

1 Integrate

Built-in Function Programmer Guide

Product version: v 1.4

Document version: v 1.1.3

Document date: 08/02/2017



Copyright 2017 1Spatial Group Limited.

All rights reserved. No part of this document or any information appertaining to its content may be used, stored, reproduced or transmitted in any form or by any means, including photocopying, recording, taping, information storage systems, without the prior permission of 1Spatial Group Limited.

1Spatial

Tennyson House

Cambridge Business Park

Cambridge

CB4 0WZ

United Kingdom

Phone: +44 (0)1223 420414

Fax: +44 (0)1223 420044

Web: www.1spatial.com

Every effort has been made to ensure that the information contained in this document is accurate at the time of printing. However, the software described in this document is subject to continuous development and improvement.

1Spatial Group Limited reserves the right to change the specification of the software. 1Spatial Group Limited accepts no liability for any loss or damage arising from use of any information contained in this document.



Contents

1 Overview	4
Prerequisites	4
2 Creating Functions	5
Class Interface	5
Public Methods	6
Casting Input Parameters	8
Returning Values from a Function	9
Destroying Geometries and Descriptor Objects	9
Creating an Extension jar	10
Testing a New Function	11
3 Sample Code	12

1 Overview

The Rule extensibility API allows you to create custom functions ("built-in functions") in Java for use in the 1Integrate rule builder.

This guide explains the 1Integrate Rule Extensibility API using sample code supplied with 1Integrate, and provides details on how to write and implement your own code.

The sample source code is located in the Documentation directory of your installation package and in "Sample Code" on page 12.

Prerequisites

This guide assumes you have the following skill set and software tools:

- ▶ A good understanding of how to use 1Integrate
- ▶ A good knowledge of Java for creating custom built-in functions
- ▶ A Java editor

2 Creating Functions

Each new built-in function is held in a self-contained Java class. This class defines the functionality, the interface, and the help tooltip, as seen in the rule builder.

Once created, built-in functions are accessed in the 1Integrate Rule Author interface.

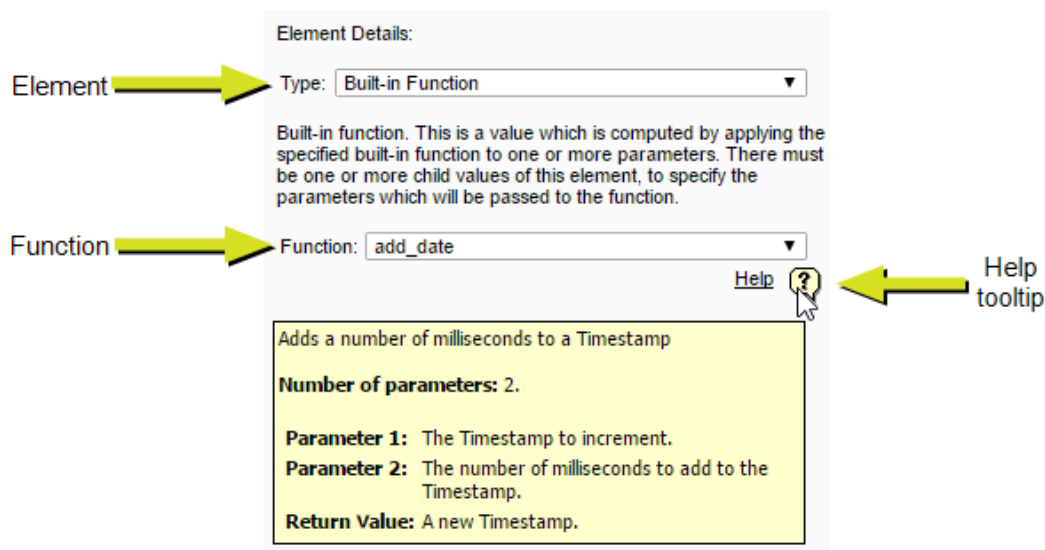


Figure 2-1: A built-in function within the 1Integrate Rule Author

Class Interface

Each function is implemented in its own class. To ensure the functionality is called correctly and the tooltip is displayed, the class defining the new function must be set up as follows:

- ▶ **Declaration** - `public class [class_name] implements BuiltinFn` declares the class as a function.

As a special case, if you want to pass 3D geometries to this function, then also implement from the `Builtin3D` class. The name of the class is not displayed in the 1Integrate interface; the results of the `getName` function is displayed.

- ▶ **Classes to import** - To ensure your new functions can access the data in the 1Integrate cache, you need to import a set of classes held in the gothic library:

- ▶ ▶ gothic.main.GothicException
- ▶ ▶ com.onespatial.rule.interfaces.BuiltInFn



Note: You can import other classes to implement functions, such as geometry manipulation. For these classes, refer to the Java API documentation `*-javadoc.jar`, within the Documentation directory of your installation package.

These classes are available in the `gothic-java-[version].jar` and `rulelibapi-[version].jar` files located in the Documentation directory of your installation package.



Note: You must either define a no argument constructor or provide no constructors and allow the default to be used.

Public Methods

The following public methods must be set up for each function:

Public Method	Description
<code>public String getName()</code>	Returns a name in the Built-in Function list in 1Integrate. The name does not need to be the same as the <code>class_name</code> .
<code>public int getMinNumArgs()</code>	Returns the minimum number of parameters to be passed to the built-in function. This can be zero or more, and represents the number of non-optional parameters.
<code>public int getMaxNumArgs()</code>	Returns the maximum number of parameters to be passed to the built-in function. This can be any number equal to or greater than the minimum number of parameters and represents the number of non-optional parameters plus the maximum number of non-optional parameters. For unlimited parameters, return <code>Integer.MAX_VALUE</code> . For a fixed number of parameters, then this method should return the same number as <code>getMinNumArgs</code> .


Public Method	Description
<pre>public Object evaluate(Object [] args) throws GothicException</pre>	<p>Contains the functionality for a built-in function.</p> <p>The return value is an Object. The arguments are an array of Objects that must be cast to the required class within this function.</p>

The following functions will be used to populate the tooltips within the Rule and Action builder user interface:



Note: When creating stings, avoid HTML reserved characters such as `<`; `>`; `&` unless they are used as valid HTML.

Function	Description
<pre>public String getVersion()</pre>	<p>Returns your version number for the built-in function as a string, for example “1.0”.</p>
<pre>public String getGeneralDescription ()</pre>	<p>Returns a general description of the function as a string.</p> <p>The return value will be embedded within HTML, so HTML formatting can be used if required.</p>
<pre>public String getArgumentDescription (int arg)</pre>	<p>Returns the description of the argument for the specific number (starting from 0).</p> <p>This will be called for each argument from 0 up to, but not including, <code>getMaxNumArgs()</code>.</p> <p>The return value will be embedded within HTML, so HTML formatting can be used if required.</p> <p>If there is an unlimited number of arguments (i.e. <code>Java.lang.Integer.MAX_VALUE</code>) then only the description for the first optional argument is displayed.</p>
<pre>public String getReturnDescription()</pre>	<p>Returns a description of the function’s return value.</p> <p>The return value will be embedded within HTML, so HTML formatting can be used if required.</p>

Function	Description
<pre>public String getGroup ()</pre>	<p>Returns the name of the group of functions within which to list this built-in in the interface.</p> <div style="border: 1px solid gray; padding: 5px; margin: 10px 0;">  Note: This does not need to be an existing group, a new group will be created if it does not already exist. </div> <p>If this is unimplemented or returns null, then the functions will be listed in a default group.</p> <p>Default groups are:</p> <ul style="list-style-type: none"> ▶ Geometric ▶ Identity ▶ Mathematical ▶ Bit Manipulation ▶ String ▶ Timestamp ▶ Topological ▶ Collection ▶ Shifting

Casting Input Parameters

The input parameters to a function are passed in to the `evaluate()` method as an array of `Java.lang.Objects`.

You should cast each array value into the correct Java class. The values in the array are in the same order as the parameters passed to the function. The mapping from 1Integrate data types to Java objects is as follows:

1Integrate data type	Java object
Boolean	<code>java.lang.Boolean</code>
Date/Timestamp	<code>gothic.descriptor.Timestamp</code>
Geometry 2D	<code>gothic.descriptor.Geometry</code>

1Integrate data type	Java object
Geometry 3D	<code>gothic.descriptor.HeightedGeometry</code>
Integer	<code>java.lang.Integer</code>
Integer64	<code>java.lang.Long</code>
Object	<code>gothic.main.GothicObject</code>
Real	<code>java.lang.Double</code>
String	<code>java.lang.String</code>

Returning Values from a Function

Typically you will return values of types listed in "Casting Input Parameters" on the previous page, which then get assigned or reported within a rule or action.

Destroying Geometries and Descriptor Objects

All objects inheriting from `gothic.descriptor.Descriptor` that are created inside the `evaluate ()` must be destroyed to prevent memory leaks and keep memory usage low during processing.

The most commonly used objects of this class are `gothic.descriptor.Geometry`.



Note: All input parameters will be destroyed by 1Integrate after the method has returned. If any `gothic.descriptor` input parameters will be modified and returned from the method, ensure that you return a copy of the object (using the `copy ()` method) before returning the value. For geometries, do not make any modifications to the input geometry before copying it. Otherwise, you will modify the original geometry passed in to the function and the Rule or Action may produce unexpected results.

To destroy the objects, call the `destroy ()` method on each object before the method returns. To ensure that this happens in all cases, put the destroy call within a `finally ()` block.

For example:

```
Geometry inputGeom = (Geometry)args[0];
Geometry bufferedGeometry = null;
try
{
```

```

        bufferedGeometry = inputGeom.bufferCreate(10.0,
10.0);
        return bufferedGeometry.getData().areaArea
    }
finally
{
    if (bufferedGeometry != null)
        bufferedGeometry.destroy();
}

```

Creating an Extension jar

Custom built-ins must be compiled and packaged into a **.jar** file.

When compiling the java files, you must ensure the `gothic-java.jar` file is on the classpath.

In order for 1Integrate to find the new built-ins, a java `ServiceLoader` configuration file must also be present within the **.jar** file. This should be a single file called `META-`

`INF/services/com.onespatial.rule.interfaces.BuiltinFn.`

The file should contain fully qualified names of any built-in function implementation classes that you have created, with one per line.



Note: The **.jar** file can be created using any standard Java development environment, such as Eclipse.

WebLogic deployment:

For WebLogic, the location of the **.jar** file must be specified using the 1SMS Installation Wizard.

When installing 1Integrate, two parameters are requested for both the 1Integrate Interface and 1Integrate Session Queue.

Tick the **Include Custom Extensions** parameter and then use the **Selected Custom Extensions** parameter to browse to your **.jar** file.



Note: If you need to replace this **.jar** file for any reason, you will need to uninstall 1Integrate and then re-install it using the 1SMS Installation Wizard.

Parameter	Description
Custom Extensions	
Include Custom Extensions	Tick this box to include custom extensions.

Parameter	Description
Selected Custom Extensions	Browse for custom extensions to be included.

Wildfly deployment:

For Wildfly, after starting the interface and session queue for the first time, copy the **.jar** file into `standalone/deployments/ms-integrate-interface-[version].ear/lib` and also into `standalone-sessionqueue/deployments/ms-integrate-sessionqueue-[version].ear/lib` then restart both applications.

Testing a New Function



Note: The entire application server must be restarted before testing the new functionality.

Test a newly created function:

1. Create a new rule with something that requires a value, such as a condition comparison.
2. Within the Element Details tab, select a **Type** of Built-in Function, then use the **Function** drop-down list to select the new function.
3. Using the **Help** tooltip, check that the number of minimum and maximum parameters, the version number, and other descriptions are correct.
4. Check you can add parameters up to, but not over, your expected maximum.
5. Create a new session and apply the rule.
6. If behaviour is not as expected, then correct any errors, rebuild the **.jar** file and re-deploy it to the application server (see "Creating an Extension jar" on the previous page).

3 Sample Code

```
package sample.radiusstudio.builtin;
import gothic.descriptor.Descriptor;
import gothic.descriptor.Geometry;
import gothic.descriptor.HeightedGeometry;
import gothic.main.GothicException;
import gothic.support.rv.GeometrySaGetDataRV;

import com.onespacial.rule.interfaces.BuiltinFn;

/**
 * Built-in function to return the 'roundness' of a polygon, calculated as
 * (area * 4 * pi / (perimeter squared)) of a geometry.
 */
public class Roundness implements BuiltinFn
{
    public String getName()
    {
        return "get_roundness";
    }

    public String getVersion()
    {
        return "1.1";
    }

    public int getMinNumArgs()
    {
        return 1;
    }

    public int getMaxNumArgs()
    {
        return 1;
    }

    public String getGeneralDescription()
    {
        return "Calculate the 'roundness' of a polygon, calculated as
        (area * 4 * pi / (perimeter squared)) of a geometry. Perfect circles
        return 1, more complex and spidery shapes return lower values.";
    }
}
```

```

public String getArgumentDescription(int arg)
{
    if (arg == 0)
    {
        return "A simple or multi polygon geometry. If empty or
non-polygon geometries are passed in then 0 is returned.
If non-geometry types are passed in then an exception is
raised.";
    }
    else
    {
        return null;
    }
}

public String getReturnDescription()
{
    return "A real value between 0 and 1. 1 means a perfect circle,
0 means a fractal of infinite complexity.";
}

public String getGroup()
{
    return "Example Group";
}

public Object evaluate(Object[] args) throws GothicException
{
    Geometry geomArg = null;
    double area, perimeter;
    GeometrySaGetDataRV info;

    try
    {
        if (args[0] instanceof Geometry)
        {
            Descriptor arg0 = (Descriptor) args[0];
            geomArgs = (Geometry) arg0.copyDescriptor();
        }
        else if (args[0] instanceof HeightedGeometry)
        {
            HeightedGeometry arg0 = (HeightedGeometry) args[0];
            geomArgs = arg0.get2DGeometry();
        }
        else
        {

```

```

        throw new GothicException("This function must be
        passed a geometry");
    }
    if (geomArg.getType() != Geometry.SIMP_AREA &&
    geomArg.getType() != Geometry.COMP_AREA)
    {
        return new Integer(0);
    }
    // empty geometries are ignored
    if(geomArg.testClear())
    {
        return new Integer(0);
    }
    info = geomArg.saGetData();
    area = info.totalArea;
    perimeter = info.perimeter;

    // return (area * 4 * pi) / perimeter squared
    return new Double(area * 4 * java.lang.Math.PI) /
    (perimeter * perimeter);
}
finally
{
    if (geomArg != null)
    {
        geomArg.destroy();
    }
}
}
}

```