variable references the persistent object ID for the state machine runtime instance that is executing the action. Compare to "boPoid" on page 21.

Usage

- State machines
  - States
  - Guards
  - Transitions

sourceObject

The `sourceObject` bound variable is defined in the method body of a **postAssociate** "special" Operation and references a class instance participating in an association and on which the **postAssociate** operation is defined. This will always be the same as "self".

Usage

- "Special" Operation **postAssociate**.

sourceRoleName

The `sourceRoleName` bound variable is defined in the method body of a **postAssociate** "special" Operation and references the role name of the `sourceObject` end of the association.

Usage

- "Special" Operation **postAssociate**.

sourceState

In a state machine's transition effect, the `sourceState` bound variable references the state from which the object is transitioning.

Usage

- State machines
  - Transitions
  - Guards

startEvent

The `startEvent` bound variable references the state machine's start event in guard conditions, transition effects, or states. The `startEvent` bound variable applies only to the top-level state machine in the case where the state machine includes other state machines.

Usage

- State machines
  - States
  - Guards
  - Transitions

state

The `state` bound variable has two meanings, depending on where you are using it. Within a state, use of this variable refers to the current state. When used as part of an expression for a guard condition, the `state` variable refers to the state from which the object is transitioning. Compare to the parent bound variable (see "parent" on page 26).

Usage

- State machines
  - States
  - Guards

targetObject

The `targetObject` bound variable is defined in the method body of a postAssociate "special" operation and references the class instance associated to the `sourceObject`.

Usage

- "Special" Operation **postAssociate**.

targetRoleName

The `targetRoleName` bound variable is defined in the method body of a **postAssociate** "special" Operation and references the role name of the `targetObject` end of the association.

Usage

- "Special" Operation **postAssociate**.

targetState

The `targetState` variable references the state to which the object is transitioning in transitions of state machines.

Usage

- State machines
  - Transitions

4

Reserved keywords

Cogility defines the following reserved keywords in action semantics.

callHTTPIWS DELETE at

Invokes the DELETE method of a modeled HTTP inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. The optional “at” keyword can be added to specify the endpoint where the web service is deployed.

> **Note:** Use of this Expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callHTTPIWS DELETE <httpInboundWebServiceFullName>(<namedParameterList>)
```

Optional **at** keyword

```
callHTTPIWS DELETE <httpInboundWebServiceFullName>(<namedParameterList>)
at (endpointURL);
```

Example

```
-- Delete existing customer given an id
res : Model::DeleteCustomer::result;
res := callHTTPIWS DELETE Model::DeleteCustomer(id: 1234);


res : Model::DeleteCustomer::result;
res := callHTTPIWS DELETE Model::DeleteCustomer(id: 1234) at
   ('http://localhost:8888/CustomerServices');
```

where the application server is located on the localhost at port 8888 and the HTTPDeployment object associated to the HTTP inbound web service is named CustomerServices.

callHTTPIWS GET at

Invokes the GET method of a modeled HTTP inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. The optional “at” keyword can be added to specify the endpoint where the web service is deployed.

> **Note:** Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callHTTPIWS GET <httpInboundWebServiceFullName>(<namedParameterList>)
```

Optional **at** keyword

```
callHTTPIWS GET <httpInboundWebServiceFullName>(<namedParameterList>) at
(endpointURL);
```

Example

```
-- Retrieve existing customer information give an id
res : Model::GetCustomerInfo::result;
res := callHTTPIWS GET Model::GetCustomerInfo(id: 1234);


res : Model::GetCustomerInfo::result;
res := callHTTPIWS GET Model::GetCustomerInfo(id: 1234) at
   ('http://localhost:8888/CustomerServices');
```

where the application server is located on the localhost at port 8888 and the HTTPDeployment object associated to the HTTP inbound web service is named CustomerServices.

callHTTPIWS POST at

Invokes the POST method of a modeled HTTP inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. The optional “at” keyword can be added to specify the endpoint where the web service is deployed.

> **Note:** Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callHTTPIWS POST <httpInboundWebServiceFullName>(<namedParameterList>)
```

Optional **at** keyword

```
callHTTPIWS POST <httpInboundWebServiceFullName>(<namedParameterList>) at
(endpointURL);
```

Example

```
-- Create a new customer given age and name
res : Model::CreateCustomer::result;
res := callHTTPIWS POST Model::CreateCustomer(age: 38,
   name: 'John Smith');


res : Model::CreateCustomer::result;
```

```
res := callHTTPIWS POST Model::CreateCustomer(age: 38, name: 'John Smithv)
   at ('http://localhost:8888/CustomerServices');
```

where the application server is located on the localhost at port 8888 and the HTTPDeployment object associated to the HTTP inbound web service is named CustomerServices.

callHTTPIWS PUT at

Invokes the PUT method of a modeled HTTP inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. The optional "at" keyword can be added to specify the endpoint where the web service is deployed.

> **Note:** Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callHTTPIWS PUT <httpInboundWebServiceFullName>(<namedParameterList>)
```

Optional **at** keyword

```
callHTTPIWS PUT <httpInboundWebServiceFullName>(<namedParameterList>) at
(endpointURL);
```

Example

```
-- Update the age of an existing customer given new age and an id
res : Model::UpdateCustomerAge::result;
res := callHTTPIWS PUT Model::UpdateCustomerAge(age: 40, id: 1234);


res : Model::UpdateCustomerAge::result;
res := callHTTPIWS PUT Model::UpdateCustomerAge(age: 40, id: 1234)
   at ('http://localhost:8888/CustomerServices');
```

where the application server is located on the localhost at port 8888 and the HTTPDeployment object associated to the HTTP inbound web service is named CustomerServices.

callIWS

Invokes a modeled inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. An instance of this web services "result" message is returned.

> **Note:** This invocation behaves similar to an operation such that it executes within the transaction of the caller and not within its own transaction.

Syntax

```
callIWS <inboundWebServiceFullName>(<namedParameterList>);
```

Example

```
-- Create a new customer given age and name
res : Model::CusomerServices::CreateCustomer::result;
res := callIWS Model::CustomerServices::CreateCustomer(age: 25,
   name: 'John Smith');
```

Related topics

- To invoke a modeled inbound web service from outside the model, see "callXIWS at" on page 35.
- To invoke an outbound web service, see "callOWS at" on page 34.

callOWS at

Invokes a modeled outbound web service from another modeled object (for example, an inbound web service or a state machine). This action takes two arguments: the fully-qualified web service name and a named parameter list. An instance of this web services "result" message is returned. The optional "at" keyword can be added to specify the endpoint where the web service is deployed.

> **Note:** Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callOWS <outboundWebServiceFullName>(<namedParameterList>);
```

Optional **at** clause

```
callOWS <outboundWebServiceFullName>(<namedParameterList>) at
(endpointURL);
```

Example

```
res : Model::OWS::ValidateEmial::IsValidEmail::result;
res := callOWS
Model::OWS::ValidateEmail::IsValidEmail(Email:'support@cogility.com');
```

Optional **at** clause

```
res : Model::OWS::ValidateEmial::IsValidEmail::result;
res := callOWS
Model::OWS::ValidateEmail::IsValidEmail(Email:'support@cogility.com')
   at ('http://www.webservicex.net/ValidateEmail.asmx');
```

Related topics

- To invoke an inbound web service, see "callXIWS at" on page 35.

callOWS with

Invokes a modeled outbound web service from another modeled object (for example, an inbound web service or a state machine). This action takes three arguments: invocation of the web service requires the fully-qualified web service name and a named parameter list; the **with** clause takes one or more name-value pairs. An instance of this web services "result" message is returned.

The **with** clause supports four optional names:

- **username** - used for authentication if required. password must be specified as well.
- **password** - used for authentication if required. username must be specified as well.
- **endpointURL** - an override for the default endpoint.
- **timeout** - number of milliseconds to wait before returning. If the timeout is exceeded, a com::cogility::common::OWSTimeoutException is thrown. If no timeout value is specified, then the caller will wait, and possibly create a hung process.

Note: Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callOWS <outboundWebServiceFullName>(<namedParameterList>) with (
option1Name: option1Value, option2Name: option2Value, ...);
```

Example

```
-- This example will attempt to invoke a third party web sevice. If a
-- response does not arrive within 5 seconds, it will timeout.
try
   res : Model::OWS::ValidateEmail::IsValidEmail::result;
   res := callOWS Model::OWS::ValidateEmail::IsValidEmail(Email:
'support@cogility.com') with (timeout: 5000);
catch (com::cogility::common::OWSTimeoutException ex)
   oclprintln('### Third Party Service Did Not Respond in time.');
endtry;
```

Related topics

- To invoke an inbound web service, see "callXIWS at" on page 35.

callXIWS at

Invokes a modeled inbound web service from another modeled object. This action takes two arguments: the fully-qualified web service name and a named parameter list. Returns an inbound web service result object. The optional "at" keyword can be added to specify the endpoint where the web service is deployed.

Note: Use of this expression requires the use of the application server. The model with the web service must be deployed to the application server and the server must be running.

Syntax

```
callXIWS <inboundWebServiceFullName>(<namedParameterList>)
```

Optional **at** clause

```
callXIWS <inboundWebServiceFullName>(<namedParameterList>) at
(endpointURL);
```

Example

```
-- Create a new customer given age and name
res : Model::CusomerServices::CreateCustomer::result;
res := callXIWS Model::CustomerServices::CreateCustomer(age: 25,
   name: 'John Smith');
```

Optional **at** clause

```
res : Model::CusomerServices::CreateCustomer::result;
res := callXIWS Model::CustomerServices::CreateCustomer(age: 25,
   name: 'John Smith') at
   ('http://localhost:8888/CustomerServices/soap');
```

where the application server is located on the localhost at port 8888 and the ContextRoot of the IWS Deployment associated to the IWS is defined as CustomerServices.

Related topics

- To invoke an outbound web service, see "callOWS at" on page 34.

countAll

Description

Syntax

```
countAll <fullClassName>(<optionalNamedParameterList>);
```

Example

```
-- count all customers
countAll Model::MIM::Customer();


-- count all customers with the first name John
countAll Model::MIM::Customer(firstName: 'John');
```

del

Deletes an instance of a business object class from the run-time database. Deleting an instance of a business object class will delete any Association and Association Class instances it plays a role in. If the business object argument evaluates to null, an exception is thrown. Factory instances cannot be deleted.

Syntax

```
del <businessObject>;
```

Example

Deleting an instance using action semantics is a two-step process. First locate the instance, then delete it using the `del` keyword. The following deletes a single instance from the current context:

```
cust : Model::MIM::Customer;
cust := locate Model::MIM::Customer(id: 1234);
del cust;
```

Related topics

- See also "deleteAll" on page 37.
- To locate one instance, see "locate" on page 41.
- To locate more than one instance, see "locateAll orderedBy" on page 41.
- To delete an association, see "del to" on page 37.

del to

Deletes an instance of an Association or Association Class that exists between two business object instances or between a business object instance and itself, from the run-time database. This expression requires three arguments, a source object, a target object, and the role the target object plays. This expression only deletes the Association or Association Class instance and leaves the business object instances. If either business object variable is null or it the business objects instances are not associated to each other, an exception is thrown.

Syntax

```
del <sourceBusinessObject> to <rolename>(<targetBusinessObject>);
```

Example

```
-- Delete the association between a given customer and order
cust : Model::MIM::Customer := locate Model::MIM::Customer(id: 1234);
ord : Model::MIM::Order := locate Model::MIM::Order(orderId: '34825');
del cust to order(ord);
```

deleteAll

Deletes a collection of business object instances from the run-time database. If you do not specify any parameter values, the action all the objects of the specified type.

Syntax

```
deleteAll <fullClassName>(<optionalNamedParameterList>);
```

Example

```
-- delete all instances of a business class
deleteAll Model::MIM::TemporaryObjects();
```

In the following example, a variable for a direct SQL string is passed.

```
-- delete all loans with status of 5 OR 6
directSQLStr : com::cogility::DirectSQLString;
directSQLStr := 'IN (5, 6)';
deleteAll <model>::MIM::Loan(status: directSQLStr);
```

Related topics

- See also "del" on page 36.
- To delete an association, see "del to" on page 37.
- To locate one instance, see "locate" on page 41.
- To locate more than one instance, see "locateAll orderedBy" on page 41.

dump

Outputs attribute values and role targets of a business object to the logging facility. You can use it to debug runtime action semantics.

Syntax

```
dump busObj;

dump busObj( attrName1, attrName2, ... );

dump busObj( roleName1, roleName2, ... );

dump busObj( AllAttrs );

dump busObj( AllRoles );

dump busObj( any combination of the arguments above );
```

If no arguments are specifed, the values of all Attributes are dumped, followed by all targets of all roles.

If arguments are specified, only those values are dumped, and, those are dumped in the order in which they appear in the argument list. And, an argument may be included more than once and will be dumped each time the argument appears.

The 'AllAttrs' and 'AllRoles' keyword are case-insensitive and cause all attributes or roles to be dumped in alphabetical order.

In the argument list, you can mix attribute names, role names and the keywords (`AllAttrs` or `AllRoles`).

Example

```
cust : Model::MIM::Customer := locate Model::MIM::Customer(id: 1234);
dump cust(firstName, lastName, AllRoles);
```

executeSQL

Executes an SQL statement on a specified legacy information model (LIM). Returns a `java.lang.Integer` that describes the number of rows affected. The operation may be an INSERT, UPDATE, or DELETE passed in as as a parameter holding an entire SQL execute statement enclosed in single quotes (`'`), or only the parameters to an execute statement object. See "Execute statements" on page 68 of the guide, *Modeling with Cogility Studio* for more information. The executeSQL keyword also executes an SQL block. See "SQL block objects" on page 73 of the guide, *Modeling with Cogility Studio* for more information.

Syntax

```
executeSQL <SQLBlockFullName>(<namedParameterList>);

executeSQL <ExecuteStatementFullName>(<namedParameterList>);

executeSQL <LIMFullName>(<stringOfSQL>);
```

> **Note:** If the passed-in values are Strings, they must be enclosed in single quotes. Because the entire SQL statement is passed as a String, any Strings within must be preceded with the escape character, a backslash (\). Likewise, if you wish to pass attribute values to the SQL statement programatically (or dynamically), you must concatenate the variable holding the attribute value to the submitted String.

Examples

Execution of a SQL Block object with named parameter list.

```
executeSQL Model::SCOTT::SalaryIncreaseAndCommission(comm: 350,
empLastName: 'ADAMS', employeeNum: 7876, salary: 1250, title: 'CLERK');
```

Execution of Execute Statement object with named parameter list.

```
executeSQL Model::SCOTT::UpdateEmployeeSalary(newSalary: 900,
   employeeNum: 7369);
```

Execution of SQL string.

```
executeSQL Model::SCOTT('UPDATE EMP SET JOB=\'ANALYST\',
   DEPTNO=20 WHERE EMPNO=7934');
```

Related topics

- See also "selectSQL" on page 52.
- For more information about working with a legacy information model, see "Legacy Information Model" on page 61 of the guide, *Modeling with Cogility Studio*.

if elseif else endif

The `if-else` statement determines whether an action is performed based on a condition. The condition is specified by `<conditional_expression>` and the action performed is specified by `<statement>`. The `<conditional_expression>` can be any Boolean expression and the `<statements>` can be zero or more statements.

If an `if` statement without an `else` clause is used in an assignment statement (for example, `num := if false then 123 endif;`), and the `if` condition evaluates to false, the assigned value will be null (even for numbers).

Syntax:

```
if <conditional_expression> then
   <statements>
elseif <conditional_expression> then
   <statements>
else
   <statements>
endif;
```

The `<elseif>` and `<else>` clauses are optional and there may be more than one elseif.

Example:

```
salary : java::lang::Double;
taxRate : java::lang::Double;
salary := 48000.0;
if salary < 7000.0 then
   taxRate := 0.0;
elseif salary < 28400.0 then
   taxRate := 15.0;
elseif salary < 68800.0 then
   taxRate := 25.0;
elseif salary < 143500.0 then
   taxRate := 28.0;
elseif salary < 311950.0 then
   taxRate := 33.0;
else
   taxRate := 35.0;
endif;
```

Related topics

- any related topics

isAssociated

Returns true if the target object is associated to the source object via the specified role name; otherwise, returns false.

Syntax

```
isAssociated <sourceBusinessObject> to
<rolename>(<targetBusinessObject>);;
```

Example

```
-- Determine if a customer is associated to an order
cust : Model::MIM::Customer := locate Model::MIM::Customer(id: 1234);
ord : Model::MIM::Order := locate Model::MIM::Order(orderId: '34825');
oclprintln(isAssociated cust to order(ord));
```

java

Invokes a JavaAction. See "Java actions" on page 61.

locate

Returns either a single business object or null. If more than one business object is returned, an exception is thrown.

Syntax

```
locate <fullClassName>(<optionalNamedParameterList>);
```

Example

```
fac : Model::InitializationFactory;
fac := locate Model::InitializationFactory();

ord : Model::MIM::Order := locate Model::MIM::Order(orderId: 18376);
```

Related topics

- To locate more than one instance, see "locateAll orderedBy" on page 41.
- To locate objects by object ID, see "LocateByIdentity" on page 69.

locateAll orderedBy

The `locateAll` (or `locateall`) keyword returns all of the objects in the current context of a specific type with attributes that match the specified parameter values. The action returns a java::util::Collection containing zero or more objects. The parameter values are optional. If you do not specify any parameter values, the action returns a collection of all the objects of the specified type.

The optional `orderedBy` (or `orderedby`) keyword causes the results to be ordered by the specified attribute value. If the attribute name is followed by :asc, the values will be in ascending order. If the attribute name is followed by :desc, the values will be in descending order. If no order qualifier exist after the attribute name, the order is ascending.

Syntax

```
locateAll <fullClassName>(<optionalNamedParameterList>);
```

```
locateAll <fullClassName>(<optionalNamedParameterList>) orderedBy
(<attrName>,<attrName>:asc,<attrName>:desc);
```

Example

```
-- Return all instances of Customer
coll : java::util::Collection;
coll := locateAll Model::MIM::Customer();


-- Return all instances of Customer whose first name is "John" and last
name is "Smith"
coll : java::util::Collection;
coll := locateAll Model::MIM::Customer(firstName: 'John',
   lastName: 'Smith');
```

Objects in the collection can be sorted in either ascending (asc) or descending (desc) order.

```
-- Return all instances of Customer whose last name is "Smith" ordered by
age from oldest to youngest (descending order).
coll : java::util::Collection;
coll := locateAll Model::MIM::Customer(lastName: 'Smith') orderedBy
   (age: desc);
```

Related topics

- To locate one instance, see "locate" on page 41.
- To locate objects by object ID, see "LocateByIdentity" on page 69.

loop do endloop

A loop statement executes a block of statements repeatedly. The loop continues to execute as long as the Boolean expression, <loop condition>, evaluates to true. The <loop_body> can contain zero or more statements.

Syntax:

```
loop <loop_condition> do
   <loop_body>
endloop;
```

Example:

```
factorial : java::lang::Integer;
count : java::lang::Integer;
factorial := 1;
count := 0;
loop count < 7 do
   count := count + 1;
   factorial := factorial * count;
endloop;
```

new

Creates a business object instance for the fully qualified name in the run-time database. Instances of Factory classes cannot be created.

> **Note:** The keyword new can be used in the same manner to create instances of Events, Messages, and Java Classes.

Syntax

```
new <fullClassName>(<optionalNamedParameterList>);
```

Example

```
cust : Model::MIM::Customer;
cust := new Model::MIM::Customer();
cust.firstName := 'John';
cust.lastName := 'Smith';

cust : Model::MIM::Customer;
cust := new Model::MIM::Customer(firstName: 'John', lastName: 'Smith');
```

A fully-qualified class is described in terms of its supporting packages, listed preceding the class and separated by double colons. In the example above, the qualified class, Customer is packaged under the MIM and within the Model model container.

Related topics

- To create a new association, see "new to" on page 43.

new to

Creates an Association or Association Class instance between two business object instnaces or a single buiness object instance and itself, in the run-time database.

Syntax

```
new <sourceBusinessObject> to <rolename>(<targetBusinessObject>);
```

Example

```
-- Create an association between a given customer and order
cust : Model::MIM::Customer := locate Model::MIM::Customer(id: 1234);
ord : Model::MIM::Order := locate Model::MIM::Order(orderId: '34825');
new cust to order(ord);

-- Create an association class between an order and an order item, where
an Association Class id defined between the two classes.
ord : Model::MIM::Order := new Model::MIM::Order();
item : Model::MIM::OrderItem := new Model::MIM::OrderItem();
item.name := 'Simple Widget';
linkObj : Model::MIM::A_orderItem_order;
linkObj := new ord to orderItem(item);
```

```
linkObj.quantity := 3
```

Related topics

- To delete an association, see "del to" on page 37.

oclAsArray

The `oclAsArray` keyword converts a List to an Array type.

Syntax

```
aList.oclAsArray(<arrayType>);
```

Example

In the following example, a List of Strings in converted to a StringArray.

```
lst : java::util::List := new java::util::ArrayList();
lst.add('one');
lst.add('two');
lst.add('three');

lst.oclAsArray(java::lang::String);
```

Related topics

- For evaluating an object's type, see "oclIsKindOf" on page 46.

oclAsList

The `oclAsList` keyword converts a Java Array to a ArrayList.

Syntax

```
<anArray>.oclAsType;
```

Example

In the following example, the split() method returns a Java StringArray. In order to iterate over the StringArray, it is converted to an ArrayList first.

```
'The quick brown fox'.split(' ').oclAsList -> iterate (word :
java::lang::String | oclprintln(word) );
```

Related topics

- For evaluating an object's type, see "oclIsKindOf" on page 46.

oclAsType

The `oclAsType` (or `oclastype`) keyword casts an object to the type specified by the `<classname>` parameter. The `<classname>` must be fully-qualified and its qualifiers must be separated by a double colon. Since OCL is a strong-typed language, you may need to cast (or coerce) an object of a more general type to a more specific type.

Syntax

```
<variableOrExpression>.oclAsType(<fullClassName>);
```

Example

In the following example, the next() function always returns a value to type java::lang::Object. However, the value needs to be cast to type Model::MIM::Customer.

```
custiter : java::util::Iterator := locateAll Model::MIM::Customer();
cust : Model::MIM::Customer;
cust := custIter.next().oclAsType( Model::MIM::Customer );
```

Related topics

- For evaluating an object's type, see "oclIsKindOf" on page 46.

oclCommit

For the open database transaction, The `oclCommit` (or `oclcommit`) keyword commits any pending changes from actions executed in an action semantics script. Following the commit, this action begins a new transaction. To use this in Cogility Action Pad, the Commit After Running option must be enabled. See "Commit After Running" on page 88 of the guide, *Model Deployment & Execution in Cogility Studio*.

This action must be enclosed in a try-catch block that will catch a com::cogility::OCLTransactionException or any of its superclasses (com::ohana::common::errors::EncapsulatingException, java::lang::Exception, or, java::lang::Throwable). A syntax error will result if no such try-catch block is detected.

Syntax

```
oclCommit;
```

Examples

- See "Batch processing" on page 130 in the guide, *Model Deployment & Execution in Cogility Studio*.

Related topics

- "oclRollback" on page 49.

oclDebugPrintln

Replaces the java action DebugPrintln().

Prints a string to standard output if debugging is configured. Lines that use oclDebugPrintln() print only if you set the following configuration parameter (in customConfigurations.txt or a project related configuration file) to "true":

```
com.cogility.println.debug=true
```

The parameter defaults to "false". See "Customizing installation configuration" on page 13 of the guide, *Model Deployment & Execution in Cogility Studio* for more information about configuration parameters. When you want to print the debug information to standard output, set the parameter to true. When you want to hide the debug statements, set the parameter to false.

Syntax

```
oclDebugPrintln(expression);
```

Example

```
coll : java::util::Collection := locateAll Model::MIM::Customer();
oclDebugPrintln(coll);
```

oclIsKindOf

The `oclIsKindOf` (or `ocliskindof`) keyword evaluates whether an object is of the type (or is a subtype of the type) specified in the `<classname>` parameter and returns a boolean. The `<classname>` must be fully-qualified and its qualifiers must be separated by a double colon.

Syntax

```
<variableOrExpression>.oclIsKindOf(<fullClassName>);
```

Example

In this example, an association exists between the Address class and the Customer class. In addition, the Customer class is defined as a superclass with subclasses CorporateCustomer, ResidentialCustomer and GovernmentCustomer. Traversing the association creates a collection of all sub types.

```
coll : address.customer;
coll -> iterate (cust : Model::MIM::Customer |
   if cust.oclIsKindOf(Model::MIM::CorporateCustomer) then
      -- process corporate customer
   elseif cust.oclIsKindOf(Model::MIM::GovernmentCustomer) the
      -- process government customer
   else
      -- process residential customer
   endif;
);
```

Related topics

- For casting an object to a type, see "oclAsType" on page 45.

oclLinkObject to

Returns a link object of a given source, target and role name.

Syntax

```
oclLinkObject <sourceBusinessObject> to
<associationClassRolename>(<targetBusinessObject>);
```

Example

```
-- print out the quantity of each item in a given order.
ord : Model::MIM::Order := locate Model::MIM::Order(orderId: 18735);
ord.orderItem -> iterate (item : Model::MIM::OrderItem;
   linkObj : Model::MIM::A_orderItem_order;
   linkObj := oclLinkObject ord to orderItem(item);
   oclprintln(linkObj.quantity);
   );
```

oclMaxLenOf

Returns the number of allowable characters (Size value) for a given String attribute for a given business object type. The specified variable or expression must evalutate to a valid business object type. This keyword can only be used with attributes of type String.

> **Note:** The default Size value of a String attribute is 0, which equates to 256 characters.

Syntax

```
<variableOrExpression>.oclMaxLenOf(<nameOfStringTypeAttr>);
```

Example

```
-- print out the size of the zip attribute
addr : Model::MIM::Address;
oclprintln(addr.oclMaxLenOf(zip));
```

oclPerform

Allows a business object (such as a MIM class, DIM class or factory) or a transient object (such as a Message, Event, or TransientClass) to perform an operation where the operation name is an expression that is computed at run time. Instead of using ‘if’ statements to perform the operation, your action semantics can simply follow an algorithm for generating the operation name and calling it.

As with normal calls to operations, the business object or transient object can be null which makes it a 'static' call to the operation. As such, within the operation body the value of 'self' will be null. For all syntaxes, there must be an expression that computes the operation's name. When business object or transient object is represented by a String expression, it must represent the fully qualified classname on which the Operation is defined. The fully qualified name can be double colon separated or dot separated.

Syntax

The syntax may take any of the following forms:

```
<variableOrExpression>.oclPerform(<operationNameExpression>);
```

Above is the basic syntax where the `OperationNameExpr` is the computed operation name.

```
<variableOrExpression>.oclPerform(<operationNameVarOrExpr>) returning
<returnTypeOfOperation>;
```

In the above syntax, if the operation returns a value, the `returning` clause specifies the type of value that is expected to be returned. The `FullXmlKeyOfReturnType` may be any model-defined type such as `ModelName::MIM::ClassName` or `java::util::Map`. If there is no `returning` clause, the oclPerform statement does not return a value, even if the operation itself does return a value.

```
<variableOrExpression>.oclPerform(<operationNameVarOrExpr>) using
(argumentMapVarOrExpr);
```

For operations that have arguments, use the syntax above. The `using` clause must have an expression that yields a `java::util::Map` that contains the arguments to the operation. For that Map, the `key` is the name of an Operation argument or parameter and the `value` is the value for that argument.

```
<variableOrExpression>.oclPerform(<operationNameVarOrExpr>) using
(argumentMapVarOrExpr) returning <returnTypeOfOperation>;
```

For operations that have both arguments and a return value, use the syntax above.

Exceptions

Since many things cannot be checked at compile time, using an oclPerform statement can result in several different runtime errors. If a runtime error occurs, a com::cogility::OCLPerformException is thrown. Some examples of problems that can result in a runtime error are:

- The `OperationNameExpr` results in a name of an operation that does not exist.
- There is no operation that matches the arguments from the `ArgumentMapExpr`.
- The data types of the `ArgumentMap` values do not match the expected data types of the operation parameters.
- The return type of the operation does not match the type in the `returning` clause.
- The oclPerform statement has a `returning` clause but the operation does not return a value.

One case where a runtime error does not result in an OCLPerformException is when another exception occurs within the operation being performed. For example, if a java::io::FileNotFoundException occurs within the performed operation, the java::io::FileNotFoundException is the exception that will be thrown.

oclPrintln

The `oclPrintln` (or `oclprintln`) keyword replaces the java action println().

Prints a string to standard output if debugging is configured.

Syntax

```
oclPrintln(expression);
```

Example

```
oclPrintln('Hello World');
```

oclRollback

For the open database transaction, The `oclRollback` (or `oclRollBack or oclrollback`) keyword rolls back any pending changes from actions executed in an action semantics script. Following the rollback, this action begins a new transaction. To use this in Cogility Action Pad, the Commit After Running option must be enabled. See "Commit After Running" on page 88 of the guide, *Model Deployment & Execution in Cogility Studio*.

This action must be enclosed in a try-catch block that will catch a com::cogility::OCLTransactionException or any of its superclasses (com::ohana::common::errors::EncapsulatingException, java::lang::Exception, or, java::lang::Throwable). A syntax error will result if no such try-catch block is detected.

Syntax

```
oclRollback;
```

Examples

- "Batch processing" on page 130 of the guide, *Model Deployment & Execution in Cogility Studio*.
- "Batch processing example" on page 133 of the guide, *Modeling with Cogility Studio*.

Related topics

- "oclCommit" on page 45.

oclTypeName

The `oclTypeName` (or `ocltypename`) keyword returns the fully qualified name of a given object as a String.

Syntax

```
<variableOrExpression>.oclTypeName;
```

Example

```
cust : Model::MIM::ResidentialCustomer;
oclprintln(cust.oclTypeName);
```

Related topics

- For evaluating an object's type, see "oclIsKindOf" on page 46.
- For casting an object to a type, see "oclAsType" on page 45.

pause

Outputs variables and their values for the set of Actions that contain the pause keyword. The pause keyword can exist in either the model or in the Action Pad. Actions executing in the application server will output to the application server log. Those executing in the Action Pad will output to the Action Pad console.

The pause keyword can perform an additional function in the Action Pad. When the "Enable 'pause' Dialog" option is enabled, the pause keyword halts execution and displays a dialog with the same variables and their values. To enable the dialog in the Action Pad, see "Enable 'pause' Dialogs" on page 89 of the guide, *Model Deployment & Execution in Cogility Studio*. The pause dialog provides various debugging options such as to continue execution, continue execution suppressing future dialogs, to terminate, and to terminate and rollback the transaction.

Syntax

```
pause (zeroOrMoreCommaSeparatedVariableNames);
```

Example

```
pause() -- display all variables and values for the set of executing
actions.

pause(cust, ord, item); -- display the variables cust, ord, and item, and
their associated values.
```

publish do

Publishes either an external (JMS) or internal event to a target business object instance.

Syntax

The following syntax applies to publishing an external event (JMS Message). A specific message model (EventToMessageConversion and Destination objects) must be defined in Modeler to support publishing an external event (JMS Message).

```
publish <subjectNameString> do
<eventClassFullName>(<namedParameterList>);

publish <subjectNameString> do <eventInstance>;
```

The following syntax appies to pubishing an internal event. An internal Event must be defined as such in Modeler.

```
publish <businessObject> do <eventClassFullName>(<namedParameterList>);

publish <businessObject> do <eventInstance>;
```

Example

Publishing External Event (JMS Message)

```
publish 'CreationAck_Topic' do SimpleModel::CreationAck_Event(customerID:
cust.customerID);
```

```
evnt : SimpleModel::CreationAck_Event := new
SimpleModel::CreationAck_Event(customerID: cust.customerID);
publish 'CreationAck_Topic' do evnt;
```

Publishing Internal Event

```
fac : Model::ProcessFactory := locate Model::ProcessFactory();
publish fac do Model::Process_Event(id: ord.id, bom: bomStr);
evnt : Model::Process_Event := new Model::Process_Event(id: ord.id,
   bom: bomStr);
publish fac do evnt;
```

publishJMS do

Publishes an "internal" Event on a fixed JMS topic, providing all the benefits of JMS without having to create the necessary Conversion and Destination objects.

The use of the these keywords will publish an "internal" Event on the **CogilityInternalEventTopic**.

Syntax

```
publishJMS <businessObject> do
<eventClassFullName>(<namedParameterList>);
```

```
publishJMS <businessObject> do <eventInstance>
```

Example

```
cust : SimpleModel::MIM::Customer;
cust := locate SimpleModel::MIM::Customer(customerID: 12345);
publishJMS cust do SimpleModel::UpdateProfileEvent(age: 22);
```

```
cust : SimpleModel::MIM::Customer;
cust := locate SimpleModel::MIM::Customer(customerID: 12345);
evnt : SimpleModel::UpdateProfileEvent;
evnt := new SimpleModel::UpdateProfileEvent(age: 22);
publish cust do evnt;
```

query

Executes a custom query on the run-time repository. It returns a list of instances of query type.

Syntax

In the following syntax, the query name must be the fully-qualified name.

```
query <customQueryFullName>(<namedParameterList>);
```

Example

The following example is from the tutorial. It requires two inputs dateSince (string) and minSalesAmount (double).

```
list : java::util::List;
list := query SimpleModel::SalesQuery(dateSince: '20040101',
minSalesAmount: 10.00);
list -> iterate (
   qry : SimpleModel::SalesQuery |
      oclPrintln(qry.employeeName.
         concat('\t\t\t\t').concat(qry.customerState).
         concat('\t\t\t\t').concat(qry.productName).
         concat('\t\t\t\t').concat(qry.numberSold).
         concat('\t\t\t\t').concat(qry.saleAmount));
);
```

Related topics

- See "Custom queries" on page 101.
- See also, "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*.

return

Returns a value of the Action Semantic statement which follows it, then exits.

If a return type is defined for an operation, the implementing method must return a value of that type.

Syntax

```
return;
```

```
return <expression>;
```

The `<expression>` may be a variable, a constant or a complex expression.

Examples

```
return price;
```

```
return 3.14159 * radius * radius;
```

selectSQL

Reads the data table of a specified legacy information model (LIM). Returns a `java.sql.ResultSet` object. You may pass in an entire SQL select statement as a parameter enclosed in single quotes (`'`), or only the parameters to a select statement object. See "Select objects" on page 70 of the guide, *Modeling with Cogility Studio* for more information.

Syntax

```
selectSQL <SelectStatementFullName>(<namedParameterList>);
```

```
selectSQL <LIMFullName>(<stringOfSQL>);
```

Note that if the passed-in values are Strings, they must be enclosed in single quotes. Because the entire SQL expression is passed as a String, any Strings within must be preceded with the escape character, a backslash (\).

Likewise, if you wish to pass attribute values to the SQL statement programatically (or dynamically), you must concatenate the variable holding the attribute value to the submitted String.

Examples

In this example, a Select Statement object is invoked using the selectSQL keyword. The Select Statement is defined with a single parameter named 'jobName'.

```
rs : java::sql::ResultSet;
rs := selectSQL Model::SCOTT::GetSalesmen(jobName: 'SALESMAN');
loop rs.next() do
   oclprintln(rs.getString('ENAME').
      concat(' - ').concat(rs.getString('EMPNO')));
endloop;
```

The following example yields the same result; however, the selectSQL keyword is used to execute a SQL select statement.

```
rs : java::sql::ResultSet;
rs := selectSQL Model::SCOTT(
   'SELECT * FROM EMP WHERE JOB = \'SALESMAN\'');
loop rs.next() do
   oclprintln(rs.getString('ENAME').
      concat(' - ').concat(rs.getString('EMPNO')));
endloop;
```

Related topics

- See also "executeSQL" on page 39.
- For more information about working with a legacy information model, see "Legacy Information Model" on page 61 of the guide, *Modeling with Cogility Studio*.

super

Calls an operation in a superclass of the class where the method is defined.

Syntax

```
super.<operation>(<arg_name>:<value>,<arg_name>:value,...);
```

Examples

```
super.initialize(nameParam: nameParam, addressParam: addressParam);
```

Related Topics

- To call operations of the object's own class, see "self" on page 26.

try catch endtry

An exception is a signal that indicates the occurrence of an error condition. The try-catch construct is a mechanism to deal with the exceptions. Exceptions thrown during execution of the `try` block can be caught and handled by one or more `catch` blocks.

Syntax: try-catch statement

```
try
   <statements>
catch(<exception_type><exception_variable>)
   <statements>
endtry;
```

Example:

```
s : java::lang::String;

try
-- The next statement causes an Exception (indexes out of bounds)
   s := 'abcdefgh'.substring( 20, 25 );
-- Execution never gets here. The Exception causes the execution
-- to go straight to the 'catch' block.
   s := 'Never Gets Here';
catch ( java::lang::Exception ex )
-- Caught Exception handled here
   s := 'A Default Error Value';
endtry;
-- Execution gets here when: the try block completes, or,
-- the catch block completes;
```

throw

A `throw` statement throws an exception that can be caught by an enclosing `try-catch` statement.

Syntax: throw exception statement

```
throw <exception>;
```

Example:

```
result : java::lang::String;
sum : java::lang::Long;
sum := 0;
count : java::lang::Integer;
count := 0;
try
   loop count < 1000 do
      count := count + 1;
      sum := sum + count;
      if sum > 10000 then
         -- Short circuit loop by throwing an Exception that
         -- causes execution to continue in the catch block.
         exc : java::lang::Exception;
         exc := new java::lang::Exception('Sum exceeds 10000');
```

```
         throw exc;
      endif;
   endloop;
   result := 'Summation did NOT reach 10000.';
catch (java::lang::Exception ex)
   result := 'Summation exceeds 10000 for count='.concat( count );
endtry;
```

$

The $ character is used to generate a collection of link objects (AssociationClass instances)

Syntax: throw exception statement

```
businessObj$associationClassRoleName;
```

Example:

```
ord : SimpleModel::MIM::Order := locate
   SimpleModel::MIM::Order(orderID: 33646);
lnkObjColl : java::util::Collection;
lnkObjColl := ord$product;
oclprintln('There are '.concat(lnkObjColl.size()).
   concat(' line item(s) in the order.'));
```

Collection operations

The arrow operator (->) combines an iteration over a collection (java.lang.Collection or any subclass) with an operation on the members of the collection. This is a compact looping construct similar to Smalltalk constructs like `do:`, `collect:`, `select:`, `detect:`. The basic syntax is as follows:

```
<result var> := aColl -> <operation> (varName : classSpec | <statements>);
```

In the line above, `<result var>` is the variable that holds the return value of the operation, `aColl` is the Collection object over which you want to iterate, `<operation>` is the keyword that performs an operation of the types listed below, `varName` holds a the current member of the Collection, the `<statements>` evaluate the members for the operations, and the `classSpec` describes the fully-qualified classname of the members of the Collection.

The `<operation>` keyword defines a particular type of operation which returns a particular Java type. The operation types, defined by the corresponding keyword are as follows. Note that the action symantics require a variable of the particular type to which the result of the operation must be returned.

collect

Performs an iteration over a java.lang.Collection, collects members or elements of the members in the Collection and returns them in an ArrayList. Returns an empty ArrayList if the Collection is empty. Used in conjunction with the arrow operator (->).

Syntax

```
aList := aColl -> collect(varName : classSpec | <statements that evaluate
to an Object>);
```

Example

```
-- Convert POIDs to Customers
listOfPoids : java::util::ArrayList;
-- <action that loads the listOfPoids ArrayList goes here>
listOfCusts : java::util::ArrayList;
listOfCusts := listOfPoids -> collect ( aPoid : java::lang::String |
    Model::MIM::Customer( poid: aPoid ) );
```

detect

Performs an iteration over a java.lang.Collection, returns the first member in the Collection for which the statements evaluate to true. Returns null if the Collection is empty or if none of the members evaluates to true. Used in conjunction with the arrow operator (->).

Returns the same type as the variable declaration within the detect statement's block (the type declaration between the single colon and the vertical bar).

```
str : java::lang::String;
str := aCollOfStrings -> detect ( currStr : java::lang::String |
   currStr.length() > 10 );
```

Syntax

```
anObj := aColl -> detect (varName : classSpec | <statements evaluate to
boolean>);
```

Example

```
-- Get rid of one bad apple
collOfApples : java::util::Collection;
-- <action that loads the collOfApples Collection goes here>
badApple : java::lang::Object;
badApple := collOfApples -> detect ( apple : Model::MIM::Apple |
   apple.isBad );
if badApple <> null then
   del badApple;
endif;
```

exists

Performs an iteration over a java.lang.Collection, returns a Boolean true if the statements that evaluate any of the members return true. Returns false if the Collection is empty or if none of the members evaluates to true. Used in conjunction with the arrow operator (->).

Syntax

```
aBool := aColl -> exists  (varName : classSpec | <statements evaluate to
boolean>);
```

Example

```
-- Determine whether a Customer has Orders that have not been filled
customer : Model::MIM::Customer;
-- <action that loads the customer goes here>
needsFilling : java::lang::Boolean;
needsFilling := customer.orders -> exists ( order : Model::MIM::Order |
   not order.isFilled );
```

forAll

Performs an iteration over a java.lang.Collection, returns a Boolean false if the statements that evaluate any of the members return false. Returns true if the Collection is empty or if none of the members evaluates to false. Used in conjunction with the arrow operator (->).

Syntax

```
aBool := aColl -> forAll  (varName : classSpec | <statements evaluate to
boolean>);
```

Example

```
-- Determine whether all students passed the test
allStudents : java::util::Collection;
-- <action that loads the allStudents Collection goes here>
allPassed : java::lang::Boolean;
allPassed := allStudents -> forAll ( student : Model::MIM::Student |
   student.score >= 70 );
```

iterate

Performs an iteration over a java.lang.Collection, and does not return a value. Performs the actions in the statements after the vertical bar (|) for each member of the Collection. Used in conjunction with the arrow operator (->).

Syntax

```
aColl -> iterate (varName : classSpec | <statements>);
```

```
aMap -> iterate ( keyVarName : <typeDecl>, valueVarName : <typeDecl> |
<statements> );
```

Example

```
-- Output a simple report on a Customer's Orders
customer : Model::MIM::Customer;
-- <action that loads the customer goes here>
customer.order -> iterate ( order : Model::MIM::Order |
   java com::cogility::Println( 'Product='.concat( order.productName ) );
   java com::cogility::Println( '  Price='.concat( order.price ) );
   java com::cogility::Println( '  Quant='.concat( order.quantity ) );
);
```

```
--Map example
aMap : java::util::HashMap := new java::util::HashMap();
aMap.put('Peter', 45);
aMap.put('Paul', 38);
aMap.put('Mary', 45);

aMap -> iterate (name : java::lang::String, age: java::lang::Integer |
   oclprintln(name.concat('\t\t').concat(age));
   );
```

reject

Performs an iteration over a java.lang.Collection, returns a Collection of members for which the statements evaluate to false (rejecting those that evaluate to true). Returns an empty Collection if the Collection is empty. Used in conjunction with the arrow operator (->).

> **Note:** The return is of the same type as the Collection - that is, if the Collection is a HashSet, the return will be a HashSet.

Syntax

```
aCol2 := aColl -> reject  ( varName : classSpec | <statements evaluate to
boolean>);
```

Example

```
-- Reduce a List to a set of Customer with good credit
allCustomers : java::util::ArrayList;
-- <action that loads the allCustomers ArrayList goes here>
goodCreditCusts : java::util::ArrayList;
goodCreditCusts := allCustomers reject ( customer : Model::MIM::Customer |
   customer.hasBadCredit() );
```

select

Performs an iteration over a java.lang.Collection, returns a Collection of members for which the statements evaluate to true. Returns an empty Collection if the Collection is empty. Used in conjunction with the arrow operator (->).

> **Note:** The return is of the same type as the Collection - that is, if the Collection is a HashSet, the return will be a HashSet.

Syntax

```
aCol2 := aColl -> select ( varName : classSpec | <statements evaluate to
boolean>);
```

Example

```
-- Select a List of Customers to receive a promotional eMail
allCustomers : java::util::ArrayList;
-- <action that loads the allCustomers ArrayList goes here>
```

```
eMailableCusts : java::util::ArrayList;
eMailableCusts := allCustomers select ( customer : Model::MIM::Customer |
   customer.eMailAddr <> null );
```

5 Java actions

In Cogility Studio, you can work with Java in two ways: you can define types with Java classes and you can integrate Java expressions directly into your action semantics. For example, the following expression declares a variable of the Java Collection type:

```
coll : java::util::Collection;
```

You then have access to the members of the Java class the same way as any other class in your action semantics. See "Variable declarations" on page 11, "Class attributes" on page 13 and "Class operations" on page 13 for more information.

To integrate Java directly into your action semantics, use Java actions. A Java action is compiled Java invoked during the execution of the action semantics. Java actions start with the `java` keyword, followed by the classpath qualifiers, the Java action expression and its parameters. The syntax is as follows:

```
java <full qualified Java action>(<parameters>);
```

Several predefined Java actions are available and described below. You can also define your own Java actions. See "Custom Java actions" on page 89.

In qualifying your Java classes, you may refer to these by an alias that abbreviates the fully-qualified classname. You define the alias in your model using a Java package alias. See "Java package alias" on page 318 of the guide, *Modeling with Cogility Studio*.

CreateInstanceOfClass

Creates an instance of a desired class, including MIMClass, DIMClass, Event, Message, and TransientClass.

Syntax

```
java com::cogility::CreateInstanceOfClass(aClassname,attributeValuesMap);
```

Parameters

- `aClassname` - (java::lang::String) the fully qualified name of the class to be instantiated.
- `attributeValuesMap` - (java::util::HashMap) a map of attribute names to attribute values of the attributes to be set. The attribute values must be of the correct type for the attribute, or, must be a String that can be converted to the correct type for the attribute.

Returns

An instance of an object. Depending on what is to be done with the object, it may need to be cast to the desired type.

Exceptions

CreateInstanceException will be thrown if:

- either parameter is null.
- the classname cannot be found or (business object class) is abstract.
- the attribue values map contains an unexpected attribute name.
- the attribue values map contains an attribute value of the wrong type.

Example

```
try
   -- inherited attributes
   attrMap : java::util::HashMap := new java::util::HashMap();
   attrMap.put('name', input.name);
   attrMap.put('contact', input.contact);
   attrMap.put('phoneNum', input.phone);

   -- custType -> (Corporate, Government, Internal, Residential)
   -- append custType to define the fully qualified name
   className : java::lang::String :=
'Model::MIM::'.concat(input.type).concat('Customer');

   -- define as the superclass Model::MIM::Customer, cast if necessary
   cust : Model::MIM::Customer;
   cust := (java com::cogility::CreateInstanceOfClass(className,
attrMap)).oclAsType(Model::MIM::Customer);

   output.custId := cust.customerId;
   output.status := 'Successfully created';
catch (java::lang::Throwable ex)
   output.status := 'Exception Thrown: '.concat(ex.getMessage());
endtry;
```

Related topics

None

CreateInstanceOfXSDType

Takes an XML string and creates an instance of an XSD type. The XSD type definition must already exist in the database (from a model-defined XSD artifact) and the XML must match that XSD type and supply the values for the attributes.

Syntax

```
java com::cogility::CreateInstanceOfXSDType(xmlString,fullXSDTypeDef);
```

Parameters

- `xmlString` - (java::lang::String) the XML string from which the instance's data is extracted. The XML string contains the attribute values for an instance of the XSD type, usually, an input to an inbound web service.
- `fullXSDTypeDef` - (java::lang::String) the fully-qualified XSD type definition. This is enclosed in single quotes and qualified using double colons. Using a string variable will generate a compilation error.

Returns

An instance of the specified XSD type definition, populated with values from the `xmlString`.

Exceptions

EncapsulatingException will be thrown if:

- the `xmlString` is malformed or not null.

Example

```
try
   per : cust::person;
   per := java com::cogility::CreateInstanceOfXSDType(input.xmlString,
'cust::person');

   cust : Model::MIM::Customer;
   cust := locate Model::MIM::Customer(firstName: per.firstname,
middleInitial: per.middleinitial, lastName: per.lastname);

   if cust <> null then
      output.status := 'Customer already exists';
      return 0;
   endif;

   cust := new Model::MIM::Customer(firstName: per.firstname,
middleInitial: per.middleinitial, lastName: per.lastname);

   output.id := cust.customerId;
   output.status := 'Success';
catch (java::lang::Throwable ex)
   output.status := 'Exception Thrown: '.concat(ex.getMessage());
endtry;
```

Related topics

See "XSD artifacts in action semantics" on page 196 of the guide, *Modeling with Cogility Studio*.

DebugPrintln

This keyword has been depricated and is replaced with the keyword oclDebugPrintln. Like `oclPrintln()`, this prints a string to standard output if debugging is configured. However, lines that use `oclPrintln()` always print. Lines that use `DebugPrintln()` print only if you set the following configuration parameter (in your CustomConfigurations.txt file) to "true":

```
com.cogility.println.debug=true
```

The parameter defaults to "false". See "Customizing installation configuration" on page 13 of the guide, *Model Deployment & Execution in Cogility Studio* for more information about configuration parameters. When you want to print the debug information to standard output, set the parameter to true. When you want to hide the debug statements, set the parameter to false.

Syntax

```
java com::cogility::DebugPrintln(toBePrinted);
```

Parameters

- `toBePrinted` - a (java::lang::Object) to be printed to standard output.

Returns

Nothing

Exceptions

None

Example

```
java com::cogility::DebugPrintln('Begin StateMachine Execution');
```

Related topics

See "oclDebugPrintln" on page 45.

See "oclPrintln" on page 48

See "Println" on page 71.

DumpDataToFile

Writes data to a file.

Syntax

```
java com::cogility::DumpDataToFile(fileName, dataToDump, appendToFile);
```

Parameters

- `fileName` - (java::lang::String) for the name of the file in which to dump the data.
- `dataToDump` - (java::lang::String) data to be output to the file.
- `appendToFile` - (java::lang::Boolean) indicating, if `true`, to append the String to the existing contents of the file, or, if `false`, to overwrite the existing contents of the file.

Returns

Nothing

Exceptions

EncapsulatingException will be thrown if:

- the file cannot be found.

Example

```
try
   custColl : java::util::Collection;
   custColl := locateAll Model::MIM::Customer() orderedBy (lastName:
asc);
   fileName, dataToDump : java::lang::String;
   fileName :=
'%OEMHOME%\\Examples\\OCL\\DumpDataToFile\\customerDump.txt';

   custColl -> iterate (cust : Model::MIM::Customer |
      dataToDump := cust.lastName
         .concat(' ')
         .concat(cust.firstName)
         .concat(' ')
         .concat(cust.middleInitial)
         .concat(' - ')
         .concat(cust.customerId)
         .concat('\n');

      java com::cogility::DumpDataToFile(fileName, dataToDump, true);
      );
   output.status := 'Success';
catch (java::lang::Throwable ex)
   output.status := 'Exception Thrown: '.concat(ex.getMessage());
endtry;
```

ExecuteOCLScript

Executes an action semantics script that may be a top-level script that orchestrates a number of sub scripts to perform a sequence of tasks, such as in a regression test.

Syntax

```
java com::cogility::ExecuteOCLScript(filename, argc,
executeInSeparateTransaction);
```

Parameters

- `filename` - (java::lang::String) of the path to the file name for the script.
- `argc` - (java::util::List) of the arguments for the script.
- `executeInSeparateTransaction` - (java::lang::Boolean) that, if `true`, indicates that the script should execute within its own database transaction.

Returns

A java::lang::Object that holds the return value from the script.

Exceptions

EncapsulatingException will be thrown if:

- the file cannot be found.

Example

```
-- the calling script
argc : java::util::ArrayList := new java::util::ArrayList();
argc.add('Hello World!');
argc.add(100);
argc.add(true);

java
com::cogility::ExecuteOCLScript('%OEMHOME%\\Examples\\OCL\\ExecuteOclScri
pt\\CalledScript.ocl', argc, false);


-- the called script
oclprintln('ARGC SIZE='.concat(argc.size()));
cnt : java::lang::Integer := 0;
loop cnt <= (argc.size() - 1) do
   obj : java::lang::Object := argc.get(cnt);
   oclprintln(obj.concat(' - ').concat(obj.oclTypeName));
   cnt := cnt + 1;
endloop;
```

Note: In the script that is called, argc does not need to be defined.

Related topics

See "Action semantics batch files" on page 56 of the guide, *Using Actions in Cogility Studio*.

GetExtendedStateData

Obtains a serializable value from the extended state data of a state.

Syntax

```
java com::cogility::GetExtendedStateData(state, key);
```

Parameters

- `state` (java::lang::Object) that represents the state runtime instance whose extended data is accessed. In the case of a model where a state is entered repeatedly, each time the state is entered it has a new RuntimeStateInstance, and each RuntimeStateInstance has its own independent extended state data.
- `key` - (java::lang::String) for the name of the extended state value to look up.

Returns

The java::lang::Object for the value stored by SetExtendedStateData.

> **Note:** The returned object may need to be cast to the appropriate type.

Exceptions

EncapsulatingException will be thrown if:

- the `state` is null.

Example

```
cnt : java::lang::Integer;
cnt := java com::cogility::GetExtendedStateData(parent,
'counter').oclAsType(java::lang::Integer);

oclprintln('State2: Performing Repeated Step'.concat(' - ').concat(cnt));
```

InvokeDelegatedOperation

Invokes a specified Operation (`operationName`) defined on a business object which is the target of a source business object. The association(s) between the source and target classes must be defined withe the “Delegates Events” checkbox checked. All targets of the source object are considered. If `operationName` does not exist for the target, then it is ignored. If the Operation does exist, then the target object is checked to determine if it defines an Operation named “isApplicable”. If the isApplicable Operation doesn’t exist, then the `operationName` is invoked. If the isApplicable Operation does exist, its method is executed. If true then `operationName` is invoked. If false `operationName` is ignored.

Syntax

```
java com::cogility::InvokeDelegatedOperation(operationName,  instance,
operArgsMapOrNull,  isApplArgsMapOrNull);
```

Parameters

- `operationName` - (java::lang::String) is a String with the name of the Operation to be invoked.
- `instance` - (java::lang::Object) is the source BusinessObject from which the delegated target BusinessObjects are determined (any target of any Association that has the "Delegates Events" checkbox checked).
- `operArgsMapOrNull` - (java::util::Map) is a Map of argument values for the "operationName" Operation parameters (KEY=parameter name, VAL=argument value {of the correct type, or, a String that can be converted to the correct type}), or, null if "operationName" has no arguments.
- `isApplArgsMapOrNull` - (java::util::Map) is a Map of argument values for the "isApplicable" Operation's parameters (KEY=parameter name, VAL=argument value {of the correct type, or, a String that can be converted to the correct type}), or, null if "isApplicable" has no arguments, or, null if "isApplicable" is known not to exist.

Returns

Nothing

Exceptions

InvokeOperationException will be thrown if:

- the `operationName` is null.
- the `instance` is null.
- either Map contains keys that are not Strings.
- either Map contains keys that don't match the Operation parameters.

Example

```
argsMap : java::util::HashMap := new java::util::HashMap();
argsMap.put('strVal','BUY');
appMap : java::util::HashMap := new java::util::HashMap();
appMap.put('testVal', 1);
java com::cogility::InvokeDelegatedOperation('opX',
   clsA, argsMap, appMap);
```

Related topics

None

InvokeOperation

Allows you to call an Operation where the instance, type, and/or, operationName is not known until runtime.

Syntax

```
java com::cogility::InvokeOperation(operationName, classnameOrNull,
instanceOrNull, argumentsMapOrNull);
```

Parameters

- `operationName` - (java::lang::String) the name of the Operation to be invoked.
- `classnameOrNull` - (java::lang::String) the fully qualified classname where the 'static' Operation can be found, or, null if instanceOrNull is not null.
- `instanceOrNull` - (java::lang::Object) the instance whose Operation is to be called (with 'self' = to instanceOrNull), or, null if classnameOrNull is not null.
- `argumentsMapOrNull` - (java::util::Map) is a Map of argument values for the "operationName" Operation parameters (KEY=parameter name, VAL=argument value {of the correct type, or, a String that can be converted to the correct type}), or, null if "operationName" has no arguments.

Returns

the return value of the invoked operation or null.

Exceptions

InvokeOperationException will be thrown if:

- the `operationName` parameter is null.
- the `classnameOrNull` and the `instanceOrNull` parameters are both non-null.
- the class definition associated to the classnameOrNull or the instanceOrNull must be a class definition that can have operations.
- the Map contains keys that are not Strings.
- the Map contains keys that don't match the Operation parameters.
- the Map values don't match the required parameter types.
- the operationName must match a defined Operation for the target class definition.

Example

```
sumOfTwoLongs : java::lang::Long;
argsMap : java::util::HashMap := new java::util::HashMap;
argsMap.put( 'aLong1', 1234 );     -- Arg value of required type
argsMap.put( 'aLong2', '2345' );   -- Arg value of String that will be
converted to the required type
returnedValue : java::lang::Object;
-- Make a 'static' call ( classnameOrNull is non-null, instanceOrNull is
null )
try
   returnedValue := java com::cogility::InvokeOperation( 'sumOrTwoLongs',
      'Model::MIM::MIMClass', null, argsMap );
   sumOfTwoLongs := returnedValue.oclAsType( java::lang::Long );
   oclPrintln( 'sumOfTwoLongs='.concat(sumOfTwoLongs) );
catch ( com::cogility::InvokeOperationException ex )
   oclPrintln( 'sumOfTwoLongs failed: '.concat( ex.getMessage() ) );
endtry;
```

Related topics

None

LocateByIdentity

Locates either a business object or behavior runtime instance using its persistent object identifier (POID).

Each object in the run-time repository has a persistent object ID (POID) attribute of type String. You can use the Java action, `LocateByIdentity` to locate a specific object in the database by its POID. You can also use `LocateByIdentity` to locate runtime instances of objects such as state machines as well as business objects.

Although this java action can be used to locate either a business object or a behavior runtime instance, it is primary used for the latter. The keyword `locate` is primarily used to find a specific business object and it allows you to specify one or more attribute-value pairs to identify the correct object. Behavior runtime instances are meta-data and the associated schema is pre-defined. However, the key issue is that multiple behavior runtime instances, based on the same behavior definition, can exist on the same business object. In general, when a trigger event is sent to a business object, it is delivered to the first behavior runitime instance that can consume it. This method is non-

deterministic. Thus, if a trigger event must be sent to a specific behavior, they that specific behavior must be located.

It is assumed that the behavior which is waiting for the trigger, initiated a request for information, and is waiting for the response. Support is provided in the form of the bound variable smPoid. This variable (along with boPoid - business object poid) are accessible within the action fields of the various state types and the transition objects of the State Machine. The smPoid is added to the outgoing event providing the returning process sufficient information for the return trip.

> **Note:** Internal Events can not be sent to a Behavior Runtime Instance. Thus the use of this java action to location a Behavior Runtime Instance should be defined only in a business object locator (BOL) field.

To access a business object by its POID, you must explicitly define a `poid` attribute on the business object's class. See "POID attribute" on page 37 of the guide, *Modeling with Cogility Studio*.

Syntax

```
java com::cogility::LocateByIdentity(idString);
```

Parameters

- `idString` - (java::lang::String) for the Persistent Object ID (POID) of the object to be located.

Returns

A BusinessObjectWithIdentity.

Exceptions

EncapsulatingException will be thrown if:

- the return type cannot be determined, can occur if the `idString` is a non-poid string.

Example

The following example locates a state machine using its POID. A variable of type String is created and the bound variable `jmsMapMessage` (see "jmsMapMessage" on page 24) gets the value of the `deliverToPoid` attribute and assigns that value to the variable. Finally, the `LocateByIdentity` Java action returns the instance which has the POID value represented by the POID variable.

```
ord : Model::MIM::Order;
ord := (java com::cogility::LocateByIdentity(input.boPoid))
   .oclAsType(Model::MIM::Order);
publish ord do Model::TriggerEvent();
```

> **Note:** When the java action is used in an action field that is not a business object locator (BOL) field, you must cast the returned object to the desired business object type.

The following example is defined in the business object locator in an M2EConversion object, and is used to identify the behavior runtime instance to which an event will be sent.

```
smPoid : java::lang::String;
smPoid := jmsMapMessage.getString('smPoid');
return java com::cogility::LocateByIdentity(smPoid);
```

Example boPoid: BO^Model.MIM.Order^-601289ec:12730912b82:-7fe8

Example smPoid: BehaviorRTInstance^-601289ec:12730912b82:-7fca

You can also use bound variables to get the POID: For information about the `smPoid` and `boPoid` bound variables, see "boPoid" on page 21 and "smPoid" on page 27.

LockBusinessObject_Oracle

Takes a single, persistent BusinessObject as an argument and places a lock on that BusinessObject. The lock remains in effect until the next commit or rollback of the enclosing transaction.

> **Note:** This java action only applies to an Oracle run-time database.

Syntax

```
java com::cogility::LockBusinessObject_Oracle(aPersistentBusinessObject);
```

Parameters

- `aPersistentBusinessObject` - (java::lang::Object) a persistent business object.

Returns

Nothing

Exceptions

EncapsulatingException will be thrown if:

- the persistent business object is null.

SQLException will be throw if:

- the persistent business object is aleady locked.

Examples

```
cust : Model::MIM::Customer :=
   locate Model::MIM::Customer(customerId: input.custId);
java com::cogility::LockBusinessObject_Oracle(cust);
```

Println

Prints a line of text to standard output for debugging purposes.

Syntax

```
java com::cogility::Println(toBePrinted);
```

Parameters

- `toBePrinted` - (java::lang::Object) to be printed to standard output.

Returns

Nothing

Exceptions

None

Example

```
java com::cogility::Println('Hello World !');
```

SendEmail

Sends an e-mail message to a single recipient, using a named SMTP server to send the mail.

Syntax

```
java com::cogility::SendEmail(smtp,from,to,subject,body);
```

Parameters

- `smtp` - (java::lang::String) for the name of the mail server used to send the mail. If this is null or an empty string, the value at the system property com.ceira.ocl.action.mail.smtpserver is used instead. If that value is also null, an exception is thrown. This should be a local SMTP server that can forward mail to the desired recipient's destination.
- `from` - (java::lang::String) for the sender's e-mail address. If this is null or an empty string, the value at the system property com.ceira.ocl.action.mail.sender is used instead. If that value is also null, an exception is thrown.
- `to` - (java::lang::String) for the recipient's e-mail address.
- `subject` - (java::lang::String) for the subject of the message.
- `body` - (java::lang::String) for the body text of the message.

Returns

Nothing

Exceptions

EncapsulatingException will be thrown if:

- the `smtp`, `from` or `to` arguments have invalid values.

Example

```
java com::cogility::SendEmail('', '', 'foo@example.com', 'Greeting',
   'Hello', messageText);
```

SetAttributeValuesOfInstance

Sets zero or more attribute values for a given business object.

Syntax

```
java com::cogility::SetAttributeValuesOfInstance(anInstance,
attributeValuesMap);
```

Parameters

- `anInstance` - (java::lang::Object) an instance of a MIMClass or DIMClass
- `attributeValuesMap` - (java::util::Map) of the arguments for the script.

Returns

nothing

Exceptions

SetAttributeValuesException will be thrown if:

- either parameter is null.
- the attribues are not valid for the instance.
- the attribue values map contains an unexpected attribute name.
- the attribue values map contains a key that is not a string.
- a value cannot be converted to the expected type.

Example

```
attrMap : java::util::HashMap := new java::util::HashMap();
attrMap.put('name', input.name);
attrMap.put('contact', input.contact);
attrMap.put('phoneNum', input.phone);

-- locate the customer
cust : Model::MIM::ResidentialCustomer := locate
Model::MIM::ResidentialCustomer(customerId: input.custId);

java com::cogility::SetAttributeValuesOfInstance(cust, attrMap);
```

SetExtendedStateData

Places a serializable value in the extended state data of a state.

Syntax

```
java com::cogility::SetExtendedStateData(state,key,value);
```

Parameters

- `state` - (java::lang::Object) whose extended data is to be accessed. In the case of a model where a state is entered repeatedly, each time it is entered the state has a new RuntimeStateInstance, and each RuntimeStateInstance has its own independent extended state data.
- `key` - (java::lang::String) for the name of the extended state value to look up.

- `value` - (java::io::Serializable) with the value to be stored.

Returns

Nothing

Exceptions

EncapsulatingException will be thrown if:

- the `state` parameter is null.

Example

```
cnt : java::lang::Integer := 1;
java com::cogility::SetExtendedStateData(parent, 'counter', cnt);
```

SortBusinessObjects

Sorts a collection of business objects. You may specify the attribute on which to sort and whether the order is ascending or descending.

Syntax

```
java com::cogility::SortBusinessObjects(objsToSort, keyAttrName,
   ascendingOrder);
```

Parameters

- `objsToSort` - (java::util::Collection) of the business objects to be sorted.
- `keyAttrName` - (java::lang::String) name of the attribute on which to sort.
- `ascendingOrder` - (java::lang::Boolean) `true` if the objects are to be sorted in ascending order, `false` for descending order.

Returns

A java::util::List of sorted business objects. In ascending order, null attribute values sort in front of non-null values. For Booleans, false sorts in front of true. For Strings, uppercase characters sort in front of lowercase characters.

Exceptions

EncapsulatingException will be thrown if:

- the `keyAttrName` is not valid for the business object.
- the datatype of the attribute values is not sortable.

Example

```
coll : java::util::Collection;
list : java::util::List;
coll := locateAll MyModel::MIM::Customer;
list := java com::cogility::SortBusinessObjects(coll,'lastName',true);
```

WalkAssoc

Locates a single target of an association or association class. This Java action should only be used when the target multiplicity is 1 or when determining if one or more targets exist.

Syntax

```
java com::cogility::WalkAssoc(sourceObject, roleName);
```

Parameters

- `sourceObject` - (java::lang::Object) modeled business object that is the source object for the association.
- `roleName` - (java::lang::String) for the name of the role whose target is to be located.

Returns

A single business object if one or more targets exist, or null if no target is found. If more than one target exists, the returned business object is non-deterministic and may not be the same from call to call.

Exceptions

EncapsulatingException will be thrown if:

- the `sourceObject` is null.

Example

```
ord : Model::MIM::Order := locate Model::MIM::Order(orderId:
input.orderId);

if ord = null then
   output.status := 'Cannot find Customer. Order does not exist.';
   return 0;
endif;

cust : Model::MIM::Customer;
cust := java com::cogility::WalkAssoc(ord, 'customer');
```

XmlDec

Converts a string that contains special, escape-encoded XML characters (`< > &` and so forth) into decoded characters (`< >` and `&`).

Syntax

```
java com::cogility::XmlDec(dataToDecode);
```

Parameters

- `dataToDecode` - (java::lang::String) the XML to be decoded.

Returns

A java::lang::String with the decoded XML.

Exceptions

None

Example

```
encodedValue : java::lang::String;
decodedValue : java::lang::String;

-- Decode an encoded value that has been extracted from XML
encodedValue := 'a &lt; b &amp;&amp; c &gt;= d';
decodedValue := java com::cogility::XmlDec( encodedValue );
java com::cogility::Println( decodedValue );
```

This writes the following to standard output:

```
a < b && c >= d
```

Related topics

- See "XmlEnc" on page 76.

XmlEnc

Converts an XML string that contains special XML characters (`< > &` and so forth) into escape-encoded characters (`< > &`).

Syntax

```
java com::cogility::XmlEnc(dataToEncode);
```

Parameters

- `dataToEncode` - (java::lang::String) with the XML to be encoded.

Returns

A java::lang::String with the encoded XML.

Exceptions

None

Example

```
javaSnippet : java::lang::String;
encodedValue : java::lang::String;
validXML : java::lang::String;

-- Encode a snippet of Java as an attribute in XML
javaSnippet := 'a < b && c >= d';
```

```
encodedValue := java com::cogility::XmlEnc( javaSnippet );
validXML := '<java><snippet>'.concat(encodedValue)
   .concat('</snippet></java>');
java com::cogility::Println( validXML );
```

This writes the following to standard output:

```
<java><snippet>a &lt; b &amp;&amp; c &gt;= d</snippet></java>
```

Related topics

- See "XmlDec" on page 75.

XmlGetBoolean

Returns a boolean value from an XML document.

Syntax

```
java com::cogility::XmlGetBoolean(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87)
- `xmlPath` - (java::lang::String) specifying the path to the value within `xmlDoc`.

Returns

A java::lang::Boolean with the value, or null if the value is empty, invalid or if the path doesn't exist.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
booParam : java::lang::Boolean;
booAttr : java::lang::Boolean;
booError : java::lang::Boolean;

xmlString := '<Demo Param="false"><Attr>true</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
booParam := java com::cogility::XmlGetBoolean( xmlDoc, '/Demo/@Param' );
booAttr := java com::cogility::XmlGetBoolean( xmlDoc,
   '/Demo/Attr/text()' );
booError := java com::cogility::XmlGetBoolean( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( booParam );
java com::cogility::Println( booAttr );
java com::cogility::Println( booError );
```

This writes the following to standard output:

```
false

true
```

```
null
```

XmlGetByte

Returns a Byte value (a signed 8-bit number [-128 to 127]) from an XML document.

Syntax

```
java com::cogility::XmlGetByte( xmlDoc, xmlPath );
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87)
- `xmlPath` - (java:lang::String) specifying the path to the value within `xmlDoc`.

Returns

A java::lang::Byte with the value, or null if the value is empty, invalid or if the path doesn't exist.

Exception

An EncapsulatingException is thrown if:

- the `xmlPath` argument cannot be parsed.
- the specified value is not a number.
- the value exceeds 8 bits.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
byteParam : java::lang::Byte;
byteAttr : java::lang::Byte;
byteError : java::lang::Byte;

xmlString := '<Demo Param="-1"><Attr>22</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
byteParam := java com::cogility::XmlGetByte( xmlDoc, '/Demo/@Param' );
byteAttr := java com::cogility::XmlGetByte( xmlDoc, '/Demo/Attr/text()' );
byteError := java com::cogility::XmlGetByte( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( byteParam );
java com::cogility::Println( byteAttr );
java com::cogility::Println( byteError );
```

This writes the following to standard output:

```
-1

22

null
```

XmlGetDouble

Returns a Double value from an XML document.

Syntax

```
java com::cogility::XmlGetDouble(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value within `xmlDoc`.

Returns

A java::lang::Double value, or null if the value is empty, invalid or if the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed, or if the specified value is not a number.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
dblParam : java::lang::Double;
dblAttr : java::lang::Double;
dblError : java::lang::Double;

xmlString := '<Demo Param="3.14"><Attr>2</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
dblParam := java com::cogility::XmlGetDouble( xmlDoc,'/Demo/@Param' );
dblAttr := java com::cogility::XmlGetDouble( xmlDoc,
   '/Demo/Attr/text()' );
dblError := java com::cogility::XmlGetDouble( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( dblParam );
java com::cogility::Println( dblAttr );
java com::cogility::Println( dblError );
```

This writes the following to standard output:

```
3.14

2.0

null
```

XmlGetFloat

Returns a Float value from an XML document.

Syntax

```
java com::cogility::XmlGetFloat(xmlDoc, xmlPath);
```

The arguments are as follows:

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value within `xmlDoc`.

Returns

A java::lang::Float with the value, or null if the value is empty, invalid or if the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed, or if the specified value is not a number.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
fltParam : java::lang::Float;
fltAttr : java::lang::Float;
fltError : java::lang::Float;

xmlString := '<Demo Param="3.14"><Attr>2</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
fltParam := java com::cogility::XmlGetFloat( xmlDoc, '/Demo/@Param' );
fltAttr := java com::cogility::XmlGetFloat( xmlDoc, '/Demo/Attr/text()' );
fltError := java com::cogility::XmlGetFloat( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( fltParam );
java com::cogility::Println( fltAttr );
java com::cogility::Println( fltError );
```

This writes the following to standard output:

```
3.14

2.0

null
```

XmlGetInteger

Returns an Integer value from an XML document.

Syntax

```
java com::cogility::XmlGetInteger(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).

- `xmlPath` - (java::lang::String) specifying the path to the value within `xmlDoc`.

Returns

A java::lang::Integer with the value, or null if the value is empty, invalid or if the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed, or if the specified value is not an integer.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
intParam : java::lang::Integer;
intAttr : java::lang::Integer;
intError : java::lang::Integer;

xmlString := '<Demo Param="33"><Attr>2</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
intParam := java com::cogility::XmlGetInteger( xmlDoc, '/Demo/@Param' );
intAttr := java com::cogility::XmlGetInteger( xmlDoc,
   '/Demo/Attr/text()' );
intError := java com::cogility::XmlGetInteger( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( intParam );
java com::cogility::Println( intAttr );
java com::cogility::Println( intError );
```

This writes the following to standard output:

```
33

2

null
```

XmlGetLong

Returns a Long value from an XML document.

Syntax

```
java com::cogility::XmlGetLong(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value from `xmlDoc`.

Returns

A java::lang::Long with value, or null if the value is empty, invalid or if the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed, or if the specified value is not an Integer.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
longParam : java::lang::Long;
longAttr : java::lang::Long;
longError : java::lang::Long;

xmlString := '<Demo Param="22"><Attr>3</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
longParam := java com::cogility::XmlGetLong( xmlDoc, '/Demo/@Param' );
longAttr := java com::cogility::XmlGetLong( xmlDoc, '/Demo/Attr/text()' );
longError := java com::cogility::XmlGetLong( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( longParam );
java com::cogility::Println( longAttr );
java com::cogility::Println( longError );
```

This writes the following to standard output:

```
22

3

null
```

XmlGetNodeCount

Returns the number of times the specified path (or node) appears in the XML document.

Syntax

```
java com::cogility::XmlGetNodeCount(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value from `xmlDoc`.

Returns

A java::lang::Integer with the number of nodes for the specified path.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
countLog : java::lang::Integer;
countLoad : java::lang::Integer;
countError : java::lang::Integer;

xmlString :=
  '<Log>
  <Init>10:05am</Init>
  <Load>10:07am</Load>
  <Load>10:09am</Load>
  <Init>10:15am</Init>
  <Load>10:17am</Load>
  </Log>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
countLog := java com::cogility::XmlGetNodeCount( xmlDoc, 'Log' );
countLoad := java com::cogility::XmlGetNodeCount( xmlDoc, 'Log/Load' );
countError := java com::cogility::XmlGetNodeCount( xmlDoc, 'Not/Found' );

java com::cogility::Println( countLog );
java com::cogility::Println( countLoad );
java com::cogility::Println( countError );
```

This writes the following to standard output:

```
1

3

0
```

XmlGetNodeFromNodeList

Returns the node at the specified index in the NodeList object returned from "XmlGetNodeList" on page 84. That node can be used as an XML document for most of the other XML Java actions.

Syntax

```
java com::cogility::XmlGetNodeFromNodeList(nodeList,nodePosition);
```

Parameters

- `nodeList` - (java::lang::Object) from XmlGetNodeList() (see "XmlGetNodeList" on page 84).
- `nodePosition` - (java::lang::Integer) of the index of the node to return.

Returns

An org::w3c::dom::Node object for the specified node.

Exceptions

An EncapsulatingException is thrown if:

- the index is out of bounds.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
nodeList : org::w3c::dom::NodeList;
node : org::w3c::dom::Node;
nodeCount : java::lang::Integer;
index : java::lang::Integer;
loadTime : java::lang::String;

xmlString :=
  '<Log>
  <Init>10:05am</Init>
  <Load>10:07am</Load>
  <Load>10:09am</Load>
  <Init>10:15am</Init>
  <Load>10:17am</Load>
  </Log>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
nodeList := java com::cogility::XmlGetNodeList( xmlDoc, 'Log/Load' );
nodeCount := java com::cogility::XmlGetNodeCount( xmlDoc, 'Log/Load' );
index := 0;
loop index < nodeCount do
  node := java com::cogility::XmlGetNodeFromNodeList( nodeList, index );
  loadTime := java com::cogility::XmlGetString( node, 'text()' );
  java com::cogility::Println( loadTime );
  index := index + 1;
endloop;
```

This writes the following to standard output:

```
10:07am

10:09am

10:17am
```

XmlGetNodeList

Returns a NodeList object with the nodes from the XML document that matches the specified path.

Syntax

```
java com::cogility::XmlGetNodeList(xmlDoc,xmlPath);
```

Parameters

- `xmlDoc` - (org::w3c::dom::dom::Node) from XmlLoad() See "XmlLoad" on page 87.
- `xmlPath` - (java::lang::String) specifying the path to the value from `xmlDoc`.

Returns

An org::w3c::dom::NodeList of the list of nodes that matches the specified `xmlPath`.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed.

Example

See the example under "XmlGetNodeFromNodeList" on page 83.

XmlGetShort

Returns a Short value (a signed 16-bit number from -32768 to 32767) from an XML document.

Syntax

```
java com::cogility::XmlGetShort(xmlDoc,xmlPath);
```

The arguments are as follows:

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value from `xmlDoc`.

Returns

A java::lang::Short of the value, or null if the value is empty, invalid or if the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed.
- the specified value is not a number.
- the value exceeds 16 bits.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
shortParam : java::lang::Short;
shortAttr : java::lang::Short;
shortError : java::lang::Short;

xmlString := '<Demo Param="44"><Attr>55</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
shortParam := java com::cogility::XmlGetShort( xmlDoc,'/Demo/@Param');
shortAttr := java com::cogility::XmlGetShort( xmlDoc,
   '/Demo/Attr/text()');
shortError := java com::cogility::XmlGetShort( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( shortParam );
```

```
java com::cogility::Println( shortAttr );
java com::cogility::Println( shortError );
```

This writes the following to standard output:

```
44

55

null
```

XmlGetString

Returns a String from an XML document.

Syntax

```
java com::cogility::XmlGetString( xmlDoc, xmlPath );
```

The arguments are as follows:

- `xmlDoc` - (org::w3c::dom::Node) from XmlLoad() (see "XmlLoad" on page 87).
- `xmlPath` - (java::lang::String) specifying the path to the value from `xmlDoc`.

Returns

A java::lang::String with the value, or null if the value is empty or the path doesn't exist.

Exceptions

An EncapsulatingException is thrown if:

- the `xmlPath` cannot be parsed.

Example

```
xmlString : java::lang::String;
xmlDoc : org::w3c::dom::Document;
strParam : java::lang::String;
strAttr : java::lang::String;
strError : java::lang::String;

xmlString := '<Demo Param="param"><Attr>a &lt; b</Attr></Demo>';
xmlDoc := java com::cogility::XmlLoad( xmlString );
strParam := java com::cogility::XmlGetString( xmlDoc, '/Demo/@Param' );
strAttr := java com::cogility::XmlGetString( xmlDoc,'/Demo/Attr/text()');
strError := java com::cogility::XmlGetString( xmlDoc, '/Demo/Attr' );

java com::cogility::Println( strParam );
java com::cogility::Println( strAttr );
java com::cogility::Println( strError );
```

This writes the following to standard output:

```
param

a < b
```

```
null
```

XmlLoad

Converts an XML string into an XML document object.

Syntax

```
java com::cogility::XmlLoad(xmlString);
```

Parameters

- `xmlString` - (java::lang::String) of the XML string to be loaded into an XML document object.

Returns

An org::w3c::dom::Document object.

Exceptions

An EncapsulatingException is thrown if:

- the XML cannot be parsed.

Example

```
-- Assign the XML to a String variable.
-- The XML usually comes from a File or a Message.
xmlString : java::lang::String;
xmlString := '<yourXMLHere />';
xmlDoc : org::w3c::dom::Document;
xmlDoc := java com::cogility::XmlLoad( xmlString );
```

XmlToString

Converts into an XML document object into an XML string .

Syntax

```
java com::cogility::XmlLoad(xmlDoc);
```

Parameters

- `xmlDoc` - (org::w3c::dom::Document) from XmlLoad() (see "XmlLoad" on page 87).

Returns

An java::lang::String.

Exceptions

An EncapsulatingException is thrown if:

- the XML document is malformed.

Example

```
xmlString : java::lang::String;
xmlString := '<yourXMLHere >';
xmlDoc : org::w3c::dom::Document;
xmlDoc := java com::cogility::XmlLoad( xmlString );

str : java::lang::String;
str := java com::cogility::XmlToString(xmlDoc);
```

xslTransformString

Converts an XML string into an XML document object.

Syntax

```
java com::cogility::XslTransformString(xsltDoc, xmlDocToTransform);
```

Parameters

- `xsltDoc` - (java::lang::String) XSL Transformation document.
- xmlDocToTransform - (java::lang::String) XML document to transform.

Returns

An org::w3c::dom::Document object.

Exceptions

An EncapsulatingException is thrown if:

- either the XML or the XSLT documents are malformed.

Example

```
xmlDoc : java::lang::String;
xmlDoc :=
'<?xml version="1.0" encoding="ISO-8859-1"?>
<contacts>
   <cd>
      <name>Cogility Software</name>
      <phone>949-223-1700</phone>
      <email>support@cogility.com</email>
      <street>111 North Market Street</street>
      <city>San Jose</city>
      <state>CA</state>
      <zip>95113</zip>
   </cd>
</contacts>';

xslt : java::lang::String;
xslt :=
'<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="/">
   <html>
   <body>
   <h2>My Contacts</h2>
      <table border="1">
         <tr bgcolor="#9acd32">
            <th>Name</th>
            <th>Phone</th>
         </tr>
         <xsl:for-each select="contacts/cd">
         <tr>
            <td><xsl:value-of select="name"/></td>
            <td><xsl:value-of select="phone"/></td>
         </tr>
         </xsl:for-each>
      </table>
   </body>
   </html>
</xsl:template>
</xsl:stylesheet>';

resultString : java::lang::String;
resultString := java com::cogility::XslTransformString(xslt, xmlDoc);
```

Custom Java actions

You can define your own Java actions, thus allowing access to your deployed model at run-time using the Java language. The Java class you define must do the following:

- Extend `com.ceira.pm.ocl.JavaAction`, thereby providing access to the class using the `java` keyword in action semantics. See "Extending JavaAction" on page 89.
- Specify the inputs (and their types) for the Java action by defining the abstract method, buildInputPins(). See "Defining buildInputPins()" on page 90.
- Specify the outputs (and their types) for the Java action by defining the abstract method, buildOutputPins(). See "Defining buildOutputPins()" on page 91.
- Describe the execution by defining the abstract method, execute(). See "Defining execute()" on page 91.

Each of these are described below with a simple example. In this example, the custom Java action class is called QualifiesForSeniorDiscount. It takes two arguments, a Customer object from the model and an integer representing the age that qualifies a Customer for a senior discount. The action returns a Boolean, true if the Customer qualifies for a senior discount, false if not.

Extending JavaAction

The Cogility JavaAction class provides an interface to action semantics with the `java` keyword. Your custom Java action class must extend the JavaAction class in order to work like any other Java action. For the foregoing example, therefore, the imports include the necessary Cogility libraries:

```
import java.util.ArrayList;
import java.util.List;
```

```
import com.ceira.pm.ocl.InputPin;
import com.ceira.pm.ocl.JavaAction;
import com.ceira.pm.ocl.OutputPin;
import com.ceira.pm.ocl.Procedure;
import com.ceira.pm.ocl.execution.ActionExecution;
import com.ceira.pm.ocl.marker.OCLAttributeDefinitionMarker;
import com.ceira.pm.ocl.marker.OCLClassDefinitionMarker;
import com.ceira.pm.ocl.parser.Parameters;
import com.ohana.common.errors.EncapsulatingException;
```

The class declaration follows the usual format. The class must implement a constructor method as shown below. The constructor takes three arguments, a Procedure, a Parameters and an oclLineNum object. The super() method passes these up for use by the compiler. They are completely transparent and require no further use in your definition.

```
/**
 * JavaAction to return whether a Customer qualifies for a "Senior"
discount.
 */
public class QualifiesForSeniorDiscount extends JavaAction {

   /**
    * Constructor.
    */
   public QualifiesForSeniorDiscount( Procedure aProcedure, Parameters
aParameters, Integer oclLineNum ) throws EncapsulatingException {
      super( aProcedure, aParameters, oclLineNum );
   }
```

Defining buildInputPins()

You may have multiple inputs to your Java action, and they may be of any type, though you should specify the type in order to take advantage of native type checking. The inputs are held in an ArrayList, and in this example, their variables are labeled `inputPin0` and `inputPin1` to correspond with their indeces in the ArrayList. For each input, you define an inputPin object; for a model-defined class you must pass to its constructor the results of the JavaAction methods getProcedure(), getMetaModel(), and classNamed() in that order as shown below. It is through these methods that the Java action has access to the class defined in your model (the metamodel). In this example, that class is qualified as Model.MIM.Customer, where Model is the name of the model. The class must be fully-qualified. For inputs of a simple type, you simple pass to the InputPin constructor the class attribute for the type, in this example, `Integer.class`. For each InputPin, you call the JavaAction's setAction() and add the InputPin to the ArrayList, as shown below.

```
/**
    * Define the inputs (and their types) for this JavaAction
    */
   public List buildInputPins() throws EncapsulatingException {
      ArrayList result = new ArrayList();

      InputPin inputPin0 = new InputPin(
this.getProcedure().getMetaModel().classNamed( "Model.MIM.Customer" ) );
      inputPin0.setAction( this );
      result.add( inputPin0 );
```

```
        InputPin inputPin1 = new InputPin( Integer.class );
        inputPin1.setAction( this );
        result.add( inputPin1 );

        return result;
    }
```

Defining buildOutputPins()

You may have only one output from your Java action; it may be of any type. In the example below, the custom Java action will return a Boolean.

```
/**
     * Define the output (and its type) for this JavaAction
     */
    public List buildOutputPins() throws EncapsulatingException {
        ArrayList result = new ArrayList();

        OutputPin outputPin = new OutputPin( Boolean.class );
        outputPin.setAction( this );
        result.add( outputPin );

        return result;
    }
```

Defining execute()

The execute method uses an ActionExecution object to retrieve and set values. It first retrieves the inputs from the ArrayList. If the inputs are run-time objects, as Customers are in this example, the Java action must have access to the both the metamodel and the object model when it executes. The method first retrieves the model object using the getPinValue method, and places it in an Object variable named `theCustomer` in this example. The getPinValue() method always returns an Object. Next, it needs an OCLClassDefinitionMarker object (`custClass`) that describes the model Object (`theCustomer`) and the object model for the run-time application. Likewise, it needs an OCLAttributeDefinitionMarker (`ageAttr`) to hold the Age attribute definition. Note that it uses the metamodel's getAttribute() method to get a definition for the Age attribute based on the class definition of the Customer, `custClass`. Then it retrieves the value of the Age attribute through the object model's getAttribute() method, casts it to Integer and places it in the `age` attribute.

Now the action needs to retrieve the value for the age that qualifies the Customer for the senior discount. The getPinValue() method again returns the input in an Object variable, named `theAgeLimit` in this example. The theAgeLimit value is then cast to an Integer and placed in an int called `ageLimit`.

The actual operation is simply a test of the Customer age value. If it is greater than or equal to the ageLimit, the value of the `isSenior` boolean is true. Note that this is not the return value of the execute() method. The method returns true if it completes, and this is provided for by defining the return as true.

```
/**
     * Execute this JavaAction
     */
    public boolean execute( ActionExecution exe ) throws
```

```
        EncapsulatingException {
        InputPin inputPin0 = this.getInput(0);
        InputPin inputPin1 = this.getInput(1);

        Object theCustomer = exe.getPinValue( inputPin0 );
        OCLClassDefinitionMarker custClass =
           exe.getObjectModel().getObjectClass( theCustomer );
        OCLAttributeDefinitionMarker ageAttr =
           exe.getMetaModel().getAttribute( custClass, "Age" );
        Integer age = (Integer) exe.getObjectModel().getAttribute(
           theCustomer, ageAttr );

        Object theAgeLimit = exe.getPinValue(inputPin1);
        int ageLimit = ((Integer) theAgeLimit).intValue();

        boolean isSenior = age.intValue() >= ageLimit;

        OutputPin outputPin = this.getOutput(0);
        exe.setPinValue( outputPin, new Boolean(isSenior) );

        return true;
    }
}
```

6

Functions

OCL requires that the following functions be supported in action semantics.

abs

The absolute value of a number.

Syntax:

```
<number>.abs()
```

Example:

```
n := -1.2;
n.abs(); --returns 1.2
```

concat

Concatenate the string values of two objects, and return a string.

Syntax:

```
<object>.concat( object )
```

Example 1:

```
'max='.concat( 9999 ) --returns 'max=9999'
```

Example 2:

```
'tall='.concat( someObj.height > 72 ) --returns 'tall=true'
```

Example 3:

```
123.concat( 456 ) --returns '123456'
```

Notes

The concat() function is inefficient when concatenating long strings. There is an option that makes string building much more efficient.

Use a StringBuffer which is optimized for building long strings.

```
sb : java::lang::StringBuffer;
sb := new java::lang::StringBuffer();
sb.append('abc'); sb.append('def'); sb.append('ghi'); sb.append('jkl');
sb.toString();
```

div

The number of times that a number fits completely within another number.

Syntax:

```
<number>.div( number )
```

Example 1:

```
n := 5;
x := 2;
n.div(x); --returns 2
```

Example 2:

```
n := -5.0;
x := 2;
n.div(x); --returns -2.0
```

floor

The largest integer/long which is less than or equal to a number.

Syntax:

```
<number>.floor()
```

Example 1:

```
n := 1.8;
n.floor(); --returns 1
```

Example 2:

```
n := 5;
n.floor(); --returns 5
```

Example 3:

```
n := -1.2;
n.floor(); --returns -2;
```

max

The maximum of two numbers.

Syntax:

```
<number>.max( number )
```

Example 1:

```
n := 5;
x := 6;
n.max(x); --returns 6
```

Example 2:

```
n := -1.5;
x := -2.6;
n.max(x); --returns -1.5
```

Example 3:

```
n := 2;
x := 1.1;
n.max(x); --returns 2.0
```

min

The minimum of two numbers.

Syntax:

```
<number>.min( number )
```

Example 1:

```
n := 5;
x := 6;
n.min(x); --returns 5
```

Example 2:

```
n := -1.5;
x := -2.6;
n.min(x); --returns -2.6
```

Example 3:

```
n := 1
x := 2.2;
n.min(x); --returns 1.0
```

mod

The result of a number modulo another number.

Syntax:

```
<number>.mod( number )
```

Example 1:

```
n := 12;
x := 5;
n.mod(x); --returns 2
```

Example 2:

```
n := -12;
x := 5;
n.mod(x); --returns -2
```

Example 3:

```
n := -12;
x := -5;
n.mod(x); --returns -2
```

round

The integer that is closest to a number. When there are two such integers, the largest one.

Syntax:

```
<object>.round( object )
```

Example 1:

```
n := 1.2;
n.round(); --returns 1
```

Example 2:

```
n := 1.8;
n.round(); --returns 2
```

Example 3:

```
n := 1.5;
n.round(); --returns 2
```

Example 4:

```
n := -1.2;
n.round(); --returns -1
```

Example 5:

```
n := -1.8;
n.round(); --returns -2
```

Example 6:

```
n := -1.5;
n.round(); --returns -1
```

size

The number of characters in a string.

Syntax:

```
<string>.size()
```

Example 1:

```
'aString'.size(); --returns 7
```

Example 2:

```
''.size(); --returns 0
```

substring

A substring of another string. The substring begins at the specified lowerIndex and extends to the character at upperIndex.

> **Note:** This differs from Java where the substring begins at the specified lowerIndex and extends to the character at upperIndex - 1.

Syntax:

```
<string>.substring( lowerIndex, upperIndex )
```

Example 1:

```
'abcd'.substring( 1, 2 ); --returns 'bc'
```

Example 2:

```
'abcd'.substring( 1, 1 ); --returns 'b'
```

Example 3:

```
'abcd'.substring( 1, 0 ); --returns ''
```

toLower

Converts all uppercase characters in a string to lowercase characters.

Syntax:

```
<string>.toLower()
```

Example 1:

```
'This Is Test 21!'.toLower(); --returns 'this is test 21!';
```

toUpper

Converts all lowercase characters in a string to uppercase characters.

Syntax:

```
<string>.toUpper()
```

Example 1:

```
'This Is Test 21!'.toUpper();     --returns 'THIS IS TEST 21!';
```

7 Queries

There are two ways to query the run-time repository using action semantics. For simple queries involving one class and a few attributes, direct SQL, discussed next, is easy and efficient. For more complex queries, use custom queries; see "Custom queries" on page 101.

Note that the use of direct SQL and custom queries applies only to the run-time repository. To query and perform operations on a legacy database with a legacy information model, see "Legacy Information Model" on page 61 of the guide, *Modeling with Cogility Studio*.

Direct SQL

Direct SQL lets you specify search criteria ranges in your queries with the locate and locateAll orderedBy keywords (see "locate" on page 41). For example, you could query for all customers whose age is 25, as in the following:

```
locateall MyModel::MIM::Customer(age: 25);
```

However, to locate all customers whose age is greater than 25, you would use the following action semantics:

```
directSQL : com::cogility::DirectSQLString;
directSQL := new com::cogility::DirectSQLString( '> 25');
locateall MyModel::MIM::Customer (age: directSQL);
```

The limited, direct use of SQL allows these kinds of queries to be performed. A Direct SQL query is specified on a per attribute basis, using DirectSQLString type to hold the query string.

Note that the use of direct SQL applies only to the run-time repository. To query and perform operations on a legacy database with a legacy information model, see "Legacy Information Model" on page 61 of the guide, *Modeling with Cogility Studio*.

DirectSQLString

The DirectSQLString type holds an SQL query string for use with locate and locateAll orderedBy keywords. Its fully-qualified class name is as follows:

```
com::cogility::DirectSQLString;
```

To use the DirectSQLString, you first declare a variable of type DirectSQLString, then assign a valid SQL string enclosed in single quotes to that variable, and finally pass the variable as a parameter for the locate or locateAll orderedBy keyword. The following example locates all Customers who are older than 25 and whose status is either 1 or 3 or 8:

```
ageSQL := new com::cogility::DirectSQLString('> 25');
```

```
statusSQL := new com::cogility::DirectSQLString('IN (1, 3, 8)'));
locateall MyModel::MIM::Customer(age: ageSQL, status: statusSQL);
```

Note that the query string must be held in a DirectSQLString variable in order to avoid OCL compile errors. When you pass an argument to the locate or locateAll orderedBy keywords, a key value pair in the format `<attr_name>:<value>` is expected. The `<value>` must match the type for the `<attr_name>`. If the `<attr_name>` is of type Integer and you pass a `<value>` of type String, a compile error results. For example the following results in a compile error if `age` is of type Integer:

```
locateall MyModel::MIM::Customer(age: '< 25');
```

When you pass a `<value>` of type DirectSQLString, it works with any type defined for the `<attr_name>`. Thus, in the following example, `age` may be an Integer and the value `'> 25'` may be a DirectSQLString:

```
directSQL : com::cogility::DirectSQLString;
directSQL := new com::cogility::DirectSQLString( '> 25');
locateall MyModel::MIM::Customer (age: directSQL);
```

SQL operators

The following are the valid SQL operators that you may use in your query:

=	equal to
>	greater than
<	less than
<>	not equal to
>=	greater than or equal to
<=	less than or equal to
IN	in a set
NOT IN	not in a set
BETWEEN p AND q	value between p and q inclusive
LIKE 'xyz%'	matches xyz followed by zero or more characters
LIKE 'xyz_'	matches xyz followed by one character
IS NULL	is null
IS NOT NULL	is not null

Escape characters

When you need to specify a string component within the SQL, its begin and end string characters have to be "escaped" with backslashes (\), as in the following example:

```
directSQL := new com::cogility::DirectSQLString(' LIKE \'%main%\' ');
locateall MyModel::MIM::Customer(address: directSQL);
```

This locates all customers where the string “main” (with wildcards %) appears anywhere in the address. Because the value of the DirectSQLString parameter is enclosed in quotes, any string components within must be escaped.

When you specify SQL to look for all records with the string “main” included, the ordinary SQL looks like the following:

```
SELECT * WHERE A_ADDRESS LIKE '%Main%'
```

Limitations

You can use direct SQL where the SQL operators qualify the value for an attribute name. That is, self-contained SQL strings including operators may be used as values for the DirectSQLString variable, but you cannot use these operators to qualify the attribute name itself.

For example, the following SQL statement looks for records with an age value of greater than the square root of 625:

```
SELECT * FROM PEAR_MYMODEL_MIM_CUSTOMER WHERE A_AGE_ > SQRT(625)
```

You can define such an expression with action semantics. It would look like the following:

```
sql :=  new com::cogility::DirectSQLString('> SQRT (625)');
locateall MyModel::MIM::Customer(age: sql);
```

Note that the greater than operator (>) is not used to qualify the `age` attribute, but rather the value returned in the SQL string.

Conversely, the following query cannot be performed with action semantics with direct SQL:

```
SELECT * FROM PEAR_MYMODEL_MIM_CUSTOMER WHERE SQRT(AGE) > 5
```

Taking the square root of the age attribute is not allowed. Nor is selecting for the name in upper case, as in the following example:

```
SELECT * FROM PEAR_MYMODEL_MIM_CUSTOMER WHERE UPPER(NAME) = 'HENRY'
```

In both of the preceding examples, the standard use of SQL permits this type of query, but it cannot be performed with action semantics.

Custom queries

For simple queries involving one class and a few attributes, you can use direct SQL statements. This is described in "Direct SQL" on page 99. For more complex queries on the run-time repository involving several classes, several attributes and multiple associations you can use custom queries. A custom query allows queries across logical sets of tables, connected by associations, and allows you to select only some of the columns of data as required. In database parlance, this type of query allows you to perform database joins.

You develop custom queries with model artifacts as described in "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*. You use the queries with the SQL statements according to the syntax described in this section.

Note that the use of custom queries applies only to the run-time repository. To query and perform operations on a legacy database with a legacy information model, see "Legacy Information Model" on page 61 of the guide, *Modeling with Cogility Studio*.

Expression syntax

To use a custom query, you first create the query artifacts in Cogility Modeler (see "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*). Doing so requires the use of shorthand for tables, classes and attributes of the run-time database. This shorthand is described next.

You also use action semantics to execute the query in state machines, web services, or wherever appropriate. Executing a custom query is described in "Query execution action semantics" on page 104.

Shorthand for SQL statements

You write the various statements of an SQL expression in the custom query fields in Cogility Modeler. See "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio* for information about creating custom queries. The SQL statements you write are identical to ordinary SQL except for the shorthand that describes the tables, columns, and attributes of the run-time database. For example, in the following statement, the #C specifies the class table for the Address class and the #A specifies the attribute named state.

```
#C[QSample::MIM::Address].#A[state]
```

When you specify the class, you must present the fully-qualified name of the class; in this case the Address class is a MIM class of the QSample model.

The following table describes each shorthand notation and how it is used.

Shorthand	Description	Examples	Meaning
#C	Class table	`#C[X].#A[abc]`	Attribute abc of class X.
#A	Attribute		
#AC	Association class	`#AC[Z].#A[abc]`	Attribute abc of association class Z.
#R	Relationship	`#C[X].#R[y]`	Relationship between class X and role y of class Y.
		`#AC[Z].#R[x]`	Relationship between association class Z and role x of class X. Note that association class Z associates classes X and Y, which is not discernible from the notation.
@	Subclass	`#C[X].#R[y@Y1]`	Relationship between class X and role y of class Y. Class Y1 is a subclass of class Y. This traverses the relationship between X and Y to allow access to Y1.
*	Outer Join	`#C[X*].#R[y]`	Relationship between class X and class Y via role y of class Y. This returns all instances of X even if there are no associated instances of Y. This is the equivalent of X left outer join Y.
		`#C[X].#R[y*]`	Relationship between class X and class Y via role y of class Y. This returns all instances of Y even if there are no associated instances of X. This is the equivalent of X right outer join Y.
		`#C[X*].#R[y*]`	Relationship between class X and class Y via role y of class Y. This returns all instances of X and Y even if they are not associated. This is the equivalent of X full outer join Y.
		`#C[X].#R[y@Y1*]`	Relationship between class X and role y of class Y. Class Y1 is a subclass of class Y. This traverses the relationship between X and Y to allow access to Y1 with a right outer join on Y1.
Alias	Alias	`#C[X as XAlias].` `#R[y as YAlias]`	Relationship between class X and role y of class Y. Class X is aliased to XAlias; class Y is aliased to YAlias.
		`#C[X as XAlias].` `#R[y@Y1 as Y1Alias]`	Relationship between class X and role y of class Y. Class Y1 is a subclass of class Y. This traverses the relationship between X and Y to allow access to Y1 Class X is aliased to XAlias; class Y1 is aliased to Y1Alias.
		`#C[X as XAlias*].` `#R[y@Y1 as Y1Alias*]`	Same as above, but with full outer join specified for this relationship.
#input		`#input[argument]`	Query argument.

Outer join

Care should be taken when creating multiple, chained outer joins. It is best to specify the relationship in a meaningful and ordered manner such that the target class represented by the #R notation in the first relationship is specified in the #C notation in the second relationship, and so on. This ensures that the SQL that is generated is chained correctly.

For example, if Class A is associated with Class B, and Class B is associated with Class C, you should specify the relationships in the following manner:

```
#C[A].#R[b*]
```

```
#C[B*].#R[c]
```

In this example, the outer join specification ensures that the result includes all instances of B even if there are no associated instances of A or C.

Alias

An Alias must be declared in the relationship specification section, either in the #C[…] notation or in the #R[…] notation by using the "AS" keyword. There are three ways to declare an alias.

- Declare an alias XAlias for Class X:

  ```
  #C[X as XAlias]
  ```

- Declares an alias YAlias for Class Y which is the class at role y:

  ```
  #R[y as YAlias]
  ```

- Declared an alias Y1Alias for Class Y1 which is a subclass of the class at role y:

  ```
  #R[y@y1 as Y1Alias]
  ```

Aliases cannot be declared for an association class in the #AC[…] notation.

Aliases can be used in the Attribute, Conditions and Other Clauses sections of the query definition. An alias can be used in any place where a class name is appropriate. This includes the #C[…] notation or after the @ symbol in the #R[…] notation. For example, you can specify the attribute abc on class X by using X's alias XAlias:

```
#C[XAlias].#A[abc]
```

If you have declared an alias for a class, this class must be referred to by its alias and you must use the `@` symbol. It is no longer possible to refer to this class by its class name. This causes a syntax error in the generated SQL. For example, if you declared an alias for class Y, to specify a relationship beween Class X and Class Y, you would write the statement as:

```
#C[XAlias].#R[y@YAlias]
```

Writing `#C[XAlias].#R[y]` it would cause an error because Class Y at role y has already been aliased and the alias must be used.

Special care should be given when declaring and using aliases in the relationship specification section. You must declare and alias before you can use it. For example, the following block causes an error because the alias XAlias is used in in the first line before its declaration inthe second line:

```
#C[XAlias].#R[y as YAlias1]
```

```
#C[X as XAlias].#R[ay as YAlias2]
```

Query execution action semantics

Before executing a custom query with action semantics, first set it up in Cogility Modeler as described in "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*. The syntax for an execution statement is as follows:

```
query <query name>(<argument>:<value>, <argument>:<value>, ...);
```

Example

An example model that illustrates a custom query is available as %DCHOME%\MB\Examples\Queries\QuerySample.cog. You can load the sample model and run it with the accompanying action semantics script. This section describes that script. For a complete overview of the model, including the class diagram that describes the relationships between classes, see "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*.

The %DCHOME%\MB\Examples\Queries\DataSetUp.ocl script populates the database with some Customers and Sales and the %DCHOME%\MB\Examples\Queries\RunSalesQuery.ocl script runs a custom query against that database.

```
coll : java::util::Collection;
coll := query SimpleModel::SalesQuery(dateSince: '20040101',
minSalesAmount: 10.00);

java com::cogility::Println('Size of Coll = '.concat(coll.size()));
java com::cogility::Println(
'Employee\t\t\tState\t\t\tProductName\t\t\tItemsSold\t\t\tSaleAmount');
java com::cogility::Println(
'========\t\t\t=====\t\t\t==========\t\t\t========\t\t\t==========');

qry : SimpleModel::SalesQuery;
iter : java::util::Iterator;
iter := coll.iterator();
loop (iter.hasNext()) do
   qry := iter.next().oclAsType(SimpleModel::SalesQuery);
   java com::cogility::Println( qry.employeeName.
      concat('\t\t\t\t').concat(qry.customerState).
      concat('\t\t\t\t').concat(qry.productName).
      concat('\t\t\t\t').concat(qry.numberSold).
      concat('\t\t\t\t\t').concat(qry.saleAmount));
endloop;
```

To call the query, a Collection variable is defined and the `query` statement is executed with a `dateSince` argument and a `minSalesAmount` argument. This performs the query and assigns a Collection of SalesQuery instances to the `coll` variable.

Next, the listing outputs the size of the Collection and some headers for the columns of data that will be output.

Then, a `qry` variable is defined to hold the instances of the `SalesQuery` that in the `coll` Collection. Then, the listing iterates over the Collection and assigns the next instance to the `qry` variable. For each instance, the `SalesQuery` values are output under the column titles.

The penultimate lines print the values of the result object attributes.

To load and run the sample model associated with this script, see "Custom Queries" on page 179 of the guide, *Modeling with Cogility Studio*.

8

Special

Common to many systems is the need to periodically process large volumes of data. Usually this processing involves thousands, even hundreds of thousands of business objects. Batch processing groups the records into batches and commits them to the database one batch at a time. This way, if any of the objects in the batch causes an exception, the database transaction may be rolled back for just that batch. Previous batches are already committed and therefore unaffected; subsequent batches are also treated in their own transactions.

Batch processing may be handled with a state machine as described in "Batch processing" on page 130 of the guide, *Behavior Modeling with Cogility Studio*. However, that approach has the disadvantage of being bound by the limitations of the application server because the state machine runs in the application server process which usually imposes a limit on the time allowed for database transactions. See "Limitations" on page 130 of the guide, *Behavior Modeling with Cogility Studio*.

The preferred approach is to run the batch process against the database directly, avoiding the application server and its limitations. With this approach, you create, compile and execute a script in Cogility Action Pad. When you process a batch in its own process, as described here, you must begin, commit or roll back the transaction explicitly. In a state machine, because the transaction boundaries are defined by the state's do activity, beginning and committing the transaction happen automatically. The scripts described here show you how to define your own transaction boundaries.

The oclCommit and oclRollback actions define transaction boundaries in action semantics scripts. See "oclCommit" on page 45 and "oclRollback" on page 49 of the guide, *Using Actions in Cogility Studio*. The transaction begins when the script executes - usually by clicking the Run button in Cogility Action Pad. These actions complete by also begining a new transaction so that processing may resume.

The example scripts for running a batch process independent of the application server are located in the directory, %DCHOME%\MB\Examples\Behavior\BatchProcess:

- **CreateBatchData.ocl** - creates 100 records on the database of the Subscriber class defined in the sample model (see below).
- **CreateAndProcessBatch.ocl** - creates a batch of Subscriber objects and processes them in sub-batches. It demonstrates how to begin, commit and roll back a transaction for processing the sub-batches. See "CreateAndProcessBatch" on page 108.
- **BatchFromPoid.ocl** - creates and processes Subscriber objects in sub-batches, just like CreateAndProcessBatch.ocl, but instead of retrieving the Subscriber objects directly, it references the objects by POID. See "BatchFromPoid" on page 110.

To work with these example scripts, you will need to turn over and push the BatchExample model. However, you will only push it onto the database. The application server will not be involved. See "Running the batch processing scripts" on page 111 for full instructions.

CreateAndProcessBatch

The CreateAndProcessBatch script retrieves Subscriber objects, places them in batches and processes them. It refers to Subscriber objects defined in the BatchExample model. See "Model objects" on page 133 of the guide, *Modeling with Cogility Studio* for a description of this model. Note that for the purposes of this script, only the class definition of the Subscriber object is pertinent. The other model artifacts are used in the batch processing in state machines example described in that section.

The first lines of the script declare the BusinessObjectsBatch object and call the newBatchFor method to set up the sub-batches. See "BusinessObjectsBatch" on page 130 and "newBatchFor" on page 130 of the guide, *Modeling with Cogility Studio*. The last lines below assign the batch object's POID to a local variable, `s` and print it to standard output.

```
batch : com::cogility::poal::BusinessObjectsBatch;
map : java::util::HashMap;
map := new java::util::HashMap();
directSQL : com::ceira::DirectSQLString;
directSQL := new com::ceira::DirectSQLString('BETWEEN 1000 and 1500');
map.put('subscriberId', directSQL);

batch := (null.oclAsType(com::cogility::poal::BusinessObjectsBatch))
                .newBatchFor('BatchExample.MIM.Subscriber', map, '', 5);
s : java::lang::String;
s := batch.getIdString();

java com::cogility::Println(batch.getIdString());
```

The next group of lines set a variable for the batch count and one to hold Subscriber objects during processing.

```
------- set a variable for the batch count
cnt : java::lang::Integer;
cnt := 1;
sub : BatchExample::MIM::Subscriber;
```

Processing

Processing takes place in a loop that iterates through the sub-batches while the batch has more instances. Each sub-batch in process is reported to standard output. The nextBatch action retrieves the next sub-batch and places it in a `curBatch` List, then iterates through the Subscriber objects in the List. It tests each Subscriber object's age attribute for a value greater than 21, and if the value is greater than 21, sets the premium attribute to true. It prints to the console the subscriberId and age. As long as no Subscriber object in the sub-batch throws an exception, the sub-batch of processed Subscribers is committed to the database with `oclCommit`. See "oclCommit" on page 45 of the guide, *Using Actions in Cogility Studio*. Note that the current transaction was begun when the script was executed - usually by clicking the Execute button in Cogility Action Pad. The oclCommit action also begins a new transaction for the next sub-batch before focus falls to the `endtry` that closes the try/catch block.

```
loop (batch.hasMoreInstances()) do
   java com::cogility::Println('Processing Batch #: '.concat(cnt));
   try
      curBatch : java::util::List;
      curBatch := batch.nextBatch();
      cbIter : java::util::Iterator;
      cbIter := curBatch.iterator();
```

```
      loop cbIter.hasNext() do
         sub := cbIter.next().oclAsType(BatchExample::MIM::Subscriber);
         if sub.age > 21 then
            java
com::ceira::Println(sub.subscriberId.concat('\t').concat(sub.age));
            sub.premium := false;
         endif;
      endloop;
      oclCommit;
```

Those Subscriber objects that don't have a value for the age attribute will throw an exception, which will be handled with any other exceptions in the `catch` block. When an exception is encountered, the current transaction is rolled back with `oclRollback`. See "oclRollback" on page 49 of the guide, *Using Actions in Cogility Studio*. The `oclRollback` action first rolls back the current transaction, then begins a new transaction so that processing may resume.

```
   catch (java::lang::Throwable ae)
      java com::cogility::Println('##### Exception was caught
               - rolling back transaction.');
      oclRollback;
```

At this point the batch object refers to a sub-batch which has Subscriber objects in error. When the transaction for that sub-batch was rolled back, the batch object itself, which holds its own state information as a persistent database object, has also been rolled back. To retrieve the batch object in its rolled-back state, you must explicitly locate the batch object by its POID with `LocateByIdentity` again before you can work with the sub-batch in error. Note that the sub-batch in error is no longer the *current* sub-batch (which is no longer defined), but the *next* sub-batch, according to the batch object's state information. To work with the sub-batch, you call the `setAsideNextBatchAsErrored` method, then commit the unprocessed batch with `oclCommit`. This also begins a new transaction for the next sub-batch. The last line increments the batch count for the next pass through the loop.

```
      batch := java com::ceira::oem::LocateByIdentity(s)
               .oclAsType(com::cogility::poal::BusinessObjectsBatch);
      batch.setAsideNextBatchAsErrored();
      oclCommit;
   endtry;
   cnt := cnt + 1;
endloop;
```

When the sub-batch is set aside, each object in the sub-batch is placed in its own sub-batch (each new sub-batch has a batch size of 1). This is how the objects in error are isolated; the new sub-batches are re-processed after the initial sub-batches (in this example, those with a batch size of 5). After all of the initial sub-batches and all of the new sub-batches are processed, there will be set aside a number of new sub-batches each with an object in error. Notice that each Subscriber object in error has been tested twice; once in its original sub-batch along with the other Subscriber objects, then a second time in its own sub-batch.

The last block of code checks the batch object for errored sub-batches, gets those sub-batches, and prints the `subscriberId` for each errored Subscriber to standard output.

```
if (batch.hasErroredInstances()) then
   erroredList : java::util::List;
   iter : java::util::Iterator;
   erroredList := batch.getErroredInstances();
   java com::ceira::Println('There are  '.concat(erroredList.size()
```

```
                .concat(' errored instances')));
   iter := erroredList.iterator();
   loop (iter.hasNext()) do
      sub :=  iter.next().oclAsType(BatchExample::MIM::Subscriber);
      java com::ceira::Println('Could not process subscriber: '
                .concat(sub.subscriberId));
   endloop;
endif;
```

BatchFromPoid

The BatchFromPoid script is just like the CreateAndProcessBatch script ("CreateAndProcessBatch" on page 108) except that instead of retrieving the Subscriber objects directly, it references the objects by POID. To reference the POID of a business object, its class must define a poid attribute. See "POID attribute" on page 37 of the guide, *Modeling with Cogility Studio*.

Like the CreateAndProcessBatch script, BatchFromPoid sets up a DirectSQLString to specify the subset of Subscriber objects. It then places these Subscribers in a Collection, and prints to standard output the number of objects in the Collection.

```
directSQL : com::ceira::DirectSQLString;
directSQL := new com::ceira::DirectSQLString('BETWEEN 1000 and 1500');

coll : java::util::Collection;
coll := locateall BatchExample::MIM::Subscriber(subscriberId: directSQL);

java com::cogility::Println('Size = '.concat(coll.size()));
```

It then defines an ArrayList to hold the list of Subscriber POIDS, and an iterator to run through the Collection of Subscriber objects.

```
list : java::util::ArrayList;
list := new java::util::ArrayList();

iter : java::util::Iterator;
iter := coll.iterator();
```

The next block sets up a `sub` variable to hold Subscriber objects and loops through the Collection, adding each Subscriber POID to the ArrayList `list`.

```
sub : BatchExample::MIM::Subscriber;
loop (iter.hasNext()) do
   sub := iter.next().oclAsType(BatchExample::MIM::Subscriber);
   list.add(sub.poid);
endloop;
```

Now the script declares the BusinessObjectBatch object and calls the newBatchFor method. This time, the Subscribers are already selected, and the method may refer to the `list` variable to retrieve them by POID. The variables to hold the batch POID and the sub-batch count are also defined.

```
batch : com::cogility::poal::BusinessObjectsBatch;
batch := (null.oclAsType(com::cogility::poal::BusinessObjectsBatch))
                .newBatchFor(list, 5);
s : java::lang::String;
s := batch.getIdString();
```

```
cnt : java::lang::Integer;
cnt := 1;
```

From here, the script is just like the CreateAndProcessBatch script. See "Processing" on page 108.

Running the batch processing scripts

To run the batch processing sample scripts:

1. In Cogility Modeler, import the model, %DCHOME%\MB\Examples\Behavior\BatchProcess\BatchExample_v1.cog.

   See "Message definitions" on page 106 of the guide, *Modeling with Cogility Studio* for loading instructions.
2. Turn over and push the model.
3. Run Cogility Action Pad.

   See "Running Cogility Action Pad" on page 80. Cogility Action Pad opens in a new window.
4. In Cogility Action Pad, open the following script files in the %DCHOME%\MB\Examples\Behavior\BatchProcess directory:
   - **CreateBatchData.ocl**
   - **CreateAndProcessBatch.ocl**
   - **BatchFromPoid.ocl**
   - **DeleteBatchData.ocl**

   See "Opening FilePads" on page 82.
5. Run the **CreateBatchData.ocl** script.

   See "Running action semantics in an action pad" on page 90.
6. Inspect the data on the database.

   Use a tool like DbVisualizer. On a an Oracle database the populated table is labeled, PEAR_BATCHEXAMPLE_MIM_SU_1. You can also use SQL Worksheet (if you are running Oracle) and submit the following query:

   ```
   select * from PEAR_BATCHEXAMPLE_MIM_SU_1 WHERE A_SUBSCRIBERID_ BETWEEN
   1000 AND 1100;
   ```

   Notice that the data includes 100 Subscriber records. Four of the records lack an age value, and these will generate exceptions during the batch processing.
7. In Cogility Action Pad, run the script, **CreateAndProcessBatch.ocl** or **BatchFromPoid.ocl**.

   Notice the transactions rolled back and the two records that could not be processed.
8. Once again, inspect the data.

   Notice the column for the Premium attribute now has values.
9. In Cogility Action Pad, run the **DeleteBatchData.ocl** script.

Global data user cache

This feature allows model logic to access a global cache (a user-defined registry) to improve performance. Action semantics defined in a web service or state machine run in multiple threads inside the J2EE application server. Each thread executes in its own transaction. The user-defined

registry allows frequently accessed read-only objects to be cached once and read, without going all the way to the database many times.

Methods

getRegistryObject(object, key) - returns the value to which the specified key is mapped, or null if there is no mapping for the specified key.

putRegistryObject(object key, object val) - associates the specified value with the specified key.

removeKey(object key) - removes the mapping for this key.

Example

For example, new orders come in via a web service called PlaceNewOrder, which determines if the requested product exists in the product catalog. If it does, then the web service calculates the total order cost by looking up the price of the product. Each web service invocation executes in its own transaction. You do not want each invocation (there could be millions over time) to have to go to the database for the same 50 products.

Furthermore, you don't want to write a separate piece of logic to go proactively cache these products in the user-defined registry, but use the logic in the PlaceOrder web service to do this. Under this scenario, the user-defined registry is lazily initialized, meaning that the cache is populated when the information is first needed, not when the application server comes up.

The action semantics in this web service might look like the following:

```
product : MySchema::MIM::Product;
oal : com::ceira::pm::objaccess::OAL;
product := oal.getOAL().getRegistryObject(input.itemName)
   .oclAsType(MySchema::MIM::Product);
if (product = null) then
   -- this must be the first time someone's looking this product up,
   -- it is not yet in the cache.
   product := locate MySchema::MIM::Product(name: input.itemName);
   oal.getOAL().putRegistryObject(input.itemName, product);
else
   -- do something else
endif;

-- calculate total order cost using product.price
```

Busness Object Methods

JSON Support

JSON (JavaScript Object Notation) is a lightweight data-interchange format and is simple to read and write. Despite is relationship to JavaScript, it is considered to be language independent and is used as an alternative to XML.

The JSON object is expressed as an unordered set of name/value pairs. An ordered list is not supported.

Two methods exist on Business Objects:

- The name/values pairs are unordered
- Attributes with null values are not returned (empty string '' will be returned)
- BO Type and POID are included

toJSONStirng()

Example:

```
cust : Model::MIM::Customer := new Model::MIM::Customer();
cust.firstName :=  'John';
cust.lastName := 'Doe';
cust.isActive := true;
cust.creationDate := new com::cogility::Timestamp();
oclprintln(cust.toJSONString());
```

Returns:

{"id":76542,"isActive":true,"lastName":"Doe","creationDate":1283894638187,"bo type":"Model.MIM.Customer","poid":"BO^Model.MIM.Customer^-3dbe38dc:12aed7bbc5a:-7ffa","firstName":"John"}

toJSONStringNicelyFormatted()

- Attributes of type Timestamp are converted to YYYY-MM-DD HH:MM:SS format.

Returns:

{"id":76541,"isActive":true,"lastName":"Doe","creationDate":"2010-09-07 14:22:18","bo type":"Model.MIM.Customer","poid":"BO^Model.MIM.Customer^-3dbe38dc:12aed7bbc5a:-7ffb","firstName":"John"}

Index

Symbols

A

B

I

J

K

T

W

X