IBM Cúram Social Program Management
8.0.2

Cúram Express Rules Reference Manual

IBM

Note

Before using this information and the product it supports, read the information in "Notices" on page
224

Edition

This edition applies to IBM® Cúram Social Program Management 8.0.0, 8.0.1, and 8.0.2.

Licensed Materials - Property of IBM.

Contents

Figures

Tables

x

Chapter 1. Cúram Express Rules Reference

Cúram Express® Rules are used to perform business calculations. A development environment for authoring and testing Cúram express rule sets is available. Rules can be executed at run time. The CER editor is a tool for business and technical users to view create and manage CER rule sets.

Related concepts

Working with Cúram Express Rules

Developing with Eligibility and Entitlement by using Cúram Express Rules

How to Build a Product

Related information

Working with Cúram Express Rules

CER Overview

An overview of CER including its key concepts, benefits, how it is used in IBM Cúram Social Program Management, and its guiding principles.

What is CER?

A description of the Cúram Express Rules (CER) rule language, development environment, and runtime features.

CER includes the following:

- A language for expressing the rules for business calculations (in rule sets).
- A development environment for authoring and testing these rule sets.
- A runtime environment for executing rules.

Rules Language

CER is a language for defining the questions that can be asked and the rules that determine the answers to the questions.

Each question specifies its name, the type of data that provides the answer to the question, and the rules for providing the answer if the question is asked.

The answer to a question can be as simple as Yes or No. For example, the question "Is this person eligible to receive benefit?" However, you can define answer types to be as complex as you need. For example, the question "Which groups of people in the household have an urgent need?" can be answered by providing a list of household groups, with each household group that contains a list of people.

The rules for determining the answer to a question again can also be as simple or as complex as you need, for example, the rule for the answer to the question "What is the claimant's date of birth?" is likely to (trivially) be "the date that the claimant declared their date of birth to be", whereas the rule for answering the question "Is this person eligible to receive a benefit?" is likely to involve further questions such as "What level of income does this person have?" or "How many children does this person have?".

CER Language Terminology

The following is a list of the key terminology for the CER language.

Rule Class

A rule class is a type of "thing" that has data, such as a Person, an Income, or a Claim. A new rule class can be created in the CER Editor. See "Technical View" on page 87

Rule Object

A rule object is an instance of a Rule Class, for example, John Smith (a Person), John Smith's income from a part-time job (Income), or John Smith's application for child support benefits (Claim).

Rule Attribute

A rule attribute is a question that can be asked. It is defined on a Rule Class and might be asked of any Rule Object of that class, for example, the Person rule class might define the dateOfBirth rule attribute, and so the John Smith rule object might be asked its dateOfBirth, (for example, 3rd October 1970). A new Attribute can be created for the selected rule class in the CER Editor. See "Technical View" on page 87

Expression

An expression is a calculation step that can be used to answer a question, for example, if the eligibility of a claim depends on a person's total income being below a certain threshold, we can use a "sum" expression to calculate the total income and then use a "compare" expression to compare that total with the threshold amount. To create an expression, a "sum" rule element can be dragged to the rule attribute in the CER Editor. See "Business View" on page 86

Rule Set

A rule set is a collection of rule classes, typically centered around a specific purpose, for example, a rule set for child benefit determination might include the rule classes Claim, Person, and Income. A new rule set can be created in the Rules and Evidence section of the Administration Interface.

Note: Rule sets are no longer stand-alone. A class in one rule set can extend a rule class from another rule set; the data type of a rule attribute in one rule set can be a rule class from another rule set; and expressions to read or create rule objects can use rule classes from other rule sets.

Rule Session

A rule session controls the running of rules. For example, your application might create a rule session to determine John Smith's eligibility for child benefit, by invoking the appropriate rule set and asking eligibility questions about John's personal circumstances.

Authoring and Testing Environment

Cúram Express Rules (CER) rule sets are created and maintained in the CER Editor. CER rule sets are stored as XML data on the application database. The XML data for a CER rule set adheres to the CER-supplied rule schema.

CER also includes a comprehensive rule set validator that can detect errors in your rule set before it allows your rules to run. You can validate your rule set in the CER Editor. For more information, see the **CER Editor Global Menu section.**

CER supports you running rule sessions in:

- A production environment
- A stand-alone test environment

Running Rules in Production

In a production environment, CER integrates with your application to answer questions when needed. CER rule sets are fully dynamic. In production environments, CER supports the uploading of changes to rule sets that take effect when published. No rebuilding or redeploying of your application is required.

Testing Rule Sets

The testing of CER rule sets can be at whatever level suits you. You can provide detailed test data for a complete "business scenario." You can create isolated tests for parts of your rule set without having to carefully set up large amounts of input data. You also can select both options.

For example, the determination of a person's eligibility for child benefit might be a complex calculation that involves (among other things) the comparison of the person's total income to a certain threshold. Furthermore, the calculation of the person's total income is itself a complex calculation, involving decisions that regard whether certain types of income are "countable" for the purposes of child support eligibility.

When the user tests the eligibility calculation, in traditional development you might have to set up income data carefully to provide a calculated total income, which in turn you can use to test the eligibility calculation. Depending on the complexity of the calculations, this set up of data might be tedious to do and be brittle to change.

By comparison, in CER you can "stub out" a calculation without having to supply low-level detailed data. CER makes it straightforward to produce a test that effectively says, "for the purposes of this test, the total income is $10 - do not attempt to calculate the total income during this test."

This CER facility makes it easy to test all the functions in your rule set at a level that makes sense.

RuleDoc
> An HTML extract of the structure of your rule sets.

SessionDoc
> An HTML representation of the data in your rule objects.

Coverage Tool
> Reports the extent of your rule set that was exercised by your tests.

The authoring environment also includes tools to assist with development and testing of your rules.

CER Editor Global Menu

The Cúram Express Rules (CER) Editor provides common functions as part of the global menu for easier access.

Runtime Environment

CER executes rules on demand at runtime. CER also includes features that allow you to complete the following tasks.

- Store rule objects on the database, so that the rule objects are available for future processing, and
- Integrate with the Dependency Manager to detect when input data has changed, and to automatically update calculation results which depend on these input data items (akin to familiar spreadsheet processing).

Benefits of CER

A brief overview of the key benefits of CER.

CER offers the following key benefits.

Simplicity
> CER rule sets are only as complex as your business requirements. Business users and technical users can read CER rule sets and easily understand 'what's going on'. Rule sets are simple to write and simple to test.

Flexibility
> Rule sets are easy to change. You can add new questions at any time and CER guarantees that existing behavior is entirely unaffected. You can make changes to the way an existing questions is answered, and CER will show you which calculations depend on that question, so that you are fully in control of the effect of your change.

Localization Support
> CER can produce localizable output, so that answers to questions can be displayed to your end users according to their language and locale preferences.

Validity
> CER works hard to detect errors in your rule set before you run it. The CER rule set validator reports as many errors as it can so you can fix them in one go. CER finds technical problems in your rule set so that you only have to concentrate on the functionality of your rule set.

Testability

You can test your CER rule sets at the granularity that suits you. CER allows you to control your large rule sets by creating tests for discrete sections of your rules.

Dynamic Support

You can make changes to your CER rule set in a running system and your changes take effect immediately upon publication.

Spreadsheet-like Behavior

Construction of CER rules is like layering formulae in the cells of a spreadsheet (which will be familiar to many users). When input data (such as evidence, personal data or payment rates) change, CER integrates with the Dependency Manager to automatically recalculate derived values which are affected by the change.

CER Usage

Learn about how CER is used by other application components in IBM Cúram Social Program Management. You can also use CER to perform calculations for your own custom business areas.

The CER development environment is tightly integrated with the wider application development environment. CER is used by the following application areas.

Universal Access Responsive Web Application

The Universal Access Responsive Web Application uses CER to determine potential eligibility for social services programs when citizens use the self service module to perform self-screening. Based on the data that is captured, CER determines which programs the citizen is potentially eligible for, and provides a textual explanation of why that citizen is or is not potentially eligible.

Advisor

The Advisor uses CER rules to compute advice to display to users. This advice is automatically updated when circumstances change.

Eligibility and entitlement engine

The case eligibility and entitlement engine tightly integrates with CER to provide determinations of a case's eligibility and entitlement (and explanations of how these were derived) across the full lifetime of the case. The case eligibility and entitlement engine relies on the integration between CER and the dependency manager to identify when to recalculate a case's determination, either due to case-specific changes such as evidence data, or from wider data changes such as changes to personal data or product-wide rate data.

Custom business area calculations

You can also use CER to perform calculations for your own custom business areas. The CER runtime does not have any in-built business concepts; instead, each business area communicates with CER through the use of:

- Interface rule classes that encapsulate the business concepts, and/or
- Extensions to the CER language which are business-concept aware.

For more information about how application components utilize CER, see the business and technical information for that component.

Guiding Principles

CER upholds certain key principles. Knowing a little about these principles can help you to understand the CER approach.

- **Questions are answered only when needed**

  The work done to answer a question is only done when that question is asked.

- **All data is immutable**

  The answer to a question is a value which cannot be accidentally changed by something outside CER. If the answer to a question is recalculated, then a fresh answer value is created.

- **Allow multiple questions**

A rule set does not execute once through, rather a rule set allows as many questions to be asked as needed.

- **Specify rules, not order of execution**

  You specify the rules for answering a question, and leave CER to efficiently answer those questions at runtime.

- **No volatility**

  Identical inputs processed by identical rules always produces the same output.

- **No "working storage"**

  There no counters or running totals. An count or total is a question in its own right - you provide the rules for answering it, and CER worries about the order of execution when answering it.

- **Name what you need to**

  You only have to provide names for business concepts and questions. You do not have to think up descriptive names for interim results (unless you want to).

- **Development and testing go hand-in-hand**

  CER provides strong support for managing the testing of your rules.

- **No in-built business concepts**

  The CER runtime intentionally contains no business concepts. You define the business concepts you need, leaving the CER runtime to be a general-purpose environment.

- **Rule implementation maps neatly to rule requirements**

  The implementation of your rule requirements is as complex as those requirements, but no more so. CER rule sets make sense when viewed by the business analysts who gathered the original requirements.

- **Lean on well-known Java support**

  CER does not reinvent the wheel - functionality provided by existing Java™ technology is easily reused in CER rule sets.

- **Management of calculation dependencies**

  CER integrates with the Dependency Manager to manages calculation dependencies so that you don't have to. When an input data item changes, the Dependency Manager and CER know what to recalculate. You do not have to write any special processing which guesses at which calculated outputs *might* be affected.

Developing and Testing CER Rule Sets

Use this information to learn about how to develop and testing your CER rule sets. This includes creating, localizing, and validating rule sets. By using the rule set interpreter and test code generator, you can run rules and write test code. You can publish rule sets and generate rule documentation.

CER Rules Editor

The CER editor is a user-friendly environment and interface for business and technical users to create, edit, and validate rule sets and their rule classes.

For more detailed information about using the CER editor, see .

Localizing CER Rules

You can localize calculated data that is returned by a CER rule attribute, so that the output can be displayed to users in different locales. You can localize the descriptions for artifacts in your CER rule sets so that users can view rule sets in the CER editor in their own locale.

Important: The names of rule set elements such as rule classes and attributes cannot be localized, as they are used within the CER language as identifiers. For example, you cannot localize the name of an attribute in a `reference` expression.

Instead, you can localize the descriptions of rule elements, leaving the names of elements unchanged.

Localizing Calculated Data

CER supports the Java class `String`. You can use strings in the initial phases of developing your rule set. However, if your rules contain output which needs to be displayed to users in different locales, you can take advantage of CER's support for localization.

The String value "Hello World" in this sample code is fine for users who read English, but what if they do not?

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="HelloWorldRuleSet"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="HelloWorld">

    <Attribute name="greeting">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="Hello, world!"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

CER includes a `curam.creole.value.Message` interface which enables the conversion of a value to locale-specific String at runtime.

For a list CER expressions which can create an instance of `curam.creole.value.Message`, see .

This code sample rewrites the `HelloWorld` example to be localized.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="LocalizableHelloWorldRuleSet"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="HelloWorld">

    <Attribute name="greeting">
      <type>
        <!-- Use Message, not String -->
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- Look up the value from a localizable
             property, instead of hard-coding a
             single-language String -->
        <ResourceMessage key="greeting"
          resourceBundle="curam.creole.example.HelloWorld"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Localization of "Hello, world!" in English.

```
# file curam/creole/example/HelloWorld_en.properties

greeting=Hello, world!
```

Localization of "Hello, world!" in French.

```
# file curam/creole/example/HelloWorld_fr.properties

greeting=Bonjour, monde!
```

How will this message behave at runtime? Any code which interacts with your rule set must invoke the `toLocale` method on any messages, to convert them to the required Locale.

This example shows interaction with the localized rule set.

```java
package curam.creole.example;

import java.util.Locale;

import junit.framework.TestCase;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import
 curam.creole.ruleclass.LocalizableHelloWorldRuleSet.impl.HelloWorld;
import
 curam.creole.ruleclass.LocalizableHelloWorldRuleSet.impl.HelloWorld_Factory;
import curam.creole.storage.inmemory.InMemoryDataStorage;
import curam.creole.value.Message;

public class TestLocalizableHelloWorld extends TestCase {

  /**
   * Runs the class as a stand-alone Java application.
   */
  public static void main(final String[] args) {

    final TestLocalizableHelloWorld testLocalizableHelloWorld =
        new TestLocalizableHelloWorld();
    testLocalizableHelloWorld.testLocalizedRuleOutput();

  }

  /**
   * A simple test case, displaying output localized into different
   * locales.
   */
  public void testLocalizedRuleOutput() {

    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    final HelloWorld helloWorld =
        HelloWorld_Factory.getFactory().newInstance(session);

    // returns a Message, not a String
    final Message greeting = helloWorld.greeting().getValue();

    // to decode the message, we need to use the user's locale
    final String greetingEnglish =
        greeting.toLocale(Locale.ENGLISH);
    final String greetingFrench = greeting.toLocale(Locale.FRENCH);

    System.out.println(greetingEnglish);
    System.out.println(greetingFrench);

    assertEquals("Hello, world!", greetingEnglish);
    assertEquals("Bonjour, monde!", greetingFrench);
  }
```

```
  }
```

If used within a localizable message, the following data types are formatted at runtime in a locale-aware way:

- Rule objects by using the value of the rule object's `description` attribute
- Dates by using `curam.util.type.Date`
- Code table items
- Nested localizable message

All other objects are displayed by using their `toString` method.

Localizing CER Rule Artifact Descriptions

The CER editor allows you to add descriptions to certain rule set artifacts by using the Label annotation. The descriptions are stored as property files that you can localize.

You can add descriptions to the following rule set artifacts.

- Rule Set
- Rule Class
- Rule Attribute
- Expression

When a CER rule set is published in the Administration Application, the descriptions on these rule set artifacts are stored in the application's resource store as property files (named RULESET - (rule set name) (rule set version number). You can localize these property files as for any other resource in the resource store.

Validating Rule Sets

CER includes a validator that checks the structure of your rule sets. You can validate your rule sets by using the Administration Application or the CER editor.

For rule sets on the file system, run the following command to validate the structure of the rule sets:

build creole.validate.rulesets

The target runs the CER rule set validator on your rule sets and reports any errors and warnings.

Tip: The CER rule set validator also reports any warnings about any non-fatal problems in your rule sets. These warnings do not prevent you running and testing your rules, but you should address them to ensure an optimal rule set.

Using the Rule Set Interpreter

CER includes an interpreter that you can use to run dynamic rule sets. Sample code is provided that uses the CER rule set interpreter to run rules from a `HelloWorldRuleSet`.You can run the sample either as a stand-alone Java application, through its `main` method, or as a JUnit test.

```
package curam.creole.example;

import junit.framework.TestCase;
import curam.creole.execution.RuleObject;
import curam.creole.execution.session.InterpretedRuleObjectFactory;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import curam.creole.parser.RuleSetXmlReader;
import curam.creole.ruleitem.RuleSet;
import curam.creole.storage.inmemory.InMemoryDataStorage;

public class TestHelloWorldInterpreted extends TestCase {

  /**
```

```
  * Runs the class as a stand-alone Java application.
  */
 public static void main(final String[] args) {

   final TestHelloWorldInterpreted testHelloWorld =
       new TestHelloWorldInterpreted();
   testHelloWorld.testUsingInterpreter();
 }

 /**
  * Reads the HelloWorldRuleSet from its XML source file.
  */
 private RuleSet getRuleSet() {

   /* The relative path to the rule set source file */
   final String ruleSetRelativePath = "./rules/HelloWorld.xml";

   /* read in the rule set source */
   final RuleSetXmlReader ruleSetXmlReader =
       new RuleSetXmlReader(ruleSetRelativePath);

   /* dump out any problems */
   ruleSetXmlReader.validationProblemCollection().printProblems(
       System.err);

   /* fail if there are errors in the rule set */
   assertTrue(!ruleSetXmlReader.validationProblemCollection()
       .containsErrors());

   /* return the rule set from the reader */
   return ruleSetXmlReader.ruleSet();
 }

 /**
  * A simple test case, using the fully-dynamic CER rule set
  * interpreter.
  */
 public void testUsingInterpreter() {

   /* read in the rule set */
   final RuleSet ruleSet = getRuleSet();

   /* start a session which creates interpreted rule objects */
   final Session session =
       Session_Factory.getFactory().newInstance(
           new RecalculationsProhibited(),
           new InMemoryDataStorage(
               new InterpretedRuleObjectFactory()));

   /* create a rule object instance of the required rule class */
   final RuleObject helloWorld =
       session.createRuleObject(ruleSet.findClass("HelloWorld"));

   /*
    * Access the "greeting" rule attribute on the rule object -
    * the result must be cast to the expected type (String)
    */
   final String greeting =
       (String) helloWorld.getAttributeValue("greeting")
           .getValue();

   System.out.println(greeting);
   assertEquals("Hello, world!", greeting);
 }

}
```

These two ways of running this class. When you write your own code to run rule sets, choose the one that suits you best. The test method `testUsingInterpreter` performs key functions that are described in the following sections.

- It reads in the rule set.
- It starts a CER session.
- It creates a new rule object instance in the session.
- It executes rules by retrieving an attribute's value from the rule object.

Reading the Rule Set

First, the `testUsingInterpreter` method reads in the rule set as follows. This line calls a utility method to read the rule set from an XML source file.

```
/* read in the rule set */
            final RuleItem_RuleSet ruleSet = getRuleSet();
```

Next, the rule set is explicitly validated to ensure that it contains no errors.

```
/* dump out any problems */
            ruleSetXmlReader.validationProblemCollection().printProblems(
            System.err);

            /* fail if there are errors in the rule set */
            assertTrue(!ruleSetXmlReader.validationProblemCollection()
            .containsErrors());
```

Starting a CER Session

The following lines create a new CER session for the rule set.

```
/* start a session which creates interpreted rule objects */
            final Session session =
            Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
            new InterpretedRuleObjectFactory()));
```

A session manages the rule objects created for the classes in the rule set. In this example, we are using a session which creates fully-dynamic rule objects by using `InterpretedRuleObjectFactory`. In an interpreted session, each reference to a rule class or an attribute name happens through an API call that takes that name as a String parameter.

Creating a New Rule Object

The following code sample creates a new rule object, that is, an instance of the `"HelloWorld"` rule class and stores the rule object in the CER session's memory.

```
/* create a rule object instance of the required rule class */
            final RuleObject helloWorld =
            session.createRuleObject("HelloWorld");
```

Executing Rules

This line retrieves the value of the `"greeting"` attribute from the rule object created above.

```
/*
 * Access the "greeting" rule attribute on the rule object -
 * the result must be cast to the expected type (String)
 */
final String greeting =
(String) helloWorld.getAttributeValue("greeting")
.getValue();
```

When the attribute's value is requested, CER executes the rules for deriving the attribute's value, that is, returning the constant string `"Hello, world!"`.

Note: When you run an interpreted session, you must cast the output of `getValue` to the expected data type.

In this example, we requested the value of one attribute. However, while the session is still active, code can request the value of any attribute on any rule object in the session. CER remembers values that are already calculated and only performs a calculation the first time it is requested.

Using the CER Test Code Generator

CER includes a code generator that you can use to generate Java wrapper classes for your rule classes. These generated classes can simplify the writing of your test code and allow the compiler to detect problems which otherwise would only occur at runtime.

The CER rule set interpreter allows you to reference to rule class and attribute names through strings. This allows fully dynamic configuration of rule sets. However, for testing purposes it can be cumbersome to use strings and cast attribute values. If you enter a rule class or attribute name incorrectly, or use the wrong type of cast, then your code might compile with no errors but you will get errors at runtime.

> ⚠️ **Warning:** The generated code is only intended for use in test environments where it is a straightforward to recompile changes to code. The generated code is not portable across machines because it contains absolute paths to the rule sets on the local machine. *Do not* use the generated code in any production environment where rule sets may change dynamically.

This code sample rewrites the code for running the HelloWorldRuleSet to illustrate running rules with the CER-generated test rule classes.

```
package curam.creole.example;

import junit.framework.TestCase;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import curam.creole.ruleclass.HelloWorldRuleSet.impl.HelloWorld;
import
 curam.creole.ruleclass.HelloWorldRuleSet.impl.HelloWorld_Factory;
import curam.creole.storage.inmemory.InMemoryDataStorage;

public class TestHelloWorldCodeGen extends TestCase {

  /**
   * Runs the class as a stand-alone Java application.
   */
  public static void main(final String[] args) {

    final TestHelloWorldCodeGen testHelloWorld =
        new TestHelloWorldCodeGen();
    testHelloWorld.testUsingGeneratedTestClasses();

  }

  /**
   * A simple test case, using the CER-generated test classes for
   * strong typing and ease of coding tests.
   */
  public void testUsingGeneratedTestClasses() {

    /* start a strongly-typed session */
    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    /*
     * create a rule object instance of the required rule class, by
     * using its generated factory
     */
    final HelloWorld helloWorld =
        HelloWorld_Factory.getFactory().newInstance(session);

    /*
     * use the generated accessor to get at the "greeting" rule
     * attribute - no cast necessary, and any error in the
     * attribute name would lead to a compile error
     */
    final String greeting = helloWorld.greeting().getValue();

    System.out.println(greeting);
    assertEquals("Hello, world!", greeting);
  }
```

```
}
```

Note the following comparisons with the TestHelloWorldInterpreted code:

- The session used with the code generator uses `StronglyTypedRuleObjectFactory`, so named as it deals with generated Java classes rather than dynamic `RuleObject` instances.
- Loading the rule set is not required (no utility method).
- The reference to the `HelloWorld` rule class is through an identically-named Java interface. Any typing error in the name is detected by the Java compiler.
- The reference to the `greeting` rule attribute is through an identically-named Java method on the interface.
- No casting of the return type is required. The generated `greeting` method returns the correct type (`String`.

Generating Code

Use this information to learn about how to run the CER code generator to generate your test classes.

To run the CER code generator, run the following command:

build creole.generate.test.classes

This target also runs the CER rule set validator on your rule sets. If there are any errors, the CER rule set validator reports the errors and processing stops. If there are no errors, the CER generator outputs the generated Java classes and interfaces for your CER rule sets and rule classes.

Tip: The CER rule set validator also reports any warnings about any non-fatal problems in your rule sets. These warnings will not prevent you running and testing your rules, but should be addressed to ensure an optimal rule set.

The CER code generator places its output in the `EJBServer/build/svr/creole.gen/source` directory.

This code sample shows a generated Java interface for the `HelloWorld` rule class.

```
/*
 * Generated by Curam CREOLE Code Generator
 * Generator Copyright 2008-2010 Curam Software Ltd.
 */
package curam.creole.ruleclass.HelloWorldRuleSet.impl;
/**
 * Code-generated interface for tests.
 * <p/>
 * Clients must not implement this interface.
 */
public interface HelloWorld extends
 curam.creole.execution.RuleObject {
   /**
    * Code-generated accessor for tests.
    * @return container for the greeting attribute value
    */
   public curam.creole.execution.AttributeValue<? extends
 java.lang.String> greeting();
}
```

Tip: It is best practice to regenerate your test classes if you make the following structural changes to your rule sets on the file system.

- Create a new rule set or remove an existing rule set.
- Add a new rule class to a rule set or remove an existing rule class to a rule set.

Note: You do not need to regenerate the test classes if your changes relate to implementing a rule attribute, that is, its derivation expressions. Derivations are always processed dynamically from the rule set at run time and are not present in the generated test classes.

Reporting on Rule Set Coverage

CER includes a tool to report on the parts of a rule set that are covered at runtime. You can report coverage statistics for any processing that requests values from CER, including the running of the online application and JUnit test execution.

To capture coverage data, in `Bootstrap.properties`, set the environment property `curam.creole.coverage.logfile` () to the location of a file. As rules execute, lines that contain coverage information are appended to the file when the CER expressions are evaluated.

Tip: To clear the coverage data, delete the file specified in your `curam.creole.coverage.logfile` setting.

Over time, the coverage data file can become big. Turn off the capturing of coverage data when not required, by removing or commenting out your `curam.creole.coverage.logfile` setting.

To create a coverage report, run the following target:

build creole.report.coverage -Dfile.coverage.log= *file location*

A simple color-coded drillable report is written to the following location: `.../EJBServer/build/svr/creole.gen/coverage/index.html`, where

- Green = covered
- Yellow = partially covered
- Red = not covered

Rule attributes with a derivation of <specified> are intentionally excluded from the report.

Sample Coverage Report

A sample coverage report is shown here.

CREOLE Coverage Report

Generated: 23-Mar-2011 16:33:07

Coverage for Rule Set: SimpleTestProductEligibilityEntitlementRuleSet

Rule Class Name	Rule Attribute Name	Rule Expressions	Fully Covered		Partially Covered		Not Covered	
Rule Set Summary		623	231	37.08%	1	0.16%	391	62.76%
AgeRangeCalculator		47	45	95.74%	0	0.00%	2	4.26%
	description	2	0	0.00%	0	0.00%	2	100.00%
	homeHelpAgeTimeline	5	5	100.00%	0	0.00%	0	0.00%
	isAliveTimeline	10	10	100.00%	0	0.00%	0	0.00%
	isHomeHelpAgeTimeline	8	8	100.00%	0	0.00%	0	0.00%
	isInAgeRangeTimeline	10	10	100.00%	0	0.00%	0	0.00%
	maximumAge	1	1	100.00%	0	0.00%	0	0.00%
	maximumAgeTimeline	5	5	100.00%	0	0.00%	0	0.00%
	minimumAge	1	1	100.00%	0	0.00%	0	0.00%
	minimumAgeTimeline	5	5	100.00%	0	0.00%	0	0.00%
	personCalculator	0	0		0		0	
CREOLEBonus		0	0		0		0	
	amount	0	0		0		0	
	evidenceID	0	0		0		0	
	type	0	0		0		0	
CREOLEBonusCalculator		5	3	60.00%	0	0.00%	2	40.00%

Figure 1. Example Coverage Report

Note: Rule sets and classes that are included in other rule sets (using the <Include> mechanism) essentially become part of the source of the outermost including rule set. Bear this in mind when analyzing coverage reports.

Maintaining and Publishing Rule Sets

The Administration Application includes screens where existing CER rule sets are listed. You can use the screens to maintain and publish your rule sets. You can extract rule sets that were edited in the application to the file system to synchronize them or place them under source control.

You can:

- View an existing rule set in the CER editor and view historical versions of any CER rule set.
- Once opened, make changes to an existing rule set in the CER editor.
- Create a new rule set and open it in the CER editor to add rules.
- Remove an existing rule set.

Changes to CER rule sets do not take effect immediately, but instead are stored in the publication area until published.

You can accumulate many changes to CER rule sets in this area; indeed, you may *have* to accumulate changes to many rule sets, if the change you are making affects more than one rule set.

You can choose to validate your pending changes at any time. When you are happy with your changes, you can choose to publish the changes. The system will re-validate that the rule sets are valid, and if so will allow publication to continue.

Publication of the CER rule set changes occurs in deferred processing, as existing CER rule objects will need to be updated in line with any changes to CER rule classes, and/or recalculations queued for an attributes whose derivations have changed.

If you want to synchronize rule sets that you have edited in the application with the CER Editor with the version on your files system, you can extract the rule sets from the application, see

Generating the Rule Set Schema and Catalog

The schema for CER rule sets is assembled dynamically to take into account the fixed schema for CER rule sets, classes and attributes, and contributions to CER expressions and annotations by application components. Ordinarily, the dynamically-assembled schema resides in memory when CER performs validation processing. However, you can also generate the schema and a catalog that points to it on the file system.

To generate the CER schema file (`EJBServer/build/svr/creole.gen/schema/RuleSet.xsd`), run the following command:

`build creole.generate.schema`

To generate a catalog (`EJBServer/build/svr/creole.gen/catalog/CREOLECatalog.xml`) pointing to the CER schema file, run the following command:

`build creole.generate.catalog`

Generating Rule Documentation

Use this information to learn how to generate RuleDoc and SessionDoc to help you to develop and test your CER rules.

Generating RuleDoc

RuleDoc is documentation that you can automatically generate from your CER rule sets and rule classes. Run the CER RuleDoc generator to output the RuleDoc for your rule sets and rule classes.

RuleDoc is useful for the following.

- Discussing the behavior of your CER rule set with a non-technical audience.
- Visualizing the dependencies between your rule attributes, particularly as your rule sets grow in complexity.
- Understanding the impacts of any changes you make to a derivation of a rule attribute.

To generate the RuleDoc, run the following command:

`build creole.generate.ruledoc`

The target also runs the CER rule set validator on your rule sets. If there are any errors, the CER rule set validator reports the errors and processing stops. If there are no errors, the CER generator outputs the RuleDoc for your rule sets and rule classes.

Tip: The CER rule set validator also reports any warnings about any non-fatal problems in your rule sets. These warnings do not prevent you running and testing your rules, but you should address them to ensure an optimal rule set.

The CER RuleDoc generator places its output in the `EJBServer/build/svr/creole.gen/ruledoc` directory.

Simple RuleDoc Example

Sample code for a simple Hello World rule set and its generated RuleDoc.

The XML is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="HelloWorldRuleSet"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="HelloWorld">

    <Attribute name="greeting">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="Hello, world!"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Here is the generated RuleDoc for the above rule set, listing its single rule class:

Figure 2. RuleDoc for the `HelloWorldRuleSet`

Clicking on the `HelloWorld` rule class shows its RuleDoc:

Type

Message

Derivation summary

- Default rule object description.

Directly used by

None.

Back to top

---

greeting

Type

String

Derivation summary

- "Hello, world!"

Directly used by

None.

Back to top

Figure 3. RuleDoc for the `HelloWorld` rule class

And clicking on the `greeting` attribute scrolls to its derivation:

Type

Message

Derivation summary

- Default rule object description.

Directly used by

None.

Back to top

---

greeting

Type

String

Derivation summary

- "Hello, world!"

Directly used by

None.

Back to top

Figure 4. RuleDoc for the `greeting` rule attribute

More Complex RuleDoc Example

Sample code for a more complex rule set and its generated RuleDoc.

For more complex rule sets, RuleDoc can help you with the following tasks:

- Navigating the dependencies between the rule classes in your rule set;
- Understanding each rule attribute's derivation calculation; and
- Checking which rule attributes depend on a particular rule attribute.

The following XML shows a more complex rule set for a retirement year calculator.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="RetirementYearRuleSet"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="RetirementYear">

    <Attribute name="yearOfBirth">
      <type>
        <javaclass name="Number"/>
```

```
        </type>
        <derivation>
          <Number value="1970"/>
        </derivation>
      </Attribute>

      <Attribute name="ageAtRetirement">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <Number value="65"/>
        </derivation>
      </Attribute>

      <Attribute name="yearOfRetirement">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <arithmetic operation="+">
            <reference attribute="yearOfBirth"/>
            <reference attribute="ageAtRetirement"/>
          </arithmetic>
        </derivation>
      </Attribute>

  </Class>

</RuleSet>
```

Here is the generated RuleDoc for the retirement year calculator rule set.

Figure 5. RuleDoc showing derivation and usage

The example shows the following:

- The derivation of the `yearOfRetirement` rule attribute (with links to the `yearOfBirth` and `ageAtRetirement` rule attributes on which it depends); and
- The `yearOfBirth` rule attribute, with a link to the `yearOfRetirement` rule attribute which directly depends on it.

Generating SessionDoc

SessionDoc provides you with a record of all of the rule objects that you created during the session and is a powerful debugging aid when you use it with your tests.

It can be useful to use the JUnit `tearDown` hook point to force all the test methods in your test class to output their SessionDoc:

```
@Override
        protected void tearDown() throws Exception {
        /*
        * Write out SessionDoc, to a directory named after the test
```

```
           * method.
           */
          final File sessionDocOutputDirectory =
          new File("./gen/sessiondoc/" + this.getName());
          sessionDoc.write(sessionDocOutputDirectory);

          super.tearDown();
          }
```

Here is the main SessionDoc page for a `testSelfMadeMillionaireScenario` test :

CREOLE Session

Generated: 13-Jul-2012 12:08:08

Session Type

- Recalculation strategy: curam.creole.execution.session.RecalculationsProhibited
- Data storage: curam.creole.storage.inmemory.InMemoryDataStorage
- Rule object factory: curam.creole.execution.session.StronglyTypedRuleObjectFactory

Options

- "Used by" links included: true

Rule Objects (by Rule Set)

- FlexibleRetirementYearRuleSet

Figure 6. SessionDoc for the `testSelfMadeMillionaireScenario` test

Some key details on this page are:

- The date and time that the SessionDoc was created;
- The strategies that the session was created with;
- A list of rule sets whose rule objects were captured in the SessionDoc; and
- Whether "used by" links are included.

Clicking on the link for the sole rule set `FlexibleRetirementYearRuleSet` shows its rule objects:

FlexibleRetirementYearRuleSet

Generated: 13-Jul-2012 12:08:08

External rule objects

Details	Type	Description	Action
details	FlexibleRetirementYearRuleSet.FlexibleRetirementYear	Undescribed instance of rule class 'FlexibleRetirementYear', id '1'	Created during this session

Internal rule objects

Details	Type	Description	Action

Figure 7. Rule Objects for the `FlexibleRetirementYearRuleSet` rule set

- The "external" (bootstrap) rule objects created by client code during this session (only one was created in the test); and
- The "internal" rule objects created by rules during this session (none calculated for this test).

Clicking on the details link for the sole `FlexibleRetirementYear` rule object shows its SessionDoc:

Created externally

Action during this session

Created during this session

Attributes

Name	Declared type	State	Value	Derivation	Depends on	Used by
ageAtRetirement	Number	CALCULATED	50	**If** • retirementCause == / **Then**; • "Lottery winner" / • 35; • "Self-made millionaire" / • 50; Otherwise / • 65	• retirementCause	• yearOfRetirement
description	Message	CALCULATED	Undescribed instance of rule class 'FlexibleRetirementYear', id '1'	• Default rule object description.	*None*	*None*
retirementCause	String	SPECIFIED	Self-made millionaire	• Specified externally.	*None*	• ageAtRetirement
yearOfBirth	Number	SPECIFIED	1980	• Specified externally.	*None*	• yearOfRetirement
yearOfRetirement	Number	CALCULATED	2030	• Arithmetic: ○ yearOfBirth ○ + ○ ageAtRetirement	• ageAtRetirement • yearOfBirth	*None*

Figure 8. SessionDoc for the `FlexibleRetirementYear` rule object

At the top (not shown) are summary details for the rule object, and then every rule attribute on the rule object is listed, with the following details:

- **Name**

  The name of the rule attribute;

- **Declared type**

  The type of the rule attribute as declared in the rule set. The actual runtime value may be from a subtype of this declared type;

- **State**

  The state of the value, that is, whether it was:

  - **CALCULATED**

    Calculated by rules;

  - **SPECIFIED**

    Explicitly specified by client code, replacing any defined calculation, or initialized as part of a `create` expression;

    Note: Before Cúram V6, the state INITIALIZED was used. From Cúram V6, the state SPECIFIED is used instead.

– **NOT_YET_CALCULATED**

Not explicitly specified, not calculated during rules execution (because the value was never requested by any other calculations or tests); or

– **ERROR**

An error occurred during the calculation of the value (see the application logs or console output for details and calculation stack of the error).

- **Value**

A display representation of the value. If the value was never calculated (NOT_YET_CALCULATED), or is in error (ERROR) then "?" will be displayed. If the value is a rule object, then the value will be shown as a navigable hyperlink so that you can see the details of that rule object; and

- **Derivation**

The RuleDoc derivation of the attribute (without any links). For more information on RuleDoc, see "Generating Rule Documentation" on page 14.

- **Depends on**

Links to the attributes which were used to calculate this value.

- **Used by**

(Optional) Links to the attributes which used this value when calculating their values.

Tip: Implementation of a `description` rule attribute on each rule class may make your SessionDoc easier to understand. See "Creating a Rule Attribute Description" on page 117 for more details.

If you are running SessionDoc against a large database, then it may be useful to suppress the output of "used by" links, since the inclusion of such links might cause very many rule objects to be output by SessionDoc. To suppress the output of "used by" links, use the `curam.creole.execution.session.SessionDoc.write(File, boolean)`, passing *false* as the second parameter.

For rule objects that are stored on CER's database tables, you can create SessionDoc for these stored rule objects by running the `curam.creole.util.DumpOutRuleObjects` class, with a single argument being the name of a directory in which to create the SessionDoc. The DumpOutRuleObjects utility retrieves all rule objects from CER's database tables, and thus the action for each external rule object will be "retrieved". Any internal rule objects will be created (as they are not stored) and thus will show an action of "created".

Tip: The DumpOutRuleObjects utility can be a useful way of "browsing" the rule objects stored on CER's database tables, and can be a useful debugging aid once you have reached the point of integrating CER rules into your online application.

You can browse the rule objects to see the values of calculated attributes on rule objects, together with a technical view of how any calculation result was arrived at.

To extract the rule objects snapshot for a case determination to SessionDoc style html output, run the following (one-line) command from the runtime directory:

build creole.extract.ruleobjects

The following input parameters are used to run the `creole.extract.ruleobjects` build target:

- `outputDir`The folder where the HTML output pages will be placed by the tool. This should be a writable folder. This is a mandatory parameter.
- `caseDeterminationID`The unique identifier of the case determination for which you are extracting the rule objects snapshot. This is the *CreoleCaseDetermination.creoleCaseDeterminationId* field on the database. This is a mandatory parameter.
- `logFileDir`The folder where a separate log file, determinationTrace.log will be placed by the tool. This should be a writable folder. This is an optional parameter. If missing, no separate log file will be created by the tool. For optimal efficiency of the tool, it is recommended that this parameter be used for troubleshooting only.

To extract the rule objects snapshot for a program group determination to SessionDoc style html output, run the following (one-line) command from the runtime directory:

build creole.extract.programgroupruleobjects

The following input parameters are used to run the `creole.extract.programgroupruleobjects` build target:

- `outputDir`The folder where the HTML output pages will be placed by the tool. This should be a writable folder. This is a mandatory parameter.
- `programGroupDeterminationID`. The unique identifier of the program group determination for which you are extracting the rule objects snapshot. This is the *CreoleProgGrpDetermination.creoleProgGrpDeterminationId* field on the database. This is a mandatory parameter.

Reporting on Unused Rule Attributes

You can generate a report of rule attributes that are not referenced from any other calculations in your rule set and are candidates for removal.

⚠️ **Warning:** A rule attribute can be a 'top-level' attribute that is referenced only from client code. These attributes might be reported as 'unused' by this report. Do not remove any seemingly unused rule attributes from your rule set unless you are certain that no client code or tests depend on them.

To run the CER unused attribute report, run the following command:

build creole.report.unused.attributes

CER validates your rule sets and reports any unused rule attributes to the console.

Consolidating Rule Sets

CER allows you to split your rule set into smaller files, which can help the concurrent development of rule sets between several documents. Before your CER rule set is loaded, the application build scripts consolidate it into a single rule set file.

The **build creole.consolidate.rulesets** target consolidates the rule sets.

Note: The CER rule set consolidator collapses only `Include` statements that contain a `RelativePath` location. All other types of `Include` statements remain unchanged in the consolidated output.

For information about how to split up your rule set, see "Include Statement" on page 133.

Consolidated Rule Set Example

This sample code illustrates a CER rule set that includes another rule set.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Include"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- This rule class is defined directly in this rule set -->
  <Class name="Person">
    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <!-- Include a rule set defined in another file.

      When assembled into a single rule set, the
```

```
      names of all the rule classes must be unique. -->
  <Include>
    <RelativePath value="./HelloWorld.xml"/>
  </Include>

</RuleSet>
```

This sample code illustrates the same rule set after consolidation.

```
<?xml version="1.0" encoding="UTF-8"?><RuleSet
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 name="Example_Include" xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- This rule class is defined directly in this rule set -->
  <Class name="Person">
    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <!-- Include a rule set defined in another file.

       When assembled into a single rule set, the
       names of all the rule classes must be unique. -->

<!--Start inclusion of ./HelloWorld.xml-->
<Class name="HelloWorld">

    <Attribute name="greeting">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="Hello, world!"/>
      </derivation>
    </Attribute>

  </Class>
<!--End inclusion of ./HelloWorld.xml-->


</RuleSet>
```

Handling Data

A description of how CER handles data. CER performs calculations on data in order to come up with
results that are also data. CER stores rule object data on the database and in snapshots. Use this
information to learn how to create CER sessions, rule objects, specify data types on attributes and
expressions, and handle data that changes over time.

Creating CER Sessions

The main data items used with CER are rule objects and attribute values. All interactions with CER rule
objects and attribute values occur within a CER session.

Rule Objects
A rule object is an instance of a rule class from a CER rule set. For example, the rule object for the
person John Smith.

Attribute Values
An attribute value is the value of a CER rule attribute on a particular rule object. For example, John
Smith's date of birth.

Interactions include the creation, retrieval and/or removal of CER rule objects, and any calculation or
recalculation of attributes on rule objects.

Each CER session is created by using the `curam.creole.execution.session.Session_Factory` class.

When you create a CER session, you must specify the following:

Recalculation Strategy

The strategy for handling a request to recalculate a CER attribute value within a CER session

Note: The ability for CER to perform recalculations directly is now superseded by the Dependency Manager (see "Recalculating CER Results with the Dependency Manager" on page 64). The CER recalculation strategy interface and implementations are provided for backwards compatibility only.

Data Storage

The storage mechanism to use for persistent rule objects. For example in-memory only (which will be lost when the session goes out of scope), database storage, or snapshot storage.

Note: Since Cúram V6, CER offers a choice of data storage implementations. These data storage implementations affect whether rule objects, created or changed in one transaction, can be retrieved or changed in another transactions.

The choice of data storage does *not* affect the semantics of your rule expressions. This means that you can use a light-weight (in-memory) data storage for your JUnit tests (so that large numbers of JUnit tests can execute quickly), and a persistent (database) data storage for your production logic (so that rule objects are persisted across transactions), *without* any differences in your underlying calculations.

In turn, implementations of data storage must specify a rule object factory to use. This factory governs whether rule objects are created in a strongly-typed or interpreted-only way.

The application includes the following implementations:

- Recalculation Strategy:

  – **curam.creole.execution.session.RecalculationsProhibited**

    Throws an error if any recalculation within the CER session is attempted.

  – **curam.core.sl.infrastructure.propagator.impl.ImmediateRecalculationStrategy**

    Performs recalculations immediately (synchronously, within the same database transaction).

  – **curam.core.sl.infrastructure.propagator.impl.DeferredRecalculationStrategy**

    Defers recalculations (for stored attribute values) to another database transaction.

    Defers recalculations (for stored attribute values) to another database transaction.

- Data Storage:

  – **curam.creole.storage.inmemory.InMemoryDataStorage**

    Keeps rule objects in memory only. Rule objects in this data storage are only available while the data storage is in scope - typically for a single database transaction only.

  – **curam.creole.storage.database.DatabaseDataStorage**

    Note: As an optimization, only external rule objects and their attribute values are retrieved from data in the Cúram database. Internal rule objects and their attributes, by their nature, can be reliably recomputed later, whereas external rule objects typically contain data from external sources and thus cannot be recomputed.

    Retrieves external rule objects from either:

  – **curam.creole.execution.session.RuleObjectsSnapshot.SnapshotDataStorage**

    Creates an XML document containing details of a set of rule objects involved in the dependencies of one or more attribute calculations. Provides an "audit trail" of the data ultimately used in that calculation. Typically the XML document can be stored on a database table so that the snapshot of rule objects can be queried (but not altered) by a subsequent database transaction.

- a Rule Object Converter registered with the Database Data Storage. Each rule object converter nominates the rule classes that it handles, and when invokedthe rule object converter will read underlying business tables to obtain the appropriate data and populate rule objects in memory and return them to Database Data Storage; or

- CER's own database tables for storing rule objects, if no rule object converter is registered to handle the rule class requested.

Note: Note that each rule object converter is permitted to impose limits on its support for "readall" on page 183 and "readall" on page 183 expressions. For example, some rule object converters may not support the execution of a "readall" on page 183 (without a nested `match`), or place restrictions on which rule attributes can be named in the `match`'s `retrievedattribute` value. Any violations of the rule object converter's limitations will result in an exception being thrown at runtime. You should ensure that your rule set tests include logic which invokes the rule object converters (that is, which runs against Database Data Storage) - in contrast to the majority of your rules logic tests which will use In Memory Data Storage (and which thus do not invoke the rule object converters). See the documentation for each rule object converter implementation to understand any limits it imposes on its support for "readall" on page 183.

External rule objects are available for retrieval and manipulation by subsequent database transactions.

- **curam.creole.storage.hybrid.HybridDataStorage**

    Combines behavioral aspects of InMemoryDataStorage and DatabaseDataStorage. *Reserved for Cúram internal use only.*

- Rule Object Factory:

    - **curam.creole.execution.session.StronglyTypedRuleObjectFactory**

        Creates and retrieves rule objects as instances of Java classes generated by the CER Test Code Generator (see "Using the CER Test Code Generator" on page 11).

        Note: Must not be used in production code.

    - **curam.creole.execution.session.InterpretedRuleObjectFactory**

        Creates and retrieves rule objects in a fully-interpreted (and thus dynamic) way. (see "Using the Rule Set Interpreter" on page 8).

Important: In general you should not use more than one CER session within one database transaction. The in-memory copies of rule objects in a CER session are independent of in-memory copies of rule objects in all other CER Sessions.

Behavior is not guaranteed if more than one CER session (in the same transaction) attempts to retrieve or query the same rule object from the database.

- **Unit tests**

    Use an InMemoryDataStorage (for speed of execution) with a StronglyTypedRuleObjectFactory (so that generated Java classes can be used in tests), and RecalculationsProhibited (so that tests do not accidentally change data part-way through).

- **Production logic with dynamic rule sets**

    Use a DatabaseDataStorage (so that rule objects are available across transactions) with an InterpretedRuleObjectFactory (so that rule sets are fully dynamic), and RecalculationsProhibited, and use the features provided by the Dependency Manager (see "Recalculating CER Results with the Dependency Manager" on page 64) to perform any requests to recalculate CER values in a new (and independent) CER session.

- **Snapshots**

    Use a SnapshotDataStorage (so that rule objects are read from an unalterable XML document) with an InterpretedRuleObjectFactory (so that rule sets are fully dynamic), and RecalculationsProhibited (snapshots do not support changes).

Creating External and Internal Rule Objects

You can create external and internal rule objects. External rule objects are created outside of the context of a calculation and usually represent agency concepts such as a Person or Case. Internal rule objects are created by CER rules or in Java code called out to by CER rules.

Internal rule objects tend to be calculators or derived data for an interim step in a complex chain of calculations. The distinction between external and internal rule objects is explained as follows and affects whether:

- A rule object can be retrieved by using the expression.
- A rule object can be stored on the database for retrieval in subsequent transactions.

Creating External Rule Objects

Use this information to learn about external rule objects, when to use them, and how to create them.

Create external rule objects for "bootstrap"' or top-level rule objects that must always exist for your client code to ask meaningful questions and rule objects that are created based on external data. For example, a Person or a Case.

CER allows client code to ask questions of a rule object. CER executes rules to provide the answers to those questions. For client code to ask a question of a rule object, that rule object must be known to both the client code and CER. Therefore, the CER rule session must have at least one "bootstrap" rule object that is created or retrieved by client code. This client code can be test code or code that integrates CER with an application.

An external rule object is the starting point for the client code to ask questions. The answer to a question might provide a rule object or list of rule objects that were either created from rules or retrieved from other external rule objects.

Important: Once calculations have started, the `RecalculationsProhibited` strategy prevents the creation of any more rule objects which would invalidate any previously-calculated calculations.

To avoid these errors, structure your client code or tests so that the creation of all your test rule objects occurs *before* any calculations, that is, before any execution of `getValue`.

Example of External Rule Object Creation

An example rule set is shown in this sample code.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_externalRuleObjects"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- These attributes must be specified at creation time -->
    <Initialization>
      <Attribute name="firstName">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>

      <Attribute name="lastName">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
    </Initialization>

    <Attribute name="incomes">
      <type>
        <javaclass name="List">
          <ruleclass name="Income"/>
        </javaclass>
```

```
        </type>
        <derivation>
          <!-- Read all the rule objects of
               type "Income" -->
          <readall ruleclass="Income"/>
        </derivation>
      </Attribute>

    </Class>

    <Class name="Income">
      <Attribute name="amount">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

    </Class>

  </RuleSet>
```

In the rule set, the "readall" on page 183 expression is used to retrieve all instances of the `Income` rule class.

To create an external rule object, your client code or tests must specify the session when creating the rule object.

For example:

```
package curam.creole.example;

import junit.framework.TestCase;
import curam.creole.calculator.CREOLETestHelper;
import curam.creole.execution.RuleObject;
import curam.creole.execution.session.InterpretedRuleObjectFactory;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import curam.creole.parser.RuleSetXmlReader;
import
 curam.creole.ruleclass.Example_externalRuleObjects.impl.Income;
import
 curam.creole.ruleclass.Example_externalRuleObjects.impl.Income_Factory;
import
 curam.creole.ruleclass.Example_externalRuleObjects.impl.Person;
import
 curam.creole.ruleclass.Example_externalRuleObjects.impl.Person_Factory;
import curam.creole.ruleitem.RuleSet;
import curam.creole.storage.inmemory.InMemoryDataStorage;

/**
 * Tests external rule objects created directly by client code.
 */
public class TestCreateExternalRuleObjects extends TestCase {

  /**
   * Example showing the creation of external rule objects using
   * generated code.
   */
  public void testUsingGeneratedTestClasses() {

    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    /**
     * Note that the compiler enforces that the right type of
     * initialization arguments are provided.
     */
    final Person person =
        Person_Factory.getFactory().newInstance(session, "John",
            "Smith");
```

```
      CREOLETestHelper.assertEquals("John", person.firstName()
          .getValue());

    /**
     * These objects will be retrieved by the
     *
     * <readall ruleclass="Income"/>
     *
     * expression in the rule set.
     */
    final Income income1 =
        Income_Factory.getFactory().newInstance(session);
    income1.amount().specifyValue(123);

    final Income income2 =
        Income_Factory.getFactory().newInstance(session);
    income2.amount().specifyValue(345);

  }

  /**
   * Example showing the creation of external rule objects using
   * the CER rule set interpreter.
   */
  public void testUsingInterpreter() {

    /* read in the rule set */
    final RuleSet ruleSet = getRuleSet();

    /* start an interpreted session */
    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new InterpretedRuleObjectFactory()));

    /**
     * Note that the compiler cannot enforce that the right type of
     * initialization arguments are provided - if these are wrong
     * CER will report a runtime error.
     */
    final RuleObject person =
        session.createRuleObject(ruleSet.findClass("Person"),
            "John", "Smith");
    CREOLETestHelper.assertEquals("John", person
        .getAttributeValue("firstName").getValue());

    /**
     * These objects will be retrieved by the
     *
     * <readall ruleclass="Income"/>
     *
     * expression in the rule set.
     */
    final RuleObject income1 =
        session.createRuleObject(ruleSet.findClass("Income"));
    income1.getAttributeValue("amount").specifyValue(123);

    final RuleObject income2 =
        session.createRuleObject(ruleSet.findClass("Income"));
    income2.getAttributeValue("amount").specifyValue(345);
  }

  /**
   * Reads the Example_externalRuleObjects from its XML source
   * file.
   */
  private RuleSet getRuleSet() {

    /* The relative path to the rule set source file */
    final String ruleSetRelativePath =
        "./rules/Example_externalRuleObjects.xml";

    /* read in the rule set source */
    final RuleSetXmlReader ruleSetXmlReader =
        new RuleSetXmlReader(ruleSetRelativePath);

    /* dump out any problems */
    ruleSetXmlReader.validationProblemCollection().printProblems(
        System.err);

    /* fail if there are errors in the rule set */
```

```
    assertTrue(!ruleSetXmlReader.validationProblemCollection()
        .containsErrors());

    /* return the rule set from the reader */
    return ruleSetXmlReader.ruleSet();
  }

}
```

Creating Internal Rule Objects

CER allows your rules to create internal rule objects as the result or a by-product of performing calculations. Use this information to learn about internal rule objects, when to use them, and how to create them from rules.

You can use this mode to conditionally create rule objects according to rules, or where the initialization attribute values are calculated from other rules, or where the rule object is only an interim step in a calculation. For a very complex calculation, expect to have one external rule object that holds an attribute for the calculation result, several external rule objects for input data from outside of rules, and possibly a very large number of internal rule objects for the intermediate calculation steps.

If you have rule objects that will always exist and need to be accessed by "readall" on page 183 expressions, consider creating external rule objects directly in your code instead.

Example of Internal Rule Object Creation

To create a rule object, use the "create" on page 155 expression, specifying the initialization argument values required by the rule class and/or additional values to specify on the created rule object.

This sample code is an example CER rule set that uses the "create" on page 155 expression to conditionally create rule objects from rules.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_internalRuleObjects"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Uses <create> to get a new rule object.

        Other calculations (on this or other rule objects) can
        access this newly-created rule object by referring to
        this attribute, i.e.

        <reference attribute="minorAgeRangeTest"/> .

        The rule object created CANNOT be retrieved using a
        <readall> expression.
     -->
    <Attribute name="minorAgeRangeTest">
      <type>
        <ruleclass name="AgeRangeTest"/>
      </type>
      <derivation>
        <!-- Create an age-range test which checks whether this
            person is aged between 0-17 inclusive (i.e. is
            under 18 years).
         -->
        <create ruleclass="AgeRangeTest">
          <this/>
          <Number value="0"/>
          <Number value="17"/>
        </create>
      </derivation>
```

```
      </Attribute>

      <!-- Uses the age-range check to determine whether this person
           is a minor. -->
      <Attribute name="isMinor">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <reference attribute="isPersonInAgeRange">
            <reference attribute="minorAgeRangeTest"/>
          </reference>
        </derivation>
      </Attribute>

      <!-- Uses <create> to get a new rule object, within
           another expression.

           Because the new rule object is created "anonymously",
           it is not available to any other rule objects (but
           it will still show up as "created" in any SessionDoc.
      -->
      <Attribute name="isOfWorkingAge">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <!-- Create an age-range test which checks whether this
               person is legally permitted to work (i.e. is aged
               at least 16 and less than 65), and then check
               whether the test passes. -->
          <reference attribute="isPersonInAgeRange">
            <create ruleclass="AgeRangeTest">
              <this/>
              <Number value="16"/>
              <Number value="64"/>
            </create>
          </reference>
        </derivation>

      </Attribute>

  </Class>

  <!-- A generic test which checks whether the
       person's age lies within a specified (inclusive)
       range. -->
  <Class name="AgeRangeTest">
    <Initialization>
      <Attribute name="person">
        <type>
          <ruleclass name="Person"/>
        </type>
      </Attribute>
      <Attribute name="minimumAge">
        <type>
          <javaclass name="Number"/>
        </type>
      </Attribute>
      <Attribute name="maximumAge">
        <type>
          <javaclass name="Number"/>
        </type>
      </Attribute>
    </Initialization>

    <Attribute name="isPersonInAgeRange">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <all>
          <fixedlist>
            <listof>
              <javaclass name="Boolean"/>
            </listof>
            <members>
              <compare comparison=">=">
                <reference attribute="age">
                  <reference attribute="person"/>
                </reference>
```

```
                 <reference attribute="minimumAge"/>
              </compare>
              <compare comparison="&lt;=">
                 <reference attribute="age">
                    <reference attribute="person"/>
                 </reference>
                 <reference attribute="maximumAge"/>
              </compare>
           </members>
        </fixedlist>
     </all>
  </derivation>
</Attribute>

  </Class>

</RuleSet>
```

Important: Rule objects created by using the "create" on page 155 expression *cannot* be retrieved by using "readall" on page 183 expressions, because CER cannot guarantee that all internal rule objects have been or will be created (depending on whether a "create" on page 155 expression is encountered in the execution path for a calculation).

Note: As for all CER expressions, the "create" on page 155 expressions will only be calculated if requested, for example, the *minorAgeRangeTest* rule object is only created if its value or the value of *isMinor* is requested by client code or another calculation.

In the example, *isOfWorkingAge* uses the technique of creating a rule object "anonymously" by wrapping the creation of a rule object within an expression which references some attribute on the newly-created rule object. Such rule objects are not available to other calculations but will still show up in any generated SessionDoc.

An anonymous rule object is useful when you need to access rule attributes in a created rule object, but you do not need to make that created rule object itself available to any other calculations.

Pooling of Internal Rule Object

Since Cúram V6, CER keeps a "pool" of internal rule objects that have been created during a Session.

The Session's pool is queried whenever a "create" on page 155 expression is evaluated. If a rule object with the same initialization and/or specified parameters has already been created, then it will be reused from the pool rather than a new rule object created.

This pooling approach improves efficiency in the situation where many "create" on page 155 statements attempt to create "identical" rule objects. The use of a single rule object means that any calculated attributes on the single rule object are calculated at most once, instead of identical calculations occurring on many "identical" rule objects.

Re-use of pooled rule objects is guaranteed to be safe, because CER's core principles ensure that any calculation depends only on its inputs; and thus identical inputs guarantee identical outputs.

Handling Data Types

When requested, every attribute and expression in a CER rule set returns a piece of data. CER supports the following flexible data types: Rule Classes, Java Classes, and Code Tables. You must specify a data type on every attribute, and on some expressions. in your rule set.

Rule Classes

You can use any CER rule class that you define in your rule set as a data type in the same rule set. Use this information to learn how to specify the rule class data type, inheritance for rule classes, and root rule class extension.

This sample code illustrates a rule class data type.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_ruleclassDataType"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation=
 "http://www.curamsoftware.com/CreoleRulesSchema.xsd">
   <Class name="Person">

     <Attribute name="firstName">
       <type>
         <javaclass name="String"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

     <Attribute name="favoritePet">
       <type>
         <!-- The type of this attribute is a rule class
              defined elsewhere in this rule set.        -->
         <ruleclass name="Pet"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

   </Class>

   <Class name="Pet">

     <Attribute name="name">
       <type>
         <javaclass name="String"/>
       </type>
       <derivation>
         <specified/>
       </derivation>
     </Attribute>

   </Class>

 </RuleSet>
```

Inheritance for Rule Classes

CER supports simple implementation inheritance for rule classes.

A rule class can optionally specify an *extends* declaration to sub-class another rule class in the same rule set.

A sub-rule class inherits the calculated rule attributes of all its ancestor classes, and optionally may override any of these attributes to provide different derivation calculation rules.

A sub-rule class also inherits the initialized rule attributes of all its ancestor classes, and any "create" on page 155 expression for the sub-rule class must specify the value of the initialized attributes for all the ancestor rule classes of the sub-rule class *ahead of* any declared on the sub-rule class itself.

CER allows an attribute to be declared "abstract" on page 139. Every rule class that defines or inherits (but does not override) an abstract attribute must itself be declared abstract. An abstract class cannot be used in a "create" on page 155 expression.

CER will allow a rule object instance of a rule class to be returned wherever one of its ancestor rule classes is expected.

The CER rule set validator will report an error if an expression in your rule set attempts to return an incompatible value:

```
 <?xml version="1.0" encoding="UTF-8"?>
 <RuleSet name="Example_ruleclassInheritance"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation=
 "http://www.curamsoftware.com/CreoleRulesSchema.xsd">

   <!-- The CER rule set validator will insist that this
        rule class is marked abstract, because it contains
```

```
          an abstract rule attribute. -->
<Class name="Resource" abstract="true">
  <Initialization>
    <!-- Whenever a Resource rule object is created,
         its owner must be initialized.

         Since Resource is abstract, it cannot itself be
         used in a <create> expression, only concrete
         subclasses can. -->
    <Attribute name="owner">
      <type>
        <ruleclass name="Person"/>
      </type>
    </Attribute>
  </Initialization>


  <!-- The monetary value of the resource. -->
  <Attribute name="value">
    <type>
      <javaclass name="Number"/>
    </type>
    <derivation>
      <!-- Every resource has an amount, but it's
           calculated in a sub-class-specific way. -->
      <abstract/>
    </derivation>
  </Attribute>
</Class>

<!-- A building is a type of resource. -->
<Class name="Building" extends="Resource">
  <!-- The physical address of the building,
       e.g. 123 Main Street.

       The address value must be specified
       in addition to the inherited owner
       rule attribute, which is an
       initialized attribute on the super-rule
       class. -->
  <Initialization>
    <Attribute name="address">
      <type>
        <javaclass name="String"/>
      </type>
    </Attribute>
  </Initialization>


  <!-- Building is a concrete class
       (no pun intended!), and so the CER
       rule set validator will insist that this
       class inherits or declares a calculation
       for all inherited abstract rule attributes. -->
  <Attribute name="value">
    <type>
      <javaclass name="Number"/>
    </type>
    <derivation>
      <arithmetic operation="-">
        <reference attribute="purchasePrice"> </reference>
        <reference attribute="outstandingMortgageAmount"/>
      </arithmetic>

    </derivation>
  </Attribute>

  <!-- The price originally paid for the building. -->
  <Attribute name="purchasePrice">
    <type>
      <javaclass name="Number"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <!-- The amount of outstanding loans or mortgages against
       this building. -->
  <Attribute name="outstandingMortgageAmount">
    <type>
      <javaclass name="Number"/>
```

```
          </type>
          <derivation>
            <specified/>
          </derivation>
        </Attribute>

    </Class>

    <Class name="Vehicle" extends="Resource">
      <Initialization>
        <Attribute name="registrationPlate">
          <type>
            <javaclass name="String"/>
          </type>
        </Attribute>
      </Initialization>

      <Attribute name="value">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <!-- The value of this type of Resource is
               directly specified, rather than calculated.-->
          <specified/>

        </derivation>
      </Attribute>

    </Class>

    <Class name="Person">

      <!-- A sample attribute showing how initialized attributes
           are inherited. -->
      <Attribute name="sampleBuilding">
        <type>
          <ruleclass name="Building"/>
        </type>
        <derivation>
          <create ruleclass="Building">
            <!-- The first initialized rule attribute
                 is inherited from Resource.

                 Set this person to be the owner -->
            <this/>
            <!-- The second initialized rule attribute
              is specified directly on Building.

              Set the address of the Building. -->
            <String value="123 Main Street"/>
          </create>
        </derivation>
      </Attribute>

      <!-- a sample attribute which shows how a Building can be
           returned as a Resource (because a Building *IS* a
           Resource -->
      <Attribute name="sampleResource">
        <type>
          <ruleclass name="Resource"/>
        </type>
        <derivation>
          <reference attribute="sampleBuilding"/>
        </derivation>
      </Attribute>

    </Class>

 </RuleSet>
```

The Root Rule Class

If a rule class does not specify another rule class to extend, then the rule class automatically extends CER's "root" rule class, which contains a single description rule attribute.

The `description` rule attribute provides a localizable description of the rule object instance. Rule classes are free to override the derivation of the `description` rule calculation for their rule object instances.

Every rule class ultimately inherits from the root rule class (and thus contains a `description` rule attribute), similar to the way that all Java classes ultimately inherit from `java.lang.Object`.

The default implementation of the `description` rule attribute, supplied by the root rule class, uses the "defaultDescription" on page 162 expression.

Java Classes

Use this information to learn about how to handle java classes including package names, immutable objects,inheritance, and parametrized classes.

You can use any Java class on your application's classpath as a data type in your CER rule set.

Important: When storing rule objects on the database, you can use only data types for which there is a type handler registered with CER. CER includes type handlers for most commonly-used data types.

Package Names

The name of a Java class must be fully qualified with its package name, except for classes in the following packages:

- `java.lang.*`
- `java.util.*`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_javaclassDataType"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">
    <Attribute name="isMarried">
      <type>
        <!-- java.lang.Boolean does not need its
            package specified -->
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="dateOfBirth">
      <type>
        <!-- Fully qualified name to a Cúram class -->
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Tip: You cannot use primitive Java types such as `boolean` in CER. Use their class equivalents instead, for example, `Boolean`.

Immutable Objects

A core principle of CER is that each value, once calculated, cannot be changed.

To comply with this principle, any Java classes that you use must be *immutable*.

Important: If you use a *mutable* Java class as a data type in your CER rule set, you must ensure that no Java code attempts to modify the value of any objects of that Java class. CER cannot guarantee the reliability of calculations if values are being changed "underneath it"!

Fortunately, there are a wide range of immutable classes which will typically cater for most of your data type requirements. In general, to determine if a Java class is immutable refer to the Javadoc information.

Listed here are some useful immutable classes which in all likelihood will prove sufficient for your needs:

- `java.lang.String`;
- `java.lang.Boolean`;
- Implementations of `java.lang.Number`; in any case, CER converts instances of `Number` to its own numerical format (backed by `java.math.BigDecimal`) before performing any arithmetic or comparison;
- Implementations of `java.util.List` which do *not* support its optional operations (see the [JavaDoc for List](#));
- `curam.util.type.Date`;
- Implementations of `curam.creole.value.Message`; and
- `curam.creole.value.CodeTableItem`.

Inheritance of Java Classes and Interfaces

CER recognizes the inheritance hierarchy of Java classes and interfaces.

CER will allow a value of a Java class to be returned wherever one of its ancestor Java classes or interfaces is expected:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_javaclassInheritance"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">
    <Attribute name="isMarried">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="isMarriedAsObject">
      <type>
        <!-- For the sake of example, returning this
             value as an java.lang.Object (which is
             unlikely to be useful in a "real" rule
             set. -->
        <javaclass name="Object"/>
      </type>
      <derivation>
        <!-- This is ok, as a Boolean *IS* an Object. -->
        <reference attribute="isMarried"/>
      </derivation>
    </Attribute>

    <Attribute name="isMarriedAsString">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <!-- The CER rule set validator would report the error
             below (as a Boolean *IS NOT* an String):

             ERROR    Person.isMarriedAsString
             Example_javaclassInheritance.xml(28, 41)
             Child 'reference' returns 'java.lang.Boolean',
             but this item requires a 'java.lang.String'. -->
        <!-- <reference attribute="isMarried"/> -->
```

```
        <!-- (Declaring as specified so that this example
              builds cleanly) -->
        <specified/>

      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Parameterized Classes

Java 5 introduced support for parameterized classes, and CER enables you to use parameterized Java classes in your rule set.

The parameters to a parameterized class are simply listed within the `<javaclass>` declaration:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_javaclassParameterized"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">
    <Attribute name="favoriteWords">
      <type>
        <!-- A list of Strings -->
        <javaclass name="List">
          <javaclass name="String"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="luckyNumbers">
      <type>
        <!-- A list of Numbers -->
        <javaclass name="List">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="children">
      <!-- A list of Person rule objects.

           Because java.util.List can be parameterized with
           any Object, we can use a rule class as a parameter.
      -->
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>


    <!-- The dogs owned by this person. -->
    <Attribute name="dogs">
      <type>
        <javaclass name="List">
          <ruleclass name="Dog"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The cats owned by this person. -->
```

```
      <Attribute name="cats">
        <type>
          <javaclass name="List">
            <ruleclass name="Cat"/>
          </javaclass>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>


      <!-- All the pets owned by this person. -->
      <Attribute name="pets">
        <type>
          <javaclass name="List">
            <ruleclass name="Pet"/>
          </javaclass>
        </type>
        <derivation>
          <joinlists>
            <fixedlist>
              <listof>
                <javaclass name="List">
                  <ruleclass name="Pet"/>
                </javaclass>
              </listof>
              <members>
                <!-- all the dogs - dogs are a type of pet -->
                <reference attribute="dogs"/>
                <!-- all the cats - cats are a type of pet -->
                <reference attribute="cats"/>

                <!-- CER will not allow "children" in this
                     expression; a child is not a pet regardless
                     of whether he or she is adorable or claws
                     the furniture.  -->
                <!-- CANNOT BE USED -->
                <!-- <reference attribute="children"/> -->
                <!-- CANNOT BE USED -->

              </members>
            </fixedlist>

          </joinlists>
        </derivation>
      </Attribute>

  </Class>

  <Class abstract="true" name="Pet">

    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Dog" extends="Pet">

    <Attribute name="favoriteTrick">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Cat" extends="Pet">

    <Attribute name="numberOfLives">
      <type>
        <javaclass name="Number"/>
```

```
        </type>
        <derivation>
          <!-- It's well known that every cat
                has 9 lives. -->
          <Number value="9"/>
        </derivation>
      </Attribute>

  </Class>

</RuleSet>
```

For parameters that allow `java.lang.Object`, CER allows you to use any type (including rule objects).
CER enforces "strong typing" even though the parameter, for example, a rule class, is dynamically defined.
CER also recognizes the inheritance hierarchy of rule classes when deciding whether one parameterized
class is assignable to another.

Code Tables

You can use any application code table as a data type in your CER rule set. Use this information to learn
about how to create an instance of a code table entry.

Tip: The code table does *not* necessarily need to exist at development time. If an administrative user uses
the online application to create a new code table, that code table can then be used as a data type in
dynamically-defined CER rule sets.

To create an instance of a code table entry, that is, to refer to a particular item in the code table, use the
"Code" on page 152 expression.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_codetableentryDataType"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="gender">
      <type>
        <!-- The value of this attribute will
              be an entry from the "Gender" Cúram
              code table. -->
        <codetableentry table="Gender"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="isMale">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- Use "Code" to create a codetableentry value
        for comparison. -->
        <equals>
          <reference attribute="gender"/>
          <Code table="Gender">
            <!-- The code from the code table -->
            <String value="MALE"/>
          </Code>
        </equals>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Where to Specify Data Types

For every attribute, calculated and initialized, you must specify a data `type`.

For more information on attributes, see "Attribute" on page 136.

For most expressions, the `type` is either fixed, for example, the "all" on page 142 expression always returns a `Boolean` or can be inferred. For example, a "reference" on page 187 returns the type declared by the referenced `Attribute`.

However, for the following expressions, you must explicitly specify the type.

- "call" on page 148
- "choose" on page 149

Additionally, the "fixedlist" on page 167 expression declares the type of item in the `List` returned within its `listof` statement.

Handling Data that Changes Over Time

A CER timeline is a value that varies over time. Use this information to learn about what is timeline data and what is not timeline data, compare timeline and point-in-time perspectives, create timelines, perform operations on timeline data, and test timeline outputs.

What Is Timeline Data?

A Timeline is a sequence of values of a give type, where each value is effective from a particular date (up until it is superseded by another value). For any given date, a timeline has a value applicable to that date.

Important: It is important when you model data to be clear about whether the data item varies over time. Only use Timeline for data items where the value can vary over time.

Examples of Timeline Data

The following are examples of data that can vary over time:

- A person's total income varies over time as the person receives pay rises or moves between employments. Given that a person's income at a point in time can be represented as a Number, then a person's varying income over time can be represented as a timeline of Numbers, which for simplicity we will write here as Timeline<Number>.

  Note: The notation used in this guide intentionally borrows from that of Java Generics.

- Regardless of total income, a person's employment record varies over time as the person moves between jobs (or has periods of time when the person has no job). If at any one time a person has at most one *primary* employment, then the person's history of primary employment can be represented as a Timeline<Employment>, where Employment is some rule class or Java type which holds employment details, and during periods of no primary employment the value of the timeline is some special marker value such as `null` (to represent "no employment").

- A person might own an asset which is eventually disposed of, for example a person might buy and subsequently sell a car. On any date, the person either does or does not own the car, which can be modeled as a Boolean. Over time, the condition of whether the car is owned on a particular date can be modeled as Timeline<Boolean>. The timeline's value will be `false` prior to the car's purchase date, and `true` from the purchase date up to and including the date of sale (or "until further notice" if the car is not known to be sold).

- Similarly, a person has a date of birth and, eventually a date of death. Persons who are still alive have a blank date of death recorded. On any date, the person is either alive or dead, and so the derived value for "is the person alive?" can be modeled as a Boolean. Over time, the condition of whether the person is alive can be modeled as Timeline<Boolean>. The timeline's value will be `false` prior to the person's date of birth, and `true` from the person's date of birth up to and including the date of death (or "until further notice" if the person has no date of death).

- A parent may have many children, born on different dates. On any particular date, the parent has a list of children who are alive on that date, which can be modeled as List<Person>. Over time, the list of children will change as more children are born or children reach the age of maturity (or, less happily, die in childhood), which can be modeled as Timeline<List<Person>>

The examples above are shown in graph form here.
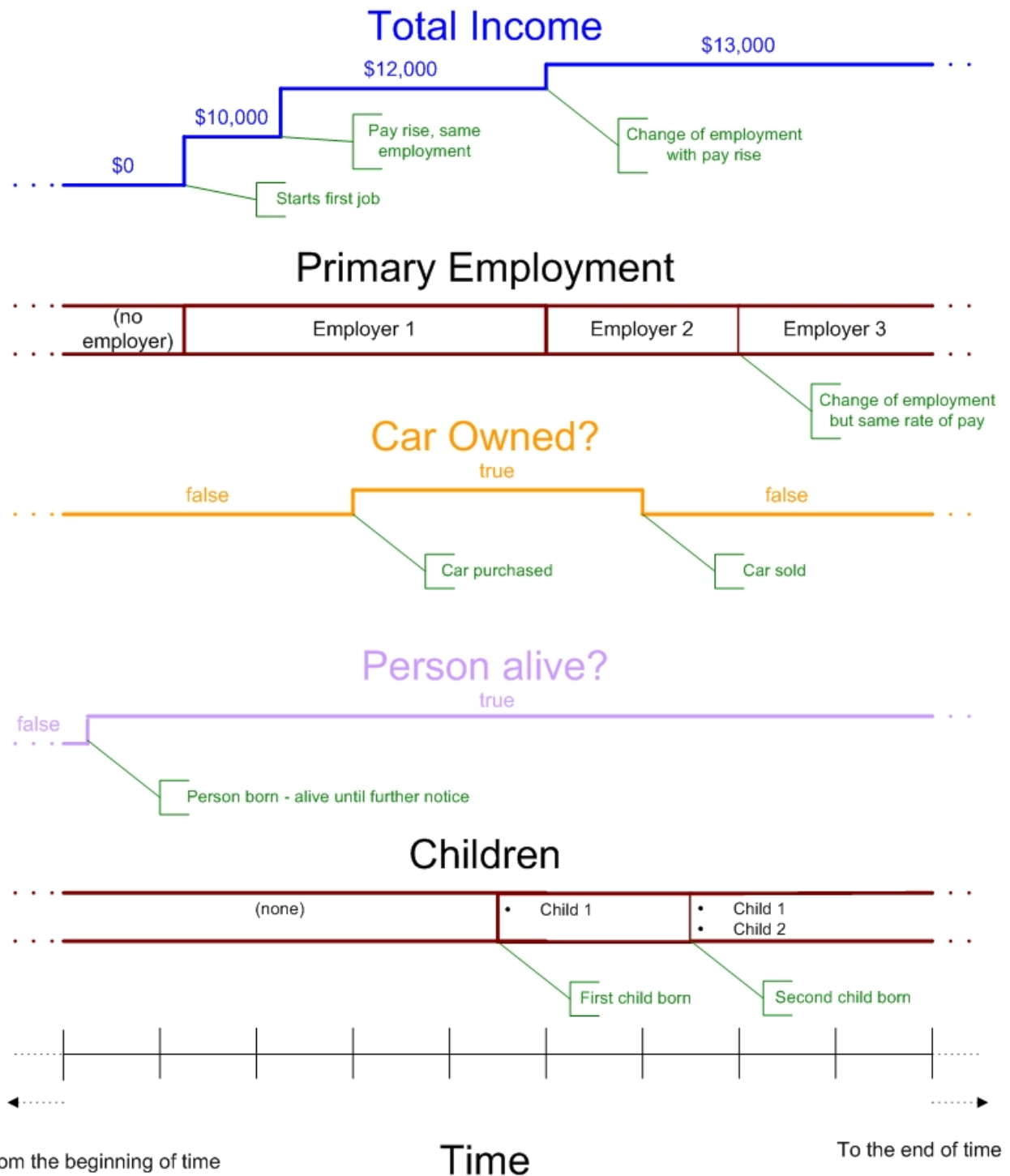
Figure 9. Examples of Timeline Data

Examples of Non-Timeline Data

Data that is not suitable for Timeline includes the following:

- **Unique identifiers**

  One of the features of a unique identifier is it intentionally does not vary over time, for example each person might be assigned a unique social security number. The social security number should be modeled as a Number, not a Timeline<Number>. By contrast, a person's name may vary over time, for example, due to marriage or change of name by deed poll, which is one of the reasons it is a poor choice as an identifier (along with lack of uniqueness).

- **Dates**

  Data of type date does not vary over time. For example, a person's date of birth is one particular date, and so should be modeled as a Date, not as a Timeline<Date>. Dates may be used to *construct* a Timeline, for example, a person's date of birth and date of death would be used to construct a Timeline<Boolean> for whether the person is alive, but the dates themselves are not Timelines.

  Note: In Cúram, certain pieces of data, such as Evidence, can have *two* kinds of history stored:

  - A history of *successions* of data regarding changes of circumstances in the real world, that is, where an event occurs in the real world which means the system's representation of that data is now out-of-date, for example, a change in a person's income due to a pay rise and
  - A history of *corrections* to data held on the system, where the system is found to have an incorrect representation of real-world circumstances, for example, a person's start date of employment is found to have been recorded incorrectly.

  As such, for data items of type Date, the data may be *corrected* in the system, but never *succeeded*. Thus there may be a *correction history* for the data item, but this correction history that is the dates on which the data item was created, is rarely of interest when creating CER rules.

  Typically, data types used in CER rules should model real-world circumstances, rather than corrections to the system representation. Use Timeline where those real-world circumstances can change over time, and do not use Timeline where the real-world data cannot vary over time.

- **Point-in-time data**

  Some data intentionally captures data which applies to a particular date only. For example, a data item for `surnameAtBirth` should be modeled as a String, not a Timeline<String>. A data item for `incomeAtRetirement` should be modeled as a Number, not a Timeline<Number>.

Comparing Timeline and Point-in-time Perspectives

When designing CER rules you can mentally switch back and forth between a point-in-time perspective and a timeline perspective. Use the examples to learn about these perspectives.

The first example is a point-in-time perspective, which does not involve timelines. Then we revisit the example from a timeline perspective.

Example of point-in-time perspective

Let's say we have a simple business requirement for a derivation. From the simple requirement, it is possible to construct a simple truth table.

The following rule is the business requirement.

Rule: On any given date, a person is considered to be a *lone parent of a minor* if, on that date, the person is:

- *not married* and
- has a *dependent who is younger than 16 years of age*.

From this simple business requirement, it is possible to construct a simple truth table for whether a person is considered to be a lone parent of a minor, *on a given date*:

Figure 10. Truth Table for Lone parent of a minor rule

Let's introduce a narrative for an example real-world change of circumstances. On 1st January 2001, Mary and Joe marry. Joe has a son, James, from a previous marriage which ended in divorce on 30th November 1998. James was born on 1st June 1990. On 30th April 2004, Joe sadly dies (and so Mary's marriage ends in widowhood).

We can use the truth table above to determine whether each of the persons is considered to be a *lone parent of a minor* on various dates:

- On 1st October 1997 (to pick a date somewhat at random), Mary is not a lone parent of a minor because on that date she is not married, but she does not have any dependents;
- On 2nd October 1997 Mary is still not a lone parent of a minor, as her circumstances haven't changed since the day before;
- On 30th November 1998 Joe is not a lone parent of a minor, because on that date his son was under 16 but Joe was still married;
- On 1st December 1998 Joe becomes a lone parent of a minor, because on that date his son was still under 16 but Joe was no longer married;
- On 1st January 2001 Mary is still not a lone parent of a minor; although she now has a dependent under 16, she is now married, so for slightly different reasons that earlier she is still not a lone parent;
- On 1st January 2001 Joe is no longer a lone parent of a minor; although he still has a dependent under 16, he is now married again;
- On 1st May 2004 Mary becomes a lone parent of a minor, because her marriage ended due to Joe's death, but James is still her dependent and is under 16;
- On 1st June 2006 Mary stops being a lone parent of a minor, because James turns 16;
- On 1st March 2009 (again somewhat at random) James is not a lone parent of a minor because although he is unmarried, he has no dependents.

Note that we have to evaluate the truth table for various dates in order to build up a picture of when each person is or is not a lone parent of a minor. To some extent, we either have to try dates which we suspect might be "interesting", or try out dates somewhat at random. For example, we suspected that Mary's date of marriage might be interesting, but it turns out that her marriage to Joe does *not* affect her lone-parent-of-a-minor status. However, Joe's lone-parent-of-a-minor status *does* change when he

marries Mary. We didn't think to test whether Joe was a lone parent of a minor on dates before James was born.

Example of Timeline Perspective

Now let's revisit the example rule and circumstances from a timeline perspective.

First, let's slightly reword the requirement as follows:

Rule (reworded): A person is considered to be a *lone parent of a minor* whenever the person is:

- *not married*; and
- has a *dependent who is younger than 16 years of age*.

(We have removed the phrases "On any given date" and "on that date", and used "whenever" instead. This rewording is subtle, but can be key to help making the mental switch from thinking about points in time, to thinking about data which changes over time.)

Now let's draw timelines (of type Timeline<Boolean> for when each of the persons is:

- married; and
- has a dependent under 16 years of age.
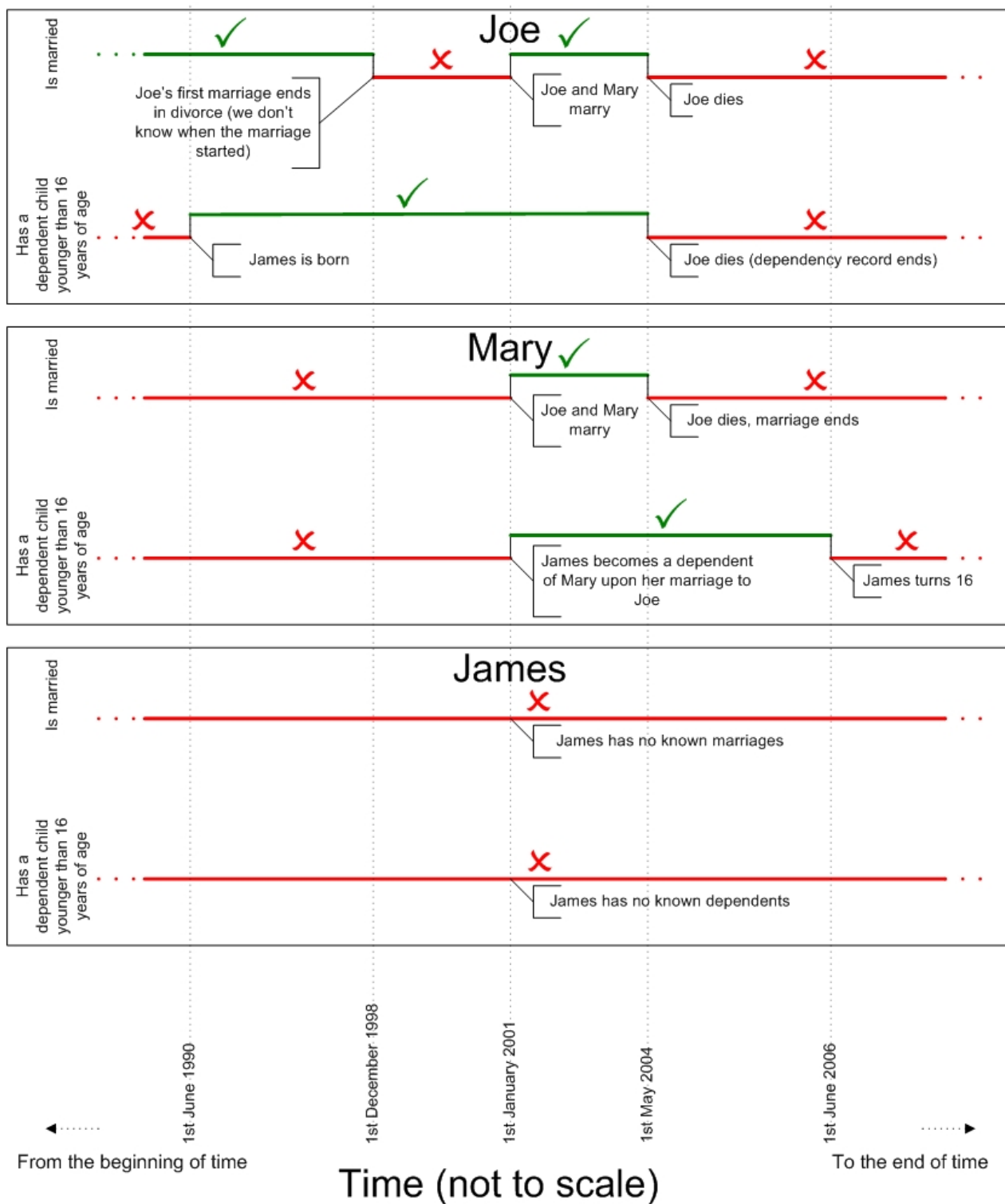
Figure 11. Timelines for Joe, Mary and James's circumstances

From these timelines of Joe, Mary and James's circumstances, let's draw new timelines to derive how their lone-parent-of-a-minor statuses change over time:
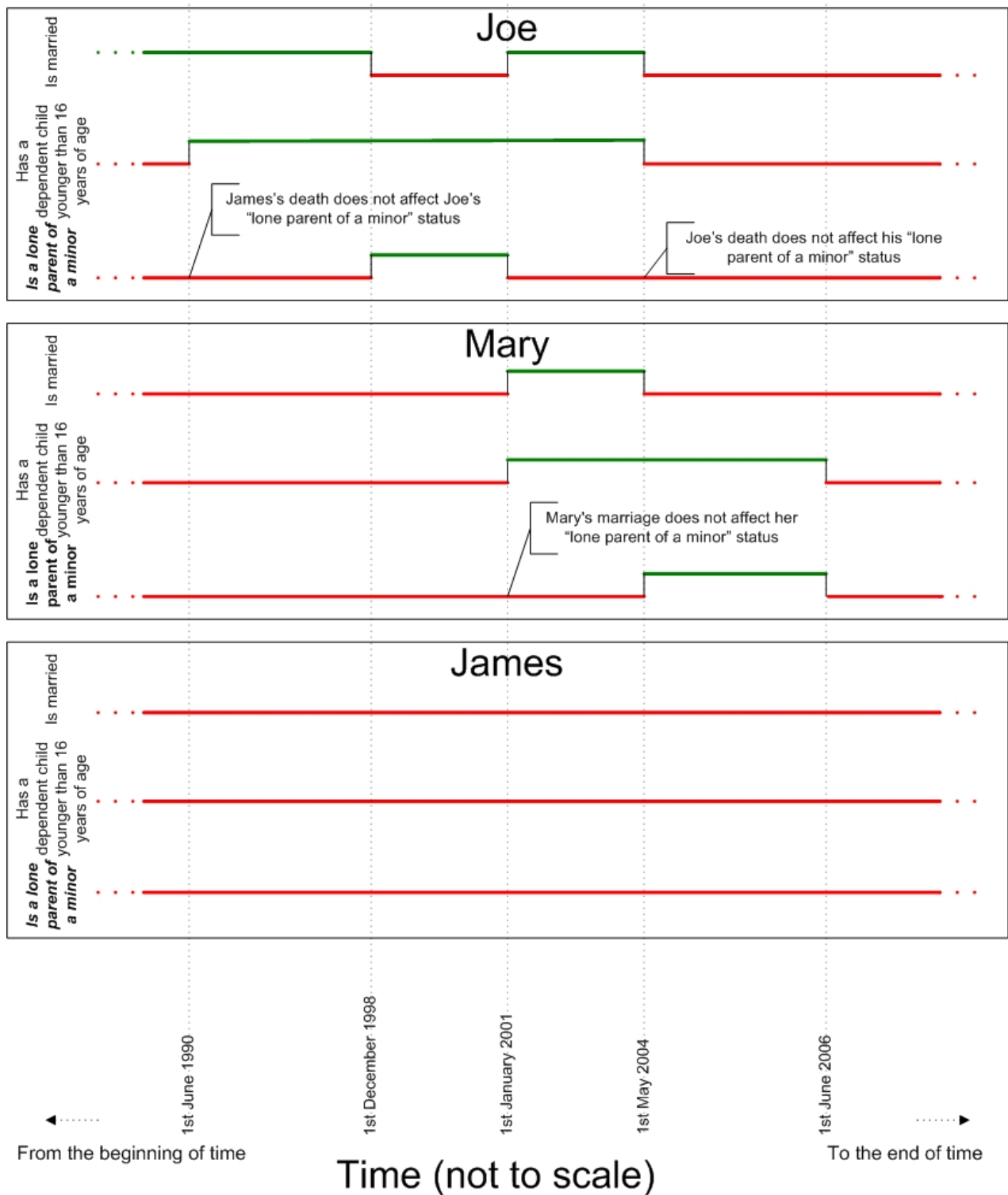
Figure 12. Timelines for Joe, Mary and James's status as a lone parent of a minor

Note that we can immediately read how the lone-parent-of-a-minor status of each person changes, without having to guess at interesting dates:

• Joe is a lone parent of a minor from 1st December 1998 to 31th December 2000 inclusive;

• Mary is a lone parent of a minor from 1st May 2004 to 30th June 2006 inclusive; and

• James is never a lone parent of a minor.

Creating Timelines

The previous example showed how to apply expressions to pre-existing timeline data to calculate data that is a timeline. This section describes how timeline data is created. There are two ways that timeline data is created.

A timeline is created:

- In Java code, either by a client of CER, like the Eligibility and Entitlement Engine, or within Java code that is invoked during a callout from CER.
- In CER rules, by using CER expressions that create timeline data from primitive (non-timeline) data.

Creating Timelines in Java Code

In Java, each piece of timeline data is an instance of the `curam.creole.value.Timeline` parameterized class. Use the example and sample code to learn how to create a Timeline in Java.

For full details of the `curam.creole.value.Timeline` class, see its JavaDoc available at `EJBServer/components/CREOLEInfrastructure/doc` in a development installation of the application.

Each Timeline holds a collection of Intervals, where an Interval is *value* applicable from a particular *start date*. A collection of appropriate intervals must be passed to the Timeline's constructor.

For example, suppose you need to create a Timeline<Number> with these intervals (recall that a timeline stretches infinitely far into the past and future):

- 0 up to and including 31st December 2000;
- 10,000 from 1st January 2001 up to and including 30th November 2003; and
- 12,000 from 1st December 2004 until further notice.

This sample Java code shows how to create such a timeline.

```
package curam.creole.example;

import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class CreateTimeline {

  /**
   * Creates a Number Timeline with these interval values:
   * <ul>
   * <li>0 up to and including 31st December 2000;</li>
   * <li>10,000 from 1st January 2001 up to and including 30th
   * November 2003; and</li>
   * <li>12,000 from 1st December 2004 until further notice.</li>
   * </ul>
   */
  public static Timeline<Number> createNumberTimeline() {

    return new Timeline<Number>(

    // first interval, application from the "start of time"
        new Interval<Number>(null, 0),

        // second interval
        new Interval<Number>(Date.fromISO8601("20010101"), 10000),

        // last interval (until further notice)
        new Interval<Number>(Date.fromISO8601("20041201"), 12000)

    );

  }
}
```

As another example, here is some sample Java code, which is callable a CER `call` expression, to calculate a timeline for a person's age, up to the person's 200th birthday (recall that age timelines must be artificially limited so that the timeline contains a finite number of value changes):

```java
package curam.creole.example;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Collection;

import curam.creole.execution.session.Session;
import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class AgeTimeline {

  /**
   * Creates a timeline for the age of a person, artificially
   * limited to 200 birthdays.
   * <p>
   * Can be invoked from CER rules via a &lt;call&gt; expression.
   */
  public static Timeline<? extends Number> createAgeTimeline(
      final Session session, final Date dateOfBirth) {

    /**
     * The artificial limit, so that the age timeline has a finite
     * number of value changes.
     */
    final int NUMBER_OF_BIRTHDAYS = 200;

    final Collection<Interval<Integer>> intervals =
        new ArrayList<Interval<Integer>>(NUMBER_OF_BIRTHDAYS + 2);

    /*
     * age before date of birth will still be recorded as 0 -
     * create an initial interval application from the
     * "start of time"
     */
    final Interval<Integer> initialInterval =
        new Interval<Integer>(null, 0);
    intervals.add(initialInterval);

    /*
     * Identify each birthday up to the limit. Note that the person
     * is deemed to be age 0 even before the date-of-birth (see
     * above); so the interval here from the date of birth up to
     * the first birthday will be merged into a single interval by
     * the timeline; no matter (it's clearer to keep the logic as
     * is).
     */

    for (int age = 0; age <= NUMBER_OF_BIRTHDAYS; age++) {

      // compute the birthday date
      final Calendar birthdayCalendar = dateOfBirth.getCalendar();

      /*
       * NB use .roll rather than .add to get the correct
       * processing for leap years
       */
      birthdayCalendar.roll(Calendar.YEAR, age);
      final Date birthdayDate = new Date(birthdayCalendar);

      /*
       * the age applies from this birthday until the next birthday
       */
      intervals.add(new Interval<Integer>(birthdayDate, age));
    }

    final Timeline<Integer> ageTimeline =
        new Timeline<Integer>(intervals);

    return ageTimeline;

  }
}
```

Note: In general, timeline data tends to be created outside of rules, by clients of CER.

In particular, the eligibility and entitlement processing contains logic to help convert evidence into timeline data.

For more information, see Developing with Eligibility and Entitlement by using Cúram Express Rules

Creating Timelines in CER Rules
Timeline data is typically created outside of rules by clients of CER, and used to populate the value of a CER attribute by using the specify mechanism. However, CER also contains some expressions that allow you to create timelines directly in CER rules:

You can use the following expressions to create timeline in CER rules:

* `Timeline`, when used with `Interval`, and
* `existencetimeline`.

Timeline and Interval

You can create a Timeline natively in CER rules by first explicitly creating a list of intervals and then using this list to create a timeline.

In practice, these fixed timelines tend to be useful only as a temporary measure while you flesh out your rule set. This code sample shows how to use Timeline and Interval to create a timeline

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Timeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="CreateTimelines">

    <!-- This example uses <initialvalue> to set the value valid
         from the start of time. -->
    <Attribute name="aNumberTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <Timeline>
          <intervaltype>
            <javaclass name="Number"/>
          </intervaltype>
          <!-- Value from start of time -->
          <initialvalue>
            <Number value="0"/>
          </initialvalue>
          <!-- The remaining intervals -->
          <intervals>
            <fixedlist>
              <listof>
                <javaclass name="curam.creole.value.Interval">
                  <javaclass name="Number"/>
                </javaclass>
              </listof>
              <members>
                <Interval>
                  <intervaltype>
                    <javaclass name="Number"/>
                  </intervaltype>
                  <start>
                    <Date value="2001-01-01"/>
                  </start>
                  <value>
                    <Number value="10000"/>
                  </value>
                </Interval>
                <Interval>
                  <intervaltype>
                    <javaclass name="Number"/>
                  </intervaltype>
```

```
                    <start>
                      <Date value="2004-12-01"/>
                    </start>
                    <value>
                      <Number value="12000"/>
                    </value>
                  </Interval>

              </members>
            </fixedlist>

        </intervals>
      </Timeline>

  </derivation>
</Attribute>


<!-- This example does not use <initialvalue>. -->
<Attribute name="aStringTimeline">
  <type>
    <javaclass name="curam.creole.value.Timeline">
      <javaclass name="String"/>
    </javaclass>
  </type>
  <derivation>
    <Timeline>
      <intervaltype>
        <javaclass name="String"/>
      </intervaltype>

      <!-- The list of intervals must include one valid from the
           null date (start of time), otherwise an error will
           occur at runtime, if this expression is evaluated. -->
      <intervals>
        <fixedlist>
          <listof>
            <javaclass name="curam.creole.value.Interval">
              <javaclass name="String"/>
            </javaclass>
          </listof>
          <members>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <!-- "from the start of time" -->
                <null/>
              </start>
              <value>
                <String value="Start of time string"/>
              </value>
            </Interval>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <Date value="2001-01-01"/>
              </start>
              <value>
                <String value="2001-only String"/>
              </value>
            </Interval>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <Date value="2002-01-01"/>
              </start>
              <value>
                <String value="2002-onwards String"/>
              </value>
            </Interval>
          </members>
        </fixedlist>
      </intervals>
    </Timeline>
  </derivation>
</Attribute>
```

```
    </Class>
</RuleSet>
```

existencetimeline

Some business objects have natural start and end dates, which together specify a period for which the business object *exists*. Either or both of the start and end dates may be optional, in which case the existence period for the business object is open-ended.

Examples can include:

- An employment, which starts and later ends.
- An asset, which is purchased and later sold.
- A person who is born and later dies.

The start and end dates for a business object can be used to divide up time into these three periods (or fewer, if either of the start date or end date is blank):

Pr-existence period
    The period of time before the business start date (if the start date exists)

Existence period
    The period of time from the business start date up to and including the business end date

Post-existence period
    The period of time after the business end date (if the end date exists).

It can often be convenient to ascribe a different value to each of these periods for a business object, and to create a timeline from these values. CER contains an `existencetimeline` expression to create a timeline of pre-existence/existence/post-existence values based on optional start and end dates.

If the start date does not exist, then there will be no pre-existence interval in the timeline. For example, if an asset does not have a purchase date recorded, then its effective value will apply from the start of time, with no "zero value" period.

If the end date does not exist, then there will be no post-existence interval in the timeline. For example, if an asset does not have a sold date, then the asset's value will hold until further notice (i.e. arbitrarily far into the future)

See for details on how to use existence timeline in the CER Editor.

Operating on Timelines

CER timelines are useful for storing data which varies over time. CER includes a timeline operation feature that allows CER expressions to operate on timeline data items to produce timeline results.

Preserving change dates
All expressions can operate on timelines in a way in which the change dates for the input timeline values map naturally through to the change dates for the resultant timeline value.

The examples above have introduced the concept of operating on timelines (*is married* timeline, *has a dependent under 16 years of age* timeline) to produce an output timeline (*is lone parent of a minor* timeline).

More formally, for any CER expression that operates on one or more values, CER allows that expression to also operate on a timeline of those values. In general, any operation that can be applied to primitive types, for example, Date, Number, String, Boolean to come up with a result, can instead be applied to Timelines of those types, for example, Timeline<Date>, Timeline<Number>, Timeline<String>, Timeline<Boolean> to come up with a result that is a Timeline value.

CER contains special expressions named and which shield the other CER expressions from "knowing" that they are operating on timelines.

For example, CER contains a expression to add a list of numbers. If a person has several incomes, then we can sum those incomes at a point in time in order to derive the person's total

income at that point in time. However, if instead we have *timelines* of how those income amounts change over time, then we can just as easily use the "sum" on page 198 expression to derive how the total income changes over time:
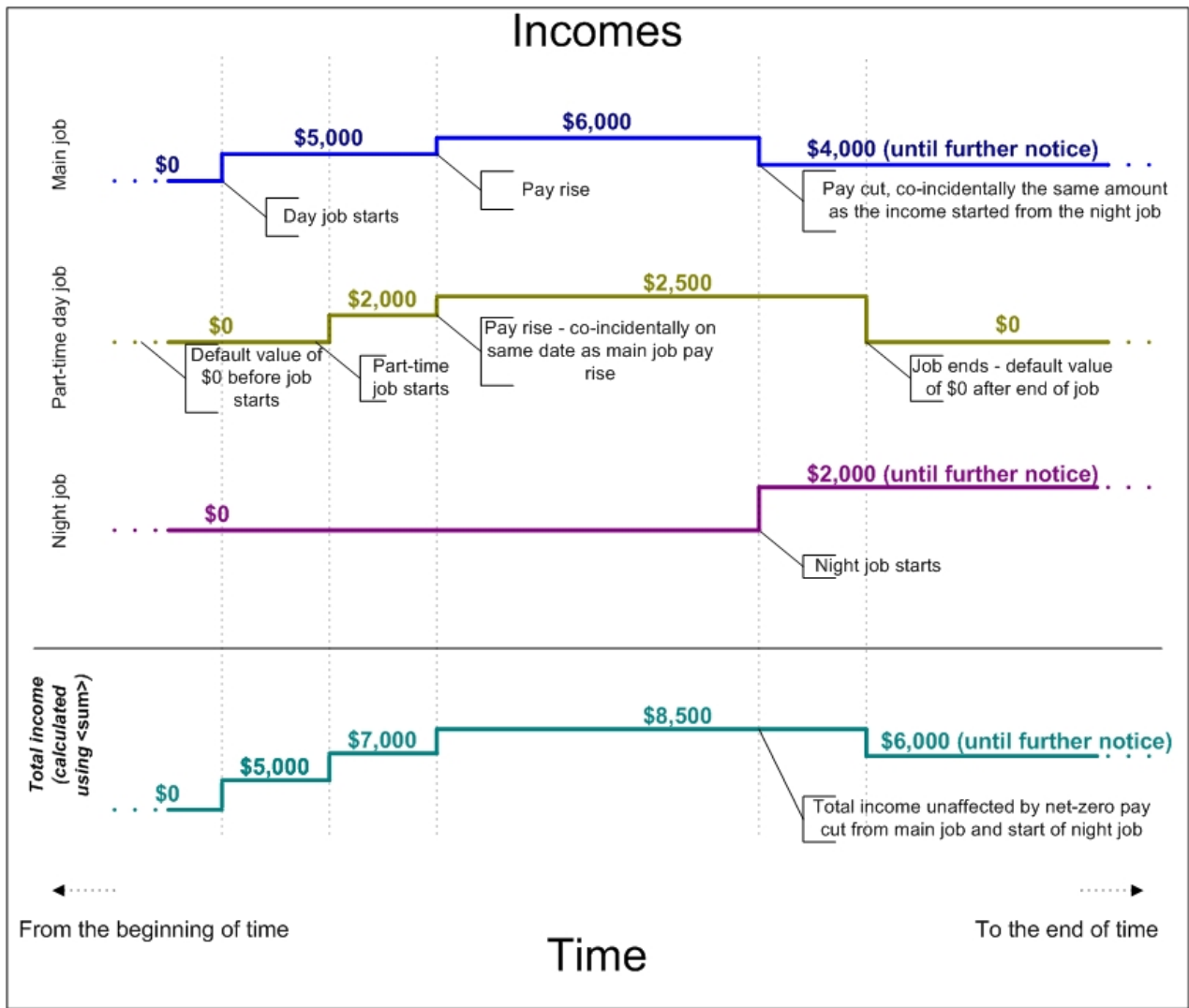
Figure 13. A Total Income Timeline, Calculated using sum

Date-shifting

Use this information to learn how to create a timeline that is based on another timeline, where the dates of value change in the resultant timeline are different from those in the input timeline.

CER does not include any expressions for date-shifting, as the types of date-shifting required tend to be business specific. The recommended approach is to create a static Java method to create your required timeline and invoke the static method from rules by use of the "call" on page 148 expression.

Important: When implementing a date-shifting algorithm, take care to ensure that there is no attempt to create a timeline with more than one value on any given date, as such an attempt will fail at runtime.

The tests for your algorithm should include any tests for edge cases, such as leap-years or months which have different numbers of days.

Date addition Example

You have a business requirement as follows: a person may not apply for a type of benefit within three months of receiving that benefit.

To implement this business requirement, you already have a timeline `isReceivingBenefitTimeline` that shows the periods of time for which a person is receiving benefit.

You now need another timeline `isDisallowedFromApplyingForBenefitTimeline` which shows the periods when it is invalid for that person to reapply for the benefit. This timeline is a date-addition of 3 months to the value-change dates in `isReceivingBenefitTimeline`:
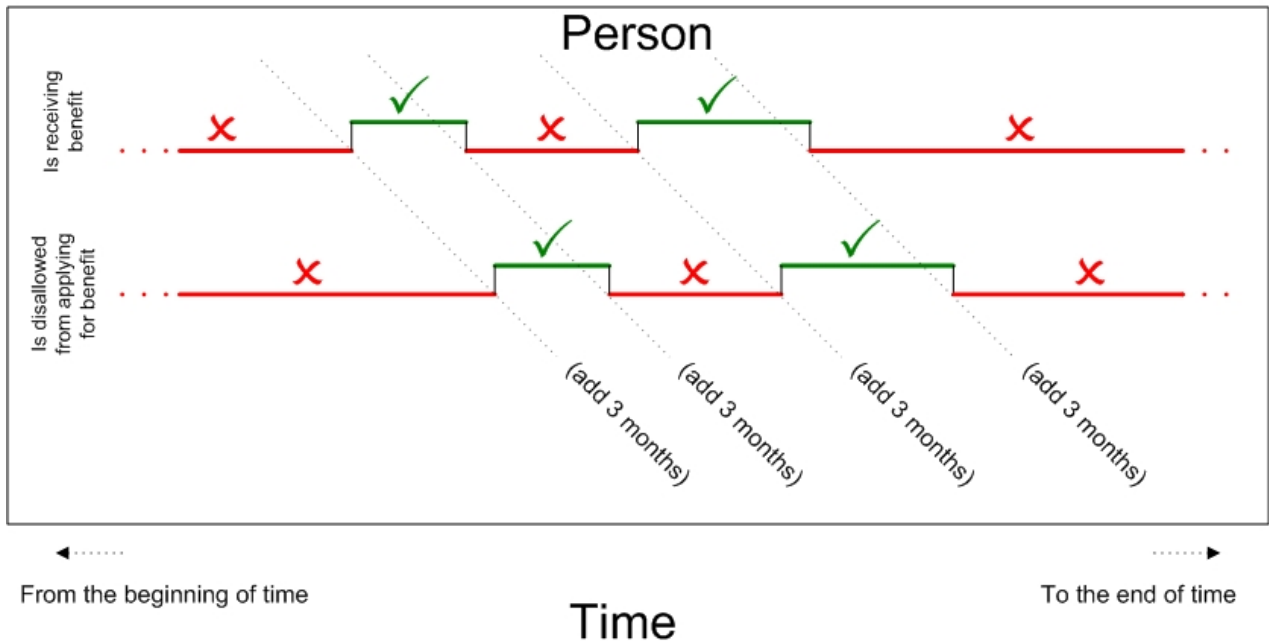
Figure 14. A Requirement for a Date-Addition Timeline

Here is a sample implementation of a static method which can be called from CER rules:

```
package curam.creole.example;

import java.util.Calendar;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import curam.creole.execution.session.Session;
import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class DateAdditionTimeline {

  /**
   * Creates a Timeline based on the input timeline, with the date
   * shifted by the number of months specified.
   * <p>
   * Note that the timeline's parameter can be of any type.
   *
   * @param session
   *          the CER session
   * @param inputTimeline
   *          the timeline whose dates must be shifted
   * @param monthsToAdd
   *          the number of months to add to the timeline change
   *          dates
   * @param <VALUE>
   *          the type of value held in the input/output timelines
   * @return a new timeline with the values from the input
   *          timeline, shifted by the number of months specified
   */
  public static <VALUE> Timeline<VALUE> addMonthsTimeline(
      final Session session, final Timeline<VALUE> inputTimeline,
      final Number monthsToAdd) {

    /*
```

```
       * CER will typically pass a Number, which must be converted to
       * an integer
       */
      final int monthsToAddInteger = monthsToAdd.intValue();

      /*
       * Find the intervals within the input timeline
       */
      final List<? extends Interval<VALUE>> inputIntervals =
          inputTimeline.intervals();

      /*
       * Amass the output intervals. Note that we map by start date,
       * because when adding months, it is possible for several
       * different input dates to be shifted to the same output date.
       *
       * For example 3 months after these dates: 2002-11-28,
       * 2002-11-29, 2002-11-30, are all calculated as 2003-02-28
       *
       * In this situation, we use the value from the earliest input
       * date only - input dates are processed in ascending order
       */
      final Map<Date, Interval<VALUE>> outputIntervalsMap =
          new HashMap<Date, Interval<VALUE>>(inputIntervals.size());

      for (final Interval<VALUE> inputInterval : inputIntervals) {
        // get the interval start date
        final Date inputStartDate = inputInterval.startDate();

        /*
         * Add the number of months - but n months after the start of
         * time is still the start of time
         */

        final Date outputStartDate;
        if (inputStartDate == null) {
          outputStartDate = null;
        } else {
          final Calendar startDateCalendar =
              inputStartDate.getCalendar();

          startDateCalendar.add(Calendar.MONTH, monthsToAddInteger);
          outputStartDate = new Date(startDateCalendar);

        }

        // check that this output date has not yet been processed
        if (!outputIntervalsMap.containsKey(outputStartDate)) {

          /*
           * the output interval uses the same value as the input
           * interval, but with a shifted start date
           */

          final Interval<VALUE> outputInterval =
              new Interval<VALUE>(outputStartDate,
                  inputInterval.value());
          outputIntervalsMap.put(outputStartDate, outputInterval);
        }
      }

      // create a timeline from the output intervals
      final Collection<Interval<VALUE>> outputIntervals =
          outputIntervalsMap.values();
      final Timeline<VALUE> outputTimeline =
          new Timeline<VALUE>(outputIntervals);
      return outputTimeline;

  }
}
```

Date spreading example

You have a business requirement as follows: a car must be taxed for any month where the car is "on the road" for one or more days in that month.

Note: If a car is put back on the road part-way through a month, the keeper of the car must ensure that tax is retrospectively paid for the entire month.

To implement this business requirement, you already have a timeline `isOnRoadTimeline` that shows the periods of time for which a car is "on the road".

You now need another timeline `taxDueTimeline` which shows the periods when the car must be taxed. This timeline is a spread-out of the dates within `isOnRoadTimeline`:
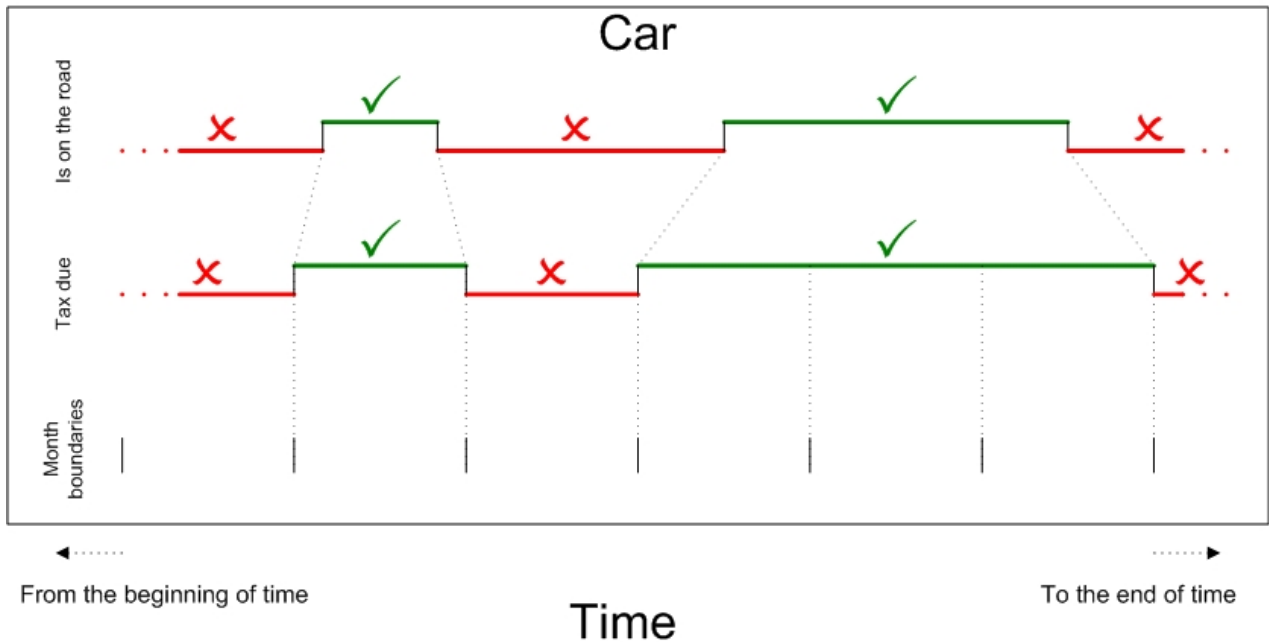
Figure 15. A Requirement for a Date-Spreading Timeline

Here is a sample implementation of a static method which can be called from CER rules:

```
package curam.creole.example;

import java.util.Calendar;
import java.util.Collection;
import java.util.GregorianCalendar;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import curam.creole.execution.session.Session;
import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class DateSpreadingTimeline {

  /**
   * Creates a Timeline for the period for which a car must be
   * taxed.
   * <p>
   * The car must be taxed for the entire month for any month where
   * that car is on-the-road for one or more days during that
   * month.
   */
  public static Timeline<Boolean> taxDue(final Session session,
      final Timeline<Boolean> isOnRoadTimeline) {

    /*
     * Find the intervals within the input timeline
     */
    final List<? extends Interval<Boolean>> isOnRoadIntervals =
        isOnRoadTimeline.intervals();

    /*
     * Amass the output intervals. Note that we map by start date;
     * a car may go off the road during a month, which would imply
     * that no tax is required at the start of the next month, only
     * to return to the road part-way through the next month, in
     * which case it does require taxing after all.
```

```
 *
 * For example, car is put back on the road 2001-01-15, so tax
 * is required (retrospectively) from 2001-01-01.
 *
 * On 2001-01-24 the car is taken back off the road, so it's
 * possible that the car does not require taxing from
 * 2001-02-01.
 *
 * However, on 2001-02-05 the car is put back on the road, and
 * so it does require taxing from 2001-02-01 after all. The
 * resultant timeline will merge these periods to show that the
 * car requires taxing from 2001-01-01 onwards (thus covering
 * from 2001-02-01 too).
 */
final Map<Date, Interval<Boolean>> taxDueIntervalsMap =
    new HashMap<Date, Interval<Boolean>>(
        isOnRoadIntervals.size());

for (final Interval<Boolean> isOnRoadInterval :
isOnRoadIntervals) {
    // get the interval start date
    final Date isOnRoadStartDate = isOnRoadInterval.startDate();

    if (isOnRoadStartDate == null) {
        // at the start of time, the car must be taxed if it is on
        // the road
        taxDueIntervalsMap.put(null, new Interval<Boolean>(null,
            isOnRoadInterval.value()));
    } else if (isOnRoadInterval.value()) {
        /*
         * start of a period of the car being on-the-road - the car
         * must be taxed from the start of the month containing the
         * start of this period
         */

        final Calendar carOnRoadStartCalendar =
            isOnRoadStartDate.getCalendar();
        final Calendar startOfMonthCalendar =
            new GregorianCalendar(
                carOnRoadStartCalendar.get(Calendar.YEAR),
                carOnRoadStartCalendar.get(Calendar.MONTH), 1);
        final Date startOfMonthDate =
            new Date(startOfMonthCalendar);

        /*
         * Add to the map of tax due periods - note that this will
         * push out of the map any "tax not due" interval
         * speculatively added if the car went off-the-road during
         * the previous month
         */
        taxDueIntervalsMap.put(startOfMonthDate,
            new Interval<Boolean>(startOfMonthDate, true));
    } else {
        /*
         * Start of a period of the car being off the road -
         * speculate that from the start of next month, the car may
         * not require tax. This speculation will hold unless the
         * car is subsequently found to be put back on the road
         * next month, in which case this speculation will be
         * discarded (i.e. pushed out of the map).
         */

        final Calendar carOffRoadStartCalendar =
            isOnRoadStartDate.getCalendar();
        final Calendar startOfNextMonthCalendar =
            new GregorianCalendar(
                carOffRoadStartCalendar.get(Calendar.YEAR),
                carOffRoadStartCalendar.get(Calendar.MONTH), 1);
        startOfNextMonthCalendar.add(Calendar.MONTH, 1);

        final Date startOfNextMonthDate =
            new Date(startOfNextMonthCalendar);

        /*
         * Add to the map of tax due periods - note that this will
         * push out of the map any "tax not due" interval
         * speculatively added if the car went off-the-road during
         * the previous month
         */
        taxDueIntervalsMap.put(startOfNextMonthDate,
            new Interval<Boolean>(startOfNextMonthDate, false));
    }
```

```
    }

    // create a timeline from the tax due intervals
    final Collection<Interval<Boolean>> taxDueIntervals =
        taxDueIntervalsMap.values();
    final Timeline<Boolean> taxDueTimeline =
        new Timeline<Boolean>(taxDueIntervals);
    return taxDueTimeline;

  }
}
```

Testing Timeline Outputs

Use the examples to learn how to test timeline outputs with input timelines that have a constant value. You can also use this information learn about the different test approaches that you can take according to your needs:

You can test a rule attribute that returns a timeline of values in your JUnit tests in a similar way to tests for non-timeline values.

To simplify your tests, you do not need to test that "timelineoperation" on page 202 correctly accumulates change dates (unless you want to). To keep your tests simple, generally you can use input timelines which have a constant value forever.

For example, let's say you have a rule attribute which calculates (in a timeline way) the total from a list of numbers:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_NumberSumTimeline"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Totalizer">

    <!-- The timelines to total -->
    <Attribute name="inputNumberTimelines">
      <type>
        <javaclass name="List">
          <javaclass name="curam.creole.value.Timeline">
            <javaclass name="Number"/>
          </javaclass>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The resultant total -->
    <Attribute name="totalTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <timelineoperation>
          <sum>
            <dynamiclist>
              <list>
                <reference attribute="inputNumberTimelines"/>
              </list>
              <listitemexpression>
                <intervalvalue>
                  <current/>
                </intervalvalue>
              </listitemexpression>
            </dynamiclist>

          </sum>
        </timelineoperation>
      </derivation>
    </Attribute>
```

```
   </Class>
</RuleSet>
```

Then you can write a simple test which uses input timelines which have a constant value for all time:

```
package curam.creole.example;

import java.util.Arrays;

import junit.framework.TestCase;
import curam.creole.calculator.CREOLETestHelper;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer_Factory;
import curam.creole.storage.inmemory.InMemoryDataStorage;
import curam.creole.value.Timeline;

public class TestForeverValuedTimelines extends TestCase {

  public void testNumberSumTimeline() {

    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    final Totalizer totalizer =
        Totalizer_Factory.getFactory().newInstance(session);

    // use input values that do not vary over time

    final Timeline<Number> inputTimeline1 =
        new Timeline<Number>(1);
    final Timeline<Number> inputTimeline2 =
        new Timeline<Number>(2);
    final Timeline<Number> inputTimeline3 =
        new Timeline<Number>(3);

    totalizer.inputNumberTimelines().specifyValue(
        Arrays.asList(inputTimeline1, inputTimeline2,
            inputTimeline3));

    // check that the resultant timeline is 6 forever
    CREOLETestHelper.assertEquals(new Timeline<Number>(6),
        totalizer.totalTimeline().getValue());

  }
}
```

Tip: The Timeline class has a convenience constructor to create a timeline with a constant value forever.

In some situations, for example, where you have created your own date-shifting algorithm, or you genuinely need to test that the change dates for input timelines are accurately reflected in resultant timelines, then there are different approaches you can take according to your needs:

- **Strict checking**

  Check that the resultant timeline is exactly equal to an expected timeline value. (The equality semantics of the Timeline class behave as you would expect - two timelines are equal if they contain exactly the same collection of intervals, i.e. the values from the two timelines are identical for every possible date).

- **Relaxed checking**

  Check that the resultant timeline has the value you expect on particular dates.

This example shows the strict testing of a resultant timeline.

```
package curam.creole.example;
```

```
import java.util.Arrays;

import junit.framework.TestCase;
import curam.creole.calculator.CREOLETestHelper;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer_Factory;
import curam.creole.storage.inmemory.InMemoryDataStorage;
import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class TestStrictTimelineChecking extends TestCase {

  public void testNumberSumTimeline() {

    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    final Totalizer totalizer =
        Totalizer_Factory.getFactory().newInstance(session);

    // use input values that vary over time

    final Timeline<Number> inputTimeline1 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 1),
          new Interval<Number>(Date.fromISO8601("20010101"), 1.1)
        ));

    final Timeline<Number> inputTimeline2 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 2),
          new Interval<Number>(Date.fromISO8601("20020101"), 2.2)
        ));

    final Timeline<Number> inputTimeline3 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 3),
          new Interval<Number>(Date.fromISO8601("20030101"), 3.3)
         ));

    totalizer.inputNumberTimelines().specifyValue(
        Arrays.asList(inputTimeline1, inputTimeline2,
            inputTimeline3));

    // strictly check the exact value of the resultant timeline
    CREOLETestHelper.assertEquals(

    new Timeline<Number>(Arrays.asList(
      new Interval<Number>(null, 6),
      new Interval<Number>(Date.fromISO8601("20010101"), 6.1),
      new Interval<Number>(Date.fromISO8601("20020101"), 6.3),
      new Interval<Number>(Date.fromISO8601("20030101"), 6.6)
    )),

    totalizer.totalTimeline().getValue());

  }
}
```

The example shows more relaxed testing of a resultant timeline.

```
package curam.creole.example;

import java.util.Arrays;

import junit.framework.TestCase;
import curam.creole.calculator.CREOLETestHelper;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
```

```
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer;
import
 curam.creole.ruleclass.Example_NumberSumTimeline.impl.Totalizer_Factory;
import curam.creole.storage.inmemory.InMemoryDataStorage;
import curam.creole.value.Interval;
import curam.creole.value.Timeline;
import curam.util.type.Date;

public class TestLaxTimelineChecking extends TestCase {

  public void testNumberSumTimeline() {

    final Session session =
        Session_Factory.getFactory().newInstance(
          new RecalculationsProhibited(),
          new InMemoryDataStorage(
            new StronglyTypedRuleObjectFactory()));

    final Totalizer totalizer =
        Totalizer_Factory.getFactory().newInstance(session);

    // use input values that vary over time

    final Timeline<Number> inputTimeline1 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 1),
          new Interval<Number>(Date.fromISO8601("20010101"), 1.1)
        ));

    final Timeline<Number> inputTimeline2 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 2),
          new Interval<Number>(Date.fromISO8601("20020101"), 2.2)
        ));

    final Timeline<Number> inputTimeline3 =
        new Timeline<Number>(Arrays.asList(
          new Interval<Number>(null, 3),
          new Interval<Number>(Date.fromISO8601("20030101"), 3.3)
        ));

    totalizer.inputNumberTimelines().specifyValue(
        Arrays.asList(inputTimeline1, inputTimeline2,
          inputTimeline3));

    /*
     * Do not strictly check that the resultant timeline is exactly
     * as expected - instead check the resultant timeline's value
     * on particular dates.
     *
     * It is possible that the timeline has incorrect values on
     * other dates, but depending on the purpose of your test, you
     * may wish to trade strictness for improved readability.
     */

    final Timeline<? extends Number> resultantTimeline =
        totalizer.totalTimeline().getValue();
    CREOLETestHelper.assertEquals(6.1,
        resultantTimeline.valueOn(Date.fromISO8601("20010101")));
    CREOLETestHelper.assertEquals(6.6,
        resultantTimeline.valueOn(Date.fromISO8601("20130101")));
  }
}
```

Timeline Properties

Each CER timeline includes some important properties that you must know before working with Timelines in your CER rule sets, rule tests, and any code that is a client of CER.

CER timelines include the following properties:

- Each CER timeline is immutable (like all data types used in CER).
- Each reference to timeline is parameterized with the type of value held in the Timeline, which can be primitives such as String, Date, Number, Boolean etc. or an arbitrarily complex type such as a Rule Class

or another parameterized type such as List. As for other parameterized types in CER, the parameter itself should be an immutable object.

- Each CER timeline extends infinitely far into the past and infinitely far into the future. In other words, each CER timeline has a value on *any* date, no matter how far in the past or future that date might be.

  Note: Each timeline covers an infinite amount of time, but can only contain a finite number of dates on which its value changes.

- When a CER timeline is created, the timeline is divided into a collection of *intervals*, with each interval holding a constant value for a period of time within the timeline. Contiguous intervals *always* have different values, otherwise they would have been merged into a single interval. Each CER timeline extends infinitely far into the past and infinitely far into the future;

  Note: Same or different values are detected by the semantic of the Java `Object.equals(...)`. All types that are used as a parameterized type to Timeline must have sensible implementations of `Object.equals(...)` and `Object.hashCode()`.

The properties have a number of consequences.

- It is not possible to have a "gap" in the middle of a timeline - all intervals in a timeline are contiguous.
- It is not possible to have a timeline which starts on a particular date; in certain circumstances a sensible default will need to be chosen, for example if you have a Timeline<Number> for the income arising from an Employment, then that income should be 0 on all dates before the Employment start date.
- It is not possible to have a timeline which ends on a particular date - the last value in the timeline applies "until further notice", that is, arbitrarily far into the future. In certain circumstances a sensible default must be chosen. For example, if you have a Timeline<Number> for the income arising from an Employment, then if there is a known end date for that Employment, then the income should be 0 on all dates after the Employment has ended. Otherwise, if there is no known end date for that Employment, then the latest income should apply until further notice.
- Any attempt to create a timeline that does not have a value for any date will fail. In particular, each timeline must have a value which applies from the *start of time*, signified by a start date of `null`.
- Each timeline can contain a finite number of value changes. This presents a limitation for timelines that represent values that change an arbitrary number of times. For example, you might have a Timeline<Number> to represent a person's age, with the value 0 up until the person's first birthday, the value 1 up until the person's second birthday and so on. For Persons who are still alive it is not possible to predict how many more birthdays they will have, and so a practical limit (for example, 200) must be imposed. In practice, this limitation should not present any difficulties.

Example of Timeline Intervals

In the example of Joe, Mary and James's circumstances, we saw that Mary was not a lone parent of a minor before she married Joe, and that when she married Joe she was still not a lone parent of a minor, but for different reasons.

Under the hood, when Mary's `isLoneParentOfMinorTimeline` value is being calculated, the input timelines that are used are Mary's `isMarriedTimeline` and her `hasMinorDependentsTimeline`.

CER identifies each date on which the input timelines change, for each of those dates calculates the resultant value (on that date) for whether Mary is a lone parent of a minor on that date, as follows:

- Dates on which Mary's `isMarriedTimeline` changes:
  - 1st January 2001; and
  - 1st May 2004.
- Dates on which Mary's `hasMinorDependentsTimeline` changes:
  - 1st January 2001; and
  - 1st June 2006.
- Therefore dates on which one or more inputs changed are:
  - 1st January 2001 (both of Mary's input timelines happen to change on this date); and

- 1st May 2004 (only Mary's `isMarriedTimeline` changes on this date); and
- 1st June 2006 (only Mary's `hasMinorDependentsTimeline` changes on this date).

So, for each of these dates, calculate the required value for `isLoneParentOfMinorTimeline`, using primitive Boolean/truth table logic:

Table 1. Calculation of interval values for Mary's `isLoneParentOfMinorTimeline` value

Date on which one or more input timelines changes value	Value of `isMarriedTimeline` on this date	Value of `hasMinorDependentsTimeline` on this date	Required value of `isLoneParentOfMinorTimeline` on this date
start of time (this date is always included)	FALSE	FALSE	FALSE
1st January 2001	TRUE	TRUE	FALSE
1st May 2004	FALSE	TRUE	TRUE
1st June 2006	FALSE	FALSE	FALSE

Finally, a timeline is constructed with the required values for `isLoneParentOfMinorTimeline` - at this point the construction of the timeline recognises that the value for start-of-time (FALSE) and 1st January 2001 (FALSE) are identical, and these intervals are merged into a single interval which stretches from the start of time up to (but not including) 1st May 2004 (when the value becomes TRUE).

Note: The resultant timeline has value changes on 1st May 2004 and 1st June 2006 only.

The timeline intentionally does *not* hold any record that 1st January 2001 was used during is construction, as the timeline's value did not change on that date - that date holds no relevance at all for the resultant timeline.

Triggering Recalculation When Data Changes

CER perform recalculations directly by using the Dependency Manager. It is recommended that you use the Dependency Manager instead of the CER recalculation strategies.

For more information, see .

Recalculating CER Results with the Dependency Manager

CER and its clients, such as the Eligibility and Entitlement Engine and Advisor, integrate tightly with the Dependency Manager to support the recalculation of CER results when inputs that are used in CER calculations change.

Dependency Manager Concepts

An overview of the key Dependency Manager concepts. The Dependency Manager stores and manages dependencies between input data items that are called precedents and output data items that are called dependents.

When the value of one data item is derived from the values of one or more other data items, the derived value depends on the values used to derive it. If one or more of the values depended on subsequently change, then the derived data item must be recalculated to obtain its new value.

The following is a list of the key Dependency Manager concepts.

Dependent
A derived data item whose value is calculated from other data items (precedents).

Precedent
    A data item whose value may be used to calculate derived data items (dependents).

Dependency
    A record of the fact that the value of a particular dependent depends on the value of a particular precedent.

Dependency Change Item
    A record of the fact that the value of a particular precedent has changed in some way.

Precedent Change Item
    A record of the fact that the value of a particular precedent has changed in some way.

Precedent Change Set
    A set of precedent changes items, grouped together for processing; used to identify potentially affected dependents which require recalculation.

Dependent Recalculation
    The recalculation of a dependent which is potentially affected by one or more of the changes to precedents in a precedent change set.

Identification
    Each Dependent and Precedent must have a type and identifier.

The concepts are best explained with an example. Let's say that a claimant's entitlement to benefit is calculated from data such as:

- The claimant's personal details
- Evidence gathered for the claimant's case
- Benefit rates and income thresholds

Joe, a claimant, has two cases (123 and 124) and Mary, another claimant, has one case (125). There are cases and personal details for other claimants too, and also some allowance rates used for other calculations.

In this example, the calculated entitlement for each case is a *dependent* and the personal details, evidence and rates/thresholds are *precedents*.

We can draw a *sparse matrix* which shows the dependencies between the dependents and the precedents (an "X" signifies the presence of a dependency):

Table 2. Example Dependency Matrix

Precedent	Case 123's Entitlement	Case 124's Entitlement	Case 125's Entitlement	Case 126's Entitlement
Joe's Personal Details	X	X		
Mary's Personal Details			X	
Frank's Personal Details				
Case 123's Evidence	X			
Case 124's Evidence		X		
Case 125's Evidence			X	
Case 126's Evidence				X

Table 2. Example Dependency Matrix (continued)				
Precedent	**Case 123's Entitlement**	**Case 124's Entitlement**	**Case 125's Entitlement**	**Case 126's Entitlement**
Benefit Rates	X	X	X	X
Income Thresholds	X	X	X	X
Allowance Rates				

So, to look at some examples from the dependency matrix:

- The entitlement for case 123 depends on Joe's personal details but not on Mary's;
- Joe's personal details are used in the calculation of both his cases (123 and 124); and
- All the cases use the benefit rates and income thresholds but not the allowance rates.
- No case's entitlement depends on Frank's personal details.

Note that the matrix can be read:

- by column, to understand all the precedents upon which a particular dependent depends - these are the set of dependencies that must be maintained every time a dependent is calculated; and/or
- by row, to understand all the dependents which depend upon a particular precedent - these are the set of dependents that must be recalculated every time the value of the precedent changes.

As the number of precedents and dependents in the system grow, the dependency matrix becomes very large. Because the matrix is only sparsely populated (i.e. each dependent depends on only a small fraction of the available precedents), the data in the matrix is stored only for dependencies which are present, as follows:

Table 3. Example Dependency Storage		
Dependent		**Precedent**
Case 123's Entitlement	depends on	Joe's Personal Details
Case 123's Entitlement	depends on	Case 123's Evidence
Case 123's Entitlement	depends on	Benefit Rates
Case 123's Entitlement	depends on	Income Thresholds
Case 124's Entitlement	depends on	Joe's Personal Details
Case 124's Entitlement	depends on	Case 124's Evidence
Case 124's Entitlement	depends on	Benefit Rates
Case 124's Entitlement	depends on	Income Thresholds
Case 125's Entitlement	depends on	Mary's Personal Details
Case 125's Entitlement	depends on	Case 125's Evidence
Case 125's Entitlement	depends on	Benefit Rates
Case 125's Entitlement	depends on	Income Thresholds
Case 126's Entitlement	depends on	Case 126's Evidence
Case 126's Entitlement	depends on	Benefit Rates
Case 126's Entitlement	depends on	Income Thresholds

(Note that the table above is shown sorted by dependent, which makes it easy to see the set of dependencies for each dependent, but it could also be shown sorted by precedent which would make it easy to see the dependents potentially affected by a change in that precedent's value.)

Let's say that Joe's personal details change. Because dependencies are recorded against Joe's personal details, the Dependency Manager is able to identify that cases 123 and 124 require recalculation. When the cases are recalculated, their entitlement values change (due to the change in Joe's personal details); but note that in typical situations, the dependencies themselves do not change - prior to the recalculation, case 123 depended on Joe's person details, the evidence stored against the case, the benefit rates and income thresholds, and the same is true after the recalculation.

It is possible for more than one precedent value to change simultaneously, for example if the agency chooses to alter both its benefit rates and its income thresholds then all cases must be recalculated. Naively, each case would be identified twice (once from the change to benefit rates and once again for the change to income thresholds); however, the Dependency Manager supports the grouping of these two precedent changes into one Precedent Change Set. When the Dependency Manager processes the Precedent Change Set it automatically filters out any duplicate dependents identified so that the minimum work necessary is performed to recalculate the dependents.

Dependency Manager Database Considerations

Each dependent or precedent is represented on the database as a type plus an identifier.

Type is a code table value. Examples of type are:

- ENTITY_ROW - 'Entity Row'
- STORED_AV - 'Stored Attribute Value'
- ACTEVID - 'Active Evidence'
- RULESETDEF - 'CREOLE Rule Set Definitions'

Identifier is a string of up to 1000 characters and the format of the identifier depends on the type of the dependent or precedent. Examples of identifier corresponding to each of the above four types are as follows:

- "ConcernRoleRelationship.concernRoleRelationshipID=501"

  Identifies a row in the ConcernRoleRelationship table.
- "2487501571575775232"

  Identifies a stored attribute value.
- "1622810443120640000"

  Identifies an integrated case.
- "IncomeAssistanceProductComparisonDecisionDetailsRuleSet"

  Identifies a rule set.

Since the identifier field is so long (1000 characters) and used in some high frequency queries, it is backed by an integer hashcode. This is a number which is derived from the string value of the identifier and enables the SQL queries used by the Dependency Manager which reference the identifier to run more efficiently by also referencing the corresponding hashcode. This also reduces database storage overhead because considerably less space is required to index the hashcode fields than would be required to index the identifier fields.

These hashcode fields are used on two Dependency Manager tables - Dependency and PrecedentChangeItem - and are populated automatically by the infrastructure as these records are written.

Dependency Manager Functions

The Dependency Manager performs a number of main functions that are described in this section.

The following are the main functions of the Dependent Manager.

- Stores dependency records identified by a client. For example, by the Eligibility and Entitlement Engine when calculating an initial assessment determination.
- Captures changes in precedent values which potentially affect the values of dependents.
- Identifies the dependents which are potentially affected by items in a precedent change set.
- Controls the recalculation of these identified dependents.

Storage of Dependency Records

Use this information to learn about how the Dependency Manager stores dependency records on the database. The Dependency Manager is responsible for the creation of new dependency records on the database, and for the removal of existing dependency records which are no longer required.

Note: Each dependency record does not contain any modifiable information, and the Dependency Manager never *modifies* any existing dependency records - it only ever creates new records or removes existing records.

Every time that a client of the Dependency Manager calculates the value of a dependent, the client is responsible for identifying the precedents used in that calculation, and for passing that dependent and its set of precedents to the Dependency Manager. The Dependency Manager uses that dependent to retrieve its existing set of stored dependencies (if any) from the database, and creates or removes dependency records in line with the new set of precedents identified by the client.

Typically, the first time that the Dependency Manager is invoked for a dependent, the Dependency Manager will create several new rows on the database to store the dependencies on the precedents identified.

However, on subsequent invocations of the Dependency Manager for the same dependent, it is very common for the Dependency Manager to find that the new set of required dependencies passed in exactly matches those already stored on the database, and so under these circumstances there are no database writes for the Dependency Manager to perform. Occasionally the Dependency Manager will find that a small number of new dependency rows are required, and/or a small number of existing dependency rows are now extraneous and must be removed; and under these circumstances the Dependency Manager performs a small number of database writes to bring the stored rows up-to-date with the required dependencies, leaving the bulk of the dependency records unchanged for the dependent.

Clients of the Dependency Manager may identify that dependency records are no longer required for a dependent, and can instruct the Dependency Manager to remove all dependency records for that dependent.

Note: When writing CER rules, it is possible that a rule set could cause some dependencies to be generated which refer to a precedent with an ID of zero. The Dependency Manager may prevent these zero ID dependencies from being written to the database during rule execution, with the following guidance:

- If the precedent identified is an entity row which resolves to an ID of zero, then the dependency *will not be* stored on the database. For example: `Person.concernRoleID=0`
- If the precedent identified is from a 'readall/match' search, and the search criteria resolves to a zero value on a *primary key attribute*, then the dependency *will not be* stored on the database. For example: `MyRuleSet.ConcernRoleRelationship.concernRoleID=0`
- If the precedent identified is from a 'readall/match' search, and the search criteria resolves to a zero value on *non-primary key attribute*, then the dependency *will be* stored on the database. For example: `MyRuleSet.ConcernRole.residencyAbroadInd=0`

Example of Storage of Dependency Records
The following example describes how the Dependency Manager stores new dependency records when entitlement for a case is performed for the first time.

When an entitlement for a case is performed for the first time, the Dependency Manager stores new dependency records to show that the case entitlement depends on the claimant's personal details, the evidence recorded against the case, rates, and so on.

If the case is subsequently recalculated (either automatically by the Dependency Manager in response to a change in personal details, say, or manually requested by a user), then after recalculation the Dependency Manager compares the dependencies identified during the calculation with those already on the database and finds no differences.

If a new member of the household is added to the case, then when entitlement is recalculated a new dependency will be identified - namely that the case's entitlement now also depends on the new household member's personal details, in addition to the existing dependencies already stored against the case. The Dependency Manager creates a new dependency record on the database to store the additional dependency.

If the new household member is later removed, then when entitlement is recalculated there will be no dependency on the now-removed household member's personal details. The Dependency Manager identifies that the stored dependency on that household member's personal details is now extraneous and removes it from the database, leaving the other dependency records (on the claimant's personal details, the evidence recorded against the case, rates) still intact.

When the case is eventually closed, it will no longer require support for recalculations and so the dependency records are no longer required.

Note: Assuming a reassessment strategy of "Do not reassess closed cases". See the *Inside Cúram Eligibility and Entitlement Using Cúram Express Rules* guide.

For good housekeeping, the Dependency Manager is invoked to remove all dependency records for the case's entitlement. If the case is subsequently reopened then its entitlement can be recalculated and the Dependency Manager recreates all the required dependency records.

No Understanding of Dependents or Precedents
The Dependency Manager intentionally does *not* maintain any records of all known dependents or precedents in the system.

To do so would:

- Duplicate data held elsewhere in the system.
- Become a bottleneck during processing, potentially leading to concurrency issues when the system becomes aware of new data used as precedents in calculations.

Rather, the Dependency Manager only records information about dependencies. Each dependency is merely a link between a particular dependent and a particular precedent. If a particular dependent has no precedents, or vice versa, then there will simply be no dependency records stored for it.

The Dependency Manager does not "understand" the dependent and precedent information stored in a dependency record; instead, each type of dependent and each type of precedent has a "handler" registered with the Dependency Manager, and the Dependency Manager calls upon these handlers to perform business-specific processing appropriate to the type, e.g. to decode a precedent or dependent into a human-readable description, or to recalculate a dependent when required.

Dependency Storage is Optional
Use of the Dependency Manager to store dependency records is optional. Clients of the Dependency Manager can choose whether or not dependency records are required. That is, whether or not the client requires the Dependency Manager's ability to automatically identify and recalculate dependents.

For example, the Eligibility and Entitlement Engine uses the Dependency Manager to store dependency records for Case Assessment Determinations, that is, determinations which typically lead to financial

payments and/or bills. The Eligibility and Entitlement Engine requires that the Dependency Manager notify it when a case must be reassessed, which is why the dependency records must be stored.

By contrast, the Eligibility and Entitlement Engine also contains a feature for a case worker to manually check the eligibility and entitlement of a case, based off in-edit evidence. These manual eligibility/entitlement calculations use the same calculation methods but do not require any dependency storage, as the system is never required to recalculate such determinations. Rather, manual eligibility/entitlement calculations are always triggered by an explicit request from a case worker.

Granularity of Dependencies

The Dependency Manager does not know or care about the meanings of the dependencies that it stores between dependents and precedents . It is the responsibility of the clients of the Dependency Manager to attach meanings to these and to store dependencies at an appropriate granularity.

Note that the precedent data items that are used in the earlier example are deliberately vague. The term "personal details" would in all likelihood cover a great many individual data fields such as dates of birth/death, demographics, and so on.

The choice of granularity involves finding an acceptable trade-off between the following two extremes:

Very fine granularity
> Very accurate dependencies are stored between dependents and individual data fields, enabling an extremely tight identification of dependents affected by precedent changes, but at the cost of very high numbers of dependency records being stored

Very course granularity
> Very broad dependencies are stored between dependents and groupings of large numbers of individual data fields into one "data item", leading to low numbers of dependency records being stored, but at risk of spurious recalculations being requested, that is, recalculations which turn out not to be needed because the calculation is not affected by the particular data field which changed.

It is the responsibility of designers of clients of the Dependency Manager to consider these trade-offs and make sensible choices about the level at which to store dependency information in the Dependency Manager.

For example, let's say that the system records these personal details about a claimant, in a realistic system there might be many more fields that are considered personal details:

- Date of birth (used in entitlement calculations)

- Number of children (used in entitlement calculations)

- Mother's birth surname (the answer to a security question, used only to confirm the claimant's identity - not used in entitlement calculations)

A very fine-grained set of dependencies would show that a case's entitlement depends on the date of birth and number of children, but not on the mother's birth surname (as it was not accessed during calculations):

Table 4. Example Fine-Grained Dependency Matrix	
Precedent	**Case 127's Entitlement**
Frank's date of birth	X
Frank's number of children	X
Frank's mother's birth surname	

This fine-grained dependency storage could end up requiring many rows to be stored; but note that only changes to the date of birth and/or number of children will trigger a recalculation of the case's entitlement - if a typo is corrected in the mother's birth surname only, then no case entitlement recalculation will be triggered.

By contrast, a very coarse-grained set of dependencies would show a much simpler record that the case's entitlement depends on the overall personal details:

Table 5. Example Coarse-Grained Dependency Matrix	
Precedent	**Case 127's Entitlement**
Frank's personal details	X

This coarse-grained dependency storage stores fewer dependency records but if a typo is corrected in the mother's birth surname then the overall personal details have changed and a recalculation of the case's entitlement will be triggered, even though the recalculation will show that the calculation result has not changed.

Capture of Precedent Change Items

Clients must notify the Dependency Manager whenever the value of a precedent has changed. The Dependency Manager accumulates these changes into a Precedent Change Set for later processing. The Dependency Manager supports a queue for deferred processing (default) and a queue for batch processing for dealing with precedent changes.

The most prevalent example of notifications when the value of a precedent has changed is the Eligibility and Entitlement Engine, which contains rule object propagators. These propagators are responsible for listening to changes in entities and evidence, and for notifying these changes to the Dependency Manager.

Queue for Deferred Processing
This is the default mode for handling precedent changes. If any precedent change items are identified during the scope of a database transaction, then the Dependency Manager does the following.

- Creates a single new precedent change set on the database.
- Adds all the precedent change items identified during the transaction to this new precedent change set.
- Queues a request for a deferred process to process this new precedent change set.

Queue for Batch Processing
This mode is used only by clients that identify changes in precedent values that are likely to lead to a large number of dependents being recalculated. The Dependency Manager maintains a special system-wide batch precedent change set which accumulates precedent changes from across potentially many different transactions.

If any precedent change items are identified during the scope of a database transaction, the Dependency Manager:

- Retrieves the special system-wide batch precedent change set.
- Adds all the precedent change items identified during the transaction to this batch precedent change set.
- Writes an informational message to the application logs to prompt an administrator to schedule batch processing to process this batch precedent change set.

Example of Precedent Change Processing
An example of precedent change processing. Use this information to learn about how the Dependency Manager handles precedent changes.

Joe's personal details are updated on the system. The Entity Rule Object Propagator notifies the Dependency Manager of the change and the Dependency Manager writes the precedent change to a new precedent change set and enqueues a deferred process. The deferred processing identifies that Joe's two cases are potentially affected and reassesses them.

Later an administrator publishes some changes to CER rule sets. The Dependency Manager records these changes to CER rule sets by adding a precedent change item record to the system-wide batch precedent change set. The administrator also publishes some changes to rates, and the Dependency Manager records these changes to rates by adding another precedent change item record to the system-wide batch precedent change set. The administrator arranges to run the Dependency Manager batch suite to identify and reassess the affected cases.

Lifecycle of a Precedent Change Set

An overview of the precedent change set lifecycle. The transitions between each state occur differently depending on the mode in which precedent changes were captured.

Each precedent change set goes through the following a simple lifecycle:

- **Open**

  The state of a precedent change set when it is initially created. In this state new precedent change items can be added to the precedent change set.

- **Submitted**

  The precedent change set has been submitted into dependent-identification processing. No more precedent change items can be added to the precedent change set.

- **Complete**

  The recalculation of all dependents affected by the precedent changes has been completed. The precedent change set is kept for historical purposes only.

The transitions between each state occur differently depending on the mode in which precedent changes were captured:

- Queue for Deferred Processing:

  - the transaction that captures the precedent changes *opens* a new precedent change set and *submits* it by requesting a deferred process; and

  - the deferred process accepts the *submitted* precedent change set, identifies and recalculates affected dependents, and *completes* the precedent change set.

- Queue for Batch Processing:

  - the transaction that captures the precedent changes writes them to the currently- *open* batch precedent change set;

  - the suite of Dependency Manager batch processes performs the following steps:

    - retrieves the currently- *open* batch precedent change set and *submits* it into the next step in the batch process, and creates a new *open* batch precedent change set to capture any further precedent changes identified;

      Note: In a running system, this is how a new batch precedent change set is created, that is, by its predecessor being submitted. The *initial* open batch precedent change set is supplied by a DMX file included in the application.

    - performs streamed batch processing to identify and recalculate the dependents affected by the changes in the now- *submitted* batch precedent change set; and

    - *completes* the batch precedent change set.

Identification of Potentially Affected Dependents

Once the Dependency Manager has captured one or more precedent change items and grouped them into a precedent change set, then the Dependency Manager can identify which dependents are potentially affected by one or more of the precedent change items.

It does this by examining the stored dependency records for each precedent in the precedent change set. It is at this point that the Dependency Manager filters out any dependents which are identified more than once.

This identification of the potentially affected dependents occurs either in deferred processing or in batch processing, depending on the mode in effect when the precedent change items were captured (see).

For example, if a single database transaction writes changes to several database rows, then the deferred processing step will identify each affected case only once, even though each changed database row on its own affects a particular case.

Similarly, if the batch precedent change set contains changes to both CER rule sets and rates, then the batch processing step will again identify each affected case only once, even though the CER rule set change on its own affects a particular case, as does the rate change on its own, too.

Recalculation of Identified Dependents

Once the Dependency Manager has identified all the dependents which are potentially affected by precedent changes, it requests that each of those dependents be recalculated. The recalculation of a dependent occurs either in deferred processing or in batch processing, depending on the mode in effect when the precedent change items were captured.

The Dependency Manager does not understand what each dependent represents, so the appropriate recalculation is achieved by the Dependency Manager looking up a registered handler for each dependent and delegating the responsibility for the recalculation to the handler.

For example, the registered dependent handler for case assessment determinations understands that the appropriate recalculation for a case is to reassess the case.

Once this step has completed, the system is now up-to-date with respect to the precedent changes which were captured.

Note: Because each dependent handler may be called from online, deferred, or batch transactions, it is imperative that each dependent handler make *no* assumptions on the transaction type in effect.

Dependency Manager Deferred Processing

The Dependency Manager contains deferred processing to identify potentially affected dependents and recalculate them. You can set a system limit for deferred processes. This information also covers error handling in deferred processing and how to detect transfer of precedent change items from deferred processing to batch if this occurs.

The deferred processing initially precalculates the total amount of potentially affected dependents there are, an how many recalculations it must perform and make one of two decisions:

- If this calculation is below the system limit, then the precedent change set recalculations are done immediately by the deferred process.
- If this calculation exceeds the system limit, then the precedent change set calculations are moved to the Dependency Manager's batch processing. The precedent change set is marked with a status of 'Deferred To Batch'.

Setting the System Limit for Deferred Processing

You can set the system limit for deferred processing through the application property `curam.dependency.deferred.processing.limit`.

The default for this is set to 50, however it should be noted that the true figure for any system depends on a multitude of factors such as available memory. It is recommended that you set a value based on system testing to find a suitable value that can process most, if not all deferred processes normally, but will move any problematic ones to batch processing.

Error Handling in the Deferred Processing

If an error occurs during running of the Dependency Manager Deferred Process, after the normal number of retries, the system will move the work to batch processing instead. This protects against failures due to transaction timeouts and ensures that every avenue for processing the calculations has been tried.

Additionally, the status of the precedent change set in this case will be changed to be 'Deferred To Batch' to indicate that the deferred process was in error and a notification will be raised for the originator of the deferred process.

Detecting transfer from Deferred Processing to Batch

Under some conditions, the Dependency Manager might move the processing of precedent change items from deferred processing to batch. You can detect when this happens by writing Java code to implement an optional hook.

To implement this hook, you must add the following two customizations:

1. Register your implementation of the interface in your Guice module.
2. Provide the Java implementation of the interface to take an appropriate action.

Important: This hook method executes as part of the transaction which transfers the precedent change items from deferred processing to batch processing, so if an error occurs in this method it will result in that transaction being rolled back and the precedent change items will not be batch processed.

1. Register the custom implementation in the Guice module:

   In this example, the `PrecedentChangeSetToBatchNotification` interface is implemented by class `MyPrecedentChangeSetToBatchNotificationImpl`:

   ```
   import curam.dependency.impl.PrecedentChangeSetToBatchNotification;

   /**
    * In this example, the PrecedentChangeSetToBatchNotification
    * interface is implemented by class
    * PrecedentChangeSetToBatchNotificationImpl
    */
   public class Module extends AbstractModule {

     @Override
     protected void configure() {
       binder().bind(PrecedentChangeSetToBatchNotification.class).to(
         MyPrecedentChangeSetToBatchNotificationImpl.class);
       }
     }
   ```

2. Provide the Java implementation of the interface:

   ```
   import curam.util.exception.AppException;
   import curam.util.exception.InformationalException;

   public class MyPrecedentChangeSetToBatchNotificationImpl
   implements PrecedentChangeSetToBatchNotification {

     /**
      * Raise a notification when a precedent change set gets deferred to batch.
      *
      * NB Do not allow this method to throw exceptions, as this would interfere
      * with the dependency processing.
      *
      * @param status Indicates the status of this call to the hook. A value 0
      * indicates that the deferral was due to the number of precedents being too
      * large to process in DP, value 1 indicates that an error occurred when
      * processing the DP.
      * @param instDataID The identifier for the DP instance data record.
      * @param precedentChangeSetID The identifier for the precedent change set.
      * @param dependentCount The number of dependents affected by the change set.
      * @param userID The user who caused the deferred process to be queued.
      * @param caseID Optional - the identifier of the case whose changes caused
      * the DP to be queued, if one case caused the DP. Note that not all
      * precedent change sets are created as a result actions on a single
      * case, so if the precedent change set cannot be linked to a single case
      * then this parameter will be left blank.
      *
      * @throws AppException Standard signature, note that throwing this
      * will prevent the precedent change set from being deferred to batch.
      * @throws InformationalException Standard signature, note that throwing this
      * will prevent the precedent change set from being deferred to batch.
      */
     @Override
     public void deferredToBatch(final int status, final long instDataID,
       final long precedentChangeSetID,
   ```

```
        final long dependentCount, String userID, long caseID)
        throws AppException, InformationalException {

        // Provide your code here to get executed whenever
    }
}
```

Tip:

The `caseID` parameter is derived from the `curam.core.intf.CachedCaseHeader` class which caches accesses to a single case header record for efficiency and underpins the various case screens across the product. If a case, or any of its precedents are modified without using `CachedCaseHeader` then `caseID` parameter will be zero. For example, if you have a custom screen which generates one or more precedent changes for a case without using the `CachedCaseHeader` class, then the caseID will not be populated.

If the `caseID` parameter is not populated, the following SQL query can be used to identify which items have been affected by a precedent change set using its precedent change set identifier:

```
 SELECT DISTINCT Dependency.dependentID, Dependency.dependentType
 FROM Dependency
   INNER JOIN PrecedentChangeItem
   ON   Dependency.precedentType=PrecedentChangeItem.precedentType AND
 Dependency.precedentID=PrecedentChangeItem.precedentID
   WHERE precedentChangeItem.precedentChangeSetID = <precedent-change-set-identifier> ;
```

The following SQL querity can be used to identify which CER case determinations would be recalculated as a result of a precedent change set:

```
 SELECT DISTINCT Dependency.dependentID
 FROM Dependency
   INNER JOIN PrecedentChangeItem
   ON   Dependency.precedentType=PrecedentChangeItem.precedentType AND
 Dependency.precedentID=PrecedentChangeItem.precedentID
   WHERE dependentType = 'CADETERRES'
     AND precedentChangeItem.precedentChangeSetID = <precedent-change-set-identifier> ;
```

Dependency Manager Batch Processing

The Dependency Manager includes batch processing for calculations. An overview of the batch processing is provided including submitting and completing precedent change sets.

The Dependency Manager maintains control records on the database to point to the following batch precedent change sets:

- The currently-open batch precedent change set (there will always be exactly one batch precedent change set which is open for accepting new precedent change items).
- The batch precedent change set which is currently being processed by the Dependency Manager batch suite (if any - only populated during batch processing; most of the time there is no batch precedent change set in this state).

These control records are fundamental to the behavior of the Dependency Manager batch suite.

Whenever there are precedent changes in "queue for batch processing" mode, then the application logs will show a message notifying the administrator that a run of the Dependency Manager batch suite is required. The user who made the changes which were queued for batch processing will also receive an on-screen informational message advising that a run of the Dependency Manager batch suite is required.

The Dependency Manager batch suite includes these separate batch processes:

- **Submit Precedent Change Set**

  The start point for the batch suite. A lightweight single-stream process that submits the currently-open batch precedent change set.

- **Perform Batch Recalculations From Precedent Change Set**

The heavyweight multiple-stream process that identifies the dependents which are potentially affected by the changes in the submitted precedent change set, and recalculates them. The time taken to run this process will vary according to how many dependent recalculations are required, and may be considerable.

- **Complete Precedent Change Set**

  The end point for the batch suite. A lightweight single-stream process that completes the currently-submitted batch precedent change set.

Related concepts
Dependency Manager Batch Tooling

Submitting a Precedent Change Set

This batch process is the starting point for the batch suite. It is a lightweight single-stream process that submits the currently-open batch precedent change set, and creates a new open batch precedent change set that is used to capture any subsequent precedent changes identified and queued for batch processing.

To run this batch process, execute the following command (on one line):

```
build runbatch -Dbatch.program=
curam.dependency.intf.SubmitPrecedentChangeSet.process
-Dbatch.username=SYSTEM
```

If this batch process completes successfully, it outputs a simple message confirming that the open batch precedent change set has been submitted.

Tip: A common error is to attempt to run this batch process when another batch precedent change set is still in the submitted state.

This process will output a simple error message if there is another such batch precedent change set which has not yet been completely processed by the batch suite.

As a convenience, this batch process also outputs a list of the dependent types which are registered with the Dependency Manager - this list of dependent types is important when running the next step ("Performing Batch Recalculations From Precedent Change Set" on page 76).

The dependent types included with the application are:

- **Case Assessment Determination Result**

  The calculation of an assessment determination for a product delivery case.

  See the `Inside Cúram Eligibility and Entitlement Using Cúram Express Rules` guide.

- **Advice Context**

  The calculation of advice.

  See the `Cúram Advisor Configuration Guide`.

- **Stored Attribute Value**

  The calculation of an attribute stored on CER's database tables.

  See "Storing Dependencies for Attribute Values" on page 84.

Performing Batch Recalculations From Precedent Change Set

This batch process is the heavyweight multiple-stream process that identifies the dependents that are potentially affected by the changes in the submitted precedent change set, and recalculates them. The time that is taken to run this process varies according to how many dependent recalculations are required, and can be considerable.

The Perform Batch Recalculations From Precedent Change Set step must be ran multiple times - once for each dependent type that is registered with the Dependency Manager (see the output that is produced by the previous step "Submitting a Precedent Change Set" on page 76). You are free to choose the most appropriate order in which to process the dependent types; for example, it is likely to be more critical to your business to reassess case determinations (see the `Inside Cúram Eligibility and Entitlement Using Cúram Express Rules` guide) than it is to identify out-of-date advice (see the `Advisor Configuration Guide`). You are also free to spread the processing for different dependent types over several days, but note that further precedent change items queued for batch cannot be processed until the currently-submitted precedent change set has completed the full Dependency Manager batch suite.

The Perform Batch Recalculations From Precedent Change Set step uses Cúram's Batch Streaming Architecture (see the *Cúram Batch Performance Mechanisms Guide*) and as such the processing is divided into these phases:

- **Identify Chunks**

  A phase, which must be run as a single process, that identifies the dependents (of a given dependent type) potentially affected by changes in the submitted batch precedent change set. The IDs of the identified dependents are written to "chunks" for processing by the next phase.

- **Process Chunks**

  A phase, amenable to concurrent execution by multiple processes, that takes a chunk of identified dependents and recalculates each dependent.

To run this batch process for a particular dependent type, issue the following command on one line:

```
build runbatch -Dbatch.program=
curam.dependency.intf.PerformBatchRecalculationsFromPrecedentChangeSet.process
-Dbatch.username=SYSTEM
 -Dbatch.parameters="dependentType= code-for-dependent-type "
```

By default, the single process performs both phases; however, you can run extra "Stream" processes concurrently on other computers to perform the second phase in parallel (For more information about parallel processing and the environment variables that govern the parallel processing behavior of this Perform Batch Recalculations From Precedent Change Set process, see the *Cúram Batch Performance Mechanisms Guide*. To run a "stream" process for a particular dependent type, run the following command on one line:

```
build runbatch -Dbatch.program=
curam.dependency.intf.PerformBatchRecalculationsFromPrecedentChangeSetStream.process
-Dbatch.username=SYSTEM
 -Dbatch.parameters="dependentType= code-for-dependent-type "
```

The batch process fails to start with a unrecoverable error if any of the following occur:

- the dependentType is not supplied, or is supplied but is not the code for any dependent type that is registered with the Dependency Manager; or
- there is no batch precedent change set in the "submitted" state (that is, the Submit Precedent Change Set process has not yet run since the last run of Complete Precedent Change Set).

Otherwise, the batch process starts, and attempt to identify and recalculate the affected dependents. The result of attempting to recalculate a particular dependent is either:

- **Success**

  The dependent was found and recalculated correctly and processing continues normally.

- **Not found**

  The dependent was not found and so could not be processed. This situation can occur if a client of the Dependency Manager decides that a dependent should no longer exist, but neglects to request the Dependency Manager to remove dependency records for that dependent. Under these circumstances, the Dependency Manager automatically removes the extraneous dependency records and writes a warning to application log/batch stream output.

- **Error**

    An exception was thrown during the recalculation of the dependent. For example, if a CER calculation encountered a "division by zero" problem. The thrown exception is written to the batch stream output and recovery processing is handled by the Cúram's Batch Streaming Architecture's "skip" processing.

When the Perform Batch Recalculations From Precedent Change Set process finishes, a comprehensive report is written with details of how many dependents were processed successfully, vs. not found, vs. encountered errors. If any errors were encountered, examine the output logs from your batch streams to obtain details of the errors.

The report for the Perform Batch Recalculations From Precedent Change Set process will provide the results per product if configured to do so. The application property `curam.batch.performbatchrecalculationsfromprecedentchangeset.productlevelreporting` is used to dictate whether or not the batch reporting will display product-level results. The default value is 'NO'. If this value is set to 'YES', the results of the batch program will be displayed per product. This applies where the batch program Dependent Type Mode is 'Case Assessment Determination Result', which reassesses cases on the system.

Important: If you set the standard Cúram log level to "verbose" or higher, then before recalculating each dependent the Dependency Manager outputs:

- a human-readable description of the dependent; and
- the subset of precedent changes (from the precedent change set) that caused the dependent to be identified.

This log level is suitable for development environments only. Verbose logging can adversely affect performance and scalability in a production system.

This output can be helpful in understanding why a particular dependent was identified as requiring a recalculation.

Note: If you accidentally run this batch process more than once for the same dependent type, then recalculations of the dependents occur, but the recalculation finds that the dependent is already up-to-date.

As such, this kind of accidental extra run for a dependent type does not harm the system but it uses up valuable processing time.

Take careful note of the list of dependent types, and track which dependent types you process and which dependent types remain to be processed.

Important: To determine the cases which will be affected by the batch suite, the following SQL can be used, where the 'DependentType' SQL value is the dependentType value that will be passed to the 'Perform Batch Recalculations From Precedent Change Set' batch process:

If Dependency Hash Code support is disabled (i.e. if property 'curam.creole.dependency.hashcodes.disabled' is set to True or Yes) then you should use this query:

```
SELECT DISTINCT CaseID
FROM CaseHeader, PrecedentChangeItem, PrecedentChangeSet, Dependency
WHERE Dependency.DependentID = CaseHeader.CaseID
  AND Dependency.PrecedentType = PrecedentChangeItem.PrecedentType
  AND Dependency.PrecedentID   = PrecedentChangeItem.PrecedentID
  AND PrecedentChangeItem.PrecedentChangeSetID = PrecedentChangeSet.
PrecedentChangeSetID
  AND PrecedentChangeSet.Status = 'OPEN'
  AND Dependency.DependentType = 'CADETERRES';
```

If Dependency Hash Code support is enabled (i.e. if property 'curam.creole.dependency.hashcodes.disabled' is NOT set to True or Yes) then you should use this query:

```
SELECT DISTINCT CaseID
FROM CaseHeader, PrecedentChangeItem, PrecedentChangeSet, Dependency
WHERE Dependency.DependentID = CaseHeader.CaseID
  AND Dependency.PrecedentType   = PrecedentChangeItem.PrecedentType
```

```
    AND Dependency.PrecedentIDHash = PrecedentChangeItem.PrecedentIDHash
    AND Dependency.PrecedentID     = PrecedentChangeItem.PrecedentID
    AND PrecedentChangeItem.PrecedentChangeSetID =
 PrecedentChangeSet.PrecedentChangeSetID
    AND PrecedentChangeSet.Status = 'OPEN'
    AND Dependency.DependentType = 'CADETERRES';
```

Completing a Precedent Change Set

This is the end point for the batch suite. It is a lightweight single-stream process that completes the currently-submitted batch precedent change set.

Important: Do not run this batch process until you are sure that the previous step ("Performing Batch Recalculations From Precedent Change Set" on page 76) has completed *for each dependent type registered with the Dependency Manager*.

To run this batch process, execute the following command (on one line):

build runbatch -Dbatch.program=

curam.dependency.intf.CompletePrecedentChangeSet.process

-Dbatch.username=SYSTEM

If this batch process completes successfully, it will output a simple message confirming that the submitted batch precedent change set has been completed.

Tip: A common error is to attempt to run this batch process before the Submit Precedent Change Set has been run.

This process will output a simple error message if there is no batch precedent change set in the "submitted" state.

After the batch precedent change set has been completed, the processing checks to see if any new precedent changes were queued for further batch processing since the batch suite began. This situation can occur if

- the recalculation of any dependent during the batch run gave rise to changes in data which is also used as precedent; and/or
- the online system has been running concurrently with the batch suite and a user has made changes which resulted in precedent change items being queued for batch processing.

The processing will output a simple message reporting that either:

- there are no further precedent change items queued for batch processing (and thus the system is up-to-date with respect to batch precedent change items); or
- further precedent change items have been queued for batch processing, and another run of the Dependency Manager batch suite is required to process these further items. In this situation, you will need to decide whether to execute the additional run immediately or wait until a later time (perhaps when even more batch precedent change items have been queued for batch).

Integration between CER and the Dependency Manager

CER integrates with the Dependency Manager in a number of important ways that are described in this section.

- CER can identify dependencies for the clients of the Dependency Manager to store.
- CER uses the Dependency Manager to store dependencies for any calculated attribute values stored on CER database tables.
- The Dependency Manager requests CER to recalculate any calculated attributes values stored on CER database tables if any of their precedents change.

CER Utility to Identify Dependencies to Store

An overview of the CER utility that helps CER clients to identify dependencies to store in the Dependency Manager.

A client of CER uses CER to perform complex calculations. Often the client of CER may also need to store dependencies in the Dependency Manager so that the Dependency Manager can notify the client whenever precedents change, and that client can then re-invoke CER to recalculate its output, taking the changes to precedent data into account.

The precedents that are identified by the utility are a mixture of:

- Precedents that are identified directly by CER.
- Precedents that are identified by the rule object converters registered with CER's Database Data Storage.

The CER utility takes in an attribute value (that CER has calculated) and returns a set of precedents for that attribute value. A client of CER can then pass its dependent and the identified precedents to the Dependency Manager to store dependency records.

When CER calculates an attribute value, it keeps in memory a full tree of logical dependencies containing:

- At its root, the calculated attribute value itself.
- At its branch nodes intermediate calculate values (typically on internal rule objects).
- At its leaf nodes, the external input data retrieved during the CER calculation.

The utility is able to parse this tree of logical dependencies in order to provide a much smaller set of precedents, typically based off the leaf nodes of the tree. Typically, the intermediate calculation results are ignored and the dependencies stored reflect that calculated attribute value ultimately depends on the external input data accessed during calculations.

Note: Any non-trivial calculation, such as those typically performed by CER, will have a number of intermediate derived values "in between" the overall dependent and its input precedents.

These intermediate values are not passed to the Dependency Manager. Instead, the Dependency Manager stores dependency records which link high-level dependents (such as a case's entitlement) directly to its low-level precedents (such as entity, evidence and rate data).

Intermediate values are not of interest when storing dependencies.

Precedents Identified Directly by CER
The utility directly identifies these types of dependencies for a calculated attribute value.

The overall calculation is the calculation of the attribute itself, or on any of the internal attribute values on which it ultimately depends (the "intermediate" calculations between the calculated attribute value and its external data inputs).

Table 6. Precedents Identified Directly by CER		
Name	**When Identified**	**Trigger for Recalculation**
Stored Attribute Value	Identifies any "input" attribute value stored on CER's database tables that was retrieved during the overall calculation of the attribute value. The precedent ID refers to the internal ID for the stored attribute's database row on CER's database tables.	If the value of the stored attribute changes, then a precedent change item for the stored attribute value will be written to a precedent change set.

Table 6. Precedents Identified Directly by CER (continued)

Name	When Identified	Trigger for Recalculation
Rule Set Group	Identifies a group of Rule Set Definitions containing any of the attributes used within the overall calculation of the attribute value. The precedent ID refers to the ID of the rule set group containing one or more attribute definitions encountered during the overall calculation.	If changes to a CER rule set group are published, then a precedent change item for the changed rule set group is written to a precedent change set.
Rule Set Definitions	Identifies each rule set containing any of the attributes used within the overall calculation of the attribute value. The precedent ID refers to the name of the rule set containing one or more attribute definitions encountered during the overall calculation.	If changes to a CER rule set are published, then a precedent change item for the changed rule set will be written to a precedent change set.
'readall' search	Identifies any "readall" on page 183 (with no nested `match`) expressions encountered during the overall calculation, which retrieved rule objects stored on CER's database tables (as opposed to rule objects retrieved using rule object converters - see "Precedents Identified by Rule Object Converters" on page 84 instead). The precedent ID refers to the name of the rule class sought by the "readall" on page 183 expression.	A precedent change item for the rule class will be written to a precedent change set if: • A new rule object for that rule class is stored on CER's database tables and/or • An existing rule object for that rule class is removed from CER's database tables.
'readall/match' search	Identifies any "readall" on page 183 expressions encountered during the overall calculation, which retrieved rule objects stored on CER's database tables (as opposed to rule objects retrieved using rule object converters - see "Precedents Identified by Rule Object Converters" on page 84 instead). The precedent ID refers to the name of the rule class sought by the "readall" on page 183 expression, together with the attribute name and value used as the search criterion.	A precedent change item for the rule class and its attribute name and match value will be written to a precedent change set if: • A new rule object for that rule class is stored on CER's database tables; • An existing rule object for that rule class is removed from CER's database tables; and/or • The value of the attribute used in the search criterion changes for an existing rule object (in which case two precedent change items will be written - one for the old value of the attribute and another for the new value of the attribute).

Note: By default Rule Set Group precedents are identified instead of Rule Set Definition precedents when the system property `curam.dependency.disable.rule set.grouping` is set to the default value of 'No'. If the property is set to 'Yes', then Rule Set Definition precedents are used instead. This results in a higher number of precedents being stored in the dependency table.

These types of dependencies are best illustrated with an example.

Let's say a new system is written which uses CER to calculate a person's tax liability. This Tax Liability system uses the Dependency Manager to store dependencies, so that the tax liability can be recalculated (using CER) if the person's circumstances change.

The Tax Liability system stores system-wide "tax threshold" information in rule objects, with these rule objects stored on CER's database tables. The CER rules for calculating a person's tax liability include a "readall" on page 183 expression to retrieve all the tax thresholds in the system.

The Tax Liability system also stores system-wide "asset" information in rule objects, with these rule objects stored on CER's database tables. Each asset specifies its owner and its market value. The CER rules for calculating a person's tax liability include a "readall" on page 183 expression to retrieve all assets owned by that person (i.e. those with an `Asset.ownedByPersonID` matching the `Person.personID`). It is possible for an asset's market value to change, and/or for an asset to be transferred from one person to another, by changing its `Asset.ownedByPersonID` from one person's ID to another.

The CER rules for calculating a person's tax liability involve summing the `Asset.marketValue` of all the assets owned by that person.

The Tax Liability system contains separate CER rule sets for retrieving the input data required for the tax liability calculation vs. the actual business calculations which use that retrieved data to calculate a person's tax liability.

A user uses the Tax Liability system to calculate the tax liability for Joe (personID 456) and for Mary (personID 457), who each have one asset. The Tax Liability system uses the CER utility to identify dependencies, and passes these to the Dependency Manager for storage, resulting in the following dependencies being stored:

Table 7. Dependencies Stored For Tax Liability Example

Dependent Type	Dependent ID	Precedent Type	Precedent ID
Tax Liability	456 (Joe's person ID)	'readall' search	Rule class: TaxThreshold Stored because the calculation of Joe's tax liability caused a retrieval of all TaxThreshold rule objects.
Tax Liability	456	'readall/match' search	Rule class: Asset, with attribute value ownedByPersonID=456 Stored because the calculation of Joe's tax liability caused a retrieval of all Asset rule objects owned by Joe.
Tax Liability	456	Stored Attribute Value	789 (the internal ID of `Asset.marketValue` for Joe's asset) Stored because the calculation of Joe's tax liability accessed the stored value of marketValue on the single asset retrieved for Joe.

Table 7. Dependencies Stored For Tax Liability Example (continued)			
Dependent Type	**Dependent ID**	**Precedent Type**	**Precedent ID**
Tax Liability	456	Rule Set Groups	123 (the internal ID of the rule set group) Stored because the calculation of Joe's tax liability involved the definitions of rule attributes in two rule sets: the TaxLiabilityDataRetrievalRuleSet (used to retrieve the TaxThreshold and Asset rule objects) and the TaxLiabilityBusinessCalculationRuleSet (used to compute the overall tax liability based on input data). The group contains these two Rule Set Definitions and is defined on the CREOLERuleSetGroup table
Tax Liability	457 (Mary's person ID)	'readall' search	Rule class: TaxThreshold Stored because the calculation of Mary's tax liability caused a retrieval of all TaxThreshold rule objects.
Tax Liability	457	'readall/match' search	Rule class: Asset, with attribute value ownedByPersonID=457 Stored because the calculation of Mary's tax liability caused a retrieval of all Asset rule objects owned by Mary.
Tax Liability	457	Stored Attribute Value	780 (the internal ID of `Asset.marketValue` for Mary's asset) Stored because the calculation of Mary's tax liability accessed the stored value of marketValue on the single asset retrieved for Mary.
Tax Liability	456	Rule Set Group	123 (the internal ID of the rule set group) Stored because the calculation of Mary's tax liability involved the definitions of rule attributes in two rule sets: the TaxLiabilityDataRetrievalRuleSet (used to retrieve the TaxThreshold and Asset rule objects) and the TaxLiabilityBusinessCalculationsRuleSet (used to compute the overall tax liability based on input data). The group contains these two Rule Set Definitions and is defined on the CREOLERuleSetGroup table.

We can now see how recalculations of tax liability will be triggered by various changes in data:

Table 8. Example Precedent Change Items for Tax Liability		
Data Change	**Precedent Change Items Recorded**	**Recalculations Triggered**
The market value of Joe's asset increases from $100 to $120	• Stored Attribute Value, 789	• Joe's tax liability is recalculated

Table 8. Example Precedent Change Items for Tax Liability (continued)		
Data Change	**Precedent Change Items Recorded**	**Recalculations Triggered**
Mary sells her asset and its rule object is removed	• 'readall/match' search, Rule class: Asset, with attribute value ownedByPersonID=457	• Mary's tax liability is recalculated
Joe receives a new asset, stored as a new rule object	• 'readall/match' search, Rule class: Asset, with attribute value ownedByPersonID=456	• Joe's tax liability is recalculated
Joe transfers his first asset to Mary, thus the asset's ownedByPersonID changes from 456 to 457	• 'readall/match' search, Rule class: Asset, with attribute value ownedByPersonID=456 (old value) • 'readall/match' search, Rule class: Asset, with attribute value ownedByPersonID=457 (new value)	• Joe's tax liability is recalculated • Mary's tax liability is recalculated
An administrator introduces a new tax threshold, stored as a new rule object	• 'readall' search, Rule class: TaxThreshold	• Joe's tax liability is recalculated • Mary's tax liability is recalculated
An administrator removes an existing tax threshold, thus removing its rule object	• 'readall' search, Rule class: TaxThreshold	• Joe's tax liability is recalculated • Mary's tax liability is recalculated
An administrator publishes changes to the TaxLiabilityBusinessCalculationsRuleSet rule set	• Rule Set Definitions, Tax Liability Business Calculations RuleSet • Rule Set Group, 123	• Joe's tax liability is recalculated • Mary's tax liability is recalculated

Precedents Identified by Rule Object Converters

The Rule Object Converters that are registered with CER's Database Data Storage can also contribute to the dependencies identified by the CER utility.

See the documentation for each rule object converter for details on the dependencies identified.

Storing Dependencies for Attribute Values

For attribute values that are stored in CER, CER uses the Dependency Manager to store dependencies. For rule objects that are stored on CER's database tables, each attribute value on the rule objects can play a role.

The roles are as follows:

• Dependent, that is, an attribute value that is derived by calculating its values from input data.

• Precedent, that is, an attribute value that is used as input data during the calculation of a dependent.

CER registers a dependent handler and a precedent handler with the Dependency Manager to allow its stored attribute values to be used as dependents and/or precedents in stored dependencies.

For example, if an attribute `totalIncome` is present on a `Person` rule object stored on CER's database tables, and the calculation of `totalIncome` involved the retrieval of `Income.amount` attribute values also stored on CER's database tables, then CER would request the Dependency Manager to store the fact that the `Person.totalIncome` attribute value depends on the `Income.amount` attribute values.

Recalculating Stored Calculated Attribute Values

When precedent values change, the Dependency Manager requests that CER identify any stored attribute values in CER that depend on the changed precedent values and requests that CER recalculate its dependent values. This is done through the dependent handler that CER registers with the Dependency Manager.

For example, if an `Income.amount` attribute value stored on CER's database tables changed its value, then CER notifies the Dependency Manager that the `Income.amount` precedent has changed, and the Dependency Manager subsequently identifies that the `Person.totalIncome` dependent requires recalculation and invokes CER to recalculate it.

CER Editor Reference

The CER editor allows you to create and maintain CER rule sets. Use this information to learn about the key elements of the CER editor. The global menu provides access to commonly used features and two basic views: The Business View and Technical View. Other key elements include the diagram canvas, tools and template palettes, and rule element pop-up menus.

CER Editor Global Menu

The CER editor global menu provides easy access to common functions.

Global Menu Functions

This section lists and explains each function that is available on the **Global** menu bar.

Table 9. Menu Bar

Function Name	Description
Undo	To cancel the last edit, choose **undo** from the menu bar.
Redo	To cancel the last undo, choose **Redo** from the menu bar this action rolls back the last action taken.
Include Rule Sets	To see the included rule sets and allows rule sets to be added by providing a class path.
Export	The CER editor supports exporting all diagrams in the rule outline view to PNG images, which can be saved as a single ZIP archive file to the user's local hard disk.
Save	A rule set can be saved at any time after it is opened in the editor. Saving a rule set that has a status of published results in an **In Edit** record to be created for that rule set. Any modifications that you make are saved to the in edit record. Saving a rule set that has a status of "In Edit" saves your modifications directly. Rule sets can also have a status of **Newly Created**. Saving modifications to a newly created rule set results in the modifications to BO saved directly to that newly created rule set.

Table 9. Menu Bar (continued)	
Function Name	**Description**
Save All	The CER editor supports opening a number of rule sets from within one instance of the editor. A number of CER elements can point to classes or attributes that are defined in the other rule sets. CER elements that point to something in a different rule set has a menu option **Open Rule Set**. This function allows a user to have any number of rule sets opened in the editor at any one time. The **Save All** menu option saves all rule sets that are opened in the editor and that were modified.
Validate	Allows the user to validate their changes to a rule set. The rule set validator of the CER rule engine is called. CER rule sets are XML files that adhere to the CER-supplied rule schema. CER also includes a comprehensive rule set validator, which can detect errors in your rule set before it allows your rule set to run. The CER rule set validator reports a list of errors in the Properties and Validations pane so they can be resolved by the user. If any errors exist, the CER rule set validator reports the errors and processing halts.
Search	Allows the user to do a quick text-based search that opens a search results window that contains matching results. More information on the functions that are provided and how to refine a search can be found in the **Search Tools** section.

Search Criteria

You can perform a text-based search for criteria in a number of categories: Rules and Rule References, Descriptions, Folders, Technical Data, and Code Tables. Each is explained in the table that follows.

Table 10. Search Criteria	
Name	**Description**
Rules and Rule References	Search for rule by giving the name of the rule.
Descriptions	Search for rules or attributes within a rule by entering some of text that is used to describe the rule or attribute.
Folders	Search for a folder by giving the name of the folder.
Technical Data	Search for any attributes that are specified or not specified attributes.
Code Tables	Search for any attributes that are specified as code tables.

Business View

The **Business** view is for users who are familiar with the structure and text that is required by legislation. The view provides a rules-centric view of the elements that are available for creating and maintaining business rules.

It consists of a tree or hierarchical view of the rules, rules canvas, business element palettes, and the properties and validation pane. The **Rules Outline View** displays the top-level rules and the folders where they can be found.

Rules Outline View

A menu is available for each item within the **Outline View**.

Table 11. New Menu items	
Name	**Description**
Folder (Button)	Create a folder by providing a new folder name.
Rule (Button)	Create a Rule by providing a new rule name and the data type for this rule that is defaulted to a Boolean type. The rule is created in the selected folder. If no folder is selected, the new rule is created at the root level.
New Folder	Create a folder by providing a new folder name.
New Rule	Create a Rule by providing a new rule name and the data type for this rule that is defaulted to a Boolean type. The rule is created in the selected folder. If no folder is selected, the new rule is created at the root level.
Delete	Delete a folder or rule that depends on what is highlighted in the outline view.
Set Rule Type	Create derivation for an attribute. A list of data types is provided for a new rule.
Move Rule	This function allows the rules developer to move a rule from one folder to another folder by selecting the folder name.
Find References To This Rule	Allows the user to find all the references to the selected rule.

Technical View

The **Technical** tab is for users who are familiar with the implementation aspects of rules development. It provides a view of all classes and attributes for the rule that is being edited.

The **Class Outline View** displays all the top-level rules and the folders where they can be found. You can remove all classes and attributes by clicking the menu that is associated with each class or attribute.

Class Outline View

A menu is available for each item within the outline view.

Table 12. New Menu Items	
Name	**Description**
Class (Button)	Create a class by providing a new class name.
Attribute (Button)	Create an attribute within a class by providing a new attribute name. The button is enabled only after a class is selected from the outline view otherwise the button remains inactive.
New Attribute	Create an attribute within a class by providing a new attribute name.
Delete	Delete a class or attribute that depends on what is highlighted in the tree view.
Set Rule Type	Allows the rules developer to create a derivation for an attribute.
Find References To This Rule	Allows the user to find all the references to the selected rule.

Diagram Canvas

The diagram canvas provides you with drag and drop, technical toggle, and pan and zoom controls.

Drag and Drop

You can drag and drop a rule element from any palette to the diagram. For example, drag and drop a `rule` element to an attribute. A visual indicator appears if the target can accept the rule element being dragged.

Additionally rule elements can be dragged and dropped between rule element containers. For example, drag and drop a `rule` element from a result section of the when rule element to a result section of another when rule element.

Toggle to Show Detailed Diagrams

The CER editor provides two different types of views for the rule elements. The business view is a simple version that is aimed at the business rule writer. The technical view is more detailed and is primarily for the technical rule designer. The CER editor provides a toggle to switch between the technical and business views. The toggle icon is located above the pan and zoom controls on the rule canvas.

Pan and Zoom Controls

The CER editor provides pan and zoom controls to allow the user better view and edit the rule set. The arrows on the circular control can be used to pan around any large rules. Clicking on the center button of the pan control recenters the image to the starting position. Use the plus button to zoom in and the minus button to zoom out.

Tools and Template Palettes

The CER editor includes four types of rule element palettes and three rule templates palettes. These palettes contain the following rule elements: Business (default), Business (extended), Data Types, and Technical.

The rule templates are grouped as follows: Household Units, Financial Units, Food Assistance Units, and the Decision Table.

Business Palettes

Both the default and extended business palettes include a range of rule elements that you can use to design business logic in diagram form. Use this information to learn about the available rule elements on business palettes.

Table 13. Rule Elements on Business Palettes		
Image	**Name**	**Description**
	Rule	The Rule element provides a graphical representation of the 'reference' expression. See "Rule" on page 97.
	And Rule Group	The And Rule Group element provides a graphical representation of the 'all' expression and returns a Boolean value. See "And Rule Group" on page 99.
	Or Rule Group	The Or Rule Group element provides a graphical representation of the 'any' expression and returns a Boolean value. See "Or Rule Group" on page 99.
	Not	The Not element provides a graphical representation of the 'not' expression and negate a Boolean value. See "Not" on page 99.

Table 13. Rule Elements on Business Palettes (continued)		
Image	**Name**	**Description**
	Choose	The Choose element provides a graphical representation of the 'choose' expression and chooses a value based on a condition being met. See "Choose" on page 100.
	Compare	The Compare element provides a graphical representation of the 'compare' expression and compares a left-hand-side value with a right-hand-side value, according to the comparison provided. See "Compare" on page 100.
	When	The When element is part of the Choose element and contains a condition to test, and a value to return if the condition passes. See "When" on page 100.
	Arithmetic	The Arithmetic element provides a graphical representation of the 'arithmetic' expression and performs an arithmetical calculation on two numbers (a left-hand-side number and a right-hand-side number) and optionally rounds the result to the specified number of decimal places. See "Arithmetic" on page 100.
MIN	MIN	The Min element provides a graphical representation of the 'min' expression and determines the smallest value in a list (or null if the list is empty). See "MIN" on page 101.
MAX	MAX	The Max element provides a graphical representation of the 'max' expression and determines the largest value in a list (or null if the list is empty). See "MAX" on page 101.
Σ	Sum	The SUM element provides a graphical representation of the 'sum' expression and calculates the numerical sum of a list of Number values. See "SUM" on page 101.
	Repeating Rule	The Repeating Rule element provides a graphical representation of the 'dynamiclist' expression and creates a new list by evaluating an expression on each item in an existing list. See "Repeating Rule" on page 101.
	Filter	The Filter element provides a graphical representation of the 'filter' expression and creates a new list containing all the items in an existing list which meet the filter condition. See "Filter" on page 102.
	Size	The Size element provides a graphical representation of the 'property' expression and the name of the property is set to 'size'. See "Size" on page 103.
	Otherwise	The Otherwise element is part of the Choose element and contains a value to return. See "Otherwise" on page 103.
	Legislation Change	The Legislation Change element provides a graphic representation of the 'legislationchange' and can contain more than one Era element. See "Legislation Change" on page 103.
	Era	The Era element is a part of the Legislation Change elements and contains a date entry (empty from), and a value to return to the Legislation Change element. See "Era" on page 103.

Data Types Palette

The Data Types palette includes a range of rule elements that you can use to design data types in diagram form. Use this information to learn about the available rule elements on the Data Types palettes.

Image	Name	Description
	Boolean	The Boolean element provides a graphical representation of both 'true' and 'false' expression. By default, the Boolean element is set to true. See "Boolean" on page 104.
	String	The String element provides a graphical representation of the 'String' expression that is a literal Number constant value. See "String" on page 104.
	Number	The Number element provides a graphical representation of the 'Number' expression that is a literal Number constant value. See "Number" on page 104.
	Date	The Date element provides a graphical representation of the 'Date' expression that is a literal Date constant value. See "Date" on page 104.
	Codetable	The Codetable element provides a graphical representation of the 'Code' expression that is a literal constant value representing a code from a Cúram code table. See "Code Table" on page 105.
	Rate	The Rate element provides a graphical representation of the 'rate' expression that is a literal constant value representing a rate from a rate table. See "Rate" on page 105.
	Frequency Pattern	The Frequency Pattern element provides a graphical representation of the 'FrequencyPattern' expression that is a literal FrequencyPattern constant value. See "Frequency Pattern" on page 105.
	Resource Message	The Resource Message element provides a graphical representation of the 'ResourceMessage' expression and creates a localizable message from a property resource. See "Resource Message" on page 106.
	XML Message	The XML Message element provides a graphical representation of the 'XMLMessage' expression and creates a localizable message from a property resource. See "Xml Message" on page 106.
	Null	The null element provides a graphical representation of 'null'. The null element can be used in elements like Choose, When, Compare etc... to compare any value to 'null'. See "Null" on page 107.

Table 14. Rule Elements on Data Types Palette

Technical Palette

The Technical palette includes a range of rule elements that can be used to design the technical logic in the diagram form. Use this information to learn about the rule elements on the Technical palette.

Image	Name	Description
	Create	The Create element provides a graphical representation of the 'create' expression and gets a new instance of a rule class in the session's memory. See "Create" on page 107.
	Search	The Search element provides a graphical representation of the 'readall' expression and retrieves all rule object instances of a rule class which were created by client code. See "Search" on page 107.
	Fixed List	The Fixed List element provides a graphical representation of the 'fixedlist' expression and creates a new list from items known at rule set design time. See "Fixed List" on page 108.
	Property	The Property element provides a graphical representation of the 'property' expression. The Property element obtains a property of a Java object. See "Property" on page 108.
	Custom Expression	Custom Expression provides a graphical representation for any user defined valid XML node. See "Custom Expression" on page 109.
	Existence Timeline	The Existence Timeline element provides a graphical representation of the 'existencetimeline' expression. See "Existence Timeline" on page 109.
	Timeline	The Timeline element provides a graphical representation of the 'Timeline' expression. See "Timeline" on page 110.
	Interval	The Interval element provides a graphical representation of the 'Interval' expression.. See "Interval" on page 110.
	Combine Succession Set	The Combine Succession Set element provides a graphical representation of the 'combineSuccessionSet' expression. See "CombineSuccessionSet" on page 111.
	Call	The Call element provides a graphical representation of the 'call' expression and calls out to a static Java method to perform a complex calculation. See "Call" on page 111.
	Period Length	The Period Length element provides a graphical representation of the 'periodlength' expression and calculates the amount of time units between two dates. See "Period Length" on page 112.
	ALL	The ALL element provides a graphical representation of the 'all' expression and returns a Boolean value. See "ALL" on page 112.
	ANY	The ANY element provides a graphical representation of the 'all' expression and returns a Boolean value. See "ANY" on page 112.

Table 15. Rule Elements on Technical Palette

Table 15. Rule Elements on Technical Palette (continued)		
Image	**Name**	**Description**
	This	The This element provides a graphical representation of the 'this' expression that is a reference to the current Rule Object. See "This" on page 113.
	Sort	The Sort element provides a graphical representation of the 'sort' expression. Sort takes a list as child element and performs sorting on it.. See "Sort" on page 113.
	Shared Rule Reference	Shared Rule Reference is basically a normal Reference with a Create element in it. See "Shared Rule Reference" on page 113.
	CONCAT	The Concat element provides a graphical representation of the 'concat' expression and creates a message by concatenating a list of values. See "CONCAT" on page 114.
	Join Lists	The JoinLists element provides a graphical representation of the 'joinlists' expression and creates a new list by joining together some existing lists. See "Join Lists" on page 114.

Household Units Template

The Household Units template includes a range of rule elements that can be used to design Household Units in diagram form. Use this information to learn about the rules elements on the Household Units template palette.

Table 16. Rule Elements on Household Units Template Palette		
Image	**Name**	**Description**
	Household Composition	See "Rule Element Reference for Household Units Templates" on page 115.
	Household Category	See "Rule Element Reference for Household Units Templates" on page 115.

Financial Units Template

The Financial Units template includes a range of rule elements that can be used to design Financial Units in diagram form. Use this information to learn about the rules elements on the Financial Units template palette.

Table 17. Rule Elements on Financial Units Template Palette		
Image	**Name**	**Description**
	Financial Unit	See "Rule Element Reference for Financial Units Templates" on page 115.
	Financial Unit Category	See "Rule Element Reference for Financial Units Templates" on page 115.
	Financial Unit Member	See "Rule Element Reference for Financial Units Templates" on page 115.

Food Assistance Units Template

The Food Assistance Units template includes a range of rule elements that can be used to design Food Assistance Units in diagram form. Use this information to learn about the rules elements on the Food Assistance Units template palette.

Table 18. Rule Elements on Food Assistance Units Template Palette		
Image	**Name**	**Description**
	Food Assistance Unit	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Food Assistance Single Person Category	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Food Assistance Multi Person Category	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Meal Group Members	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Relatives	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Discard	See "Discard" on page 116.
	Member for household unit	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Optional Members	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.
	Exceptions	See food assistance single person category "Rule Element Reference for Food Assistance Units Templates" on page 115.

Decision Table Template

The Decision Table template contains the decision table element that is used to create decision tables. Use this information to learn about the decision table element.

Table 19. Elements on Decision Table Palette		
Image	**Name**	**Description**
	Decision Table	The Decision Table provides a graphical representation of a 'decision table'. See "Rule Element Reference for Decision Table Templates" on page 116.

Rule Element Pop-up Menus

The Rule Element pop-up menu provides general functions for all rule elements and specific functions for single rule element.

For information about specific rule element pop-up menu items, see the "Rule Element Reference for CER Editor Palettes" on page 96.

Table 20. General Pop-up Menu Functions	
Function Name	**Description**
Cut	Cut the rule element and be ready to move to other location.
Copy	Copy the existing rule element and be ready to create a new copy in other location.
Delete	Delete the rule element from the diagram.
Collapse	Collapse the children of the rule element.
Expand	Expand the children of the rule element.
Make Timeline	Only available for the technical view. Create a timeline for the rule element.
Make Timeline Interval	Only available for the technical view. Create a timeline interval for the rule element.
Remove Timeline	Only available for the technical view. Remove a timeline from the rule element.
Remove Timeline Interval	Only available for the technical view. Remove a timeline interval from the rule element.

Rule Element Properties

Use this information to learn about properties in the CER editor including collapsible property and validation panels, as well as general, rule class, and attribute properties.

For information about specific rule element properties, see the "Rule Element Reference for CER Editor Palettes" on page 96.

Collapsible Property and Validation Panels

Property panels are grouped in the Business and Technical tabs to show property information based on your perspective as a rules developer.

You can expand and collapse the property and validation panels by clicking on the toggle button on the panel containers border. Expanding the property and validation panels reveals the currently selected diagram property details. Collapsing the property and validation panel hides the details to provide more space to view the diagram canvas.

General Properties for all Rule Elements

The following table describes the general properties that you can set for all rule elements.

Table 21. General properties	
Name	**Description**
Display Name	This is a localizable name. The field supports multi byte and accented characters. This field is available in the Business Tab.

Table 21. General properties (continued)	
Name	**Description**
Description	The description support free-text entries for either rules or attributes. Multi byte and accented characters are supported. It requires the rules developer to enter a meaningful business description of the rule or attribute. This field is available in the Business Tab. The content of the description field is localizable and can contain multi byte or accented characters.
Legislation Link	A link to any legislation website. This field is available in the Business tab.
Display Name ID	An ID for the name of the rule element. This field is available in the Technical tab.
Tag	The tag annotation supports free-text tag entries for any element that supports annotations. It is used for searching the rule element. This field is available in the Business tab.

Rule Class Properties

The following table describes the properties that you can set for rule classes.

Table 22. Rule Class Properties	
Name	**Description**
Primary Attribute	Select an attribute from a drop-down to be a primary attribute of this rule class. This is visible in the Business Tab.
Abstract	The class is an abstract class if this property is selected. This is visible in the Business Tab.
Extends	Users can extend this rule class from other rule class that either belongs to the current or external rule set by the Change Rule Set and Rule Class Wizard. This is visible in the Business Tab.
Class	The name of the rule Class. This is visible in the Technical Tab.
SuccessionSet	The SuccessionSet annotation is added if this property is selected. Users need to select date attributes from both "Start Date Attribute" and "End Date Attribute" drop-downs. This is visible in the Technical Tab.
ActiveInEditSuccessionSet	The ActiveInEditSuccessionSet annotation is added if this property is selected. Users need to select date attributes from both "Start Date Attribute" and "End Date Attribute" drop-downs. This is visible in the Technical Tab.

Attribute Properties

The following table describes the properties that you can set for attributes.

Table 23. Attribute Properties	
Name	**Description**
Data Type	The data type of an Attribute. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

Table 23. Attribute Properties (continued)	
Name	**Description**
Display	The Display annotation is added if this property is selected. It requires users to enter a value. This is visible in the Business Tab.
Display Subscreen	The Display Subscreen annotation is added if this property is selected. This is visible in the Business Tab.
Class	The name of the Rule Class the attribute belongs to. This is visible in the Technical Tab.
Attribute	The Name of the Attribute. This is visible in the Technical Tab.
Related Succession Set	The Related Succession Set annotation is added if this property is selected. It requires users to select either one of them (none, parent, child) from the drop-down. This is visible in the Technical Tab.
Related Evidence Type	The Related Evidence Type annotation is added if this property is selected. It requires users to select either one of them (none, parent, child) from the drop-down. This is visible in the Technical Tab.
Abstract	The attribute is set to an "abstract" rule element and the class will be an abstract class if this property is selected. This is visible in the Technical Tab.

Rule Element Wizards

Some rule elements, for example, Rule and Create Rule, use the Change Rule Set and Rule Class wizard to link to another rule class of either the current rule set or external rule set. Use this information to learn out the available options in the Rule Element wizard.

The following table describes the three available options.

Table 24. Rule Element Wizard Options	
Name	**Description**
Create an Empty Rule Reference	This allows the rules writer to create a placeholder for a Rule element which can edited at a later point whilst developing the rule.
Create New Rule	This allows the rules writer to create a new rule by specifying the name of the rule and the result type of the new rule from the drop-down combo box.
Use Existing Rule	This allows the rules writer to choose from an already created rule by selecting the Rule from the drop-down combo box. If the rule is not contained in the existing rule the rules writer can select another ruleset by entering the name of the ruleset and hitting the search button. This will open another dialog box allowing the rules writer to choose the appropriate Rule Set.

Rule Element Reference for CER Editor Palettes

Definitions of the rule elements that are included with the CER editor. The rule elements are categorized by the rule element palettes that they apply to. For helpful categorizations of the rule elements, see the *Tools and Template Palettes* section.

Palettes

You can un-dock and re-dock the palettes as needed. You can also expand and collapse palettes to reveal and hide the selected palette name and textual descriptions of each of the rule elements.

Un-docking and Re-docking Palettes

Un-dock the palettes by dragging the palette from its original position onto the diagram canvas. To re-dock the palette, simply drag the palette back to the right hand side of the screen.

Expanding and Collapsing Palettes

You can expand and collapse the palette container by clicking the toggle button on the palette containers border. Expanding the palette container reveals the currently selected palette name and textual descriptions of each of the rule elements. Collapsing the palette container hides the textual descriptions and reduces the amount of space the container takes up on the diagram.

Rule Element Reference for the Default and Extended Business Palettes

Definitions of the rule elements that are available on the default and extended business palettes.

Rule

The Rule element provides a graphical representation of the 'rule' expression and negates a Boolean value. Use this information to learn about how to use the Rule element, types of scenarios to use the Rule element, and specific properties and menu items.

You cannot add any other palette elements to the Reference element. The Rule element can be added to other palette elements, for example, And Rule Group, Or Rule Group or Repeating Rule. There are seven different types of scenarios to use the Rule elements. A reference to a rule can be added when a user is in the Business View and in the Technical View. When a user is in the Business View the following options are available:

- Empty Reference - you can create an empty reference.
- Create New Rule - you can create a new rule, and
- Use Existing Rule - you can select a rule from the current rule set, or perform a search of external rule sets and select a rule from an external rule set.

When you are in the Technical View the following options are available:

- Use Existing Rule - you can select a rule from the current rule set or perform a search of external rule sets and select a rule from an external rule set, and
- Create New Rule - users can create a new rule.

If you are in the Business View and you want to add a reference to a specific Rule Class or Attribute, you must switch to the Technical View.

Table 25. Types of scenarios to use the Rule elements	
Name	**Description**
Nested Reference	A nested reference can be created on the reference that points to an attribute that is of another rule class type, for example, it is not contained in the existing class. The outer reference attribute is an object of the inner reference attribute class. This structure can be created only if the inner reference attribute, which is of another rule class type, is located in the class where the inner reference is being created.

Table 25. Types of scenarios to use the Rule elements (continued)

Name	Description
Nested Reference with Create	A nested reference can be created on the reference that points to an attribute that is of another rule class type, for example, it is not contained in the existing class, but the attribute is not located in the current class. The outer reference attribute is an object of the inner reference attribute class. This structure can be created only if the inner reference attribute (which is of another rule class type) is located in the class that is not the class where the inner reference is being created.
CurrentUnitMemeber	To refer to the member in the current unit that satisfies the exceptional test condition.
FARelationship	To refer to the class that is used as the relationship record for the meal group member.
FAException	To refer to a class that is used to check if other members in a unit satisfy the exceptional condition.
HC Current	To refer to a current list element (like Household member etc) in the household composition HCCurrent is used.
Current	To refer to a current list element in the lists like dynamiclist Current element is used. If an attribute of the current list element class has to be referred then a reference pointing to that attribute is wrapped around the current element.

The following table lists specific properties items for this element:

Table 26. Reference properties items

Name	Description
Class	The name of the Rule Class. This is visible in the Business Tab.
Attribute	The name of an attribute. This is visible in the Business Tab.
Single Item	Only one item returned from the element. This is visible in the Technical Tab.
Behavior when no items found	Return either one of these results (error, return null) when no items are found. This is active when the Single Item Box is checked.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items are found. This is active when the Single Item Box is checked.

The following table lists specific pop-up menu items for this element:

Table 27. Rule Element Pop-up Menu items

Name	Description
Wrap in OR	Wrap the Rule element in the Or Rule Group element.
Wrap in AND	Wrap the Rule element in the And Rule Group element.
Edit Rule	Edit the Rule element by choosing the rule you want to refer to.
Open Diagram For This Rule	Opens the diagram for the rule that is being referenced in a new diagram tab.

Table 27. Rule Element Pop-up Menu items (continued)	
Name	**Description**
Include Rule Logic Here	If the Rule element is referring to another rule (in the same ruleset or in another external ruleset), the logic for that rule can be included in the current rule being viewed in the editor.

And Rule Group

The And Rule Group element provides a graphical representation of the 'all' expression and returns a Boolean value. It operates on a list of Boolean values to determine whether all of the list values are true.

The list of Boolean values is typically provided by a fixedlist. You can add other palette elements, such as Reference, Repeating Rule or Choose that provide a list of Boolean values to the empty member of the And element for calculation. You can add the And Rule Group element to other palette elements, for example, And Rule Group, Or Rule Group or When.

The following table lists specific pop-up menu items for the And Rule Group element.

Table 28. And Rule Group Element Pop-up Menu items	
Name	**Description**
Wrap in AND	Wrap the And Rule Group element in another And Rule Group element.
Wrap in OR	Wrap the And Rule Group element in the Or Rule Group element.
Change to Or	Change the And Rule Group element to the Or Rule Group element.

Or Rule Group

The Or Rule Group element provides a graphical representation of the 'any' expression. You can add Boolean values to the empty member of the Or Rule Group element for calculation.

You can add the Or Rule Group element to other palette elements, for example, And Rule Group, Or Rule Group or When.

The following table lists specific pop-up menu items for this element:

Table 29. Or Element Pop-up Menu Items	
Name	**Description**
Wrap in Or Rule Group	Wrap the Or Rule Group element in another Or Rule Group element.
Wrap in And Rule Group	Wrap the Or Rule Group element in the And Rule Group element.
Change to And	Change the Or Rule Group element to the And Rule Group element.

Not

The Not element provides a graphical representation of the 'not' expression and negates a Boolean value. You can add other palette elements, for example, Reference, Or or And that return a Boolean value to the Not element.

You can add the Not element to other palette elements, for example, And Rule Group, Or Rule Group or Repeating Rule.

The following table lists specific pop-up menu items for this element:

Table 30. Not Element Pop-up Menu Items	
Name	**Description**
Wrap in And	Wrap the Not element in another And Rule Group element.

Table 30. Not Element Pop-up Menu Items (continued)	
Name	**Description**
Wrap in Or	Wrap the Not element in the Or Rule Group element.

Choose

The Choose element provides a graphical representation of the 'choose' expression and chooses a value based on a condition being met. The edit choose wizard provides nine data types: String, Number, Boolean, Date, Datetime, Codetable, Message, timeline, Codetable, Rule Class, Java Class, and Message.

You can add Only When and Otherwise elements to the Choose element. You can add the Choose element to other palette elements, for example, And Rule Group, Or Rule Group or the complex view of the Compare.

The following table lists specific pop-up menu items for this element:

Table 31. Choose Element Pop-up Menu Items	
Name	**Description**
Wrap in OR	Wrap the Choose element in Or Rule Group element.
Wrap in AND	Wrap the Choose element in the And Rule Group element.
Edit Choose	Provide a list of data types for Choose. See the Edit Choose Dialog box as follows.

Compare

The Compare element provides a graphical representation of the 'compare' expression and compares a left-side value with a right-side value, according to the comparison provided. When you add a compare to a diagram, it creates empty members for the left and right-side arguments.

You can add the Compare element to other palette elements, for example, When, And Rule Group, or Repeating Rule.

The following table lists specific pop-up menu items for this element:

Table 32. Compare Element Pop-up Menu Items	
Name	**Description**
Wrap in OR	Wrap the Compare element in Or Rule Group element.
Wrap in AND	Wrap the Compare element in the And Rule Group element.

When

The When element is part of the Choose element and contains a condition to test, and a value to return if the condition passes. You can add other palette elements, for example, Code or Any to the empty condition of the When element.

You can add other palette elements, for example, Number or Reference to the empty value of the When element.

Arithmetic

The Arithmetic element provides a graphical representation of the 'arithmetic' expression and performs an arithmetical calculation on two numbers (a left-side number and a right-side number) and optionally rounds the result to the specified number of decimal places.

You can add other palette elements, for example, Max, Min or Reference to the Arithmetic element. You can add the Arithmetic element to other palette elements, for example, When or Repeating Rule.

The following table lists specific properties items for this element:

Table 33. Arithmetic properties items	
Name	**Description**
Decimal Places	A place for a user to enter how many decimal places. This is visible in the Business Tab.
Rounding	Provide different type of rounding (ceiling, down, floor, half down, half even, half up, up). This is visible in the Business Tab.

MIN

The Min element provides a graphical representation of the 'min' expression and determines the smallest value in a list, or null if the list is empty. The Min element has a fixedlist that contains any type of comparable object.

You can add other palette elements, for example, Number or Rule Reference to the Empty Member (fixedlist) of the Min element. You can add the Min element to other palette elements, for example, When or Repeating Rule.

The following table lists the pop-up menu item for this element.

Table 34. Min Element Pop-up Menu Item	
Name	**Description**
Change To Max	Change the Min element to the Max element.

MAX

The Max element provides a graphical representation of the 'max' expression and determines the largest value in a list (or null if the list is empty). The Max element has a fixedlist that contains any type of comparable object.

You can add other palette elements, for example, Number or Rule Reference to the Empty Member (fixedlist) of the Max element. The Max element can be added to other palette elements, for example, When or Repeating Rule.

The following table lists the pop-up menu item for this element:

Table 35. Max Element Pop-up Menu Item	
Name	**Description**
Change To Min	Change the Max element to the Min element.

SUM

The SUM element provides a graphical representation of the 'sum' expression and calculates the numerical sum of a list of Number values.

The SUM element has a fixedlist that contains any type of numerical object. You can add other palette elements, for example, Number or Rule Reference to the Empty Member (fixedlist) of the SUM element. The SUM element can be added to other palette elements, for example, When or Repeating Rule.

Repeating Rule

The Repeating Rule element provides a graphical representation of the 'dynamiclist' expression and creates a new list by evaluating an expression on each item in an existing list.

You can add other palette elements, for example, Rule Reference to the empty list of the Repeating Rule element. You can add other palette elements, for example, Sum or Choose to the empty member

(listitemexpression) of the Repeating Rule element. You can add the Repeating Rule element to other palette elements, for example, And Rule Group, Or Rule Group or When.

The following table lists specific properties for this element:

Table 36. Repeating Rule properties	
Name	**Description**
Single Item	Only one item returned from the element. This is visible in the Technical Tab.
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is active when the 'Single Item' is checked.
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is active when the 'Single Item' is checked.

The following table lists specific pop-up menu items for this element:

Table 37. Repeating Rule Pop-up Menu iIems	
Name	**Description**
Remove Duplicates	Remove duplicate items in the Repeating Rule element.
Concatenate Results	Concatenate the items in the Repeating Rule element.
Join Inner Lists	Join the lists together to make one list.
Succeed On Any	Wrap the Repeating Rule element in the Any.
Succeed On All	Wrap the Repeating Rule element in the All.
Sum Items	Sum up a list of numbers.

Filter

The Filter element provides a graphical representation of the 'filter' expression and creates a new list containing all the items in an existing list which meet the filter condition

You can add other palette elements, for example, Reference to the empty list of the Filter element. You can add other palette elements, for example, Sum or Choose to the empty member (listitemexpression) of the Filter element. You can add the Filter element to other palette elements, for example, And, Or or When.

The following table lists specific properties for this element:

Table 38. Filter Properties	
Name	**Description**
Single Item	Only one item returned from the element. This is visible in the Technical Tab.
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is active when the 'Single Item' is checked.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items found. This is active when the 'Single Item' is checked.

The following table lists specific pop-up menu items for this element:

Table 39. Filter Pop-up Menu Items	
Name	**Description**
Remove Duplicates	Remove duplicate items in the Filter element.
Concatenate Results	Concatenate the items in the Filter element.
Join Inner Lists	Join the lists together to make one list.
Wrap in OR	Wrap the Filter element in Or Rule Group element.
Wrap in AND	Wrap the Filter element in the And Rule Group element.

Size

The Size element provides a graphical representation of the 'property' expression and the name of this element is set to 'size'. The Size element obtains a property of a Java object.

You can add other palette elements, for example, Rule Reference or Repeating Rule to the Size element. The Size element can be added to other palette elements, for example, When or Repeating Rule.

The following table lists specific properties item for this element:

Table 40. Size properties item	
Name	**Description**
Value	The name of the property element.

Otherwise

The Otherwise element is part of the Choose element and contains a value to return.

You can add other palette elements, for example, Number or Rule Reference to the empty value of the Otherwise element.

Legislation Change

The Legislation Change element provides a graphical representation of the 'legislationchange' and can contain more than one Era element.

The Legislation Change wizard provides ten data types (String, Number, Boolean, Date, Datetime, List, Message, Codetable, Ruleclass and Javaclass). You can use the selected data type to define a returning value of the Era element.

The following table lists specific pop-up menu items for this element:

Table 41. Legislation Change Element Pop-up Menu Items	
Name	**Description**
Edit Legislation Change	Edit the Legislation Change by choosing a type for the new rule. See the Edit Legislation Change Dialog box as follows.

Era

The Era element is a part of the Legislation Change elements and contains a date entry (empty from), and a value to return to the Legislation Change element.

You can add other palette elements, for example, Date or Rule Reference to the empty date entry of the Era element. You can add other palette elements, for example, Choose or Reference to the empty value of the Era element.

Rule Element Reference for Data Types Palette

Definitions of the rule elements that are available on the Data Types palette.

Boolean

The Boolean element provides a graphical representation of both 'true' and 'false' expressions. By default, the Boolean element is set to true.

You can add the Boolean element can to other palette elements, for example, And Rule Group, Or Rule Group, or Repeating Rule elements. No element can be added to the Boolean element.

The following table lists specific pop-up menu items for this element:

Table 42. Boolean Pop-up Menu Items	
Name	**Description**
Change To False	Change the boolean rule element to false if its value is true.
Change To True	Change the boolean rule element to true if its value is false.

String

The String element provides a graphical representation of the 'String' expression that is a literal Number constant value.

You can add the String element to other palette elements, for example, When or Repeating Rule. No element can be added to the String element.

The following table lists specific properties items for this element:

Table 43. String properties items	
Name	**Description**
Value	The value of the string. This is visible in the Business Tab.

Number

The Number element provides a graphical representation of the 'Number' expression that is a literal Number constant value.

You can add the Number element to other palette elements, for example, When or Repeating Rule. No element can be added to the Number element.

The following table lists the properties item for this element:

Table 44. Number Properties Item	
Name	**Description**
Value	The value of the number. This is visible in the Business Tab.

Date

The Date element provides a graphical representation of the 'Date' expression that is a literal Date constant value.

You can add the Date element to other palette elements, for example, PeriodLength or Era. No element can be added to the Date element.

The following table lists specific properties items for this element:

Table 45. Date Properties Items	
Name	**Description**
Value	The value of the date. The default is set to today's date. This is visible in the Business Tab.
Zero Date	By checking the 'Zero Date' checkbox, the user can use the Cúram 'Zero Date'. This is visible in the Business Tab.

Code Table

The Code Table element provides a graphical representation of the 'Code' expression that is a literal constant value representing a code from a Cúram code table.

You can add the CodeTable element to other palette elements, for example, When. No element can be added to the CodeTable element.

The following table lists specific properties items for this element:

Table 46. Code Table Properties Items	
Name	**Description**
Codetable Name	The name of the code table. Leave blank and search for all available codetables. Input a value and search for a codetable starting with the input value. This is visible in the Business Tab.
Codetable Value	A drop-down box containing all the items contained in selected codetable. This is visible in the Business Tab.

Rate

The Rate element provides a graphical representation of the 'rate' expression that is a literal constant value representing a rate from a Cúram rate table.

You can add the Rate element to other palette elements, for example, When. No element can be added to the Rate element.

The following table lists specific properties items for this element:

Table 47. Rate Properties Items	
Name	**Description**
Table Name	The name of the rate table. This is visible in the Business Tab.
Row	The row's value of the rate table. This is visible in the Business Tab.
Column	The column's value of the rate table. This is visible in the Business Tab.

Frequency Pattern

The Frequency Pattern element provides a graphical representation of the 'FrequencyPattern' expression that is a literal FrequencyPattern constant value.

You can add the Frequency Pattern element to other palette elements, for example, When or Create. No element can be added to the Frequency Pattern element.

The following table lists specific properties items for this element:

Table 48. Frequency Pattern Properties Items	
Name	**Description**
Pattern	The frequency pattern. This is visible in the Business Tab.

The following table lists specific pop-up menu items for this element:

Table 49. Filter Pop-up Menus items	
Name	**Description**
Edit Frequency Pattern	Edit the selected frequency pattern.

Resource Message

The Resource Message element provides a graphical representation of the 'ResourceMessage' expression and creates a localizable message from a property resource.

You can add the Resource Message element to other palette elements, for example, When or Create. No element can be added to the Resource Message element.

The following table lists specific properties items for this element:

Table 50. Resource Message Properties Items	
Name	**Description**
Key	Resource Bundle objects contain an array of key-value pairs. You specify the key, which must be a String, when you want to retrieve the value from the Resource Bundle. This is visible in the Business Tab.
Resource Bundle	The name of the resource bundle. This is visible in the Business Tab.

The following table lists specific pop-up menu items for this element:

Table 51. Resource Message Pop-up Menus items	
Name	**Description**
New Argument	Add a new argument to the resource message rule element.
Remove Argument	Remove an argument from the resource message rule element.

Xml Message

The Xml Message element provides a graphical representation of the 'XmlMessage' expression and creates a message from the free-form XML content.

You can add the Xml Message element to other palette elements, for example, When or Create. No element can be added to the Xml Message element.

The following table lists specific properties items for this element:

Table 52. XML Message Properties Items	
Name	**Description**
Value	The value of the Xml resource's content.

The following table lists specific pop-up menu items for this element:

Table 53. Xml Message Pop-up Menus items	
Name	**Description**
Edit Message	Provide a dialog box in which a user can enter the content of the message.

Null

The null element provides a graphical representation of the 'null'. The null element can be used in element like Choose/When, Compare etc. to compare any value to null.

Rule Element Reference for Technical Logic Palette

Definitions of the rule elements that are available on the Technical Logic palette.

Create

The Create element provides a graphical representation of the 'create' expression and gets a new instance of a rule class in the session's memory. The Create element supports two types of parameters: standard and mandatory.

You can add other palette elements, for example, Rule Reference, Repeating Rule or Choose to the parameters of the Create element. You can add the Create element to other palette elements, for example, Repeating Rule or When.

The following table lists specific properties items for this element:

Table 54. Create Properties Items	
Name	**Description**
Class	The name of the rule class that is chosen as a type. This is visible in the Technical Tab.

The following table lists specific pop-up menu items for this element:

Table 55. Create Element Pop-up Menus items	
Name	**Description**
Edit Create	Edit the Create element by choosing the rule class and rule set you want to refer to.
New Parameter	Create a new parameter by selecting the attribute you want to add a parameter for.
New Mandatory Parameter	Create a new mandatory parameter.

Search

The Search element provides a graphical representation of the 'readall' expression and retrieves all rule object instances of a rule class which were created by client code.

It can retrieve a single item from a list if the 'singleitem' expression is selected. You can add the Search element to other palette elements, for example, Filter or Create. No element can be added to the Search element.

The following table lists specific properties items for this element:

Table 56. Search Properties Items	
Name	**Description**
Class	The name of the rule class that is chosen as a type. This is visible in the Technical Tab.
Rule Set	The name of the rule set that includes the chosen rule class. This is visible in the Technical Tab.
Single Item	Only one item returned from the element. This is visible in the Technical Tab.

Table 56. Search Properties Items (continued)	
Name	**Description**
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is activated when the 'Single Item' box is checked.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items found. This is activated when the 'Single Item' box is checked.

The following table lists specific pop-up menu items for this element:

Table 57. Search Element Pop-up Menus items	
Name	**Description**
Edit Search	Edit the Search element by choosing the rule class and rule set you want to refer to.

Fixed List

The Fixed List element provides a graphical representation of the 'fixedlist' expression and creates a new list from items known at rule set design time. The Fixed List wizard provides nine data types: String, Number, Boolean, Date, Date time, Codetable Entry, Rule Class, Java Class, and Message.

The sub-list function is provided. You can add other palette elements, for example, Rule Reference, Repeating Rule or Choose to the empty member of the Fixed List element. You can add the Fixed List element to other palette elements, for example, And Rule Group, Or Rule Group or When. The fixed list element will be wrapped by other rule element depending on which data type is selected. For example, And Rule Group/Or Rule Group rule elements for the boolean type, Concat rule element for the string type, Max/Min/Sum rule elements for the number type.

The following table lists specific properties items for this element:

Table 58. FixedList Properties Items	
Name	**Description**
Data Type	The data type of the fixedlist. This should correspond to the data type of the attribute. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

Property

The Property element provides a graphical representation of the 'property' expression. The Property element obtains a property of a Java object.

You can add other palette elements, for example, Rule Reference or Repeating Rule to the Property element. You can add the Property element to other palette elements, for example, When or Repeating Rule.

The following table lists specific properties items for this element:

Table 59. Property Properties Items	
Name	**Description**
Value	The name of the property element.

The following table lists specific pop-up menu items for this element:

Table 60. Property Pop-up Menus items	
Name	**Description**
Wrap in OR	Wrap the property element in the Or Rule Group element.
Wrap in AND	Wrap the property element in the And Rule Group element.
New Argument	Add a new argument to the property rule element.
Remove Argument	Remove an argument from the Property rule element.

Important: Since Cúram V6, CER and the Dependency Manager support the automatic recalculation of CER-calculated values if their dependencies change.

If you change the implementation of a property method, CER and the Dependency Manager will *not* automatically know to recalculate attribute values that were calculated using the old version of your property method.

Once a property method has been used in a production environment for stored attribute values, rather than changing the implementation you should instead create a new property method (with the required new implementation), and change your rule sets to use the new property method. When you publish your rule set changes to point to the new property method, CER and the Dependency Manager will automatically recalculate all instances of the affected attribute values.

Custom Expression

Custom Expression provides a graphical representation of any user-defined valid XML node.

You can add Custom Expression to any element in CER and you must ensure that the Custom Expression nodes names do not match with any CER language node names.

The following table lists specific pop-up menu items for this element:

Table 61. Custom Expression Pop-up Menu Items	
Name	**Description**
Edit Custom Expression	Displays the popup to edit the custom expression.

Existence Timeline

The Existence Timeline element provides a graphical representation of the 'existencetimeline' expression.

You can add other palette elements, for example, Date or Rule Reference to the FromDate and ToDate of the Existence Timeline element. You can add other palette elements, for example, Date or Rule Reference to the preExistenceValue, postExistenceValue and ExistenceValue of the Existence Timeline element. You can add the Existence Timeline element to other palette elements, for example, Repeating Rule.

The following table lists specific properties items for this element:

Table 62. Existence Timeline Properties Items	
Name	**Description**
Data Type	The data type of the Existence Timeline. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

The following table lists specific pop-up menu items for this element:

Table 63. Existence Timeline Element Pop-up Menus items	
Name	**Description**
Edit Existence Timeline	Provide 10 data types (String, Number, Boolean, Date, Datetime, Codetable Entry, Rule class, Java Class, List and Message) for an interval type of the existence timeline rule element.

Timeline

The Timeline element provides a graphical representation of the 'Timeline' expression. You can set an initial value for a Timeline element by using its menu option.

You can add other palette elements, for example, Interval, FixedList, Repeating Rule to the Timeline element.

The following table lists specific properties items for this element:

Table 64. Existence Timeline Properties Items	
Name	**Description**
Data Type	The data type of the Timeline. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

The following table lists specific pop-up menu items for this element:

Table 65. Timeline Element Pop-up Menus items	
Name	**Description**
Add Intervals	Adds intervals to the Timeline
Add Initial Value	Adds the initial value to the Timeline
Remove Intervals	Removes intervals from the Timeline
Remove Initial Value	Removes the initial value from the Timeline
Edit Timeline	Shows the Timeline wizard to edit the Timeline

Interval

The Interval element provides a graphical representation of the 'Interval' expression.

You can add other palette elements, for example, Date or Reference to the FromDate and ToDate of the Existence Timeline element. You can add Interval only to the Intervals element of a Timeline element. You can set Interval type using the interval wizard. The Interval element contains a start element and a value element as child elements.

The following table lists specific pop-up menu items for this element:

Table 66. Interval Element Pop-up Menu Items	
Name	**Description**
Edit Interval	Shows the wizard to edit the interval

CombineSuccessionSet

The CombineSuccessionSet element provides a graphical representation of the 'combineSuccessionSet' expression.

You can add other palette elements, for example, Filter or Fixed List, to the CombineSuccessionSet element. You can add the CombineSuccessionSet element to other palette elements, for example, When or Create.

The following table lists specific pop-up menu items for this element:

Table 67. CombineSuccessionSet Element Pop-up Menu Items	
Name	**Description**
Edit CombineSuccessionSet	Edit the CombineSuccessionSet element by choosing the rule class and rule set you want to refer to.

Call

The Call element provides a graphical representation of the 'call' expression and calls out to a static Java method to perform a complex calculation.

You can add the new argument to the Call element. You can add other palette elements, for example, Date, Period Length or Rule Reference to the argument of the Call element. You can add the Period Length element to other palette elements, for example, Create.

The following table lists specific properties items for this element:

Table 68. Call Properties Items	
Name	**Description**
Class	The name of the class that has the calling method. This is visible in the Technical Tab.
Method Name	The name of the method that will be called. This is visible in the Technical Tab.
Date Type	The return type of the call. This should be the same as the data type of the attribute. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

The following table lists specific pop-up menu items for this element:

Table 69. Call Pop-up Menus items	
Name	**Description**
Wrap in AND	Wrap the call rule element in the And Rule Group element.
Wrap in OR	Wrap the call rule element in the Or Rule Group element.
New Argument	Add a new argument to the call rule element.
Remove Argument	Remove an argument from the call rule element.

Important: Since Cúram V6, CER and the Dependency Manager support the automatic recalculation of CER-calculated values if their dependencies change.

If you change the implementation of a static method, CER and the Dependency Manager will *not* automatically know to recalculate attribute values that were calculated using the old version of your static method.

Once a static method has been used in a production environment for stored attribute values, rather than changing the implementation you should instead create a new static method (with the required new implementation), and change your rule sets to use the new static method. When you publish your rule set changes to point to the new static method, CER and the Dependency Manager will automatically recalculate all instances of the affected attribute values.

Period Length

The Period Length element provides a graphical representation of the 'periodlength' expression and calculates the amount of time units between two dates.

You can add other palette elements, for example, Date, Call or Rule Reference to the Period Length element. You can add the Period Length element to other palette elements, for example, Create.

The following table lists specific properties items for this element:

Table 70. Period Length Properties Items

Name	Description
Unit	Provide four types of units (days, weeks, months, years) for the period length rule element. This is visible in the Technical Tab.
EndDateInclusion	Provide two types of date inclusions (inclusive or exclusive). This is visible in the Technical Tab.

ALL

The ALL element provides a graphical representation of the 'all' expression and returns a Boolean value.

It operates on a list of Boolean values to determine whether all of the list values are true. The ALL element does not have a fixedlist with it. You can add other palette elements, for example, Repeating Rule or Fixed List to the ALL element. You can add the ALL element to other palette elements, for example, Repeating Rule.

The following table lists specific pop-up menu items for this element:

Table 71. ALL Element Pop-up Menu Items

Name	Description
Wrap in OR	Wrap the And Rule Group element in another Or Rule Group element.
Wrap in AND	Wrap the And Rule Group element in another And Rule Group element.
Change to OR	Change the And Rule Group element to the Or Rule Group element.

ANY

The ANY element provides a graphical representation of the 'any' expression and returns a Boolean value. It operates on a list of Boolean values to determine whether any of the list values are true.

The ANY element does not have a fixedlist with it. You can add other palette elements, for example, Repeating Rule or Fixed List to the ANY element. You can add the ANY element to other palette elements, for example, Repeating Rule.

The following table lists specific pop-up menu items for this element:

Table 72. ANY Element Pop-up Menu Items

Name	Description
Wrap in OR	Wrap the Or Rule Group element in another Or Rule Group element.
Wrap in AND	Wrap the Or Rule Group element in the And Rule Group element.

Table 72. ANY Element Pop-up Menu Items (continued)	
Name	**Description**
Change to AND	Change the Or Rule Group element to the And Rule Group element.

This

The This element provides a graphical representation of the 'this' expression that is a reference to the current Rule Object.

You can add the This element can be added to other palette elements, for example, When or Any. No element can be added to the This element.

Sort

The Sort element provides a graphical representation of the 'sort' expression. Sort takes a list as a child element and sorts it.

The following table lists specific pop-up menu items for this element:

Table 73. ANY Element Pop-up Menu Items	
Name	**Description**
New Ascending Sort Items	Allows to add an ascending sort item.
New Descending Sort Items	Allows to add a descending sort item.

Shared Rule Reference

A Shared Rule Reference is a normal Reference with a Create element in it. Shared Rule reference shows a wizard that displays the names of the Rule classes that have Primary attribute set in the current rule set.

You can edit Shared Rule Reference by selecting the "Edit Shared Rule Reference" menu option on the diagram. A shared rule is a class with a primary attribute in it. The Shared Rule Wizard allows you to create a Shared Rule that can be used for creating the Shared Rule References.

The following table lists specific properties items for this element:

Table 74. Shared Rule Reference Properties Items	
Name	**Description**
Class	The name of the rule class. This is visible in the Business Tab.
Attribute	The name of an attribute. This is visible in the Business Tab.
Single Item	Only one item returned from the element. This is visible in the Technical Tab.
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is active when the Single Item Box is checked.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items found. This is active when the Single Item Box is checked.

The following table lists specific pop-up menu items for this element:

Table 75. Shared Rule Reference Element Pop-up Menus items	
Name	**Description**
Wrap in OR	Wrap the Shared Rule Reference element in Or Rule Group element.

Table 75. Shared Rule Reference Element Pop-up Menus items (continued)	
Name	**Description**
Wrap in AND	Wrap the Shared Rule Reference element in the And Rule Group element.
Edit Reference	Edit the Reference element by choosing the rule you want to refer to.
Edit Shared Rule Reference	Edit the Shared Rule Reference element by choosing the shared rule you want to refer to. See the Shared Rule Edit Reference Dialog box as follows.
New Parameter	Create a new parameter by selecting the attribute you want to add a parameter for. See the New Parameter Dialog box as follows.
New Mandatory Parameter	Create a new mandatory parameter.

CONCAT

The Concat element provides a graphical representation of the 'concat' expression and creates a localizable message by concatenating a list of values.

The Concat element has a fixedlist that contains String objects. You can add other palette elements, for example, String or Rule Reference to the Empty Member (fixedlist) of the Concat element. The Concat element can be added to other palette elements, for example, When or Repeating Rule.

The following table lists specific properties items for this element:

Table 76. Concat properties items	
Name	**Description**
Data Type	The data type of the Concat element. This must correspond with the data type of the attribute. If the user wishes to change the data type into a Timeline, check the Timeline box. If the user wishes to change the data type to a list, check the list box. This is visible in the Business Tab.

Join Lists

The JoinLists element provides a graphical representation of the 'joinlists' expression and creates a new list by joining together some existing lists.

You can add other palette elements, for example, Rule Reference or Choose to the empty member (fixedlist) of the JoinLists element. You can add the JoinLists element to other palette elements, for example, And Rule Group, Or Rule Group or When.

The following table lists specific properties items for this element:

Table 77. Join Lists Properties Items	
Name	**Description**
Single Item	Only one item returned from the element.
Behavior when no items found	Return either one of these results (error, return null) when no items found.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items found.

The following table lists specific pop-up menu items for this element:

Table 78. Join Lists Pop-up Menus items	
Name	**Description**
Remove Duplicates	Remove duplicate items in the Join Lists element.
Concatenate Results	Concatenate the items in the Join Lists element.
Join Inner Lists	Join the lists together to make one list.

Rule Element Reference for Household Units Templates

Two rules elements are available for use in Household Units templates: Household Composition and Household Category. These described in this section.

Household Composition

The **Composition** element is used within Cúram Income Support rules and represents a household composition.

Household Category

This element represents a new household category for a household within CGIS rules.

The following table lists specific pop-up menu items for this element:

Table 79. Household Category Pop-up Menu Items	
Name	**Description**
Add Mandatory Member	Add a new mandatory member to the household category rule element.
Remove Mandatory Member	Remove a mandatory member from the household category rule element.
Add Optional Member	Add a new optional member to the household category rule element.
Remove Optional Member	Remove an optional member from the household category rule element.

Rule Element Reference for Financial Units Templates

Three rule elements are available for use in Financial Units templates: Financial Units, Financial Units Templates, and Financial Unit Member.

Financial Unit

The Financial Unit element is used within Cúram Income Support rules and presents a financial unit.

Financial Unit Category

This element represents a new financial unit category for a financial unit in CGIS rules.

Financial Unit Member

This element represents a new financial unit member for a financial unit in CGIS rules.

Rule Element Reference for Food Assistance Units Templates

Several rule elements are available for use in Food Assistance Units templates including Food Assistance Unit, Food Assistance Single Person Category, and Food Assistance Multi Person Category.

Food Assistance Unit

This template represents a new food assistance unit within Income Support rules. A food assistance diagram must first be configured before it can be fully edited. Certain wizards in the editor require that

the configuration be set before they will provide food assistance specific menu options, for example, the reference wizard. A food assistance diagram configuration can be changed at any time.

Food Assistance Single Person Category
This template represents a new food assistance single person category within Income Support rules.

Food Assistance Multi Person Category
This template represents a new food assistance multiple person category within Income Support rules. The following table lists specific pop-up menu items for this element:

Table 80. Food Assistance Multi Person Category Pop-up Menus items	
Name	**Description**
Remove Meal group	Remove a meal group from the Food Assistance Multi Person Category rule element.
Remove Relatives	Remove relatives from the Food Assistance Multi Person Category rule element.
Remove Discard	Remove discard from the Food Assistance Multi Person Category rule element.
Remove Head of Household Members	Remove a head of household members from the Food Assistance Multi Person Category rule element.
Remove Optional Member	Remove an optional member from the Food Assistance Multi Person Category rule element.

Meal Group Members
This template represents new food assistance meal group members within Income Support rules.

Relatives
This template represents new food assistance relatives within Income Support rules.

Discard
This template represents new food assistance discard within Income Support rules.

Member for Household Unit
This template represents a new food assistance member of household unit within Income Support rules.

Optional Members
This template represents new food assistance optional members within Income Support rules.

Exceptions
This template represents new food assistance exceptions within Income Support rules.

Discard

This template represents new food assistance discard within Income Support rules.

Rule Element Reference for Decision Table Templates

Definitions for the Decision Table element including the available options, properties and menu items. The Decision Table element provides a graphical representation of the 'decision table' expression.

Decision Table

When a Decision Table is dragged onto a rule or an attribute the **Create Decision Table** wizard is displayed. You must set the following options:

Number of Rows
Number of rows that will be contained in the Decision table, the maximum number of rows is 99

Result Type

Return type of the decision table, the result type must match with the result type of the rule or attribute containing the decision table

Rule Class

Select the current rule class or change rule classes.

When you click **Next**, you can use an existing rule; or choose to create a new rule.

The following table lists specific properties items for this element:

Table 81. Decision Table Properties Items

Name	Description
Class	The name of the rule class that is chosen as a type. This is visible in the Business Tab.
Attribute	The name of the attribute containing the Decision Table. This is visible in the Business Tab
Single Item	Only one item returned from the element. This is visible in the Technical Tab.
Behavior when no items found	Return either one of these results (error, return null) when no items found. This is activated when the 'Single Item' box is checked.
Behavior when multiple items found	Return either one of these results (error, return null, return first, return last) when multiple items found. This is activated when the 'Single Item' box is checked.

The following table lists specific pop-up menu items for this element:

Table 82. Decision Table Pop-up Menu Items

Name	Description
Edit Decision Table	Change the Result Type or the associated attribute.
Add new row.	Ad done new row to the decision table.

CER Best Practices

To make your CER rule sets easier to develop, test, and maintain, follow these best practices.
Related information
Testing the rules

Creating a Rule Attribute Description

Each rule class inherits from a CER root rule class. This root class includes a `description` rule attribute that has a default, but not particularly helpful implementation. Creating a meaningful `description` is indispensable when understanding the behavior of your rule set.

The value of a rule object's `description` is output in RuleDoc, and also by the `toString` method on a `RuleObject` (which many Java IDEs use when you click a variable).

You can override the default `description` calculation by explicitly creating a `description` attribute on each of your rule classes. Use the CER editor to create a `description` attribute like you create a normal attribute on any rule class. The CER rule set validator issues a warning if you have a rule class which does not define (or inherit from another defined rule class) a `description` rule attribute.

The `description` attribute is a localizable message and (just like other rule attributes) its calculation can be as simple or as complex as is needed.

Here is an example rule set, where some rule classes provide an implementation of `description`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_description"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="lastName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Override the default description -->
    <Attribute name="description">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- Concatenate the person's first and last names -->
        <concat>
          <fixedlist>
            <listof>
              <javaclass name="Object"/>
            </listof>
            <members>

              <reference attribute="firstName"/>
              <String value=" "/>
              <reference attribute="lastName"/>
            </members>
          </fixedlist>
        </concat>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Income">
    <!-- The person to which this
         income record relates. -->
    <Attribute name="person">
      <type>
        <ruleclass name="Person"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
    <Attribute name="startDate">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Override the default description -->
```

```
    <Attribute name="description">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- Concatenate the person's description and the start
 date-->
        <concat>
          <fixedlist>
            <listof>
              <javaclass name="Object"/>
            </listof>
            <members>

              <reference attribute="description">
                <reference attribute="person"/>
              </reference>
              <!-- In a real rule set, this description would use
                   a <ResourceMessage> to avoid hard-coded
                   single-language Strings.  -->
              <String value="'s income, starting on "/>
              <reference attribute="startDate"/>
            </members>
          </fixedlist>
        </concat>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Benefit">
    <!-- NB no override of <description>; the CER rule set validator
         will issue a warning, and rule objects of this class will
         be more difficult to understand in RuleDoc or a Java
         integrated development environment. -->
    <!-- The person to which this
         benefit record relates. -->
    <Attribute name="person">
      <type>
        <ruleclass name="Person"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

</RuleSet>
```

And here is a test class which creates some rule objects (a `Person`, an `Income` and a `Benefit`):

```
package curam.creole.example;

import java.io.File;

import junit.framework.TestCase;
import curam.creole.execution.session.RecalculationsProhibited;
import curam.creole.execution.session.Session;
import curam.creole.execution.session.SessionDoc;
import curam.creole.execution.session.Session_Factory;
import
 curam.creole.execution.session.StronglyTypedRuleObjectFactory;
import curam.creole.ruleclass.Example_description.impl.Benefit;
import
 curam.creole.ruleclass.Example_description.impl.Benefit_Factory;
import curam.creole.ruleclass.Example_description.impl.Income;
import
 curam.creole.ruleclass.Example_description.impl.Income_Factory;
import curam.creole.ruleclass.Example_description.impl.Person;
import
 curam.creole.ruleclass.Example_description.impl.Person_Factory;
```

```
import curam.creole.storage.inmemory.InMemoryDataStorage;
import curam.util.type.Date;

/**
 * Tests the description rule attribute.
 */
public class TestDescription extends TestCase {

  /**
   * Tests the description rule attribute.
   */
  public void testDescriptions() {

    /**
     * Create a new session.
     */
    final Session session =
        Session_Factory.getFactory().newInstance(
            new RecalculationsProhibited(),
            new InMemoryDataStorage(
                new StronglyTypedRuleObjectFactory()));

    /**
     * Create a SessionDoc to report on rule objects.
     */
    final SessionDoc sessionDoc = new SessionDoc(session);

    /*
     * Create a Person rule object.
     */
    final Person person =
        Person_Factory.getFactory().newInstance(session);
    person.firstName().specifyValue("John");
    person.lastName().specifyValue("Smith");

    /*
     * Create an Income rule object.
     */
    final Income income =
        Income_Factory.getFactory().newInstance(session);
    income.person().specifyValue(person);
    income.amount().specifyValue(123);
    income.startDate().specifyValue(Date.fromISO8601("20070101"));

    /*
     * Create a Benefit rule object.
     */
    final Benefit benefit =
        Benefit_Factory.getFactory().newInstance(session);
    benefit.person().specifyValue(person);
    benefit.amount().specifyValue(234);

    /*
     * The .toString method evaluates the description rule
     * attribute
     */
    System.out.println(person.toString());

    /*
     * println calls an object's toString method to print it.
     */
    System.out.println(income);

    /*
     * The benefit rule class does not provide an implementation of
     * the description rule attribute, so we'll get a default
     * description here
     */
    System.out.println(benefit);

    /*
     * Write out SessionDoc for this session.
     */
    sessionDoc.write(new File("./gen/sessiondoc"));
>
  }

}
```

When the test is run, it produces this output, showing the rule object descriptions:

```
John Smith
John Smith's income, starting on 01/01/07 00:00
Undescribed instance of rule class 'Benefit', id '3'
```

At the end of the test, the session's rule objects are output as SessionDoc. The high-level SessionDoc summary shows the rule objects created, and lists each rule object's description:

Example_description

Generated: 13-Jul-2012 11:52:43

External rule objects

Details	Type	Description	Action
details	Example_description.Benefit	Undescribed instance of rule class 'Benefit', id '3'	Created during this session
details	Example_description.Income	John Smith's income, starting on 01/01/07 00:00	Created during this session
details	Example_description.Person	John Smith	Created during this session

Internal rule objects

Details	Type	Description	Action

Figure 16. SessionDoc showing rule object `description` values

The description for the `Benefit` rule object is the default description; in the absence of a good implementation of `description`, someone reading the SessionDoc might have to navigate to the SessionDoc for the `Benefit` rule object in order to make sense of it:

Rule Object

Generated: 13-Jul-2012 11:52:43

Type

Example_description.Benefit

Description

Undescribed instance of rule class 'Benefit', id '3'

Creation

Created externally

Action during this session

Created during this session

Attributes

Name	Declared type	State	Value	Derivation	Depends on	Used by
amount	Number	SPECIFIED	234	• Specified externally.	*None*	*None*
description	Message	CALCULATED	Undescribed instance of rule class 'Benefit', id '3'	• Default rule object description.	*None*	*None*
person	Person	SPECIFIED	John Smith	• Specified externally.	*None*	*None*

Figure 17. SessionDoc for a rule object with no `description` override

Lastly, this screen shot shows how an integrated development environment (such as Eclipse, shown) uses an object's `toString` method when debugging, which (for rule objects) calculates the `description`:
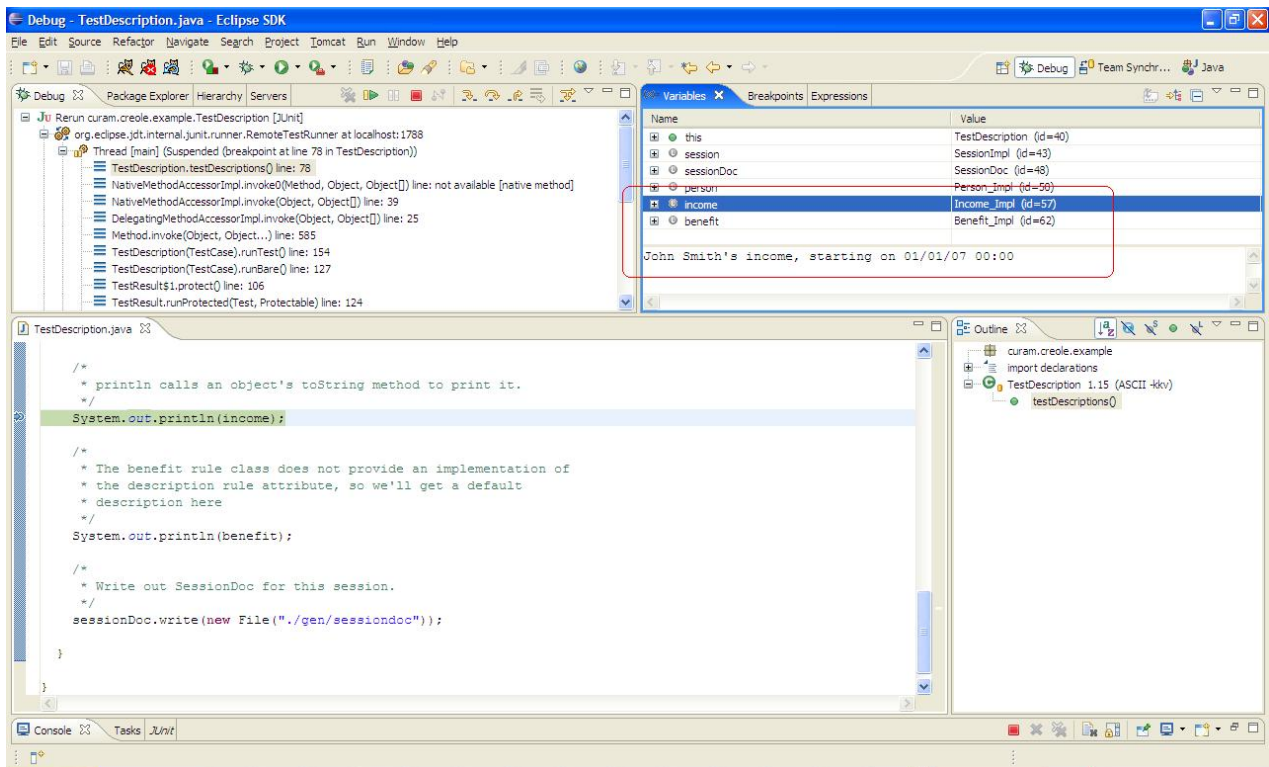
Figure 18. Use of `description` in an integrated development environment

Tip: Remember that the aim of the `description` is to describe a rule object instance, not the rule class itself.

In particular, the calculation of your `description` rule attribute should include data that means that it is easy to distinguish between different rule object instances of your rule class.

Getting a Rule Set Working Quickly

By deferring certain rule development tasks, you can get a CER rule set up and running quickly. This section provides a list of useful shortcuts.

Consider using one or more of the following shortcuts:

• Create empty rule classes, that is, no rule attributes for each of your business concepts. You can add rule attributes later. For example, you can use the CER editor to create an empty rule class and add an attribute to it later.

• Create hard-coded derivations for rule attributes; the business rules can be added later. For example, declaring the calculation of an `isEligible` rule attribute to be `<true>` may enable you to write tests and integrate CER with your own application. You can replace the hard-coded "always eligible" derivation with the real business rules later. For example, you can use the CER editor to drag the "Boolean" rule element (its default value is set to true) to an `isEligible` rule attribute.

• Create messages as plain single-locale hard-coded Strings. You can convert the Strings to localizable messages later. For example, you can use the CER Editor to drag the "Resource Message" or "XML Message" rule element to a rule attribute.

• Use String values instead of application code table values; you can convert the Strings to codes later (and update rules that test for them). For example, you can use the CER editor to drag the "String" rule element to a rule attribute.

Important: Simply taking these shortcuts does *not* remove the need to do the work more rigorously, it merely defers some of this working until after the point at which your rule set is "runnable".

You should *not* hold off on creating tests for your rules; in particular, if you take these shortcuts, a good body of rule set tests will help prevent you introducing certain types of error when you come to undo the shortcuts. Create your tests as you write your rules.

You may also be tempted to hold off on creating `description` rule attributes for your rule classes; however, in the early stages of rule set design, such a short cut can prove to be false economy, as the `description` rule attributes can greatly assist you in debugging rules, for a relatively cheap cost.

Naming Rule Elements

Name CER rule attributes to describe their business meaning. Name a rule attribute in line with the result it provides, instead of describing how the rule attribute provides it.

A rule set name can take a combination of letters (unaccented, in the standard A-Z character range), numbers and under scores. A rule set name should start with an upper case letter.

A class name can take a combination of letters (unaccented, in the standard A-Z character range), numbers and under scores. A class name should start with an upper case letter.

An attribute name can take a combination of letters (unaccented, in the standard A-Z character range), numbers and under scores. An attribute name should start with a lower case letter.

Tip: Use "camelCase" to name your rule elements.

camelCase is the practice of writing compound words or phrases in which the elements are joined without spaces, with each elements initial letter capitalized within the compound and the first letter is either upper or lower case.

Note: Each rule element has a single name which cannot be localized. For localizable descriptions, use the "Label" annotation, as described in "Localizing CER Rule Artifact Descriptions" on page 8.

Using the Reference Expression

Correct use of the `reference` expression is key to the structure of a good CER rule set. Use of the `reference` expression goes hand-in-hand with creating the correct number of rule attributes.

The CER editor provides different types of scenarios to create and use the rule reference element. See "rule" item in "Rule" on page 97.

Striking the right balance (between too few uses of `reference` and too many) is more art than science. However, here are some general guidelines:

- If you some of your expressions are nested very deeply or are otherwise complex, then you may have too few references. Consider breaking up complex expressions by creating rule attributes for a meaningful block of expressions, and using a `reference` to the new rule attribute instead.

- If your requirements have a strong concept or calculation which does not map neatly to a rule attribute, then you should consider creating such a rule attribute.

- If several expressions repeat the same kind of calculations, then your rule set might benefit from the creation of a rule attribute to implement the common logic.

- If a rule attribute is difficult to name, then the rule attribute might be an unnecessary encapsulation of logic, and you might have too many uses of `reference` in your rule set. Consider removing the rule attribute and "inlining" its derivation in the places where it is used, especially in the case where the rule attribute is only referenced from one other calculation.

Using RuleDoc

When your CER rule sets are small in size, you can open them directly in the CER editor and easily understand them in full. However, as your rule set grows in complexity, take advantage of the CER RuleDoc tool to gain insights into the structure and behavior of your rule set.

For information about how to generate RuleDoc from your CER rule sets, see "Generating Rule Documentation" on page 14.

Normalizing Common Rules

As you develop your rule set, you might notice rules that are similar across different parts of your rule set functionality. It is encouraged that you identify common rules and centralize them.

You have two options when centralizing common rules.

Inheritance
Use CER's support for implementation inheritance to allow one rule class to extend another. The CER Editor provides the inheritance mechanism to design the rule. See "extends" item in the Rule Class Properties section in "Rule Element Properties" on page 94.

Containment
Use CER's support for creating new rule objects from rules to allow a rule class to create new instances of another rule class when required. The CER Editor provides the containment mechanism to design the rule. See change rule set and rule class section in "Rule Element Wizards" on page 96.

Sometimes it can be tricky to identify which mechanism to use when centralizing common rules. In general you should use inheritance carefully, only when the sub-rule class represents a business concept which genuinely *"is an"* instance of the business concept represented by the super-class. In particular, CER does not support multiple inheritance.

An example of inheritance is where a person has resources, and each resource might be a building or a vehicle. The `Building` and `Vehicle` rule classes each extend an abstract `Resource` rule class. See the listing in "Rule Classes" on page 33.

You should use containment when the business concept represented by a rule class *"has an"* instance of the business concept represented by the rule class being contained.

An example of containment is where a person has many different age range tests applied. The `Person` rule class creates many instance of `AgeRangeTest`.

If you find you have similar rule classes across different rule sets, you should consider using CER's facilities (since Cúram V6) for allowing one rule set to reference artefacts in other. Place the common rule classes in one or more common rule sets, and place rule classes which are not common in other rule sets.

Removing Unused Rules

As you centralize common rules, you might find that some rule attributes are no longer referenced from other calculations and can be removed from your rule set as part of a decluttering exercise. CER includes support for reporting on rule attributes that are not referenced from any other calculations in your rule set and are candidates for removal.

You can also use the CER editor to delete any rule class and rule attribute. See "Technical View" on page 87.

To run the CER unused attribute report, see "Reporting on Unused Rule Attributes" on page 24.

> ⚠️ **Warning:** It is possible for a rule attribute to be a "top-level" rule attribute that is only referenced from client code. Such attributes might be reported as "unused" by this report, but any seemingly-unused rule attributes should not be removed from your rule set unless you are certain that no client code or tests depend on them.

Changing the Order of Declarations

In general, the order of declaration in your rule sets has no effect on behavior. This section describes the impact of changing the order of rule classes in a rule set, changing the order of rule attributes in a rule class, and changing the order of boolean conditions.

Order of Rule Classes in a Rule Set

You are free to reorder the rule classes in your rule set into whatever order is most useful to you. Reordering rule classes has no effect on the behavior of your rule sets.

Order of Calculated Rule Attributes in a Rule Class

Similarly, you are free to reorder the *calculated* rule attributes in your rule class into whatever order is most useful to you. Reordering *calculated* rule attributes has no effect on the behavior of your rule sets.

Order of Initialized Attributes in a Rule Class

Whenever a rule object instance of the class is created, whether within rules by use of the `create` expression or via Java code using the generated rule classes or the dynamic rule API, the values of all the initialized attributes must be specified *in the order that they are defined in the rule class*.

⚠️ **Warning:** Avoid re-ordering the attributes in an `Initialization` block unless you are prepared to also update all places, in rules or Java code, which create rule object instances of the rule class.

Order of Boolean Conditions

The order of Boolean conditions within an `all` or any expression does not affect the logical value of the result.

However, at runtime, processing of the boolean conditions stops as soon as a result is confirmed. As such, consider ordering the Boolean conditions so that the `all` or any expression tends to obtain its result as quickly as possible.

This means:

- For `all` expressions, placing boolean conditions which are likely to evaluate to *false* earlier in the list.
- For any expressions, placing boolean conditions which are likely to evaluate to *true* earlier in the list.

You can use the CER editor to change the order of the Boolean rule elements within the Any rule element. For example, arrange all Boolean rule elements with true value first.

Creating Rule Objects

Consider carefully how your rule objects are created. In particular, if you are using CER in your own application, decide early which rule objects are created by your application code versus which rule objects are created by your rules.

For more information, see "Creating External and Internal Rule Objects" on page 28.

Passing Rule Objects Instead of IDs

When you use the `create` expression to create an internal rule object, you can pass data to the new rule object by using the initialization block and `specify` elements. For data that includes a reference to external data that has an identifier, for example, a caseID, consider designing the rule class for the created rule object so that it is initialized with a rule object instead of an ID value.

The use of such a rule object for initialization can increase the 'type safety' of your data. It may help prevent another rules designer from creating their own internal rule objects for the same rule class but accidentally passing an ID that represents a different kind of external data.

It is likely that passing the wrong kind of ID will cause rules to fail at runtime, for example, because an attempt to convert a rule object for that ID will not find the underlying data. Whereas passing a rule object for type safety allows the CER rule set validator to detect the problem at design time.

Developing Static Methods

CER includes support for various expressions that likely are to provide the calculations you need. If you have a business calculation that cannot be implemented by using CER expressions, CER supports the `call` expression to allow you to callout from your rule set to a static method on a custom Java class.

The CER editor provides few rule elements, for example, "call" to allow you to define a static method on a custom Java class. See "call" rule element in "Call" on page 111.

The following code is an example rule set that calls out to a Java method:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_StaticMethodDevelopment"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Income">

    <Attribute name="paymentReceivedDate">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Person">

    <Attribute name="incomes">
      <type>
        <javaclass name="List">
          <ruleclass name="Income"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="mostRecentIncome">
      <type>
        <ruleclass name="Income"/>
      </type>
      <derivation>
        <!-- In early development, the method
             identifyMostRecentIncome_StronglyTyped would be used.

             In practice, you would not maintain two versions of
             each method; you would simply weaken the argument
             and return types, and keep a single method.
        -->

        <call class="curam.creole.example.BestPracticeDevelopment"
          method="identifyMostRecentIncome_WeaklyTyped">
          <type>
            <ruleclass name="Income"/>
          </type>
          <arguments>
            <this/>
          </arguments>
        </call>
      </derivation>
    </Attribute>
  </Class>

</RuleSet>
```

When you develop your static method, you can start by using CER's generated Java classes as argument types, return types, or both for the method:

```java
package curam.creole.example;

import java.util.List;

import curam.creole.execution.session.Session;
import
```

```
  curam.creole.ruleclass.Example_StaticMethodDevelopment.impl.Income;
import
 curam.creole.ruleclass.Example_StaticMethodDevelopment.impl.Person;

public class BestPracticeDevelopment {

  /**
   * Identifies a person's most recent income.
   *
   * Note that this calculation can be performed using CER
   * expressions, but is shown here in Java just to illustrate the
   * use of generated types when developing static Java methods for
   * use with CER.
   *
   * This method is suitable only for use in development; for
   * production, see the
   * {@linkplain #identifyMostRecentIncome_WeaklyTyped} below.
   *
   * In practice, you would not maintain two versions of each
   * method; you would simply weaken the argument and return types,
   * and keep a single method.
   */
  public static Income identifyMostRecentIncome_StronglyTyped(
      final Session session, final Person person) {

    Income mostRecentIncome = null;
    final List<? extends Income> incomes =
        person.incomes().getValue();
    for (final Income current : incomes) {
      if (mostRecentIncome == null
          || mostRecentIncome.paymentReceivedDate().getValue()
              .before(current.paymentReceivedDate().getValue())) {
        mostRecentIncome = current;
      }
    }

    return mostRecentIncome;
  }
}
```

When the Java method is working to your satisfaction, you must weaken the types to use dynamic rule objects instead of the generated Java classes (which is not be used in a production environment):

```
  /**
   * Identifies a person's most recent income.
   *
   * Note that this calculation can be performed using CER
   * expressions, but is shown here in Java just to illustrate the
   * use of generated types when developing static Java methods for
   * use with CER.
   *
   * This method is suitable for use in production; for initial
   * development, see
   * {@linkplain #identifyMostRecentIncome_StronglyTyped} above.
   *
   * In practice, you would not maintain two versions of each
   * method; you would simply weaken the argument and return types,
   * and keep a single method.
   */
  public static RuleObject identifyMostRecentIncome_WeaklyTyped(
      final Session session, final RuleObject person) {

    RuleObject mostRecentIncome = null;
    final List<? extends RuleObject> incomes =
        (List<? extends RuleObject>) person.getAttributeValue(
            "incomes").getValue();
    for (final RuleObject current : incomes) {
      if (mostRecentIncome == null
          || ((Date) mostRecentIncome.getAttributeValue(
              "paymentReceivedDate").getValue())
              .before((Date) current.getAttributeValue(
                  "paymentReceivedDate").getValue())) {
        mostRecentIncome = current;
      }
    }

    return mostRecentIncome;

  }
```

Important: If you change the implementation of a static method, CER does not know to recalculate attribute values automatically that were calculated by using the old version of your static method.

After a static method is used in a production environment for stored attribute values, rather than changing the implementation you need to create a new static method (with the required new implementation), and change your rule sets to use the new static method. When you publish your rule set changes to point to the new static method, CER automatically recalculates all instances of the affected attribute values.

Any static method (or property method) started from CER:

- needs to depend only n the data that is passed to it (that is, its behavior must be deterministic, based on its input parameters). It must not get data from other sources such as the database, environment variables, or variable globals, as to do so means that CER is unable to detect when such data changes, and thus recalculations would become unreliable; and

- needs to not initiate any side-effects (such as writing data to the database), as CER makes no guarantees about the order in which processing occurs nor how often the static/property methods are started.

Avoiding Common Pitfalls in Tests

Writing JUnit tests for your CER rule sets is straightforward. However, there are a number of pitfalls that you must be aware of. This section provides some test code examples which at first glance look fine, but each of them fail to produce the required behavior.

Avoid JUnit assertEquals

JUnit tests inherit assertion methods for testing. Typically you will assert that an *expected* result is matched by an *actual* result.

A common pitfall is to use JUnit's `assertEquals`, only to find that it does not work correctly for numerical comparisons (and with a slightly confusing error message to boot).

CER converts all instances of Number to its own numerical format (backed by `java.math.BigDecimal`) before processing, to ensure no loss of precision. This conversion can be problematic and unintuitive if you use JUnit's `assertEquals` method.

CER includes a replacement in a helper class. Use `CREOLETestHelper.assertEquals`, which will correctly compare numbers of any type.

For other data types, `CREOLETestHelper.assertEquals` behaves identically to JUnit's `assertEquals`, so in general it is good practice to use `CREOLETestHelper.assertEquals` throughout your tests, to avoid possible confusion (even in the places where technically it is unnecessary).

```
public void creoleTestHelperNotUsed() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    flexibleRetirementYear.retirementCause().specifyValue(
        "Reached statutory retirement age.");

    /**
     * Will not work - getValue returns CER's own numerical handler,
     * but 65 is an integer.
     *
     * JUnit will report the somewhat confusing message:
     * junit.framework.AssertionFailedError: expected:<65> but
     * was:<65>
     *
     * Use CREOLETestHelper.assertEquals instead.
     */
    assertEquals(65, flexibleRetirementYear.ageAtRetirement()
        .getValue());

}
```

Use .getValue()

The CER test code generator creates a Java interface for each rule class, and an accessor method on the interface for each rule attribute.

This generated accessor method returns a CER `AttributeValue`, *not* the attribute's value directly. To obtain the value, you must call the `.getValue()` method on the `AttributeValue`.

If you forget to use `.getValue()` in a test, then your test will probably compile but fail to behave correctly when it is run.

```
public void getValueNotUsed() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    flexibleRetirementYear.retirementCause().specifyValue(
        "Reached statutory retirement age.");

    /**
     * Will not work - ageAtRetirement() is a calculator, not a
     * value.
     *
     * JUnit will report the message:
     * junit.framework.AssertionFailedError: expected:<65> but
     * was: <Value: 65>
     *
     * Remember to use .getValue() on each attribute calculator!
     */
    assertEquals(65, flexibleRetirementYear.ageAtRetirement());

  }
```

Note: In the example, the value of the `AttributeValue` shows as the String "Value: 65", rather than the number 65, which is what `.getValue()` would have returned.

Specify All Values Required

In your tests, you need only to specify the values to be accessed during rules execution.

However, it can be easy to forget to specify a value. If so, when CER attempts a calculation, it might encounter an attribute whose derivation is `<specified>` but for which no value has been specified in your test code, and CER will report a stack of errors:

```
public void valueNotSpecified() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    /**
     * Will not work - a value required for calculation was marked
     * as <specified> but no value was specified for it.
     *
     * CER will report a stack of messages:
     * <ul>
     *
     * <li> Error calculating attribute 'ageAtRetirement' on rule
     * class 'FlexibleRetirementYear' (instance id '1', description
     * 'Undescribed instance of rule class
     * 'FlexibleRetirementYear', id '1'). </li>
     *
     * <li>Error calculating attribute 'retirementCause' on rule
     * class 'FlexibleRetirementYear' (instance id '1', description
     * 'Undescribed instance of rule class
     * 'FlexibleRetirementYear', id '1'). </li>
     *
     * <li>Value must be specified before it is used (it cannot be
     * calculated).</li>
     *
     * </ul>
     *
     *
     * Remember to specify all values required by calculations!
```

```
      */
    CREOLETestHelper.assertEquals(65, flexibleRetirementYear
        .ageAtRetirement().getValue());

  }
```

Do Not Specify the Same Value More than Once

CER allows you to specify a value which would otherwise be calculated.

However, when using the `RecalculationsProhibited` strategy, CER raises a runtime error if you try to specify the value of an attribute (on a particular rule object) more than once, once the value has been specified, it cannot be changed. To do so, might mean that previously-performed calculations would now be incorrect.

```
public void valueSpecifiedTwice() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    flexibleRetirementYear.retirementCause().specifyValue(
        "Reached statutory retirement age.");

    /**
     * Will not work - the same attribute value cannot be specified
     * a second time.
     *
     * CER will report the message: A value cannot be specified,
     * as the current state of this calculator is 'SPECIFIED'.
     *
     * Do not attempt to specify the same value twice!
     */
    flexibleRetirementYear.retirementCause().specifyValue(
        "Lottery winner");

  }
```

Specify the Correct Type of Value for an Attribute

Each CER `AttributeValue` has a `specifyValue` method to allow the attribute's value to be specified (rather than calculated). The method takes any value, but if you specify the wrong type of value, CER will raise a runtime error as shown:

```
public void incorrectValueType() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    /**
     * Will not work - retirementCause() expects a String, not a
     * Number.
     *
     * CER will report the message: Attempt to set the value '123'
     * (of type 'java.lang.Integer') on attribute 'retirementCause'
     * of rule class 'FlexibleRetirementYear' (which expects a
     * 'java.lang.String').
     */
    flexibleRetirementYear.retirementCause().specifyValue(123);

  }
```

Note: Technical readers might wonder why `AttributeValue.specifyValue` does not use Java 5 generics to restrict the value type it can receive. If rule class A extends rule class B, then A is free to override the derivation of any of B's attributes. A is also free to redeclare any of B's attributes to be a more-restrictive type (i.e. A's declaration of the attribute specifies its type to be a subtype of that declared by B).

The generated Java interface for A extends the generated Java interface for B. Because the accessors return calculators, rather than the value type directly, all interfaces must use wildcard extension so that

the compiler will allow A's declaration of the attribute's accessor to extend that of B's. Because wildcard extension is used, `specifyValue` cannot restrict to a type, and thus must be declared to receive any `Object`.

From another point-of-view, if a Java class C were to extend Java D, then C can define a more restrictive return type for one of D's getters, but cannot restrict D's setters to a subtype - C must implement D's setter, and detect any undesirable values at runtime. To do so might arguably violate Liskov's substitution principle).

Another justification is that when purely-dynamic rule objects are used, that is, in an interpreted session, compile-time restriction of values cannot be used.

Thus, CER uses its knowledge of declared attribute types to detect incorrect values at runtime, not compile time.

Create all Rule Objects for a Session Before Running get Value Comparisons

Your rule set tests can set up any number of rule objects in a CER session before going on to check any number of calculated values on those rule objects.

However, once calculations have commenced, the `RecalculationsProhibited` strategy prevents the creation of any rule objects which invalidate the value of a previously-calculated "readall" on page 183 calculations.

Structure your tests so that the creation of all your test rule objects occurs *before* any calculations, that is, before any execution of `getValue`. In practice this is not unduly restrictive.

If your test attempts to create a new rule object in a session after a calculation has occurred, then (if previously-calculated `readall` calculations are affected), the `RecalculationsProhibited` strategy will raise a runtime error:

```
public void newObjectsAddedAfterCalculationsStarted() {

    final FlexibleRetirementYear flexibleRetirementYear =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

    flexibleRetirementYear.retirementCause().specifyValue(
        "Reached statutory retirement age.");

    /**
     * Calculate the age at retirement and test its value
     */
    CREOLETestHelper.assertEquals(65, flexibleRetirementYear
        .ageAtRetirement().getValue());

    /**
     * Create another rule object.
     */

    /**
     * May not work - new rule objects added to the session once
     * calculations have started could invalidate earlier
     * <code><readall></code> calculations.
     *
     * {@linkplain RecalculationsProhibited} may report the
     * message: "Cannot create new rule objects for this session,
     * because this session has already accepted a calculation
     * request."
     *
     * To avoid this problem, create all your rule objects before
     * attempting any calculations!
     */
    final FlexibleRetirementYear flexibleRetirementYear2 =
        FlexibleRetirementYear_Factory.getFactory().newInstance(
            session);

}
```

⚠ **Warning:** If your rule set does not *currently* contain any `readall` expressions, you might get away with not structuring your tests so that all rule object creation occurs before any calculations.

However, if you change your rule set in the future to contain `readall` expressions, you will need to restructure your tests at that point. To avoid rework, always structure your tests so that all rule object creation is performed before calculations begin.

CER XML Dictionary

A description of the elements that make up the CER language for rule sets.

Rule Set

A rule set specifies its name and contains any number of rule classes and include statements. Optionally, a rule set can contain annotations

The XML structure of a rule set and its elements is constrained by the CER `RuleSet.xsd` schema. This schema is dynamically constructed, so that extensions to CER can contribute expressions and annotations to the schema.

Here is a sample outline of a rule set:

```
RuleSet
  Annotations (optional)
    ...
    ...
  Include
    ...
  Include
    ...
  ... more Include statements
  Class
    Annotations (optional)
      ...
      ...
  Initialization (optional)
      Attribute
        Annotations (optional)
          ...
          ...
      type
        ...
      Attribute
        type
          ...
      ... more initialized attributes
    Attribute
      Annotations (optional)
        ...
        ...
    type
        ...
      derivation
        (expression)
          Annotations (optional)
            ...
            ...
          (sub-expression)
          ...
    Attribute
      type
        ...
      derivation
        ...
      ... more calculated attributes
  ... more rule classes
```

Include Statement

You might find it convenient to break a large rule set into smaller pieces for ease of parallel development or re-use. Each rule set can contain include statements to pull in other rule sets and classes.

The root element in an included item must be either:

Class
> A single rule class.

Ruleset
> An entire rule set, which itself may contain its own `Include` statements which will be processed recursively.

Different types of `Include` statements are supported:

RelativePath
> Includes an XML file with a path relative to the containing file; this mechanism may be useful during stand-alone development of the rule set by developers using a file-based development environment;

Classpath
> Includes an XML file which is present at the named location on the runtime classpath; this mechanism may be useful to refer to common rule sets which rarely change and are built into the application

Tip: Within a rule set, there is no meaning attached to the order that `Include` statements are specified. You are free to reorder `Include` statements within a rule set without affecting the behavior of the rule set.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Include"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- This rule class is defined directly in this rule set -->
  <Class name="Person">
    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <!-- Include a rule set defined in another file.

      When assembled into a single rule set, the
      names of all the rule classes must be unique. -->
  <Include>
    <RelativePath value="./HelloWorld.xml"/>
  </Include>

</RuleSet>
```

See "Consolidating Rule Sets" on page 24 for how to collapse a rule set which contains `RelativePath` inclusions into a single rule set file.

Rule Class

A rule class defines the behavior of its rule object instances. A rule class specifies its name, which must be unique among the rule classes in the rule set, and whether it is abstract. A rule class optionally contains initialized attributes.

Initialized Attributes

Initialized attributes are a block of attributes that must have their values specified whenever a rule object instance of the class is created. They also include zero or more calculated `Attribute` statements, each describing a value that the rule class can calculate.

The `Initialization` block contains one or more `Attribute` statements, each specifying the attribute's `type`, but *not* specifying any `derivation`.

Whenever a rule object instance of the class is created, whether within rules by use of the `create` expression or via Java code using the generated rule classes or the dynamic rule API, the values of all the initialized attributes must be specified *in the order that they are defined in the rule class*.

warning: As such, you should avoid re-ordering the attributes in an `Initialization` block unless you are prepared to also update all places (in rules or Java code) which create rule object instances of the rule class.

Calculated Attributes

Calculated attributes are listed directly within the rule class.

Tip: Within a rule class, there is no meaning attached to the order that calculated attributes are specified. You are free to reorder calculated attributes within a rule class without affecting the behavior of the rule class.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_RuleClass"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Initialization>
      <!-- Initialized attributes each contain a type
           but no derivation.

           You should NOT arbitrarily reorder
           initialized attributes. -->
      <Attribute name="firstName">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
      <Attribute name="age">
        <type>
          <javaclass name="Number"/>
        </type>
      </Attribute>
    </Initialization>

    <!-- Each calculated attribute specifies both
         a type and a derivation.

         You are free to arbitrarily reorder
         calculated attributes. -->
    <Attribute name="isAdult">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <compare comparison=">=">
          <reference attribute="age"/>
          <Number value="18"/>
        </compare>
      </derivation>
    </Attribute>

    <Attribute name="isSeniorCitizen">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <compare comparison=">=">
          <reference attribute="age"/>
          <Number value="65"/>
        </compare>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Attribute

Each attribute specifies its name, which must be unique amongst the rule attributes defined on or inherited by the rule class, and contains a type and a derivation.

Each `Attribute` contains:

type
> Defines the type of value that this attribute provides (see "Handling Data Types" on page 33).

derivation
> Defines how the attribute calculates its value. Each `derivation` contains a single CER expression (see "Full Alphabetical Listing of Expressions" on page 139).

Important: There are special marker expressions which can alter the semantics of the rule attribute. For more information, see "Markers" on page 138.

Expressions

CER supports a wide range of expressions. The expressions are listed alphabetically in this section and are also logically grouped here for reference. Note that some expressions intentionally appear in more than one logical group.

Boolean Logic

List of expressions that evaluate to a boolean value.

- "true" on page 207
- "false" on page 165
- "all" on page 142
- "any" on page 144
- "not" on page 175

Value Comparison

List of expressions that support value comparisons.

- "equals" on page 162
- "compare" on page 153
- "sort" on page 194

Constants

List of expressions that provide literal constant values.

- "true" on page 207
- "false" on page 165
- "null" on page 176
- "String" on page 196
- "Number" on page 177
- "Date" on page 159
- "Code" on page 152
- "FrequencyPattern" on page 170

Conditional Logic

List of expressions to support conditional logic.

- "choose" on page 149

List Aggregations

List of expressions that aggregate a list of values into a derived value.

- all
- any
- sum
- min
- max
- concat
- singleitem

For further operations directly provided from the Java `java.util.List` interface, see "Useful List Operations" on page 212.

List Transformations

List of expressions that transform a list to create a new list.

- "dynamiclist" on page 159
- "fixedlist" on page 167
- "filter" on page 165
- "joinlists" on page 171
- "removeduplicates" on page 189
- "sort" on page 194
- "sublists" on page 196

Localizable Messages

List of expressions that allow the creation of messages which can be displayed in the end-user's language/locale.

- "concat" on page 154
- "ResourceMessage" on page 190
- "XmlMessage" on page 207

Numerical Calculations

List of expressions that support numerical calculations.

- "Number" on page 177
- "arithmetic" on page 146
- "periodlength" on page 177
- "sum" on page 198
- "max" on page 172
- "min" on page 174

References

List of expressions that allow a calculation to refer to another item.

- "reference" on page 187
- "current" on page 157
- "this" on page 199

Creation

The create expression allows the creation of a new rule object.

- "create" on page 155

Retrieval

The readall expression allows the retrieval of rule objects.

- "readall" on page 183

Java Callouts

List of expressions that allow the invocation of Java code to perform a calculation.

- "property" on page 179
- "call" on page 148

Markers

List of expressions that are special markers, not calculations.

- "abstract" on page 139
- "specified" on page 195

Timelines

List of expressions that process CER timelines.

For more details on CER timelines, see "Handling Data that Changes Over Time" on page 42 in this guide.

- "Interval" on page 170
- "Timeline" on page 200
- "existencetimeline" on page 164
- "intervalvalue" on page 171
- "timelineoperation" on page 202

Product Delivery Eligibility and Entitlement

List of expressions that provide business-specific calculations for eligibility and entitlement of product delivery cases.

- "combineSuccessionSets" on page 153
- "legislationChange" on page 172
- "rate" on page 182

For a description of these expressions, see Developing with Eligibility and Entitlement by using Cúram Express Rules

Full Alphabetical Listing of Expressions

This section alphabetically lists and defines all of the expressions that are included with CER. For helpful categorizations of these expressions, see the logical groupings of expressions that were described previously.

Note: Some expressions are for business-specific derivations in the application. These expressions are included here and you are referred to other information that describe the expressions in their business context.

For brevity, example rule sets are shown without annotations. In practice, rule sets that are saved by using the CER editor contain annotations for diagram and description information.

abstract

A marker expression to denote that the attribute's derivation must be specified on concrete sub-class or one of their superclasses.

If one or more attributes in a rule class is marked `abstract`, then the CER rule set validator will insist that the class itself is marked *abstract="true"*, and prevent the rule class from being used in any <u>"create"</u> <u>on page 155</u> expressions.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_abstract"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- Base class for all types of benefit.
      Every concrete subclass has its own
      calculation of "name" and "isEligible". -->
  <Class name="Benefit" abstract="true">
    <Initialization>
      <!-- The person for which benefit eligibility
        is being determined. -->
      <Attribute name="person">
        <type>
          <ruleclass name="Person"/>
        </type>
      </Attribute>
    </Initialization>


    <!-- The name of this type of benefit -->
    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <abstract/>
      </derivation>
    </Attribute>

    <!-- Whether the person is eligible for this benefit. -->
    <Attribute name="isEligible">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <abstract/>
      </derivation>
    </Attribute>

  </Class>

  <!-- A concrete subclass of Benefit.
      Contains concrete derivations for the inherited
      abstract attributes. -->
  <Class name="MedicalBenefit" extends="Benefit">
    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="Medical Benefit"/>
      </derivation>
```

```
        </Attribute>
    <Attribute name="isEligible">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <all>
          <fixedlist>
            <listof>
              <javaclass name="Boolean"/>
            </listof>
            <members>
              <!-- NB the person attribute is inherited from Benefit
-->
              <reference attribute="isPoor">
                <reference attribute="person"/>
              </reference>
              <reference attribute="isSick">
                <reference attribute="person"/>
              </reference>
            </members>
          </fixedlist>

        </all>

      </derivation>
    </Attribute>
</Class>


<!-- Another concrete subclass of Benefit,
  with different concrete derivations for the inherited
  abstract attributes. -->
<Class name="NeedyBenefit" extends="Benefit">
    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <String value="Medical Benefit"/>
      </derivation>
    </Attribute>
    <Attribute name="isEligible">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <all>
          <fixedlist>
            <listof>
              <javaclass name="Boolean"/>
            </listof>
            <members>
              <reference attribute="isPoor">
                <reference attribute="person"/>
              </reference>
              <any>
                <fixedlist>
                  <listof>
                    <javaclass name="Boolean"/>
                  </listof>
                  <members>
                    <reference attribute="isHungry">
                      <reference attribute="person"/>
                    </reference>
                    <reference attribute="isDeprived">
                      <reference attribute="person"/>
                    </reference>
                  </members>
                </fixedlist>
              </any>
            </members>
          </fixedlist>

        </all>

      </derivation>
    </Attribute>
</Class>

<Class name="Person">
```

```
<Attribute name="isPoor">
  <type>
    <javaclass name="Boolean"/>
  </type>
  <derivation>
    <specified/>
  </derivation>
</Attribute>

<Attribute name="isSick">
  <type>
    <javaclass name="Boolean"/>
  </type>
  <derivation>
    <specified/>
  </derivation>
</Attribute>

<Attribute name="isHungry">
  <type>
    <javaclass name="Boolean"/>
  </type>
  <derivation>
    <specified/>
  </derivation>
</Attribute>

<Attribute name="isDeprived">
  <type>
    <javaclass name="Boolean"/>
  </type>
  <derivation>
    <specified/>
  </derivation>
</Attribute>


<!-- A list of all the benefits for
     which the person is being assessed.  -->
<Attribute name="allBenefits">
  <type>
    <javaclass name="List">
      <ruleclass name="Benefit"/>
    </javaclass>
  </type>
  <derivation>
    <fixedlist>
      <listof>
        <ruleclass name="Benefit"/>
      </listof>
      <members>
        <!-- Create instances of the concrete rule classes -->
        <create ruleclass="MedicalBenefit">
          <this/>
        </create>
        <create ruleclass="NeedyBenefit">
          <this/>
        </create>
      </members>
    </fixedlist>

  </derivation>
</Attribute>

<!-- The benefits for which this person
     is eligible.

     Note that the list is of the abstract
     rule class "Benefit", but that each
     concrete instance determines its
     eligibility in its own way.      -->
<Attribute name="eligibleBenefits">
  <type>
    <javaclass name="List">
      <ruleclass name="Benefit"/>
    </javaclass>
  </type>
  <derivation>
    <filter>
      <list>
        <reference attribute="allBenefits"/>
      </list>
```

```
            <listitemexpression>
              <reference attribute="isEligible">
                <current/>
              </reference>
            </listitemexpression>
          </filter>
        </derivation>
      </Attribute>

    </Class>

  </RuleSet>
```

all

The `all` expression operates on a list of Boolean values to determine whether all of the values are *true*.

Calculation stops at the first *false* value encountered in the list. If the list is empty, this expression returns *true*.

The list of Boolean values is typically provided by a "fixedlist" on page 167 or "dynamiclist" on page 159.

Tip: The ordering of items in the list makes no difference to the value of this expression; however, for performance reasons, you may wish to structure a "fixedlist" on page 167 so that "fail fast" values are nearer the top of the list, and any values which may be more costly to calculate are nearer the bottom of the list.

Note: Since Cúram V6, CER no longer reports errors in child expressions in situations where the error does not affect the overall result.

For example, if a fixed list of three Boolean attributes has these values:

- `true`;
- <error during calculation>; and
- `false`

then the calculation of the value of `all` for these values will return `false`, because at least one of the items is false (namely the third in the list), regardless of the second item returning an error.

By contrast, if another fixed list of three Boolean attributes has these values:

- `true`;
- <error during calculation>; and
- `true`

then the calculation of the value of `all` for these values will return the error reported by the second item in the list, as this error prevents the determination of whether all items have the value `true`.

The application property `curam.creole.expression.immediateexceptionreporting` can be used to override this default error-reporting behaviour. This application property will dictate whether or not errors should be reported as soon as they are encountered during evaluation of the `all` expression. The default value is 'NO'. If this value is set to 'YES', then exceptions generated during the evaluation of the `all` expression will be reported as soon as they are encountered. This property is intended for use as a troubleshooting aid for diagnostic purposes.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_all"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="isLoneParent">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- Example of <all> operating on a <fixedlist> -->
        <!-- To be considered a "lone parent", a person must
             be both unmarried and have at least one child -->
```

```
        <all>
          <fixedlist>
            <listof>
              <javaclass name="Boolean"/>
            </listof>
            <members>
              <!-- We happen to know that most people on our
database
                  are married, so we test this condition first.

                  If it so happens that the isMarried value is not
                  specified for a Person, then if that Person has
                  no children then the <all> will return false;
                  otherwise it will return an error indicating that
                  the value of isMarried was not specified.
                  -->
              <not>
                <reference attribute="isMarried"/>
              </not>
              <not>
                <property name="isEmpty">
                  <object>
                    <reference attribute="children"/>
                  </object>
                </property>
              </not>
            </members>
          </fixedlist>
        </all>
      </derivation>
    </Attribute>

    <Attribute name="hasNoYoungChildren">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- Example of <all> operating on a <dynamiclist>.

            If it so happens that one child's age cannot be
            calculated, and there is at least one child under 5,
            then the <all> will return false; otherwise, it
            will return the error showing why the child's age could
            not be calculated.
        -->

        <!-- Check whether the children are all over 5 years of age
-->
        <all>
          <dynamiclist>
            <list>
              <reference attribute="children"/>
            </list>
            <listitemexpression>
              <compare comparison="&gt;">
                <reference attribute="age">
                  <current/>
                </reference>
                <Number value="5"/>
              </compare>
            </listitemexpression>
          </dynamiclist>
        </all>
      </derivation>
    </Attribute>

    <!-- The children of this person - each child is a person too!
-->
    <Attribute name="children">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="isMarried">
      <type>
        <javaclass name="Boolean"/>
```

```
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <Attribute name="age">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

  </Class>

</RuleSet>
```

any

The any expression operates on a list of Boolean values to determine whether any of the values are *true*.

Calculation stops at the first *true* value encountered in the list. If the list is empty, this expression returns *false*.

The list of Boolean values is typically provided by a or .

Tip: The ordering of items in the list makes no difference to the value of this expression. However, for performance reasons, you might want to structure a so that 'succeed fast' values are nearer the top of the list, and any values which may be more costly to calculate are nearer the bottom of the list.

Note: CER no longer reports errors in child expressions in situations where the error does not affect the overall result.

For example, if a fixed list of three Boolean attributes has these values:

- `false`
- <error during calculation>
- `true`

then the calculation of the value of any for these values will return `true`, because at least one of the items is true (namely the third in the list), regardless of the second item returning an error.

By contrast, if another fixed list of three Boolean attributes has these values:

- `false`
- <error during calculation>
- `false`

then the calculation of the value of any for these values will return the error reported by the second item in the list, as this error prevents the determination of whether any items have the value `true`.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_any"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="qualifiesForFreeTravelPass">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- Example of <any> operating on a <fixedlist> -->
        <!-- To qualify for a free travel pass, the person
             must be aged, blind or disabled -->
        <any>
          <fixedlist>
            <listof>
```

```xml
            <javaclass name="Boolean"/>
          </listof>
          <members>
            <!-- We happen to know that most people on our
                 database are senior citizens, so we test
                 this condition first.

                 If it so happens that the isBlind value is not
                 specified for a Person, then if that Person is
                 disabled then the <any> will return false;
                 otherwise it will return an error indicating that
                 the value of isBlind was not specified.
                 -->

            <compare comparison="&gt;=">
              <reference attribute="age"/>
              <Number value="65"/>
            </compare>
            <reference attribute="isBlind"/>
            <reference attribute="isDisabled"/>
          </members>
        </fixedlist>
      </any>
    </derivation>
  </Attribute>

  <Attribute name="qualifiesForChildBenefit">
    <type>
      <javaclass name="Boolean"/>
    </type>
    <derivation>
      <!-- Example of <any> operating on a <dynamiclist>.

           If it so happens that one child's age cannot be
           calculated, and there is at least one child under 16,
           then the <any> will return true; otherwise, it
           will return the error showing why the child's age could
           not be calculated.

        -->
      <!-- To qualify for child benefit, this person must
           have one or more children aged under 16. -->
      <any>
        <dynamiclist>
          <list>
            <reference attribute="children"/>
          </list>
          <listitemexpression>
            <compare comparison="&lt;">
              <reference attribute="age">
                <current/>
              </reference>
              <Number value="16"/>
            </compare>
          </listitemexpression>
        </dynamiclist>
      </any>
    </derivation>
  </Attribute>

  <!-- The children of this person - each child is a person too!
-->
  <Attribute name="children">
    <type>
      <javaclass name="List">
        <ruleclass name="Person"/>
      </javaclass>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <Attribute name="isBlind">
    <type>
      <javaclass name="Boolean"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>
```

```
        <Attribute name="isDisabled">
          <type>
            <javaclass name="Boolean"/>
          </type>
          <derivation>
            <specified/>
          </derivation>
        </Attribute>

        <Attribute name="age">
          <type>
            <javaclass name="Number"/>
          </type>
          <derivation>
            <specified/>
          </derivation>
        </Attribute>

    </Class>

</RuleSet>
```

arithmetic

The arithmetic expression performs an arithmetical calculation on two numbers (a left-side number and a right-side number) and optionally rounds the result to the specified number of decimal places.

The following are the supported operations:

- **Addition**

  left side + right side

- **Subtraction**

  left side - right side

- **Multiplication**

  left side * right side

- **Division**

  left side / right side.

If rounding is required, then you must specify:

- the number of decimal places to round to; and

- the rounding mode, that is, the direction in which to round when rounding is performed. See the JavaDoc for RoundingMode for the list of supported rounding modes and a detailed explanation of their behavior.

warning: For division operations, in general you should supply a rounding mode and a number of decimal places. If you do not, and at runtime an exact result cannot be calculated, then a runtime error will occur.

The CER rule set validator will issue a warning if it detects any division operation in your rule set which does not specify rounding specified.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_arithmetic"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="ArithmeticExampleRuleClass">

    <!-- 3 + 2 = 5 -->
    <Attribute name="addANumberToAnother">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="+">
          <Number value="3"/>
          <Number value="2"/>
        </arithmetic>
      </derivation>
```

```
      </Attribute>

    <!-- 3 - 2 = 1 -->
    <Attribute name="subtractANumberFromAnother">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="-">
          <Number value="3"/>
          <Number value="2"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <!-- 3 * 2 = 6 -->
    <Attribute name="multiplyANumberByAnother">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="*">
          <Number value="3"/>
          <Number value="2"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <!-- 3 / 2 = 1.5 -->
    <!-- Because the division is by 2,
         we can get away without rounding.
         A warning will still be issued by the
         CER rule set validator, though.    -->
    <Attribute name="divideANumbersByAnother">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="/">
          <Number value="3"/>
          <Number value="2"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <!-- (3 + 2) * 4 = 20 -->
    <Attribute name="chainedArithmetic">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="*">
          <arithmetic operation="+">
            <Number value="3"/>
            <Number value="2"/>
          </arithmetic>
          <Number value="4"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <!-- 1.23 + 3.45 = 4.68,
         = 4.7 when rounded to the nearest 1 decimal place-->
    <Attribute name="roundedAddition">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic decimalPlaces="1" operation="+"
          rounding="half_up">
          <Number value="1.23"/>
          <Number value="3.45"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <!-- 2 / 3, = 0.667 to 3 decimal places -->
    <!-- If no rounding is specified,
         then a runtime error will occur -->
    <Attribute name="roundedDivision">
      <type>
        <javaclass name="Number"/>
```

```
      </type>
      <derivation>
        <arithmetic decimalPlaces="3" operation="/"
          rounding="half_up">
          <Number value="2"/>
          <Number value="3"/>
        </arithmetic>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

call

The `call` expression calls out to a static Java method to perform a complex calculation.

The `call` expression declares:

- **type**

  The data type of the value returned (see "Handling Data Types" on page 33); and

- **arguments (optional)**

  A list of values to pass as arguments.

The Java method must be on a class which is on the classpath at rule set validation time. The first argument to the method must be a `Session` object, and the remaining arguments must match those specified in the rule set.

warning: You should ensure that any Java code invoked by a `call` expression does *not* attempt to mutate any values on rule object attributes.

In general, CER rule sets use immutable data types, but it is possible to use your own mutable Java classes as data types; if so, it is your responsibility to ensure that no invoked code causes the value of a custom Java data type to be modified, as doing so could mean that previously-performed calculations would now be "wrong".

```
package curam.creole.example;

import curam.creole.execution.RuleObject;
import curam.creole.execution.session.Session;

public class Statics {

  /**
   * Calculates a person's favorite color.
   *
   * This calculation is too complex for rules and so has been
   * coded in java.
   *
   * @param session
   *            The rule session
   * @param person
   *            the person
   * @return the calculated favorite color of the specified person
   */
  public static String calculateFavoriteColor(
      final Session session, final RuleObject person) {

    // Note that the retrieval of the attribute value must be
    // cast to the correct type
    final String name =
        (String) person.getAttributeValue("name").getValue();
    final Number age =
        (Number) person.getAttributeValue("age").getValue();

    final String ageString = age.toString();
    // Calculate the person's favorite color according
    // to the digits in their age and their name
    if (ageString.contains("5") || ageString.contains("7")) {
      return "Blue";
    } else if (name.contains("z")) {
      return "Purple";
    } else {
```

```
      return "Green";
    }

  }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_call"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="favoriteColor">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <!-- Call a java static method
             to perform the calculation -->
        <call class="curam.creole.example.Statics"
          method="calculateFavoriteColor">
          <type>
            <javaclass name="String"/>
          </type>
          <arguments>
            <!-- Pass in this person
                 as an argument to the
                 static method -->
            <this/>
          </arguments>
        </call>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Important: Since Cúram V6, CER and the Dependency Manager support the automatic recalculation of CER-calculated values if their dependencies change.

If you change the implementation of a static method, CER and the Dependency Manager will *not* automatically know to recalculate attribute values that were calculated using the old version of your static method.

Once a static method has been used in a production environment for stored attribute values, rather than changing the implementation you should instead create a new static method (with the required new implementation), and change your rule sets to use the new static method. When you publish your rule set changes to point to the new static method, CER and the Dependency Manager will automatically recalculate all instances of the affected attribute values.

choose
The choose expression chooses a value based on a condition being met.

The choose expression contains:

- **type**

  a data type specifier (see "Handling Data Types" on page 33) declaring the type of value which will be chosen;

- **test (optional)**

  an expression which specifies the value to test against the `condition` in each when expression in turn. If no `test` expression is specified, the `condition` in each when expression is tested in turn to check if it returns the value *true*;

- **when (1 or more)**

  each contains a `condition` to test, and a `value` to return if the condition passes; and

- **otherwise**

  an expression containing a `value` to return (so that no matter what, a value is always chosen).

The conditions are evaluated in top-down order of the when expressions, stopping at the first condition to pass the test. Subsequent conditions are not evaluated.

The `choose` expression reflects that of `if` / `else if` /.../ `else` statements typical of most programming languages. It can be effectively used to implement a decision table.

You might consider ordering your conditions so that those most likely to succeed are near the top of the list (to prevent needless calculations).

warning: For simple conditions (e.g. those which test equality to a single value), typically you can reorder your conditions without affecting the behavior of the rule set.

However, for more complex conditions (and thus in general), you must carefully consider whether reordering your conditions will introduce any unwanted behavior changes.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_choose"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="ageCategory">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <choose>
          <!-- There's no explicit <test> clause, so this
               <choose> statement will test each condition to
               see if it is TRUE. -->
          <type>
            <javaclass name="String"/>
          </type>

          <!-- Note that the order of these conditions is
               important; if we were to swap the positions of the
               "Newborn" and "Infant" tests, then all children
               under 5 (including those under 1) would be
               identified as Infants; no children would be
               identified as Newborns.  -->
          <when>
            <condition>
              <compare comparison="&lt;">
                <reference attribute="age"/>
                <Number value="1"/>
              </compare>
            </condition>
            <value>
```

```xml
              <String value="Newborn"/>
            </value>
          </when>
          <when>
            <condition>
              <compare comparison="&lt;">
                <reference attribute="age"/>
                <Number value="5"/>
              </compare>
            </condition>
            <value>
              <String value="Infant"/>
            </value>
          </when>
          <when>
            <condition>
              <compare comparison="&lt;">
                <reference attribute="age"/>
                <Number value="18"/>
              </compare>
            </condition>
            <value>
              <String value="Child"/>
            </value>
          </when>
          <otherwise>
            <value>
              <String value="Adult"/>
            </value>
          </otherwise>
        </choose>
      </derivation>
    </Attribute>

    <Attribute name="numberOfSpouses">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="maritalStatus">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <choose>
          <type>
            <javaclass name="String"/>
          </type>
          <!-- Test the number of spouses -->
          <test>
            <reference attribute="numberOfSpouses"/>
          </test>
          <!-- Note that the order of the "0" and "1" tests do not
matter -
                so you might want to order these according to whether
most
                Person instances being tested have 0 or 1 spouses.
    -->
          <when>
            <condition>
              <Number value="0"/>
            </condition>
            <value>
              <String value="Unmarried"/>
            </value>
          </when>
          <when>
            <condition>
              <Number value="1"/>
            </condition>
            <value>
              <String value="Married - single spouse"/>
            </value>
          </when>
          <otherwise>
            <value>
              <String value="Married - multiple spouses"/>
            </value>
```

```
            </otherwise>
          </choose>

        </derivation>
      </Attribute>

    </Class>

</RuleSet>
```

Code

The Code expression is a literal constant value that represents a code from an application code table.

The Code expression specifies a code table name, and takes a single argument specifying the value of the code required from the table.

Note: You must specify the code's String value; code table generated constants cannot be used, as CER is a fully dynamic language and cannot be dependent on build-time constructs.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Code"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- Boolean representation of gender -->
    <Attribute name="isMale">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Code representation of gender -->
    <Attribute name="gender">
      <type>
        <codetableentry table="Gender"/>
      </type>
      <derivation>
        <Code table="Gender">
          <choose>
            <type>
              <javaclass name="String"/>
            </type>
            <when>
              <condition>
                <reference attribute="isMale"/>
              </condition>
              <value>
                <!-- use the "MALE" code from the codetable -->
                <String value="MALE"/>
              </value>
            </when>
            <otherwise>
              <value>
                <!-- use the "FEMALE" code from the codetable -->
                <String value="FEMALE"/>
              </value>
            </otherwise>
          </choose>
        </Code>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

combineSuccessionSets

For information about this expression, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

compare

The `compare` expression compares a left-side value with a right-side value, according to the comparison that is provided.

The supported comparisons are:

- **<**

  left side "is less than" right side

- **<=**

  left side "is less than or equal to" right side

- **>**

  left side "is greater than" right side

- **>=**

  left side "is greater than or equal to" right side

The left side and right side values can be of any type of comparable object, including (but not limited to):

- `Number`;
- `String`; and
- `curam.util.type.Date`.

Note: All instances of `Number` are converted to CER's own numerical format (backed by `java.math.BigDecimal`) before comparison.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_compare"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="CompareExampleRuleClass">

    <!-- 3 >= 2 - TRUE-->
    <Attribute name="compareTwoNumbers">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <compare comparison=">=">
          <Number value="3"/>
          <Number value="2"/>
        </compare>
      </derivation>
    </Attribute>

    <!-- New Year earlier than Christmas - TRUE -->
    <Attribute name="compareTwoDates">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <compare comparison="&lt;">
          <Date value="2007-01-01"/>
          <Date value="2007-12-25"/>
        </compare>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

concat

The concat expression creates a localizable message by concatenating a list of values.

For more information, see "Localizing CER Rules" on page 6.

The concat strings together its values with no additional spaces or text; if you require more complex formatting or localizable text, consider using "ResourceMessage" on page 190 instead.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_concat"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="surname">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="dateOfBirth">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- An identifier for a person, including
     first name, surname and date of birth, e.g.
     John Smith (03 Oct 1970).

     First name and surname are plain Strings,
     but date of birth will be localized
     according to the user's locale.
     -->
    <Attribute name="personIdentifier">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <concat>
          <fixedlist>
            <listof>
              <!-- Note we use Object, as we have a
                   mixture of String and Date items
                   in the list. -->
              <javaclass name="Object"/>
            </listof>
            <members>
              <reference attribute="firstName"/>
              <!-- space separator between names -->
              <String value=" "/>
              <reference attribute="surname"/>
              <String value=" ("/>
              <reference attribute="dateOfBirth"/>
              <String value=")"/>
            </members>
          </fixedlist>
        </concat>
      </derivation>
    </Attribute>

  </Class>
```

```
    </RuleSet>
```

create

The `create` expression gets a new instance of a rule class in the session's memory.

Any initialization values that are required by the rule object must be specified as child elements of the `create` expression.

The `create` expression can be used to create an new instance of a rule class from a *different* rule set, by setting the value of the optional `ruleset` XML attribute.

Note: Rule objects created using `create` cannot be retrieved during rules execution, as to do so would violate CER's ordering principle.

There is a choice of syntax when passing values to a created rule object:

- **initialization block**

  CER continues to support a block of attributes defined within an `Initialization` element. This syntax can be useful for attributes which *must* always be set and have no default implementation; and

- **specify elements**

  CER also supports arbitrary attributes having their value overridden by use of a `specify` element which names the attribute to set and which contains the value to use. This syntax can be useful for attributes which are only sometimes set and/or have a default implementation.

Created rule objects are pooled within the session. This pool enables identical requests to create a rule object to be served by a single rule object, which can conserve memory usage and also prevent identical calculations from taking place, leading to lower CPU load. Two requests to create a rule object are considered identical if they request the same rule class and the values of all initialized and specified attributes are equal.

In the example, if a person's work phone number is identical to the person's home phone number, then a single rule object will be used for each number, and thus the derived value for `isOutOfThisArea` is calculated once. Otherwise, if the work and home phone numbers are different, then two different rule objects will be created.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_create"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Person">

    <!-- Phone number details as gathered in evidence -->
    <Attribute name="homePhoneAreaCode">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="homePhoneNumber">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="workPhoneAreaCode">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
```

```
        </Attribute>

        <Attribute name="workPhoneNumber">
          <type>
            <javaclass name="Number"/>
          </type>
          <derivation>
            <specified/>
          </derivation>
        </Attribute>

        <!-- Create PhoneNumber rule objects
             and place them in a list -->
        <Attribute name="phoneNumbers">
          <type>
            <javaclass name="List">
              <ruleclass name="PhoneNumber"/>
            </javaclass>
          </type>
          <derivation>
            <fixedlist>
              <listof>
                <ruleclass name="PhoneNumber"/>
              </listof>

              <members>

                <!-- Phone Number for the home details. -->
                <create ruleclass="PhoneNumber">
                  <!-- The value for PhoneNumber.owner -->
                  <this/>
                  <!-- The value for PhoneNumber.number -->
                  <reference attribute="homePhoneNumber"/>
                  <specify attribute="areaCode">
                    <!-- The value for PhoneNumber.areaCode -->
                    <reference attribute="homePhoneAreaCode"/>
                  </specify>
                </create>

                <!-- Phone Number for the work details.

                     If a person's work phone number is identical to the
                     person's home phone number (i.e. the area code and
                     number are the same), then this <create> expression
                     will return the same rule object as the rule object
                     returned by the <create> expression above.  If the
                     phone numbers are not identical, then two different
                     rule objects will be returned.-->
                <create ruleclass="PhoneNumber">
                  <this/>
                  <reference attribute="workPhoneNumber"/>
                  <specify attribute="areaCode">
                    <reference attribute="workPhoneAreaCode"/>
                  </specify>
                </create>

              </members>
            </fixedlist>
          </derivation>
        </Attribute>

    </Class>

    <Class name="PhoneNumber">

      <Initialization>
        <!-- The values for these attributes must be passed, in order,
             by any <create> expression. -->
        <Attribute name="owner">
          <type>
            <ruleclass name="Person"/>
          </type>
        </Attribute>
        <Attribute name="number">
          <type>
            <javaclass name="Number"/>
          </type>
        </Attribute>
      </Initialization>


      <!-- The value for this attribute may be passed by a <specify>
```

```
              element within a <create> expression, which will override
              the default derivation here. -->
    <Attribute name="areaCode">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- Default implementation, used if the <create> expression
             does not <specify> a value for this attribute. -->
        <Number value="123"/>
      </derivation>
    </Attribute>

    <!-- For a pooled rule object, this derived value will only be
         calculated once.

         For example, if a person's work phone number is identical
         to the person's home phone number, then the same rule
         object will be used for both home and work phone numbers,
         and the "isOutOfThisArea" value for this single rule
         object will be calculated only once.
    -->
    <Attribute name="isOutOfThisArea">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <not>
          <equals>
            <reference attribute="areaCode"/>
            <!-- The area code for the agency's area -->
            <Number value="123"/>
          </equals>
        </not>
      </derivation>
    </Attribute>
  </Class>
</RuleSet>
```

current

The current expression refers to an item in a list that is being processed.

The current expression might appear only within an expression that processes items in a list, such as:

- The listitem expression in a <u>"filter" on page 165</u> or the <u>"dynamiclist" on page 159</u> expression.

- The sortorder in a <u>"sort" on page 194</u> expression.

For clarity, you can assign an alias to the current expression, which must match the alias on the list expression referred to. Alias are required if there are more than current expressions in scope in the same calculation.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_listitem"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Household">

    <Attribute name="members">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="adults">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <filter>
```

```
              <list>
                <reference attribute="members"/>
              </list>
              <listitemexpression>
                <!-- The reference uses current to refer
                        to an item in the list of Person
                        rule objects. -->
                <reference attribute="isAdult">
                  <current/>
                </reference>
              </listitemexpression>
            </filter>
          </derivation>
        </Attribute>

    </Class>

    <Class name="Person">

      <Attribute name="children">
        <type>
          <javaclass name="List">
            <ruleclass name="Person"/>
          </javaclass>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <Attribute name="age">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <Attribute name="isAdult">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <compare comparison=">=">
            <reference attribute="age"/>
            <Number value="18"/>
          </compare>

        </derivation>
      </Attribute>

      <!-- The children of this person who
          are not yet adults. -->
      <Attribute name="dependentChildren">
        <type>
          <javaclass name="List">
            <ruleclass name="Person"/>
          </javaclass>
        </type>
        <derivation>
          <filter>
            <!-- Use an alias to avoid confusion (for human
                  readers of the rule set!) between the parent
                  Person and the child Person.  -->
            <list alias="child">
              <reference attribute="children"/>
            </list>
            <listitemexpression>
              <not>
                <reference attribute="isAdult">
                  <!-- The alias on the current must match
                        that on the list. -->
                  <current alias="child"/>
                </reference>
              </not>
            </listitemexpression>
          </filter>
        </derivation>
      </Attribute>

    </Class>
```

```
    </RuleSet>
```

Date

The `Date` expression is literal Date constant value, of type `curam.util.type.Date`.

The `Date` value is specified in the format *yyyy-mm-dd*.

Note: There is intentionally no function in CER to obtain the current date - such a function would be volatile in that today it returns one value, tomorrow a different value.

Volatile functions are forbidden in CER, as if a function's result can change, it could mean that previously-performed calculations would now be "wrong".

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Date"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="DateExampleRuleClass">

    <Attribute name="nullDate">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <!-- A null Date -->
        <null/>
      </derivation>
    </Attribute>

    <Attribute name="dateOfBirth">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <!-- The Date 3rd October, 1970 -->
        <Date value="1970-10-03"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

dynamiclist

The `dynamiclist` expression creates a new list by evaluating an expression on each item in an existing list.

The new list will have one entry corresponding to each entry in the existing list, with ordering preserved.

A `dynamiclist` expression specifies:

- **list**

  The existing list; and

- **listitemexpression**

  The expression to evaluate on each item in the existing list.

A `dynamiclist` can be used when the number of items in the desired list is not known as design time (i.e. it can differ from execution to execution, depending on the value of other attributes). If the number of items is fixed (i.e. is known at design time), consider using "fixedlist" on page 167 instead.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_dynamiclist"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="age">
      <type>
```

```xml
            <javaclass name="Number"/>
         </type>
         <derivation>
            <specified/>
         </derivation>
      </Attribute>

      <Attribute name="isDisabled">
         <type>
            <javaclass name="Boolean"/>
         </type>
         <derivation>
            <specified/>
         </derivation>
      </Attribute>

      <Attribute name="totalIncome">
         <type>
            <javaclass name="Number"/>
         </type>
         <derivation>
            <specified/>
         </derivation>
      </Attribute>

      <Attribute name="pets">
         <type>
            <javaclass name="List">
               <ruleclass name="Pet"/>
            </javaclass>
         </type>
         <derivation>
            <specified/>
         </derivation>
      </Attribute>

   </Class>

   <Class name="Pet">
      <Initialization>
         <Attribute name="name">
            <type>
               <javaclass name="String"/>
            </type>
         </Attribute>
      </Initialization>

   </Class>

   <Class name="Household">

      <Attribute name="members">
         <type>
            <javaclass name="List">
               <ruleclass name="Person"/>
            </javaclass>
         </type>
         <derivation>
            <specified/>
         </derivation>
      </Attribute>

      <Attribute name="containsDisabledPerson">
         <type>
            <javaclass name="Boolean"/>
         </type>
         <derivation>
            <any>
               <!-- gets a list of Booleans, corresponding
                    to the isDisabled attribute on each
                    Person members of this Household    -->
               <dynamiclist>
                  <list>
                     <reference attribute="members"/>
                  </list>
                  <listitemexpression>
                     <reference attribute="isDisabled">
                        <current/>
                     </reference>
                  </listitemexpression>
               </dynamiclist>
            </any>
```

```
      </derivation>
    </Attribute>

    <Attribute name="totalIncomeOfAdultMembers">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <sum>
          <dynamiclist>
            <list>
              <!-- filter the members down to
                   just the adults -->
              <filter>
                <list>
                  <reference attribute="members"/>
                </list>
                <listitemexpression>
                  <compare comparison=">=">
                    <reference attribute="age">
                      <current/>
                    </reference>
                    <Number value="18"/>
                  </compare>
                </listitemexpression>
              </filter>
            </list>
            <listitemexpression>
              <reference attribute="totalIncome">
                <current/>
              </reference>
            </listitemexpression>
          </dynamiclist>
        </sum>
      </derivation>
    </Attribute>

    <Attribute name="memberAges">
      <type>
        <javaclass name="List">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <dynamiclist>
          <list>
            <reference attribute="members"/>
          </list>
          <listitemexpression>
            <reference attribute="age">
              <current/>
            </reference>
          </listitemexpression>
        </dynamiclist>
      </derivation>
    </Attribute>

    <Attribute name="youngestAge">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <min>
          <reference attribute="memberAges"/>
        </min>
      </derivation>
    </Attribute>

    <!-- get all the pets in the household,
         by joining together each person's
         list of pets -->
    <Attribute name="allPets">
      <type>
        <javaclass name="List">
          <ruleclass name="Pet"/>
        </javaclass>
      </type>
      <derivation>
        <joinlists>
          <!-- a list of list of pets, one
               list for each household
               member -->
```

```
                <dynamiclist>
                  <list>
                    <reference attribute="members"/>
                  </list>
                  <listitemexpression>
                    <reference attribute="pets">
                      <current/>
                    </reference>
                  </listitemexpression>
                </dynamiclist>

              </joinlists>
          </derivation>
        </Attribute>

    </Class>

</RuleSet>
```

defaultDescription

The `defaultDescription` expression provides a default implementation of the `description` attribute that all rule classes inherit from the root rule class.

For more information, see "Handling Data Types" on page 33.

Each rule class should override the `description` attribute from the root rule class to provide a more meaningful description. If no override is provided (or inherited), a warning will be issued when the rule set is validated.

Important: The `defaultDescription` expression is for use *only* by the root rule class; you must not use it in your own rule classes.

```
<Class name="RootRuleClass" abstract="true">
    <Attribute name="description">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- For use ONLY in the RootRuleClass -->
        <defaultDescription/>
      </derivation>
    </Attribute>
  </Class>
```

equals

The equals expression determines whether two objects (a left-side object and a right-side object) are equal.

Number values are converted to CER's own numerical format (backed by `java.math.BigDecimal`) before comparison. Differences in leading or trailing zeros are ignored.

Any `null` values are compared safely. If both left side and right side are `null`, then the `equals` expression returns `true`. If only one of the left side and right side values are `null`, then the `equals` expression returns `false`.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_equals"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="EqualsExampleRuleClass">

    <!-- TRUE -->
    <Attribute name="identicalStrings">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <equals>
          <String value="A String"/>
          <String value="A String"/>
        </equals>
      </derivation>
```

```
      </Attribute>

      <!-- FALSE -->
      <Attribute name="differentStrings">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <equals>
            <String value="A String"/>
            <String value="A different String"/>
          </equals>
        </derivation>
      </Attribute>

      <!-- TRUE -->
      <Attribute name="identicalNumbers">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <equals>
            <!-- These numbers are the same,
              disregarding trivial
              differences in leading/trailing
              zeroes -->
            <Number value="123"/>
            <Number value="000123.000"/>
          </equals>
        </derivation>
      </Attribute>

      <!-- FALSE -->
      <Attribute name="differentTypes">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <equals>
            <!-- These objects are of
              different types, so are
              not equal even if they
              "look" the same.-->
            <String value="123"/>
            <Number value="123"/>
          </equals>
        </derivation>
      </Attribute>

      <!-- FALSE -->
      <Attribute name="oneNull">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <equals>
            <null/>
            <Number value="456"/>
          </equals>
        </derivation>
      </Attribute>

      <!-- TRUE -->
      <Attribute name="twoNulls">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <equals>
            <null/>
            <null/>
          </equals>
        </derivation>
      </Attribute>

  </Class>

</RuleSet>
```

existencetimeline

The `existencetimeline` expression creates a Timeline of a specified type from a pair of inclusive start and end dates, either of which is optional.

For more information, see "Creating Timelines" on page 49.

For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_existencetimeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Person">

    <Attribute name="dateOfBirth">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>


    <!-- will be null if the person is still alive -->
    <Attribute name="dateOfDeath">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Creates a timeline which is false before the Person is
         born, true while the Person is alive, and false after the
         Person dies.   If the Person has no date-of-death recorded,
         there will be no trailing "false" interval. -->
    <Attribute name="isAliveTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <existencetimeline>
          <intervaltype>
            <javaclass name="Boolean"/>
          </intervaltype>
          <intervalfromdate>
            <reference attribute="dateOfBirth"/>
          </intervalfromdate>
          <intervaltodate>
            <reference attribute="dateOfDeath"/>
          </intervaltodate>
          <preExistenceValue>
            <false/>
          </preExistenceValue>
          <existenceValue>
            <true/>
          </existenceValue>
          <postExistenceValue>
            <false/>
          </postExistenceValue>
        </existencetimeline>

      </derivation>
    </Attribute>

    <!-- Creates a timeline which is "Before Birth" before the Person
         is born, "During Lifetime" while the Person is alive, and
         "After Death" after the Person dies.   If the Person has no
         date-of-death recorded, there will be no trailing "After
         Death" interval. -->
    <Attribute name="lifeStatus">
      <type>
        <javaclass name="curam.creole.value.Timeline">
```

```
              <javaclass name="String"/>
            </javaclass>
          </type>
          <derivation>
            <existencetimeline>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <intervalfromdate>
                <reference attribute="dateOfBirth"/>
              </intervalfromdate>
              <intervaltodate>
                <reference attribute="dateOfDeath"/>
              </intervaltodate>
              <preExistenceValue>
                <String value="Before Birth"/>
              </preExistenceValue>
              <existenceValue>
                <String value="During Lifetime"/>
              </existenceValue>
              <postExistenceValue>
                <String value="After Death"/>
              </postExistenceValue>
            </existencetimeline>

          </derivation>
        </Attribute>



  </Class>
</RuleSet>
```

false

The `false` expression represents the Boolean constant value "false".

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_false"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="FalseExampleRuleClass">

    <Attribute name="isCuramExpertRulesFantastic">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <not>
          <false/>
        </not>
      </derivation>
    </Attribute>

    <Attribute name="didCookbookWinPulitzerPrize">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <false/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

filter

The `filter` expression creates a new list that contains all of the items in an existing list that meet the filter condition.

The `filter` expression contains:

- **list**

  an existing list to filter

- **listitemexpression**

  the test to apply to each item in the list.

Typically the `listitemexpression` contains one or more calculations applied to the item in the list.

The relative order of the list items in the filtered result will preserve the relative order of the list items in the original list. If none of the items in the list meet the filter condition, then an empty list is returned.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_filter"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The spouse of this person, or
      null if unmarried -->
    <Attribute name="spouse">
      <type>
        <ruleclass name="Person"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The children of this person -->
    <Attribute name="children">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>


  </Class>


  <Class name="Household">

    <!-- All the people in the household -->
    <Attribute name="members">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- All the adults in the household -->
    <Attribute name="adultMembers">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
```

```
            <derivation>
              <filter>
                <list>
                  <reference attribute="members"/>
                </list>
                <listitemexpression>
                  <compare comparison=">=">
                    <reference attribute="age">
                      <current/>
                    </reference>
                    <Number value="18"/>
                  </compare>
                </listitemexpression>
              </filter>
            </derivation>
          </Attribute>

          <!-- All the lone parents in the household -->
          <Attribute name="loneParents">
            <type>
              <javaclass name="List">
                <ruleclass name="Person"/>
              </javaclass>
            </type>
            <derivation>
              <filter>
                <list>
                  <reference attribute="members"/>
                </list>
                <listitemexpression>
                  <all>
                    <fixedlist>
                      <listof>
                        <javaclass name="Boolean"/>
                      </listof>
                      <members>

                        <!-- No spouse -->
                        <equals>
                          <reference attribute="spouse">
                            <current/>
                          </reference>
                          <null/>
                        </equals>
                        <!-- At least one child -->
                        <not>
                          <property name="isEmpty">
                            <object>
                              <reference attribute="children">
                                <current/>
                              </reference>
                            </object>
                          </property>
                        </not>
                      </members>
                    </fixedlist>
                  </all>
                </listitemexpression>
              </filter>
            </derivation>
          </Attribute>

      </Class>

 </RuleSet>
```

fixedlist

The fixedlist expression creates a new list from items known at rule set design time.

The fixedlist expression specifies:

- **listof**

  The type of item in the list returned (see "Handling Data Types" on page 33); and

- **members**

  The items in the list.

The created list will contain its members in the order listed in the rule set.

Tip: The `members` element may contain 0, 1 or many child elements.

However, if the `fixedlist` is contained within a list processing operation but only specifies 0 or 1 list members, the CER rule set validator will issue a warning, indicating that the list may be unnecessary.

If you need to create a list where the number of items in the list is not known at design time, consider using a dynamic list instead.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_fixedlist"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- The pets owned by this Person -->
    <Attribute name="pets">
      <type>
        <javaclass name="List">
          <ruleclass name="Pet"/>
        </javaclass>
      </type>
      <derivation>

        <!-- A fixed list of Pets -->
        <fixedlist>
          <listof>
            <ruleclass name="Pet"/>
          </listof>
          <members>
            <!-- Every Person has exactly two pets,
                 Skippy and Lassie -->
            <create ruleclass="Pet">
              <String value="Skippy"/>
              <String value="Kangaroo"/>
            </create>
            <create ruleclass="Pet">
              <String value="Lassie"/>
              <String value="Dog"/>
            </create>
          </members>
        </fixedlist>
      </derivation>
    </Attribute>


    <Attribute name="isEntitledToBenefits">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <all>
          <!-- A fixed list of Boolean conditions -->
          <fixedlist>
            <listof>
              <javaclass name="Boolean"/>
            </listof>
            <members>
              <!-- Must be an adult -->
              <compare comparison=">=">
                <reference attribute="age"/>
                <Number value="18"/>
              </compare>
              <!-- Must be resident in the state -->
              <reference attribute="isResidentInTheState"/>
              <!-- Must have income under the threshold for benefits
 -->
              <compare comparison="&lt;">
                <reference attribute="totalIncome"/>
                <Number value="100"/>
              </compare>
            </members>
          </fixedlist>

        </all>

      </derivation>
    </Attribute>
```

```xml
    <Attribute name="totalIncome">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- A pointless sum of one item -
             the CER rule set validator will warn that this
             fixedlist may be unnecessary. -->
        <sum>
          <fixedlist>
            <listof>
              <javaclass name="Number"/>
            </listof>
            <members>
              <!-- Sum up only the earned income -->
              <reference attribute="earnedIncome"/>
            </members>
          </fixedlist>

        </sum>

      </derivation>
    </Attribute>

    <Attribute name="earnedIncome">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>


    <Attribute name="isResidentInTheState">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Pet">

    <Initialization>

      <Attribute name="name">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>

      <Attribute name="species">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>

    </Initialization>

  </Class>

</RuleSet>
```

FrequencyPattern

The `FrequencyPattern` expression is a literal FrequencyPattern constant value, of type `curam.util.type.FrequencyPattern`.

The `FrequencyPattern` 's value is specified as a 9-digit number. See the JavaDoc for `curam.util.type.FrequencyPattern` for the meaning of the digit string.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_FrequencyPattern"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="FrequencyPatternExampleRuleClass">

    <Attribute name="nullFrequencyPattern">
      <type>
        <javaclass name="curam.util.type.FrequencyPattern"/>
      </type>
      <derivation>
        <!-- A null FrequencyPattern -->
        <null/>
      </derivation>
    </Attribute>

    <Attribute name="weeklyOnMondays">
      <type>
        <javaclass name="curam.util.type.FrequencyPattern"/>
      </type>
      <derivation>
        <!-- The Frequency Pattern string for
             "Weekly on Mondays" -->
        <FrequencyPattern value="100100100"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Interval

The `Interval` expression creates an interval of a given type, with a value valid from a specified date.

For more information, see "Handling Data that Changes Over Time" on page 42.

This expression is typically used as part of the construction of a "Timeline" on page 200.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Interval"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="CreateInterval">

    <Attribute name="aNumberTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <Timeline>
          <intervaltype>
            <javaclass name="Number"/>
          </intervaltype>
          <initialvalue>
            <Number value="0"/>
          </initialvalue>
          <!-- Another interval-->
          <intervals>
            <fixedlist>
              <listof>
                <javaclass name="curam.creole.value.Interval">
                  <javaclass name="Number"/>
```

```
                </javaclass>
              </listof>
              <members>
                <!-- Creates an interval of the specified type.
                     Typically used as input into a <Timeline>. -->
                <Interval>
                  <intervaltype>
                    <javaclass name="Number"/>
                  </intervaltype>
                  <start>
                    <Date value="2001-01-01"/>
                  </start>
                  <value>
                    <Number value="10000"/>
                  </value>
                </Interval>
              </members>
            </fixedlist>

        </intervals>
      </Timeline>

    </derivation>
  </Attribute>


  </Class>
</RuleSet>
```

intervalvalue

The `intervalvalue` expression wraps an expression which returns a Timeline and allows a containing expression to operate on the individual values within the Timeline.

This expression effectively shields an outer expression from knowing that it is operating on a Timeline. For more information, see "Handling Data that Changes Over Time" on page 42.

This expression can only be used when nested within a "timelineoperation" on page 202 expression. For further description and usage examples of `intervalvalue`, see "timelineoperation" on page 202.

joinlists

The `joinlist` expression creates a new list by joining together some existing lists.

The `joinlists` expression takes a single argument which must be a list of lists.

The order of the items in the new list is identical to the order in their source list. The lists are joined in the order they are provided.

If the lists being joined can contain duplicate items, consider wrapping the `joinlists` expression in a "removeduplicates" on page 189 expression.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_joinlists"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="pets">
      <type>
        <javaclass name="List">
          <ruleclass name="Pet"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Pet">
    <Initialization>
      <Attribute name="name">
        <type>
          <javaclass name="String"/>
```

```
            </type>
          </Attribute>
        </Initialization>

    </Class>

    <Class name="Household">

      <Attribute name="members">
        <type>
          <javaclass name="List">
            <ruleclass name="Person"/>
          </javaclass>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <!-- get all the pets in the household,
           by joining together each person's
           list of pets -->
      <Attribute name="allPets">
        <type>
          <javaclass name="List">
            <ruleclass name="Pet"/>
          </javaclass>
        </type>
        <derivation>
          <joinlists>
            <!-- a list of list of pets, one
                 list for each household
                 member -->
            <dynamiclist>
              <list>
                <reference attribute="members"/>
              </list>
              <listitemexpression>
                <reference attribute="pets">
                  <current/>
                </reference>
              </listitemexpression>
            </dynamiclist>

          </joinlists>
        </derivation>
      </Attribute>

    </Class>

  </RuleSet>
```

legislationChange
For information about this expression, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

max
The max expression determines the largest value in a list or `null` if the list is empty.

The list can contain any type of comparable object, including (but not limited to):

- Number
- String
- curam.util.type.Date

Note: All instances of Number are converted to CER's own numerical format (backed by `java.math.BigDecimal`) before comparison.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_max"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="MaxExampleRuleClass">
```

```xml
      <!-- Will pick out "Cherry" as the "largest" String value -->
      <Attribute name="alphabeticallyLastFruit">
        <type>
          <javaclass name="String"/>
        </type>
        <derivation>
          <max>
            <reference attribute="fruits"/>
          </max>
        </derivation>
      </Attribute>

      <Attribute name="fruits">
        <type>
          <javaclass name="List">
            <javaclass name="String"/>
          </javaclass>
        </type>
        <derivation>
          <fixedlist>
            <listof>
              <javaclass name="String"/>
            </listof>
            <members>
              <String value="Apple"/>
              <String value="Banana"/>
              <String value="Cherry"/>
            </members>
          </fixedlist>
        </derivation>
      </Attribute>

      <!-- Determines the number of spots on the spottiest dog -->
      <Attribute name="largestNumberOfSpots">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <max>
            <dynamiclist>
              <list>
                <reference attribute="dalmatians"/>
              </list>
              <listitemexpression>
                <reference attribute="numberOfSpots">
                  <current/>
                </reference>
              </listitemexpression>
            </dynamiclist>
          </max>
        </derivation>
      </Attribute>

      <Attribute name="dalmatians">
        <type>
          <javaclass name="List">
            <ruleclass name="Dalmation"/>
          </javaclass>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

  </Class>

  <Class name="Dalmation">

    <Attribute name="numberOfSpots">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

min

The `min` expression determines the smallest value in a list or `null` if the list is empty.

The list can contain any type of comparable object, including (but not limited to):

- `Number`;
- `String`; and
- `curam.util.type.Date`.

Note: All instances of `Number` are converted to CER's own numerical format (backed by `java.math.BigDecimal`) before comparison.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_min"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="MinExampleRuleClass">

    <!-- Will pick out New Year as the "earliest" Date value -->
    <Attribute name="eariestDate">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <min>
          <reference attribute="publicHolidays"/>
        </min>
      </derivation>
    </Attribute>

    <Attribute name="publicHolidays">
      <type>
        <javaclass name="List">
          <javaclass name="curam.util.type.Date"/>
        </javaclass>
      </type>
      <derivation>
        <fixedlist>
          <listof>
            <javaclass name="curam.util.type.Date"/>
          </listof>
          <members>
            <Date value="2007-01-01"/>
            <Date value="2007-12-25"/>
          </members>
        </fixedlist>
      </derivation>
    </Attribute>

    <!-- Determines the number of strips on the least-stripey
 zebra-->
    <Attribute name="smallestNumberOfStripes">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <min>
          <dynamiclist>
            <list>
              <reference attribute="zebras"/>
            </list>
            <listitemexpression>
              <reference attribute="numberOfStripes">
                <current/>
              </reference>
            </listitemexpression>
          </dynamiclist>
        </min>
      </derivation>
    </Attribute>

    <Attribute name="zebras">
      <type>
        <javaclass name="List">
          <ruleclass name="Zebra"/>
        </javaclass>
```

```
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Zebra">

    <Attribute name="numberOfStripes">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

not

The not expression negates a Boolean value.

Foe example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_not"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="isLivingInUSA">

      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <!-- Note that this not-within-not is somewhat contrived.
 -->
        <not>
          <reference attribute="isLivingOutsideUSA"/>
        </not>
      </derivation>

    </Attribute>

    <Attribute name="isLivingOutsideUSA">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <not>
          <equals>
            <reference attribute="country"/>
            <String value="USA"/>
          </equals>
        </not>
      </derivation>
    </Attribute>

    <!-- The country in which this person resides. -->
    <Attribute name="country">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

null

A null constant value.

Setting a value to null may be useful to indicate that no value applies.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_null"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Pet">
    <Initialization>
      <Attribute name="name">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
    </Initialization>

  </Class>

  <Class name="Person">

    <!-- This Person's favorite Pet, or
         null if the Person owns no pet. -->
    <Attribute name="favoritePet">
      <type>
        <ruleclass name="Pet"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The name of this Person's
         favorite Pet, or null if
         the Person owns no pet.

         We have to test for the favoritePet
         being null before performing the
         (simple) calculation.-->
    <Attribute name="favoritePetsName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <choose>
          <type>
            <javaclass name="String"/>
          </type>
          <when>
            <!-- if this Person has no
                 favorite pet, then calculate the
                 name of the favorite pet as null. -->
            <condition>
              <equals>
                <reference attribute="favoritePet"/>
                <null/>
              </equals>
            </condition>
            <value>
              <null/>
            </value>
          </when>
          <otherwise>
            <value>
              <!-- get the name of the favorite pet -->
              <reference attribute="name">
                <reference attribute="favoritePet"/>
              </reference>
            </value>
          </otherwise>
        </choose>

      </derivation>
    </Attribute>

  </Class>
```

```
    </RuleSet>
```

Number

A literal Number constant value.

A Number in CER is an arbitrarily-long decimal value, specified using a period (".") as the decimal separator and without any thousands separator.

CER business calculations can often involve percentage values, e.g. "Deduct 10% of the person's income". To help with the codification of such rules, CER allows a Number to be specified as a percentage, by simply suffixing the number with *%*. For example, the numbers *12.345%* and *0.12345* will behave identically in calculations (but the percentage version will display in percentage form).

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Number"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="NumberExampleRuleClass">

    <Attribute name="aPositiveInteger">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- A positive integer -->
        <Number value="1"/>
      </derivation>
    </Attribute>

    <Attribute name="aNegativeInteger">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- A negative integer -->
        <Number value="-2"/>
      </derivation>
    </Attribute>

    <Attribute name="aDecimalNumber">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- A decimal number.

             Numbers are arbitrarily long/precise, use "." for
             the decimal separator and have no thousands
             separator.
        -->
        <Number value="-12345.6789"/>
      </derivation>
    </Attribute>

    <Attribute name="aPercentage">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- A percentage
             (12.345% is equivalent to the number 0.12345) -->
        <Number value="12.345%"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

periodlength

Calculates the amount of time units between two dates.

One of the following time units must be specified:

- *days*;
- *weeks*;
- *months*; and
- *years*.

The `periodlength` expression must also specify whether the end date of the period is *inclusive* or *exclusive* or the end date (the period is always inclusive of the start date).

The calculation of the period length is always rounded down to the nearest integer.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_periodlength"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="PeriodLengthExampleClass">

    <!-- NB 1970 was not a leap year -->
    <Attribute name="firstDayOfJanuary1970">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <Date value="1970-01-01"/>
      </derivation>
    </Attribute>

    <Attribute name="lastDayOfDecember1970">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <Date value="1970-12-31"/>
      </derivation>
    </Attribute>

    <Attribute name="firstDayOfJanuary1971">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <Date value="1971-01-01"/>
      </derivation>
    </Attribute>

    <Attribute name="sameDay_LengthInDays">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- starts and ends on the same day = 1 day -->
        <periodlength endDateInclusion="inclusive" unit="days">
          <reference attribute="firstDayOfJanuary1970"/>
          <reference attribute="firstDayOfJanuary1970"/>
        </periodlength>
      </derivation>
    </Attribute>

    <Attribute name="sameDay_LengthInWeeks">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- starts and ends on the same day = 0 weeks-->
        <periodlength endDateInclusion="exclusive" unit="weeks">
          <reference attribute="firstDayOfJanuary1970"/>
          <reference attribute="firstDayOfJanuary1970"/>
        </periodlength>
      </derivation>
    </Attribute>

    <Attribute name="januaryToDecember_LengthInDays">
      <type>
        <javaclass name="Number"/>
      </type>
```

```xml
        <derivation>
          <!-- 365 days -->
          <periodlength endDateInclusion="inclusive" unit="days">
            <reference attribute="firstDayOfJanuary1970"/>
            <reference attribute="lastDayOfDecember1970"/>
          </periodlength>
        </derivation>
      </Attribute>

      <Attribute name="januaryToDecember_LengthInYearsExclusive">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <!-- 0 years (nearly 1 year, but just 1 day short) -->
          <periodlength endDateInclusion="exclusive" unit="years">
            <reference attribute="firstDayOfJanuary1970"/>
            <reference attribute="lastDayOfDecember1970"/>
          </periodlength>
        </derivation>
      </Attribute>

      <Attribute name="januaryToDecember_LengthInYearsInclusive">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <!-- 1 year (exactly) -->
          <periodlength endDateInclusion="inclusive" unit="years">
            <reference attribute="firstDayOfJanuary1970"/>
            <reference attribute="lastDayOfDecember1970"/>
          </periodlength>
        </derivation>
      </Attribute>

      <Attribute name="januaryToJanuary_LengthInYearsExclusive">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <!-- 1 year (exactly) -->
          <periodlength endDateInclusion="exclusive" unit="years">
            <reference attribute="firstDayOfJanuary1970"/>
            <reference attribute="firstDayOfJanuary1971"/>
          </periodlength>
        </derivation>
      </Attribute>

  </Class>

</RuleSet>
```

property

Obtains a property of a Java object.

The `property` expression specifies the name of the Java method to call, and:

- **object**

  The Java object on which to operate; and

- **arguments**

  Optionally, a list of arguments to pass to the Java method.

The `property` expression allows CER to leverage the power of Java classes without having to replicate an arbitrary subset of methods as CER expressions. For example, `java.util.List` contains a `size` method, and so CER contains no explicit expression for calculating the counting the number of items in a list.

However, to comply with CER's principle of immutability, only Java methods which do not alter the "value" of any object may be called. CER only allows a "property" method to be called if the method is included on the "safe list" of methods for the object's class (or one of its ancestor classes or interfaces).

A method is deemed safe if it is explicitly marked as such in the safe list. If it is not present in the safe list, then the CER rule set validator will issue an error.

Tip: The explicit setting of safety as *false* is unnecessary but can be included for documentation completeness, as is the case with the safe lists included with CER.

The safe list for a class is a properties file in the same package as the class, named *<classname>_CREOLE.properties*.

CER includes safe lists for the following Java classes and interfaces:

• `curam.creole.value.Timeline;`

• `java.lang.Object;`

• `java.lang.Number;` and

• `java.util.List.`

Safe list for `curam.creole.value.Timeline` methods.

```
# Safe list for curam.creole.value.Timeline

# safe
valueOn.safe=true
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_property"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Person">
    <Attribute name="isMinor">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <ruleclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Whether this person is a child. -->
    <Attribute name="isAChild">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <property name="valueOn">
          <object>
            <reference attribute="isMinor"/>
          </object>
          <arguments>
            <Date value="2000-01-01"/>
          </arguments>
        </property>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Safe list for `java.lang.Object` methods.

```
# Safe list for java.lang.Object

# safe
toString.safe=true

# force equality to be evaluated using <equals>
equals.safe=false

# not exposed, even though they're "safe"
```

```
    hashCode.safe=false
    getClass.safe=false
```

Safe list for `java.lang.Number` methods.

```
# Safe list for java.lang.Number

byteValue.safe=true
doubleValue.safe=true
floatValue.safe=true
intValue.safe=true
longValue.safe=true
shortValue.safe=true
```

Safe list for `java.util.List` methods.

```
# Safe list for java.util.List

contains.safe=true
containsAll.safe=true


get.safe=true

indexOf.safe=true
isEmpty.safe=true
lastIndexOf.safe=true
size.safe=true
subList.safe=true


# not exposed
hashCode.safe=false
listIterator.safe=false
iterator.safe=false
toArray.safe=false

# mutators - unsafe
add.safe=false
addAll.safe=false
clear.safe=false
remove.safe=false
removeAll.safe=false
retainAll.safe=false
```

For a description of some of the useful properties on the `List` Java interface, see .

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_property"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Person">
    <Attribute name="children">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Whether this person has any children.

        Tests the isEmpty property of List. -->
    <Attribute name="hasChildren">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <not>
          <property name="isEmpty">
            <object>
```

```
              <reference attribute="children"/>
            </object>
          </property>
        </not>

      </derivation>
    </Attribute>

    <!-- All this person's children, excluding the first child.

         Uses the subList property of List, passing in:
         - (inclusive) from item at position "1" (denoting the
  second
           member in the list; lists in Java are zero-based)
         - (exclusive) to item at position "size of list" (denoting
           the position after the last item in the list)
    -->
    <Attribute name="secondAndSubsequentChildren">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <property name="subList">
          <object>
            <reference attribute="children"/>
          </object>
          <arguments>
            <!-- The number must be converted to an integer
                 (as required by List.subList). -->
            <property name="intValue">
              <object>
                <Number value="1"/>
              </object>
            </property>
            <property name="size">
              <object>
                <reference attribute="children"/>
              </object>
            </property>

          </arguments>
        </property>

      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Important: Since Cúram V6, CER and the Dependency Manager supports the storage of calculated attribute values on the database, together with automatic recalculation of attribute values if their dependencies change.

If you change the implementation of a property method, CER and the Dependency Manager will *not* automatically know to recalculate attribute values that were calculated using the old version of your property method.

Once a property method has been used in a production environment for stored attribute values, rather than changing the implementation you should instead create a new property method (with the required new implementation), and change your rule sets to use the new property method. When you publish your rule set changes to point to the new property method, CER will automatically recalculate all instances of the affected attribute values.

rate
For information about this expression, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

readall

The readall expression retrieves all external rule object instances of a rule class, that is, those created by client code. Internal rule object instances, that is, those created from rules, are not retrieved.

For more information about the creation of rules objects, see "Creating External and Internal Rule Objects" on page 28.

The readall expression can be used to retrieve instances of a rule class from a *different* rule set, by setting the value of the optional ruleset XML attribute.

The readall expression supports an optional match element which causes the readall expression to only retrieve rule objects whose value for a particular attribute matches that in the search criterion.

Important: Prior to Cúram V6, one way of retrieving rule objects that matched a criterion was to wrap a readall within a "filter" on page 165 expression.

However, for CER Sessions that use a DatabaseDataStorage (see "Creating CER Sessions" on page 25), in general it will be more performant to use the readall / match syntax introduced in Cúram V6. The new syntax will perform better:

- when the attribute containing the readall expression is first calculated; and
- when CER and the Dependency Manager identify that the attribute containing the readall expression is out-of-date and needs to be recalculated (see "Recalculating CER Results with the Dependency Manager" on page 64).

In situations where rule objects must be matched on more than one criteria, you should use the readall / match syntax to match on the most selective attribute, and then wrap the results in a filter to filter down to the other criteria.

Tip: If you only expect there to be a singleton instance of the rule class (perhaps after filtering or matching), consider wrapping the expression in a "singleitem" on page 192 expression.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_readall"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="socialSecurityNumber">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!--
    Retrieve the one-and-only claim which will have been used to
    seed the session
    -->

    <Attribute name="claim">
      <type>
        <ruleclass name="Claim"/>
      </type>
      <derivation>
        <singleitem onEmpty="error" onMultiple="error">
          <readall ruleclass="Claim"/>
        </singleitem>
      </derivation>
    </Attribute>


    <!--
    Retrieve the benefit rule objects for this person (created from
    client code, probably by querying external storage).

    This implementation uses a <readall> with a nested <match> to
    retrieve only the matching rule objects, and (depending on data
    storage) will be more performant than the
    "benefitsFilterReadall" implementation below.
```

```
-->

    <Attribute name="benefitsReadallMatch">
      <type>
        <javaclass name="List">
          <ruleclass name="Benefit"/>
        </javaclass>
      </type>
      <derivation>
        <readall ruleclass="Benefit">
          <match retrievedattribute="socialSecurityNumber">
            <reference attribute="socialSecurityNumber"/>
          </match>
        </readall>
      </derivation>
    </Attribute>


    <!--
    Retrieves the same rule objects as for "benefitsReadallMatch"
    above, but (depending on data storage) may not be as performant.
    -->
    <Attribute name="benefitsFilterReadall">
      <type>
        <javaclass name="List">
          <ruleclass name="Benefit"/>
        </javaclass>
      </type>
      <derivation>
        <filter>
          <list>
            <!-- retrieve all Benefit rule objects from external
                 storage -->
            <readall ruleclass="Benefit"/>
          </list>
          <listitemexpression>
            <equals>
              <!-- match up the social security numbers on
                   the person rule object and the benefit
                   rule object -->
              <reference attribute="socialSecurityNumber">
                <current/>
              </reference>
              <reference attribute="socialSecurityNumber"/>
            </equals>
          </listitemexpression>
        </filter>
      </derivation>
    </Attribute>

    <!--
    Retrieves the person's benefits of type "IncomeAssistance",
    using a <match> to retrieve all the person's benefits, and then
    a <filter> to extract only the "Income Assistance" benefits from
    the benefits for that person.

    This implementation may be suitable when the
    socialSecurityNumber is the most selective attribute for a
    Benefit in the data storage (i.e. there are many Benefit rule
    objects, but each socialSecurityNumber value is present on
    relatively few Benefit rule objects).
    -->
    <Attribute name="incomeAssistanceBenefitsMatchSSNFilterType">
      <type>
        <javaclass name="List">
          <ruleclass name="Benefit"/>
        </javaclass>
      </type>
      <derivation>
        <filter>
          <list>
            <!-- retrieve all Benefit rule objects for the Person
-->
            <readall ruleclass="Benefit">
              <match retrievedattribute="socialSecurityNumber">
                <reference attribute="socialSecurityNumber"/>
              </match>
            </readall>
          </list>
          <listitemexpression>
            <equals>
              <!-- filter the Benefit rule objects for the Person
```

```
                  down to those of type "Income Assistance" only
              -->
          <reference attribute="type">
            <current/>
          </reference>
          <Code table="BenefitType">
            <!-- The value for Income Assistance -->
            <String value="BT1"/>
          </Code>
        </equals>
      </listitemexpression>
    </filter>
  </derivation>
</Attribute>


<!--
Retrieves the person's benefits of type "IncomeAssistance",
using a <match> to retrieve all the "Income Assistance"
benefits, and then a <filter> to extract only the "Income
Assistance" benefits for this Person.

This implementation may be suitable when the type is the most
selective attribute for a Benefit in the data storage (i.e.
there are few Benefit rule objects of each type).
-->
<Attribute name="incomeAssistanceBenefitsMatchTypeFilterSSN">
  <type>
    <javaclass name="List">
      <ruleclass name="Benefit"/>
    </javaclass>
  </type>
  <derivation>
    <filter>
      <list>
        <!-- retrieve all Benefit rule objects of type "Income
             Assistance" -->
        <readall ruleclass="Benefit">
          <match retrievedattribute="type">
            <Code table="BenefitType">
              <!-- The value for Income Assistance -->
              <String value="BT1"/>
            </Code>
          </match>
        </readall>
      </list>
      <listitemexpression>
        <equals>
          <!-- filter the Benefit rule objects of type "Income
               Assistance"  down to those for this Person only
            -->
          <reference attribute="socialSecurityNumber">
            <current/>
          </reference>
          <reference attribute="socialSecurityNumber"/>
        </equals>
      </listitemexpression>
    </filter>
  </derivation>
</Attribute>

<!--
Retrieves the rule objects for Benefits whose amount is greater
than 100.

Because "greater than" is not an exact match predicate, a
<filter> must be used (<match> can only be used for an exact
match criterion).
-->
<Attribute name="highPaymentBenefits">
  <type>
    <javaclass name="List">
      <ruleclass name="Benefit"/>
    </javaclass>
  </type>
  <derivation>
    <filter>
      <list>
        <!-- retrieve all Benefit rule objects for the Person
          -->
        <readall ruleclass="Benefit">
          <match retrievedattribute="socialSecurityNumber">
```

```
                          <reference attribute="socialSecurityNumber"/>
                      </match>
                  </readall>
              </list>
              <listitemexpression>
                <!-- filter the Benefit rule objects for the Person down
                       to those of amount greater than 100 only -->
                <compare comparison=">">
                  <reference attribute="amount">
                    <current/>
                  </reference>
                  <Number value="100"/>
                </compare>
              </listitemexpression>
          </filter>
        </derivation>
    </Attribute>


  </Class>

  <Class name="Benefit">

    <Attribute name="socialSecurityNumber">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="type">
      <type>
        <codetableentry table="BenefitType"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <!--
  This rule set expects that the code creating the session also
  creates a "bootstrap" single instance of this Claim rule
  class.
  -->
  <Class name="Claim">
    <Initialization>
      <Attribute name="claimIdentifier">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
      <Attribute name="claimDate">
        <type>
          <javaclass name="curam.util.type.Date"/>
        </type>
      </Attribute>
    </Initialization>
  </Class>

</RuleSet>
```

reference

The reference expression retrieves the value of an attribute from a rule object.

The `reference` expression can optionally contain a child expression which determines the rule object from which to obtain the attribute. If omitted, the rule object that contains the `reference` is used.

The `reference` expression is the key to building rules which are reusable and understandable. You can use a `reference` to a named attribute in place of any expression. The CER rule set validator will issue an error if the type of the referenced attribute does not match the type required by the expression.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_reference"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- A simple reference to another
         attribute on this rule class -->
    <Attribute name="ageNextYear">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="+">
          <!-- This <reference> has no child element,
               so the rule object is taken to be "this rule
  object"-->
          <reference attribute="age"/>
          <Number value="1"/>
        </arithmetic>
      </derivation>
    </Attribute>

    <Attribute name="favoritePet">
      <type>
        <ruleclass name="Pet"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Pet">

    <Attribute name="name">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="species">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
```

```
          <specified/>
        </derivation>
    </Attribute>

</Class>

<Class name="Household">

  <!-- All the people in the household -->
  <Attribute name="members">
    <type>
      <javaclass name="List">
        <ruleclass name="Person"/>
      </javaclass>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <!-- One special person is designated
       as the "head" of the household -->
  <Attribute name="headOfHousehold">
    <type>
      <ruleclass name="Person"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <!-- A reference to an attribute on a
       different rule object:

  name OF favoritePet OF headOfHousehold

  In a programming language, this might be
  written back-to-front using a dereferencing
  "dot" notation like this:

  headOfHousehold.favoritePet.name

  -->
  <Attribute name="nameOfHeadOfHouseholdsFavoritePet">
    <type>
      <javaclass name="String"/>
    </type>
    <derivation>

      <!-- The name... -->
      <reference attribute="name">

        <!-- ...of the favorite pet... -->
        <reference attribute="favoritePet">

          <!-- ...of the head of household. -->
          <!-- The inner-most reference must refer to
               an attribute on this rule object.  -->
          <reference attribute="headOfHousehold"/>
        </reference>
      </reference>
    </derivation>
  </Attribute>


  <!-- Identifies the dog owners in the household
       by checking which people have a favorite
       pet which is a dog. -->
  <Attribute name="dogOwners">
    <type>
      <javaclass name="List">
        <ruleclass name="Person"/>
      </javaclass>
    </type>
    <derivation>
      <filter>
        <list>
          <!-- simple reference to the members
               of the household -->
          <reference attribute="members"/>
        </list>
        <listitemexpression>
```

```
            <equals>
              <String value="Dog"/>
              <!-- A reference to an attribute on an item
                   in the list. -->

              <reference attribute="species">
                <reference attribute="favoritePet">
                  <current/>
                </reference>
              </reference>

            </equals>

          </listitemexpression>
        </filter>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

removeduplicates

The removeduplicates expression creates a new list by removing any duplicate items from an existing list.

If any item in the original list occurs more than once, the first instance only is kept. Otherwise, the ordering of items is preserved.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_removeduplicates"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- The list of relationships where
         this person is the "fromPerson". -->
    <Attribute name="relationships">
      <type>
        <javaclass name="List">
          <ruleclass name="Relationship"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The people who are related to this person.

         Any relative appears in this list only once,
         even though one person can be
         related to another in more than one way, e.g.
         my grandparent can also be my legal guardian.-->
    <Attribute name="uniqueRelatives">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <removeduplicates>
          <reference attribute="allRelatives"/>
        </removeduplicates>
      </derivation>
    </Attribute>

    <Attribute name="allRelatives">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <!-- get the relatives of this person by forming a
```

```
                list of the "toPerson" on the other end of each
                relationship. -->
          <dynamiclist>
            <list>
              <reference attribute="relationships"/>
            </list>
            <listitemexpression>
              <reference attribute="toPerson">
                <current/>
              </reference>
            </listitemexpression>
          </dynamiclist>
        </derivation>
      </Attribute>

    </Class>

    <!-- A relationship from one person to another. -->
    <Class name="Relationship">

      <Attribute name="fromPerson">
        <type>
          <ruleclass name="Person"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <Attribute name="relationshipType">
        <type>
          <javaclass name="String"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>


      <Attribute name="toPerson">
        <type>
          <ruleclass name="Person"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

    </Class>

  </RuleSet>
```

ResourceMessage

ResourceMessage creates a localizable message from a property resource.

For more information, see "Localizing CER Rules" on page 6.

Optionally, the property may specify placeholders for formatted arguments. The support and syntax for formatting is described in the JavaDoc for `MessageFormat`.

warning: As mentioned in the JavaDoc, if you need to output a single quote or apostrophe (*'*), you must specify two single quotes in the property text (*''*).

If you want to output XML or HTML, and do not require complex token formatting, or the ability to change message text without changing rules, consider using "XmlMessage" on page 207 instead.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_ResourceMessage"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="gender">
      <type>
        <javaclass name="String"/>
      </type>
```

```
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="isMarried">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="firstName">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="surname">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="income">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- Returns a greeting which can be
        output in the user's locale -->
    <Attribute name="simpleGreetingMessage">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <ResourceMessage key="simpleGreeting"
          resourceBundle="curam.creole.example.Messages"/>
      </derivation>
    </Attribute>

    <!-- Returns a greeting which contains
        the person's title and surname.
        The greeting and title are localized,
        the surname is not (it is identical
        in all locales). -->
    <Attribute name="parameterizedGreetingMessage">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- pass in arguments to
            the message placeholders -->
        <ResourceMessage key="parameterizedGreeting"
          resourceBundle="curam.creole.example.Messages">
          <!-- Title -->
          <choose>
            <type>
              <javaclass name="curam.creole.value.Message"/>
            </type>
            <when>
              <condition>
                <equals>
                  <reference attribute="gender"/>
                  <String value="Male"/>
                </equals>
              </condition>
              <value>
                <ResourceMessage key="title.male"
                  resourceBundle="curam.creole.example.Messages"/>
```

```
                        </value>
                    </when>
                    <when>
                        <condition>
                            <reference attribute="isMarried"/>
                        </condition>
                        <value>
                            <ResourceMessage key="title.female.married"
                                resourceBundle="curam.creole.example.Messages"/>
                        </value>
                    </when>
                    <otherwise>
                        <value>
                            <ResourceMessage key="title.female.single"
                                resourceBundle="curam.creole.example.Messages"/>
                        </value>
                    </otherwise>
                </choose>

                <!-- Surname -->
                <reference attribute="surname"/>

            </ResourceMessage>
        </derivation>
    </Attribute>

    <!-- Formats a number to 2 decimal places,
         with decimal point and thousands
         separator in the user's locale -->
    <Attribute name="incomeStatementMessage">
        <type>
            <javaclass name="curam.creole.value.Message"/>
        </type>
        <derivation>
            <ResourceMessage key="incomeStatement"
                resourceBundle="curam.creole.example.Messages">
                <reference attribute="income"/>
            </ResourceMessage>
        </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Example properties, English.

```
# file curam/creole/example/Messages_en.properties

simpleGreeting=Hello
parameterizedGreeting=Hello, {0} {1}
title.male=Mr.
title.female.single=Miss
title.female.married=Mrs.
incomeStatement=Income: USD{0,number,#0.00}
```

Example properties, French.

```
# file curam/creole/example/Messages_fr.properties

simpleGreeting=Bonjour
parameterizedGreeting=Bonjour, {0} {1}
title.male=M.
title.female.single=Mlle.
title.female.married=Mme.
incomeStatement=Revenue: EUR{0,number,#0.00}
```

singleitem

The singleitem expression retrieves a single item from a list.

The singleitem expression can be useful when it is expected that a list contains only a single item, for example, hen filtering a list by criteria which should pick out only a single item from the list.

The singleitem expression specifies:

- **onEmpty**

The behavior when the list is found to be empty:

– **error**

A runtime error occurs (use this option if the list is not expected to be empty); or

– **returnNull**

The value *null* is returned.

• **onMultiple**

The behavior when the list is found to contain more than one item:

– **error**

A runtime error occurs (use this option if the list is not expected to contain more than one item);

– **returnNull**

The value *null* is returned;

– **returnFirst**

The first item in the list is returned; or

– **returnLast**

The last item in the list is returned.

To retrieve an item from a specific position in a list, see get in <u>"Useful List Operations" on page 212</u>.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_singleitem"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="dateOfBirth">
      <type>
        <javaclass name="curam.util.type.Date"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="children">
      <type>
        <javaclass name="List">
          <ruleclass name="Person"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- The first child born to this person -->
    <Attribute name="firstBornChild">
      <type>
        <ruleclass name="Person"/>
      </type>
      <derivation>
        <!-- get the first child, if any
            - if no children, return null -->
        <singleitem onEmpty="returnNull" onMultiple="returnFirst">
          <!-- sort the children in date-of-birth order -->
          <sort>
            <list alias="child">
              <reference attribute="children"/>
            </list>
            <sortorder>
              <sortitem direction="ascending">
                <reference attribute="dateOfBirth">
                  <current alias="child"/>
                </reference>
              </sortitem>
            </sortorder>
```

```
            </sort>

          </singleitem>
        </derivation>

      </Attribute>

      <!-- Retrieve the single household information record
           from external storage - there should always
           be exactly one - anything else is an error.  -->
      <Attribute name="householdInformation">
        <type>
          <ruleclass name="HouseholdInformation"/>
        </type>
        <derivation>
          <singleitem onEmpty="error" onMultiple="error">
            <readall ruleclass="HouseholdInformation"/>
          </singleitem>
        </derivation>
      </Attribute>

    </Class>

    <Class name="HouseholdInformation">

      <Attribute name="householdContainsDisabledPerson">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

    </Class>

  </RuleSet>
```

sort

The `sort` expression creates a new list by sorting the members of an existing list according to a specified sort order.

A `sort` expression specifies:

- **list**

  the existing to sort (which will not be affected); and

- **sortorder**

  The order in which to sort the list.

The `sortorder` specifies one or more `sortitem` s, each of which specifies the item to sort by and whether to sort in ascending or descending order.

The `sortitem` s are listed most-significant first; each `sortitem` is only evaluated if two items being sorted are identical with regard to more significant `sortitem` s.

Within each `sortitem`, you can use "current" on page 157 to refer to the list item being sorted. Typically each `sortitem` will refer to some attribute or calculation on the "current" on page 157 list item.

If two (or more) items in the list are identical with regard to all the `sortitem` s, then they are returned in the same relative order as the source list.

The behavior of the `sort` expression is similar to SQL's ORDER BY clause.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_sort"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Household">

    <Attribute name="members">
      <type>
        <javaclass name="List">
```

```
            <ruleclass name="Person"/>
          </javaclass>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <!-- Arranges the members in order of age (oldest to youngest);
           for members which are the same age, the members are
           arranged in alphabetical order by first name. -->
      <Attribute name="sortedMembers">
        <type>
          <javaclass name="List">
            <ruleclass name="Person"/>
          </javaclass>
        </type>
        <derivation>
          <sort>
            <list>
              <reference attribute="members"/>
            </list>
            <sortorder>
              <sortitem direction="descending">
                <!-- The age of the person in the list -->
                <reference attribute="age">
                  <current/>
                </reference>
              </sortitem>
              <!-- The first name of the person in the list -->
              <sortitem direction="ascending">
                <reference attribute="firstName">
                  <current/>
                </reference>
              </sortitem>
            </sortorder>
          </sort>
        </derivation>
      </Attribute>

  </Class>

  <Class name="Person">

    <Initialization>
      <Attribute name="firstName">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
      <Attribute name="age">
        <type>
          <javaclass name="Integer"/>
        </type>
      </Attribute>
    </Initialization>

  </Class>

</RuleSet>
```

specified

The `specified` expression is a marker to denote that the attribute's value is specified externally. For example, by retrieval from a database or population by test code, rather than calculated by rules processing.

Typically, `specified` attributes denote information that comes directly from outside the system, and other attributes use this external information to derive new information.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_specified"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- This information cannot be calculated or derived -
         it must be specified from an external source -->
```

```
      <Attribute name="name">
        <type>
          <javaclass name="String"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <!-- This information cannot be calculated or derived -
           it must be specified from an external source -->
      <Attribute name="dateOfBirth">
        <type>
          <javaclass name="curam.util.type.Date"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

      <!-- Other attributes are likely to derive/calculation more
           information based on the "specified" attributes above -->
    </Class>
</RuleSet>
```

String

A `string` represents a literal String constant value.

For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_String"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="StringExampleRuleClass">

    <Attribute name="emptyString">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <!-- An empty String -->
        <String value=""/>
      </derivation>
    </Attribute>

    <Attribute name="helloWorld">
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <!-- The String "Hello, World!" -->
        <String value="Hello, World!"/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

sublists

The `sublists` expression calculates all the sublists of the list supplied, and returns these sublists as a list of lists.

For a list containing *n* elements, there are 2^n sublists, including the empty list and the original list.

The order of the list items in each of the sublists will be identical to the ordering in the original list.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_sublists"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Household">
```

```
        <Attribute name="members">
          <type>
            <javaclass name="List">
              <ruleclass name="Person"/>
            </javaclass>
          </type>
          <derivation>
            <fixedlist>
              <listof>
                <ruleclass name="Person"/>
              </listof>
              <members>
                <create ruleclass="Person">
                  <String value="Mother"/>
                </create>
                <create ruleclass="Person">
                  <String value="Father"/>
                </create>
                <create ruleclass="Person">
                  <String value="Child"/>
                </create>
              </members>
            </fixedlist>
          </derivation>
        </Attribute>

        <!-- All the different combinations of members of the household
  -->
        <Attribute name="memberCombinations">
          <!-- Note that the type is list of lists of Persons -->
          <type>
            <javaclass name="List">
              <javaclass name="List">
                <ruleclass name="Person"/>
              </javaclass>
            </javaclass>
          </type>
          <derivation>
            <sublists>
              <reference attribute="members"/>
            </sublists>
          </derivation>
        </Attribute>

  </Class>

  <Class name="Person">

    <Initialization>
      <Attribute name="name">
        <type>
          <javaclass name="String"/>
        </type>
      </Attribute>
    </Initialization>

  </Class>

</RuleSet>
```

In this example rule set, the value of *memberCombinations* is calculated as list of these 8 lists:

- an empty list (no household members);
- Mother;
- Father;
- Mother and Father;
- Child;
- Mother and Child;
- Father and Child; and
- Mother, Father and Child (the full original list).

sum

The sum expression calculates the numerical sum of a list of Number values.

If the list is empty, this expression returns *0*.

The list of Number values is typically provided by a "fixedlist" on page 167 or "dynamiclist" on page 159.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_sum"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <Attribute name="netWorth">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- Example of <sum> operating on a <fixedlist> -->
        <!-- A person's net worth is the sum of their
             cash, savings and assets -->
        <sum>
          <fixedlist>
            <listof>
              <javaclass name="Number"/>
            </listof>
            <members>
              <reference attribute="totalCash"/>
              <reference attribute="totalSavings"/>
              <reference attribute="totalAssets"/>
            </members>
          </fixedlist>
        </sum>
      </derivation>
    </Attribute>

    <Attribute name="totalAssets">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <!-- Example of <sum> operating on a <dynamiclist> -->
        <!-- The total value of a person's assets is derived by
             summing the value of each asset -->
        <sum>
          <dynamiclist>
            <list>
              <reference attribute="assets"/>
            </list>
            <listitemexpression>
              <reference attribute="value">
                <current/>
              </reference>
            </listitemexpression>
          </dynamiclist>
        </sum>
      </derivation>
    </Attribute>

    <!-- The assets of that this person owns -->
    <Attribute name="assets">
      <type>
        <javaclass name="List">
          <ruleclass name="Asset"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- NB this example doesn't show how
         total cash/savings is derived -->
    <Attribute name="totalCash">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
```

```
        </derivation>
      </Attribute>
      <Attribute name="totalSavings">
        <type>
          <javaclass name="Number"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>

  </Class>

  <Class name="Asset">

    <!-- The monetary value of the asset -->
    <Attribute name="value">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

this

The `this` expression is a reference to the current Rule Object, analogous to the keyword `this` in Java.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_this"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="Person">

    <!-- The pets owned by this Person -->
    <Attribute name="pets">
      <type>
        <javaclass name="List">
          <ruleclass name="Pet"/>
        </javaclass>
      </type>
      <derivation>
        <fixedlist>
          <listof>
            <ruleclass name="Pet"/>
          </listof>
          <members>
            <!-- Every Person has exactly two pets,
                 Skippy and Lassie -->
            <create ruleclass="Pet">
              <!-- set the owner to be THIS Person -->
              <this/>
              <String value="Skippy"/>
              <String value="Kangaroo"/>
            </create>
            <create ruleclass="Pet">
              <!-- set the owner to be THIS Person -->
              <this/>
              <String value="Lassie"/>
              <String value="Dog"/>
            </create>
          </members>
        </fixedlist>
      </derivation>
    </Attribute>

  </Class>

  <Class name="Pet">

    <Initialization>
      <Attribute name="owner">
        <type>
```

```
                <ruleclass name="Person"/>
            </type>
        </Attribute>
        <Attribute name="name">
            <type>
                <javaclass name="String"/>
            </type>
        </Attribute>
        <Attribute name="species">
            <type>
                <javaclass name="String"/>
            </type>
        </Attribute>
    </Initialization>

  </Class>

</RuleSet>
```

Timeline

The Timeline expression creates a Timeline of a given type with values that are valid from specified dates.

For more information, see "Handling Data that Changes Over Time" on page 42.

A timeline must have a value from the start-of-time (the `null` date), and so to assist with this, the `Timeline` expression contains an optional `initialvalue` element to specify the value from the start-of-time. If not used, then the collection of "Interval" on page 170 used *must* contain an interval with a `null` start date, otherwise an error will occur if this expression is evaluated at runtime.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Timeline"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="CreateTimelines">

    <!-- This example uses <initialvalue> to set the value valid
         from the start of time. -->
    <Attribute name="aNumberTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Number"/>
        </javaclass>
      </type>
      <derivation>
        <Timeline>
          <intervaltype>
            <javaclass name="Number"/>
          </intervaltype>
          <!-- Value from start of time -->
          <initialvalue>
            <Number value="0"/>
          </initialvalue>
          <!-- The remaining intervals -->
          <intervals>
            <fixedlist>
              <listof>
                <javaclass name="curam.creole.value.Interval">
                  <javaclass name="Number"/>
                </javaclass>
              </listof>
              <members>
                <Interval>
                  <intervaltype>
                    <javaclass name="Number"/>
                  </intervaltype>
                  <start>
                    <Date value="2001-01-01"/>
                  </start>
                  <value>
                    <Number value="10000"/>
                  </value>
                </Interval>
                <Interval>
                  <intervaltype>
                    <javaclass name="Number"/>
                  </intervaltype>
```

```xml
          <start>
            <Date value="2004-12-01"/>
          </start>
          <value>
            <Number value="12000"/>
          </value>
        </Interval>

      </members>
    </fixedlist>

    </intervals>
  </Timeline>

  </derivation>
</Attribute>


<!-- This example does not use <initialvalue>. -->
<Attribute name="aStringTimeline">
  <type>
    <javaclass name="curam.creole.value.Timeline">
      <javaclass name="String"/>
    </javaclass>
  </type>
  <derivation>
    <Timeline>
      <intervaltype>
        <javaclass name="String"/>
      </intervaltype>

      <!-- The list of intervals must include one valid from the
           null date (start of time), otherwise an error will
           occur at runtime, if this expression is evaluated. -->
      <intervals>
        <fixedlist>
          <listof>
            <javaclass name="curam.creole.value.Interval">
              <javaclass name="String"/>
            </javaclass>
          </listof>
          <members>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <!-- "from the start of time" -->
                <null/>
              </start>
              <value>
                <String value="Start of time string"/>
              </value>
            </Interval>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <Date value="2001-01-01"/>
              </start>
              <value>
                <String value="2001-only String"/>
              </value>
            </Interval>
            <Interval>
              <intervaltype>
                <javaclass name="String"/>
              </intervaltype>
              <start>
                <Date value="2002-01-01"/>
              </start>
              <value>
                <String value="2002-onwards String"/>
              </value>
            </Interval>
          </members>
        </fixedlist>
      </intervals>
    </Timeline>
  </derivation>
</Attribute>
```

```
      </Class>
</RuleSet>
```

timelineoperation

The timeline operation expression assembles a Timeline from repeated calls to a child expression.

For more information, see "Handling Data that Changes Over Time" on page 42. Typically `timelineoperation` is used with "intervalvalue" on page 171. Together these two expressions allow other expressions to operate on values from timelines as if they were primitive values, and then have the resultant data reassembled into a Timeline.

Tip: For each of the Timelines which are used as input into your algorithm, typically you should wrap the expression that returns the Timeline in an "intervalvalue" on page 171, and then wrap the overall result in a `timelineoperation`.

A brief description of how `timelineoperation` behaves at evaluation time is as follows:

- `timelineoperation` creates a new evaluation context to track the series of calls to make to its child expression (typically the child expression will be invoked several times, for different dates);
- `timelineoperation` invokes its single child expression with a context date of `null`, signifying the start-of-time;
- during the evaluation of the child expression (and its dependents), then whenever an "intervalvalue" on page 171 is encountered, then perform the following actions:
  - "intervalvalue" on page 171 evaluates its single child expression to obtain a Timeline, and from this Timeline get the value on the date corresponding to the current date in the `timelineoperation` 's evaluation context;
  - "intervalvalue" on page 171 checks the other dates on which the Timeline changes value, and for each of these dates add them to a queue of other dates that the `timelineoperation` should operate on (in a subsequent invocation);
- when control returns to `timelineoperation`, a value will have been calculated for a particular date, and additional dates identified on which the input timelines change value. For each date, `timelineoperation` re-invokes the child expression (on this date) until no more dates are in the queue.

The behavior described above means that the inner expressions never have to know that they are part of processing involving timelines. Furthermore, processing is efficient because expressions are only invoked for dates on which the input timelines change value.

Note: If a `timelineoperation` operates on an expression which is *not* wrapped by an "intervalvalue" on page 171, then the resultant Timeline will have a constant value for all time.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_timelineoperation"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Class name="Person">
    <!--
     true during a person's lifetime; false before date of birth,
     and false again after date of death (if any)
     -->
    <Attribute name="isAliveTimeline">
      <type>
        <javaclass name="curam.creole.value.Timeline">
          <javaclass name="Boolean"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!--
     The assets owned by the person, at some time or another.  Each
     asset's value may vary over time.
```

```
  -->
<Attribute name="ownedAssets">
  <type>
    <javaclass name="List">
      <ruleclass name="Asset"/>
    </javaclass>
  </type>
  <derivation>
    <specified/>
  </derivation>
</Attribute>


<!--
 The total value of the assets owned by the person (or by the
 person's estate, if the person has died).
 -->
<Attribute name="totalAssetValueTimeline">
  <type>
    <javaclass name="curam.creole.value.Timeline">
      <javaclass name="Number"/>
    </javaclass>
  </type>
  <derivation>
    <!--
     Total the value of all the owned assets.  The value of each
     asset may change over time.
     -->

    <!--
     <timelineoperation> will create a timeline from the series
     of <sum> calculations performed within it.

     Each execution of <sum> will calculate the total for a
     particular day; <timelineoperation> will assemble these
     daily totals into a timeline of numbers.
     -->
    <timelineoperation>

      <sum>
        <!--
         For each owned asset, get its countable-value timeline.
         -->
        <dynamiclist>
          <list>
            <reference attribute="ownedAssets"/>
          </list>
          <listitemexpression>

            <!--
             Wrap the timeline returned by
             countableValueTimeline, so that the <sum> thinks
             that it is operating on a list of numbers, not a
             list of timelines.
             -->
            <intervalvalue>
              <reference attribute="countableValueTimeline">
                <current/>
              </reference>
            </intervalvalue>
          </listitemexpression>
        </dynamiclist>

      </sum>

    </timelineoperation>

  </derivation>
</Attribute>


<!--
 The cut-off point for being entitled to benefit.  Persons with
 assets above this varying threshold do not qualify for benefit.

 -->
<Attribute name="maximumAssetsThreshold">
  <type>
    <javaclass name="curam.creole.value.Timeline">
      <javaclass name="Number"/>
    </javaclass>
```

```xml
        </type>
        <derivation>
          <!--
            In a real implementation, this value would tend to vary
            over time (e.g. from a rate table).

            However, for the sake of example, this implementation uses
            a <timelineoperation> WITHOUT a nested <intervalvalue> to
            create a Timeline which has a constant value for all time.

            This use of <timelineoperation> can often be useful for
            dummy implementations early on in rule set development.
          -->

          <timelineoperation>
            <!--
              Hard coded forever-constant value - to be replaced by a
              varying value later in rules development.
             -->
            <Number value="10000"/>
          </timelineoperation>
        </derivation>
      </Attribute>

      <!--
        The person qualified for benefit if (on any particular day)
        the person is alive and the total value of the person's assets
        does not exceed the maximum asset threshold.
       -->
      <Attribute name="qualifiesForBenefitTimeline">
        <type>
          <javaclass name="curam.creole.value.Timeline">
            <javaclass name="Boolean"/>
          </javaclass>
        </type>
        <derivation>
          <timelineoperation>
            <all>
              <fixedlist>
                <listof>
                  <javaclass name="Boolean"/>
                </listof>
                <members>
                  <!--
                    operate on the Timelines as if they were primitive
                    values
                   -->
                  <intervalvalue>
                    <reference attribute="isAliveTimeline"/>
                  </intervalvalue>

                  <compare comparison="&lt;=">
                    <intervalvalue>
                      <reference attribute="totalAssetValueTimeline"/>
                    </intervalvalue>
                    <intervalvalue>
                      <reference attribute="maximumAssetsThreshold"/>
                    </intervalvalue>
                  </compare>
                </members>
              </fixedlist>

            </all>
          </timelineoperation>

        </derivation>
      </Attribute>


  </Class>

  <!--
    An asset, owned by a person at some time or other.

    Each asset is bought, and may subsequently be sold.

    The value of an asset varies over time; the asset still has a
    value even before or after it is owned by a Person; however,
    the _countable_ value is zero outside of the period of
    ownership.
     -->
```

```
<Class name="Asset">
  <Attribute name="boughtDate">
    <type>
      <javaclass name="curam.util.type.Date"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <!-- will be null if the asset has not been sold -->
  <Attribute name="soldDate">
    <type>
      <javaclass name="curam.util.type.Date"/>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>

  <Attribute name="isOwnedTimeline">
    <type>
      <javaclass name="curam.creole.value.Timeline">
        <javaclass name="Boolean"/>
      </javaclass>
    </type>
    <derivation>
      <existencetimeline>
        <intervaltype>
          <javaclass name="Boolean"/>
        </intervaltype>
        <intervalfromdate>
          <reference attribute="boughtDate"/>
        </intervalfromdate>
        <intervaltodate>
          <reference attribute="soldDate"/>
        </intervaltodate>
        <preExistenceValue>
          <false/>
        </preExistenceValue>
        <existenceValue>
          <true/>
        </existenceValue>
        <postExistenceValue>
          <false/>
        </postExistenceValue>
      </existencetimeline>

    </derivation>
  </Attribute>

  <!--
   the varying value of the asset, regardless of whether it is
   owned by the person at the time
   -->
  <Attribute name="valueTimeline">
    <type>
      <javaclass name="curam.creole.value.Timeline">
        <javaclass name="Number"/>
      </javaclass>
    </type>
    <derivation>
      <specified/>
    </derivation>
  </Attribute>


  <!--
   The value which counts towards the person's assets - i.e. the
   value of the asset during the period when it is owned,
   otherwise 0 when it is not owned.
   -->
  <Attribute name="countableValueTimeline">
    <type>
      <javaclass name="curam.creole.value.Timeline">
        <javaclass name="Number"/>
      </javaclass>
    </type>
    <derivation>
      <!--
       reassemble the outputs from each <choose> invocation into a
       timeline
```

```
            -->
        <timelineoperation>
          <choose>
            <type>
              <javaclass name="Number"/>
            </type>
            <when>
              <condition>
                <!--
                 operate on each of the intervals of constant
                 ownership
                -->

                <intervalvalue>
                  <reference attribute="isOwnedTimeline"/>
                </intervalvalue>
              </condition>
              <value>
                <!--
                 if on a particular date, the asset is owned, then
                 its countable value on that date is simply its
                 value
                 -->
                <intervalvalue>
                  <reference attribute="valueTimeline"/>
                </intervalvalue>
              </value>
            </when>
            <otherwise>
              <value>
                <!--
                 if on a particular date, the asset is owned, then
                 its countable value on that date is zero
                 -->
                <Number value="0"/>
              </value>
            </otherwise>
          </choose>

        </timelineoperation>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Tip: If an inner expression returns a Timeline, and you forget to wrap that expression in an <u>"intervalvalue"</u> <u>on page 171</u>, you will see CER validation errors as in this example:

```
<!--
    The value which counts towards the person's assets - i.e. the
    value of the asset during the period when it is owned,
    otherwise 0 when it is not owned.
    -->
  <Attribute name="countableValueTimeline">
    <type>
      <javaclass name="curam.creole.value.Timeline">
        <javaclass name="Number"/>
      </javaclass>
    </type>
    <derivation>
      <!--
       reassemble the outputs from each <choose> invocation into a
       timeline
       -->
      <timelineoperation>
        <choose>
          <type>
            <javaclass name="Number"/>
          </type>
          <when>
            <condition>
              <!--
               operate on each of the intervals of constant
               ownership
               -->

              <!--
                **** Forgot to wrap the Timeline returned
```

```
                      in <intervalvalue> ****
                      -->
                      <reference attribute="isOwnedTimeline"/>
                  </condition>
                  <value>
                    <!--
                     if on a particular date, the asset is owned, then
                     its countable value on that date is simply its
                     value
                     -->
                    <intervalvalue>
                        <reference attribute="valueTimeline"/>
                    </intervalvalue>
                  </value>
                </when>
                <otherwise>
                  <value>
                    <!--
                     if on a particular date, the asset is owned, then
                     its countable value on that date is zero
                     -->
                    <Number value="0"/>
                  </value>
                </otherwise>
              </choose>

          </timelineoperation>
        </derivation>
      </Attribute>
```

Example error.

```
ERROR    ... Example_timelineoperation.xml(276, 19)
  AbstractRuleItem:INVALID_CHILD_RETURN_TYPE: Child 'condition' returns
  'curam.creole.value.Timeline<? extends java.lang.Boolean>',
  but this item requires a 'java.lang.Boolean'.
```

true

The `true` expression represents the Boolean constant value "true".

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_true"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="TrueExampleRuleClass">

    <Attribute name="isCuramExpertRulesFantastic">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <true/>
      </derivation>
    </Attribute>

    <Attribute name="didCookbookWinPulitzerPrize">
      <type>
        <javaclass name="Boolean"/>
      </type>
      <derivation>
        <not>
          <true/>
        </not>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

XmlMessage

The `XmlMessage` expression creates a localizable message from free-form XML content.

For more information, see "Localizing CER Rules" on page 6.

The message that is created is the literal XML content within the XmlMessage element, with one exception: The content of any replace element will be set to the expression it contains.

The replace element as a simple token replacement mechanism; if you require more complex token formatting, or the ability to change message text without changing rules, consider using "ResourceMessage" on page 190 instead.

Note: Prior to Cúram V6, XmlMessage trimmed whitespace surrounding any embedded XML characters. From Cúram V6 onwards, XmlMessage no longer trims any whitespace and preserves the source format of the XML characters (removing any XML comments).

If you require the pre-Cúram V6 trimming behavior of XmlMessage, then you must set the application environment variable *curam.creole.XmlFormat.enableWhitespaceTrimming* to the value *true* in your development environment.

In a production environment, you should ensure that if the value of the *curam.creole.XmlFormat.enableWhitespaceTrimming* environment variable is changed dynamically, you take steps to ensure that any derived data in your system that depends on rule attributes which use the XmlMessage expression must be forced to recalculate all stored attribute value instances.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_XmlMessage"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">
  <Class name="XmlMessageExampleRuleClass">

    <Attribute name="emptyMessage">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- contains no XML at all -->
        <XmlMessage/>
      </derivation>
    </Attribute>

    <Attribute name="simpleHtmlMessage">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <!-- Using XmlMessage can ensure that
             XML elements are started and
             ended correctly, e.g. <b> and </b> -->
        <XmlMessage>The following text will appear in bold in a
          browser: <b>Some in bold text.</b>
        </XmlMessage>
      </derivation>
    </Attribute>

    <Attribute name="tokenReplacementHtmlMessage">
      <type>
        <javaclass name="curam.creole.value.Message"/>
      </type>
      <derivation>
        <XmlMessage><p/>This calculated number will appear in
          italics and formatted according to locale preferences:<i>
            <replace>
              <arithmetic operation="+">
                <Number value="1.23"/>
                <Number value="3.45"/>
              </arithmetic>
            </replace>
          </i>
          <p/>And here's a resource message: <replace>
            <ResourceMessage key="simpleGreeting"
              resourceBundle="curam.creole.example.Messages"/>
          </replace>
        </XmlMessage>
      </derivation>
    </Attribute>

  </Class>
```

```
</RuleSet>
```

Annotations

CER supports annotations. An annotation is extra metadata in a rule set that is available to clients of CER, but does not affect the behavior of CER calculations.

Clients of CER can make use of annotations to govern their behavior when interacting with CER rule sets.

You can add an annotations to the following items in a CER rule set. However, each annotation can contain a validation to limit where it is placed.

- A rule set, see "Rule Set" on page 133
- A rule class, see "Rule Class" on page 134)
- A rule attribute, see "Attribute" on page 136
- An expression, see "Expressions" on page 136

Each rule item may contain zero, one or many annotations. Each type of annotation may appear at most once on any one rule item - for example, a rule set may contain both a `Label` annotation and an `EditorMetadata` annotation, but cannot contain more than one `Label` annotation.

Full Alphabetical Listing of Annotations

This section defines alphabetically lists and defines all of the annotations that are included with the application.

Note: Some annotations are for business-specific derivations in the application. The annotations are included here. However, you are referred to other information which describe those annotations in their business context.

ActiveInEditSuccessionSetPopulation

For information about this annotation, see Configuring Advisor.

Display

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

DisplaySubscreen

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

EditorMetadata

The `EditorMetadata` annotation stores diagram information for a rule set and is maintained automatically by the CER editor.

This annotation can be specified on a rule set only, see "Rule Set" on page 133.

Indexed

The `Indexed` annotation is no longer used and is included here only for backward compatibility.

Label

Provides a localized description of a rule set element:

- a rule set (see "Rule Set" on page 133);
- a rule class (see "Rule Class" on page 134);
- a rule attribute (see "Attribute" on page 136); or

- an expression (see "Expressions" on page 136).

The label contains an identifier (set by the CER editor) and a description (entered by the user).

When a CER rule set is saved or published, the values of the label annotations in a rule are used to write a property resource (in the locale of the user) to the application resource store. Conversely, when a rule set is displayed in the CER Editor, the property resource for the user's locale is retrieved from the resource store and used to populate the value of the label annotations in the rule set XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_Label"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Annotations>
    <!-- Rule-set level description -->
    <Label name="Example rule set for labels"
 label-id="annotation1"/>
  </Annotations>
  <Class name="Person">
    <Annotations>
      <!-- Rule-class level description -->
      <Label name="A Person" label-id="annotation2"/>
    </Annotations>
    <Attribute name="age">
      <Annotations>
        <!-- Attribute-class level description -->
        <Label name="The current age of the person, in years"
 label-id="annotation3"/>
      </Annotations>
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified>
          <Annotations>
            <!-- Expression-level description -->
            <Label name="This value comes directly from evidence"
 label-id="annotation4"/>
          </Annotations>
        </specified>
      </derivation>
    </Attribute>

    <Attribute name="ageNextBirthday">
      <Annotations>
        <!-- Attribute-class level description -->
        <Label name="The age of the person at the person's next
 birthday, in years"
          label-id="annotation5"/>
      </Annotations>
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <arithmetic operation="+">
          <Annotations>
            <!-- Expression-level description -->
            <Label name="Compute the person's age at next birthday"
 label-id="annotation6"/>
          </Annotations>
          <reference attribute="age">
            <Annotations>
              <!-- Expression-level description -->
              <Label name="Get the person's current age"
 label-id="annotation7"/>
            </Annotations>
          </reference>
          <Number value="1">
            <Annotations>
              <!-- Expression-level description -->
              <Label name="The number to add to get the age next
 birthday" label-id="annotation8"/>
            </Annotations>
          </Number>
        </arithmetic>
      </derivation>
    </Attribute>
```

```
    </Class>
</RuleSet>
```

Legislation

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

SuccessionSetPopulation

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

primary

Identifies the primary attribute on a rule class, as designated in the CER editor.

This annotation may be specified on a rule class (see "Rule Class" on page 134) only. The named attribute must be the exact name of an attribute declared on the rule class (it cannot be used to name an attribute which is inherited but not overridden).

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_primary"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">


  <Class name="Person">
    <Annotations>
      <!-- Declaration of the "primary" rule attribute for this rule
 class, as shown in the CER Editor -->
      <primary attribute="age"/>
    </Annotations>
    <Attribute name="age">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

relatedActiveInEditSuccessionSet

For information about this annotation, see Configuring Advisor.

relatedEvidence

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

relatedSuccessionSet

For information about this annotation, see Developing with Eligibility and Entitlement by Using Cúram Express Rules.

tags

Associates arbitrary string tags with:

- a rule set (see "Rule Set" on page 133);
- a rule class (see "Rule Class" on page 134);
- a rule attribute (see "Attribute" on page 136); or
- an expression (see "Expressions" on page 136).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="Example_tags"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <Annotations>
    <!-- Rule-set level tags -->
    <tags>
      <tag value="A rule-set tag"/>
      <tag value="Another tag"/>
    </tags>
  </Annotations>
  <Class name="Person">
    <Annotations>
      <!-- Rule-class level tags-->
      <tags>
        <tag value="A rule-class tag"/>
        <tag value="Another tag"/>
      </tags>
    </Annotations>
    <Attribute name="age">
      <Annotations>
        <!-- Attribute-class level tags -->
        <tags>
          <tag value="A rule-attribute tag"/>
          <tag value="Another tag"/>
        </tags>
      </Annotations>
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified>
          <Annotations>
            <!-- Expression-level tags -->
            <tags>
              <tag value="An expression tag"/>
              <tag value="Another tag"/>
            </tags>
          </Annotations>
        </specified>
      </derivation>
    </Attribute>

  </Class>

</RuleSet>
```

Useful List Operations

You can invoke safe Java methods by using the `property` expression.

For more information, see "Property" on page 108 and "property" on page 179.

Rule sets often contain many instances of `java.util.List`.

The safe list of methods for `java.util.List` is included with CER:

Safe list for `java.util.List` methods.

```
# Safe list for java.util.List

        contains.safe=true
        containsAll.safe=true
```

```
                get.safe=true

                indexOf.safe=true
                isEmpty.safe=true
                lastIndexOf.safe=true
                size.safe=true
                subList.safe=true


                # not exposed
                hashCode.safe=false
                listIterator.safe=false
                iterator.safe=false
                toArray.safe=false

                # mutators - unsafe
                add.safe=false
                addAll.safe=false
                clear.safe=false
                remove.safe=false
                removeAll.safe=false
                retainAll.safe=false
```

While the description of these methods is available via the JavaDoc for `java.util.List`, the most typically useful properties are included here for your reference:

- **isEmpty()**

  Returns *true* if this list contains no elements.

- **size()**

  Returns the number of elements in this list.

- **get(int index)**

  Returns the element at the specified position in this list. Note that because CER passes round numeric values as instances of `Number`, you will need to use the `intValue` to convert a `Number` to an integer.

- **contains(Object o)**

  Returns *true* if this list contains the specified element.

```xml
<?xml version="1.0" encoding="UTF-8"?>
        <RuleSet name="Example_UsefulListOperations"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
        "http://www.curamsoftware.com/CreoleRulesSchema.xsd">

        <Class name="Person">

        <!-- Exactly one Person (in each household) will
        be designated as the head of household -->
        <Attribute name="isHeadOfHousehold">
        <type>
        <javaclass name="Boolean"/>
        </type>
        <derivation>
        <specified/>
        </derivation>
        </Attribute>

        <!-- The children of this person. -->
        <Attribute name="children">
        <type>
        <javaclass name="List">
        <ruleclass name="Person"/>
        </javaclass>
        </type>
        <derivation>
        <specified/>
        </derivation>
        </Attribute>

        <!-- Whether this person has any children.

        Tests the isEmpty property of List. -->
```

```xml
<Attribute name="hasChildren">
<type>
<javaclass name="Boolean"/>
</type>
<derivation>
<not>
<property name="isEmpty">
<object>
<reference attribute="children"/>
</object>
</property>
</not>
</derivation>
</Attribute>

<Attribute name="numberOfChildren">
<type>
<javaclass name="Number"/>
</type>
<derivation>
<property name="size">
<object>
<reference attribute="children"/>
</object>
</property>
</derivation>
</Attribute>

<!-- This person's second child, if any, otherwise null -->
<Attribute name="secondChild">
<type>
<ruleclass name="Person"/>
</type>
<derivation>
<!-- We have to check whether the person has two
or more children -->
<choose>
<type>
<ruleclass name="Person"/>
</type>
<when>
<condition>
<compare comparison=">=">
<reference attribute="numberOfChildren"/>
<Number value="2"/>
</compare>
</condition>
<value>
<!-- Use the "get" property to get the second item
in the list - denoted by index 1 (lists in
Java are zero-based) -->
<property name="get">
<object>
<reference attribute="children"/>
</object>
<arguments>
<!-- The number must be converted to an integer
(as required by List.subList). -->
<property name="intValue">
<object>
<Number value="1"/>
</object>
</property>

</arguments>
</property>
</value>
</when>
<otherwise>
<!-- This person has no second child -->
<value>
<null/>
</value>
</otherwise>
</choose>
</derivation>
</Attribute>

<Attribute name="isChildOfHeadOfHousehold">
<type>
<javaclass name="Boolean"/>
</type>
```

```
<derivation>
<property name="contains">
<object>
<!-- The children of the head of household -->
<reference attribute="children">
<!-- retrieve the single Person rule object which
has isHeadOfHousehold equal to true-->
<singleitem onEmpty="error" onMultiple="error">
<readall ruleclass="Person">
<match retrievedattribute="isHeadOfHousehold">
<true/>
</match>
</readall>
</singleitem>
</reference>
</object>
<!-- check whether the list of the head of household's
children contains THIS Person -->
<arguments>
<this/>
</arguments>
</property>
</derivation>
</Attribute>

</Class>

</RuleSet>
```

Using CER with the Datastore

The application includes integration code that can create CER rule objects from entries in the application datastore. The `DataStoreRuleObjectCreator` is used by the Universal Access Responsive Web Application to convert evidence gathered by an IEG script to CER rule objects. This information describes how the `DataStoreRuleObjectCreator` works.

The `DataStoreRuleObjectCreator`

The `DataStoreRuleObjectCreator` takes a Datastore record (typically a record relating to a user or person), and navigates to all the descendant records of this "root" record (typically containing all the person's gathered evidence).

It then proceeds to create rule objects by performing a straightforward "natural mapping" between:

• The entity types and attributes in the Datastore schema.

• The rule classes and rule attribute in the CER rule set.

The `DataStoreRuleObjectCreator` also takes special action for CER rule attributes with certain names:

• **parentEntity**

  If a rule class contains a rule attribute named `parentEntity`, then the `DataStoreRuleObjectCreator` will set its value to be the rule object created from the parent record in the Datastore (if any). CER will issue a runtime error if the type of this rule attribute does not match the rule class of the parent entity's rule object; and

• **childEntities_<rule class name>**

  If a rule class contains any attributes named `childEntities_` followed by the name of a rule class, then the `DataStoreRuleObjectCreator` will set each such attribute's value to be a list of the rule objects created from the child records of that type in the Datastore (if any). CER will issue a runtime error if the type of this rule attribute is not a list of the named rule class.

DataStoreRuleObjectCreator Example

Use this example to learn about the behavior of `DataStoreRuleObjectCreator`. This example is based on the Universal Access Responsive Web Application.

An IEG script might capture income data for a household. The household can contain any number of persons, and each person can have any number of income details.

Here is a simplified view of the structure of the Datastore schema:

- Application
  - Person (0..n)
    - firstName (String)
    - lastName (String
    - Income (0..n)
      - type (Code from the IncomeType code table)
      - amount (Number)

A citizen (John) undertakes self-screening, and records evidence for his household (just John and his wife Mary) and income details (John is unemployed, Mary has two part-time jobs). John's evidence is stored as records in the Datastore:

- Application #1234
  - Person #1235
    - firstName: John
    - lastName: Smith
    - Income <no records>
  - Person #1236
    - firstName: Mary
    - lastName: Smith
    - Income #1238
      - type: Part-time
      - amount 30

The CER rule set configured to be used with this type of screening contains some rule classes as follows (NB no program rule classes are shown):

```
<?xml version="1.0" encoding="UTF-8"?>
<RuleSet name="DataStoreMappingExample"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation=
"http://www.curamsoftware.com/CreoleRulesSchema.xsd">

  <!-- NB no rule class for the CDS entity type "Application";
       the attributes stored directly on an Application are
       not required by rules, so no point creating a rule object
       which won't get used. -->

  <!-- The name of this rule class matches that of a CDS entity
       type -->
  <Class name="Person">
    <!-- The name of this rule attribute matches that of an
         attribute on the CDS entity type, and so its value will
         be automatically specified by the
         DataStoreRuleObjectRetriever. -->
    <Attribute name="firstName">
      <!-- The type of the rule attribute must agree with the
           type of the CDS attribute type, otherwise CER will
           issue a runtime error. -->
```

```
      <type>
        <javaclass name="String"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <!-- NB no rule attribute for the CDS attribute for lastName.
-->

    <!-- Rule attributes matching the "childEntities_<rule class
name>"
         pattern will receive special treatment from the
         DataStoreRuleObjectRetriever.

         The DataStoreRuleObjectRetriever will specify the value of
         this attribute to be all the rule objects created from the
         child Income records which belong to this Person's record
         in the CDS. -->
    <Attribute name="childEntities_Income">
      <!-- The type must be a list of Income rule objects -->
      <type>
        <javaclass name="List">
          <ruleclass name="Income"/>
        </javaclass>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <!-- The name of this rule class does not match any CDS entity
type,
       so the DataStoreRuleObjectRetriever will not create any rule
       objects for this rule class. -->
  <Class name="Benefit">
    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>
  </Class>

  <Class name="Income">
    <!-- A rule attribute named "parentEntity" will receive
         special treatment from the
         DataStoreRuleObjectRetriever.

         The DataStoreRuleObjectRetriever will specify the value
         of this attribute to be the rule object created from the
         Person record which is the parent of this Income record
         in the CDS. -->
    <Attribute name="parentEntity">
      <!-- The type must be a single Person rule object -->
      <type>
        <ruleclass name="Person"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="type">
      <type>
        <!-- The type of this attribute must specify the correct
             code table, matching the CDS domain definition. -->
        <codetableentry table="IncomeType"/>
      </type>
      <derivation>
        <specified/>
      </derivation>
    </Attribute>

    <Attribute name="amount">
      <type>
        <javaclass name="Number"/>
      </type>
```

```
        <derivation>
          <!-- This derivation will never be executed, as
               the DataStoreRuleObjectRetriever will automatically
               "specify" the value from that on the CDS record;
               once a value is specified CER does not attempt to
               calculate it.

               In general, attributes which expect to be populated
               by the DataStoreRuleObjectRetriever should be marked
               as <specified/> to avoid any confusion between
               stand-alone testing of your rule sets and testing
               with the DataStoreRuleObjectRetriever.
          -->
          <Number value="123"/>
        </derivation>
      </Attribute>

      <!-- This attribute is not present on the CDS entity type,
           so will not be populated.  This is exactly what we
           want, because its value is derived from other
           rule attributes in the normal CER way. -->
      <Attribute name="isCountable">
        <type>
          <javaclass name="Boolean"/>
        </type>
        <derivation>
          <choose>
            <type>
              <javaclass name="Boolean"/>
            </type>
            <test>
              <reference attribute="type"/>
            </test>
            <when>
              <condition>
                <Code table="IncomeType">
                  <String value="Full-time"/>
                </Code>
              </condition>
              <value>
                <true/>
              </value>
            </when>
            <when>
              <condition>
                <Code table="IncomeType">
                  <String value="Part-time"/>
                </Code>
              </condition>
              <value>
                <true/>
              </value>
            </when>
            <otherwise>
              <value>
                <false/>
              </value>
            </otherwise>
          </choose>
        </derivation>
      </Attribute>

      <!-- This rule attribute is not calculated, nor
           does it correspond to an attribute on the
           CDS entity type.

           If the value of this attribute is referenced
           at runtime, CER will report a runtime error:
           "Value must be specified before it is used
            (it cannot be calculated)."
      -->
      <Attribute name="employerName">
        <type>
          <javaclass name="String"/>
        </type>
        <derivation>
          <specified/>
        </derivation>
      </Attribute>
    </Class>
```

```
</RuleSet>
```

When John completes his self-screening, the Universal Access Module loads the CER rule set above, and creates a CER session.

The Universal Access Module calls the `DataStoreRuleObjectCreator`, and specifies the root Datastore record (Application #1234).

The `DataStoreRuleObjectCreator` retrieves all the descendant records of Application #1234 and processes them as follows:

- **Application #1234**

  Skipped, as there is no rule class named "Application" in the rule set;

- **Person #1235**

  creates a rule object instance of the `Person` rule class, with:

  - **firstName**

    Specified to be "John";

  - **lastName**

    Skipped, as there is no rule attribute named "lastName" on the `Person` rule class;

  - **childEntities_Income**

    Specified to be an empty list, as there are no Income records for Person record #1235;

- **Person #1236**

  creates a rule object instance of the `Person` rule class, with:

  - **firstName**

    Specified to be "Mary";

  - **lastName**

    Skipped;

  - **childEntities_Income**

    Specified to be a list containing Mary's two Income rule objects (created below);

- **Income #1237**

  creates a rule object instance of the `Income` rule class, with:

  - **parentEntity**

    Specified to be the `Person` rule object created (above) for Mary from Person record #1236;

  - **type**

    Specified to be the code "Part-time";

  - **amount**

    Specified to be the number "25";

  - **(employerName)**

    Not specified, as there is no attribute named "employerName" on the Datastore entity;

- **Income #1238**

  creates a rule object instance of the `Income` rule class, with:

  - **parentEntity**

    Specified to be the `Person` rule object created (above) for Mary from Person record #1236;

  - **type**

Specified to be the code "Part-time"; and

- **amount**

    Specified to be the number "30".

Lastly, the Universal Access Module asks the rule set its standard questions about programs, eligibility and explanation; the rule classes for the programs (not shown in the example above) access the rule objects (created by the `DataStoreRuleObjectCreator`) when answering those questions.

For more information on the application Datastore and its schemas, see the `Creating Datastore Schemas` guide.

CER Compliancy

Learn about how to develop CER rules in a compliant way. Follow these guidelines to make it easier to upgrade to future versions of the application.

Use of example artifacts

Any example artifacts that are included in this information, such as rule sets, Java code, and property files are subject to change or removal without notice.

You can freely copy example artifacts for your own use. However, note that no upgrade support is provided for example artifacts.

CER public API

The CER infrastructure has a public API that you can call in your rule set tests and application code. The examples show you how to use many features of the CER infrastructure public API.

The application does not change or remove anything in this public API without following the standards for handling customer impact.

Unless explicitly permitted in the Javadoc, you must not provide your own implementation of any CER Java interface nor any CER implementation Java subclass.

Identifying the API

The Javadoc that is included with CER is the sole means of identifying which public classes, interfaces, and methods form the CER public API.

Outside the API

The CER infrastructure also contains some public classes, interfaces, and methods that do not form part of the API.

Important: To be compliant, you must not create any dependency on any CER class or interface, or call any method that is not in the Javadoc information.

CER classes, interfaces, and methods outside of the public API are subject to change or removal without notice.

Unless explicitly permitted in the Javadoc information, you must not place any of your own classes or interfaces in the `curam.creole` package or any of its sub-packages.

CER expressions compliancy

Do not use any unsupported CER expressions in your rule sets. Only the CER expressions that are listed in documentation are supported.

For a full list of supported expressions, see "Full Alphabetical Listing of Expressions" on page 139.

The rule set schema that is included with CER contains some unsupported expressions. These expressions are experimental and are subject to change or removal without notice. Do not use these unsupported expressions in your rule sets.

Compliancy for included rule sets

Some components in the application can include CER rule sets. These included CER rule sets might have their own compliancy requirements, which are outside the scope of this guidance.

See the documentation for those components to understand whether you are permitted to customize the CER rule set, and if so, what customization restrictions apply.

The RootRuleSet that is included with CER must not be altered by customers.

CER database tables compliancy

The CER infrastructure includes a number of database tables. In general, these tables are internal to CER and the data on them can be read or written to only through the CER public API. The CREOLERuleSet database table is an exception. In certain specific conditions only, you can populate the CREOLERuleSet database table with DMX files.

Note: All CER infrastructure database tables are prefixed with the word CREOLE. However, the reverse is not true.

Tables prefixed with CREOLE are part of other application components and subject to their own compliancy statements. The following compliancy information applies only to CER infrastructure database tables.

The **CREOLERuleSet** database table

The CREOLERuleSet database table stores a row for each published CER rule set in the system. In certain conditions, you can populate the database table.

Typically, read and write access to the CREOLERuleSet database table is restricted to the CER public API.

However, you can use the Data Manager to populate the database table only when both of the following specific criteria are met.

- The value of the name column must match the rule set name that is defined in the XML for the `ruleSetDefinition` column.
- The value of the `ruleSetVersion` column must be `null`.

The following sample entry is from a compliant `CREOLERuleSet.dmx` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<table name="CREOLERULESET">
<column name="creoleRuleSetID" type="id"/>
<column name="name" type="text"/>
<column name="ruleSetDefinition" type="blob"/>
<column name="versionNo" type="number"/>
<column name="ruleSetVersion" type="number"/>
<row> ... </row>
<row>
<attribute name="creoleRuleSetID">
<!-- Use some appropriate unique identifier -->
<value>99999</value>
</attribute>
<attribute name="name">
<!-- This name must match the <RuleSet name="..."> 
value in the rule set XML held in ruleSetDefinition
below -->
<value>MyRuleSet</value>
</attribute>
<attribute name="ruleSetDefinition">
<value>./path/to/MyRuleSet.xml</value>
</attribute>
<attribute name="ruleSetVersion">
<!-- Must be an empty <value/> element, signifying a NULL
database value -->
```

```
    <value/>
    </attribute>
    <attribute name="versionNo">
    <!-- initial optimistic lock versionNo value -->
    <value>1</value>
    </attribute>
    </row>
    <row> ... </row>
    </table>
```

When you use a DMX file to add a rule set, it does not provide the same level of validation for the rule set as when you use the CER API. For example, the CER API can reference the `creole.upload.rulesets` target. The rule set XML is parsed to enforce that the rule set is correctly structured and that it conforms to the schema.

After you add the rule set, the CER semantics are not yet validated. For example, references to other classes or attributes and calls to static Java methods are not validated. As a result, invalid rule set XML can be stored on the database. If a rule set is written to the database, use the `creole.validate.rulesets` target to force validation of the new rule set.

The CER API stores a historical version, that is, a rule set snapshot, of each rule set whenever a new version is added. The DMX mechanism bypasses this snapshot. The first time the CER API is used to update a rule set that was added a DMX file, the CER API automatically creates a snapshot for it. The CER API is used to do full validation against the old and the new version of the rule set.

If the original version of the rule set was invalid, it results in later validation failures rather than validation failures when the error was introduced. To avoid validation failures at a later stage, ensure that the CER API is used successfully at least one time on each rule set. That is, use the `creole.upload.rulesets` target before you make any other changes to the rule set.

For more information about the steps to extract CREOLERuleSet data from your application and, if required, accompanying AppResource data, see Extracting Rule Sets from the Database for Testing.

The CREOLEMigrationControl database table

CREOLEMigrationControl is a single-row control table that is used to prevent concurrent publication of CER rule sets.

The data on this table is internal to CER and must not be read or written to by any other component.

The single row on this table is populated by a DMX file that is included with the application. Customers must not alter this DMX file or create any other DMX files that target the CREOLEMigrationControl table.

Other CER infrastructure database tables

The following database tables are also included with the CER infrastructure and must not be read or written to other than through the CER public API.

- CREOLEAttributeAvailability
- CREOLEAttributeInheritance
- CREOLERuleAttribute
- CREOLERuleAttributeValue
- CREOLERuleClass
- CREOLERuleClassInheritance
- CREOLERuleObject
- CREOLERuleSetDependency
- CREOLERuleSetEditAction
- CREOLERuleSetSnapshot
- CREOLEValueOverflow

Dependency Manager compliancy

The Dependency Manager is internal to Cúram and cannot be accessed from custom code. No customization is supported.

All Dependency Manager artifacts are contained in the `curam.dependency` code package and its sub-packages. Some artifacts in this code package are contributed by CER and others by the core application. You must not place any of your own classes or interfaces in the `curam.dependency` code package or any of its sub-packages.

The following database tables are owned by the Dependency Manager:

- Dependency
- PrecedentChangeSet
- PrecedentChangeItem
- PrecedentChangeSetBatchCtrl

Do not customize these database table. In addition, the data on these database tables must not be read or written other than by the Dependency Manager itself.

Initial population of these database tables by using DMX files is not supported, with the exception of DMX files included with the application. Also, it is not supported to customize or omit the population of these database tables by using the DMX files included with the application.

Notices

This information was developed for products and services offered in the United States.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing Legal and Intellectual Property Law IBM Japan Ltd. 19-21, Nihonbashi-Hakozakicho, Chuo-ku Tokyo 103-8510, Japan

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some jurisdictions do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you provide in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Director of Licensing IBM Corporation North Castle Drive, MD-NC119 Armonk, NY 10504-1785 US

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

The performance data and client examples cited are presented for illustrative purposes only. Actual performance results may vary depending on specific configurations and operating conditions.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBMproducts. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to actual people or business enterprises is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

Privacy Policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

Depending upon the configurations deployed, this Software Offering may use session cookies or other similar technologies that collect each user's name, user name, password, and/or other personally identifiable information for purposes of session management, authentication, enhanced user usability, single sign-on configuration and/or other usage tracking and/or functional purposes. These cookies or other similar technologies cannot be disabled.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details the section entitled "Cookies, Web Beacons and Other Technologies" and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at " Copyright and trademark information " at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

The registered trademark Linux is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other names may be trademarks of their respective owners. Other company, product, and service names may be trademarks or service marks of others.

IBM®

Part Number:

(1P) P/N: