# Adding User Defined Native Functions
## Getting Started Guide

Version 3.4

# Adding User Defined Native Functions

At times you might have to add new formula function for validating/processing the data.  Designer supports defining new formula functions using the Function Definition feature. You can choose to implement the operation of the function using either the formula language or the platform specific code. Once the functions are defined in a Cartridge, they are available globally in the cartridge and any cartridge that references it. This means that the function can be used in formula code from anywhere in the cartridge.

The following steps are required in creating native function that implements the operation of the function using platform specific code.

1. Create a function definition with a reference name to the native class that implements the function operation.

2. Write the native class. In case of Java platform, you need to write a Java class that implements the IInvokable interface.

   **Formatted:** Bullets and Numbering

3. Add the Java source file to the cartridge using the 'External Sources' tab of the 'Java Code Generation Settings' dialog.

4. Bind the reference name of the native class to the concrete Java class name including the package prefix.

## Creating a Native Function Defintion

Follow the steps given below to create a function defintion that uses a native class for implementing the function operation.

1. Right click the cartridge node, select 'Add Item-> New Function' as shown below.
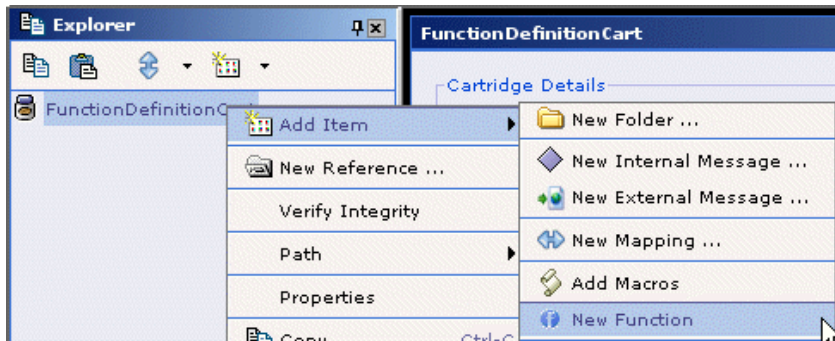
A new function definition will be added.
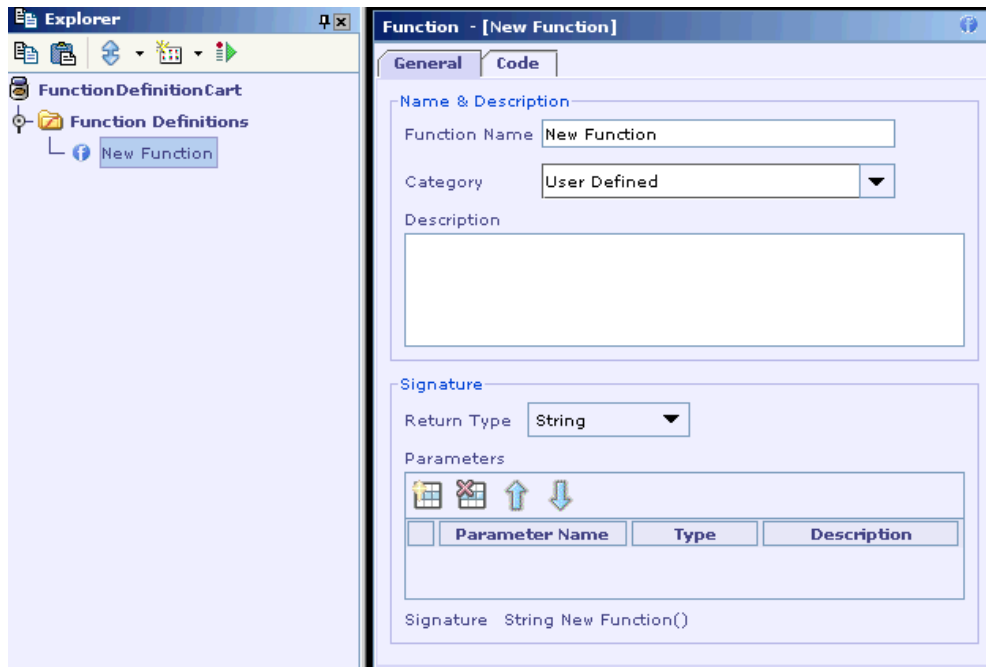


2.  Specify the name of the function in the 'Function Name' text field.

    The function should be accessed using this name. This is a mandatory property.
    The name should conform to identifier rules.

3.  Specify the category of the function in the 'Category' combo box.

    The 'User Defined' category is selected by default. You can select any other
    category from the drop down list. In the 'Edit Formula' dialog, the function will be
    listed under the specified category.

4.  Specify the function description in the 'Description' text area.

    This is not mandatory. When a function is selected in the functions list box the
    description specified here will be displayed.

5.  Specify the input parameters of the function using the 'Parameters' table.

    A function can have multiple parameters or no parameter. Parameters are not
    mandatory for a function.

    Properties of a parameter are,

**Name:** Each parameter name should be unique.

**Description:** Description of the parameter. This is not mandatory.

**Type:** The parameter type. All the supported Designer types can be specified as parameters. Along with those types, the following four new types can be specified: Any, Token, Section and Message.

6. Select the return type of the function in the 'Return Type' list box.

   The type can be any of the supported designer types. The following five new types are also supported as return types: Any, Token, Section, Message and Void.

7. In the 'Code' tab, select the 'Native Function' check box and specify a refernce name in the 'Native Class Ref' text field.
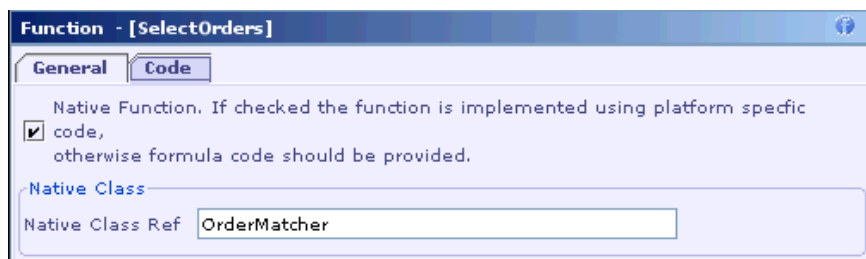


**See Also:**

[Adding User Defined Native Functions](#)

# Writing the IInvokable Implementation Class

In case of Java platform, the native class should implement the `com.tplus.transform.runtime.handler.IInvokable` interface. The API for the interface is given below for your convenience.

### Interface IInvokable

Interface for external (user defined) classes to perform a custom operation. This is used by

   Invoke External activity in a message flow
   User defined functions with native implementation

The interface defines a generic run method, which takes an Object array as a parameter. The elements of the array are the arguments to the operation. The implementing class should execute the operation and return a value. Since the run method's signature is generic, it is very important to ensure that the same set of

arguments with same types are used at the call point (Invoke External activity or Function definition).

The implementing class,

    should be public

    should have public default constructor

    should be made available to the generated JAR at runtime (by specifying in the manifest classpath, or by including the source while building)

    should be stateless. The class should not maintain call specific data.

As many instances of the implementing class will be created as required. The class should not expect all calls to be made to one particular instance.

**Note:**

For an easy way of creating an IInvokable implementation class please refer the section 'New File from Template' in **Designer Guide** documentation.

## Method Detail

```
public java.lang.Object run(java.lang.Object[] args,
                           TransformContext cxt)
                    throws TransformException
```

Executes the operation and returns the value. Primitive values are boxed into the corrsponding object wrappers.

**Parameters:**

`args` - arguments to the operation
`cxt` - the transformation context. In case of function definition this parameter should not be used.

**Returns:**

The return value of the operation.

**Throws:**

TransformException

The OrderMatcher class given below is an example of the IInvokable class.

```java
package com.volante;

import com.tplus.transform.runtime.handler.*;
import com.tplus.transform.runtime.*;

public class OrderMatcher implements IInvokable{
    public Object run(Object[] args, TransformContext cxt)
                                            throws TransformException {
        DataObjectSection orders = (DataObjectSection)args[0];
        String symbol = (String)args[1];
        DataObjectSectionImpl matchingOrders =
                        new DataObjectSectionImpl2(null, "newsection");
        for (int i = 0; i < orders.getElementCount(); ++i) {
            DataObject order = orders.getElement(i);
            if (order.getField("Symbol").equals(symbol)) {
                matchingOrders.addElement(order);
            }
        }
        return matchingOrders;
    }
}
```

It implements the IInvokable interface by implementing the run() method with the following signature and this method is invoked whenever the function definition needs to be executed.

```java
public Object run(Object[] args , TransformContext cxt)
                                    throws TransformException;
```

Here, the parameter 'args' represents the input arguments passed during invocation of this method.

It uses a 'for' loop to iterate though the elements of the section passed as the first input parameter. If the 'Symbol' field of an element matches the value of the second input parameter, then it is added to the 'matchingOrders' local variable of 'DataObjectSection' type, which will be returned as the result of the function.
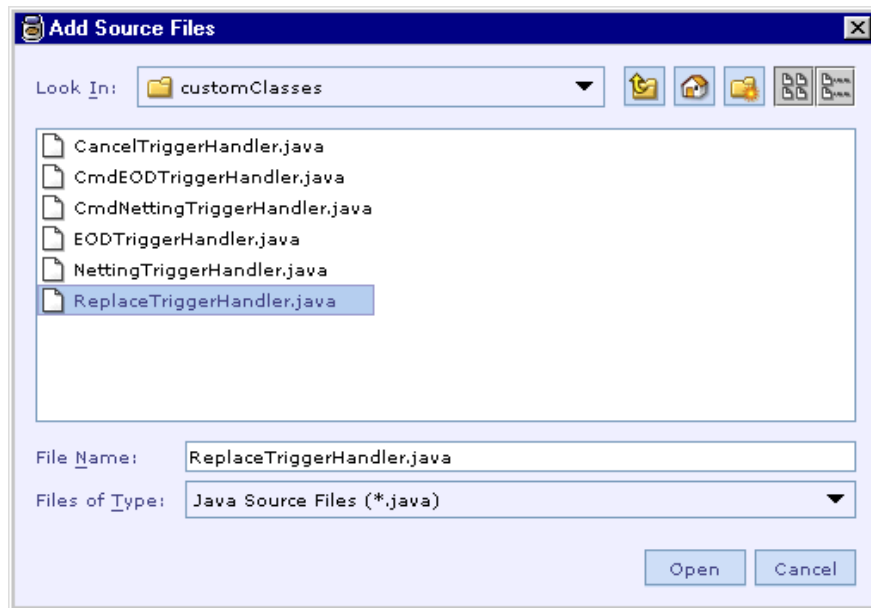
**See Also:**

Adding User Defined Native Functions

# Adding External Java Source Files

Follow the steps given below to add external Java source files to the code generation settings of a cartridge:

1. In the 'External Sources' tab of the 'Java/EJB Code Generation Settings' dialog, click on the 'Add Files'  button.

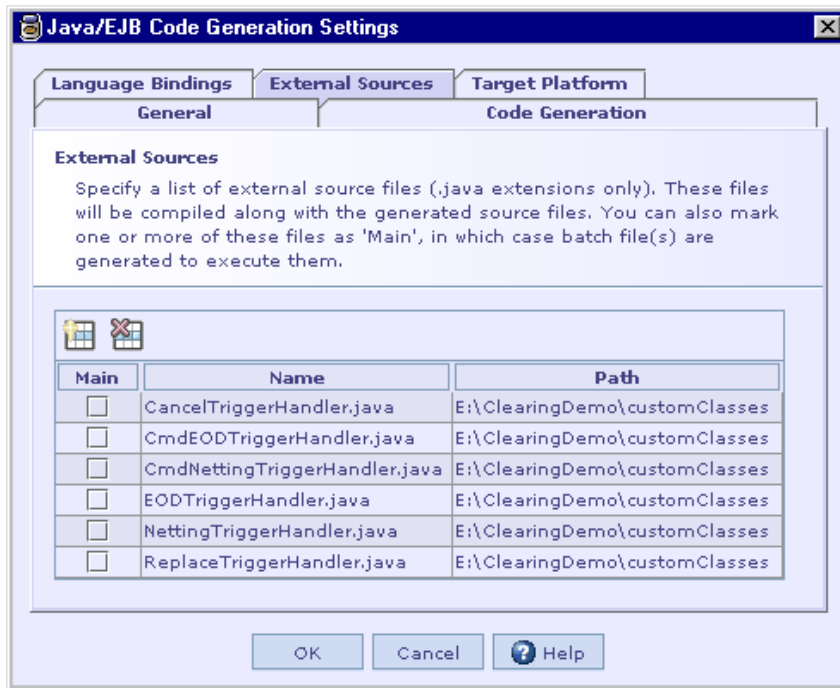2. The 'Add Source Files' dialog will be displayed.



3. Navigate to the directory that contains the required Java source files and select them.

   It is recommended that the Java source files are available under a child directory of the cartridge.

4. Click the 'Open' button.

   The files will be included to the external source files list.

5. Select the 'Main' check box for one or more source files to mark them as application class files.

   For each file marked as an application class file, a batch file that invokes the corresponding application class will be generated.

You can also add directories containing external Java source files to the code generation settings of a cartridge. In this case, all the source files in the specified directories are included in the build process.

Follow the steps given below to add directories containing Java sources files.

1. In the 'External Sources' tab of the 'Java/EJB Code Generation Settings' dialog, click on the 'Add Files' button.

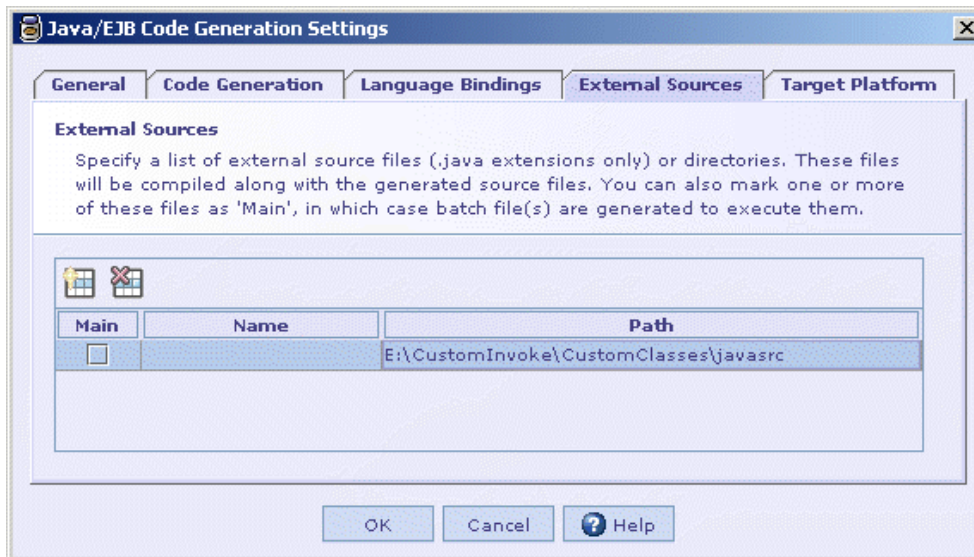2. The 'Add Source Files' dialog will be displayed.
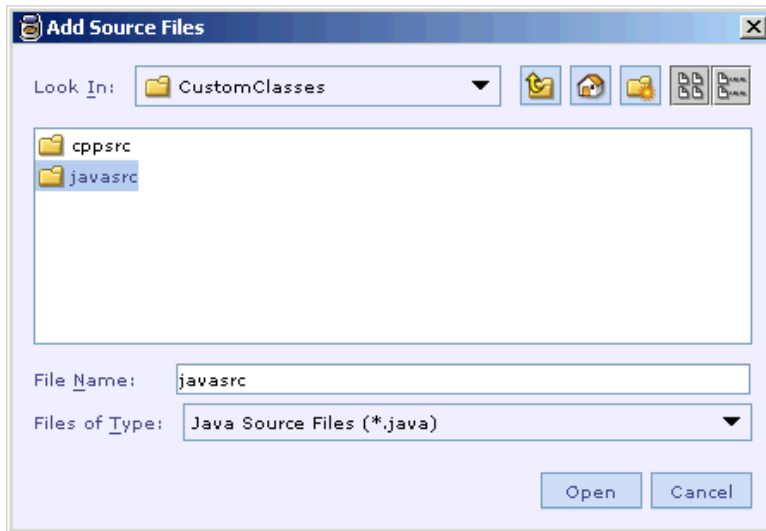
3. Navigate to the directory that contains the required Java source files and select it.

It is recommended that the Java source files are available under a child directory of the cartridge.

4. Click the 'Open' button.

The directory will be included to the external source files list.

**Note:**

You can include external Java classes to the code generation settings of a cartridge in two other ways:

> You can specify the Java class files (instead of Java source files) to be included in the generated component.

> You can bundle the required Java class files into a Jar file and add it to the manifest entry of the generated component.

**See Also:**

Adding User Defined Native Functions

# Binding a Native Class Reference to a Concrete Java Class

You can proceed with code generation and deployment, once the reference name of a native class is bound to a concrete Java class (byte code file) and the Java class is added to the code generation settings of the cartridge.

Follow the steps given below to bind a native class reference to an actual Java class.

1. In Designer, select the Build > Code Generation Settings (Java/EJB) menu item.

   The Java/EJB Code Generation Settings dialog box appears.

2. Select the **Language Bindings** tab and specify the actual Java class name in the 'Java Class' column of the row corresponding to the custom class reference.
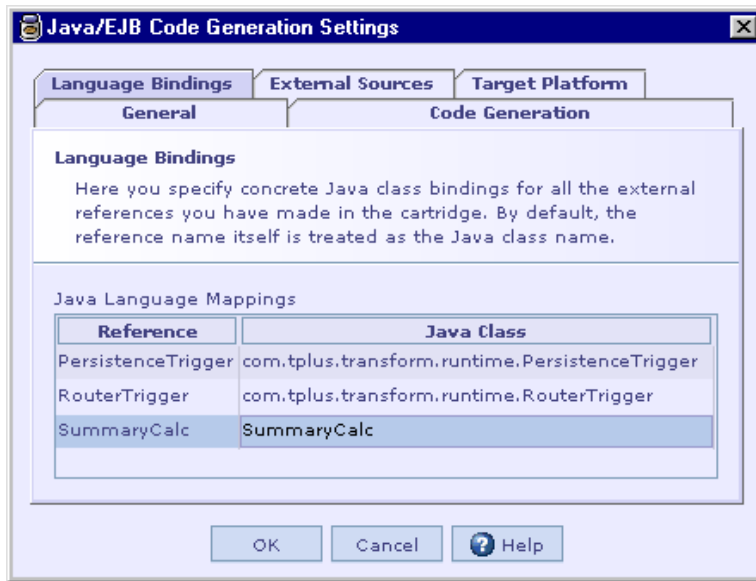
3. Click on the **OK** button to close the dialog.

Please note that you should save the cartridge to permanently save the changes.

**See Also:**

Adding User Defined Native Functions