

Defining Custom Classes in Java

Getting Started Guide

Version 10g Release 3 (10.3)

DEFINING JAVA CUSTOM CLASSES.....	3
DEFINING VALIDATION CLASSES.....	3
HOW TO PLUG IN YOUR OWN VALIDATION CLASS TO A FIELD?	4
HOW TO WRITE YOUR OWN FIELD VALIDATION CLASS?	5
DEFINING PROCESSING CLASSES.....	6
HOW TO PLUG IN YOUR OWN PRE/POST PROCESSING CLASS?	7
HOW TO WRITE YOUR OWN PROCESSING CLASS?	8
HOW TO PLUG IN YOUR OWN FIELD PROCESSING CLASS?	9
HOW TO WRITE YOUR OWN FIELD PROCESSING CLASS?	10
DEFINING MAPPING CLASSES.....	11
HOW TO PLUG IN YOUR OWN MAPPING CLASS?	12
HOW TO WRITE YOUR OWN MAPPING CLASS?	13
DEFINING IINVOKABLE CLASSES.....	19
HOW TO PLUG IN YOUR OWN IINVOKABLE CLASS TO A FUNCTION DEFINITION?	20
HOW TO WRITE YOUR OWN IINVOKABLE CLASS?	21
BINDING A CUSTOM CLASS REFERENCE TO A CONCRETE JAVA CLASS	23
ADDING EXTERNAL JAVA CLASSES.....	24
APPENDIX.....	25

Defining Java Custom Classes

A custom class adds flexibility to standard mechanism provided by Designer for defining message processing. For example, you can define validation rules, mapping rules and processing rules using the inbuilt formula language. You can also define these rules using custom classes. A custom class is more flexible compared to the formula language. For example, it can connect to a remote object to retrieve a value that can be used in validating a field. To use a custom class, you have to write a Java class that implements the corresponding interface and plug it to the item to be customized.

See Also:

[Defining Validation Classes](#)

[Defining Mapping Classes](#)

[Defining IInvokable Classes](#)

[Binding a Custom Class Reference to a Concrete Java Class](#)

[Adding External Java Classes](#)

[Appendix](#)

Defining Validation Classes

To customize validation of an internal/external message field/section, you have to [write a class](#) that implements the IFieldValidation interface and [plug it to the validation rule](#) of the corresponding field/section.

See Also:


[Defining Java Custom Classes](#)

How to plug in your own validation class to a field?

Follow the steps given below to associate a custom field validation class to the validation rule of a field.

1. [Write the field validation class](#). The field validation class must implement the `com.tplus.transform.runtime.handler.IFieldValidation` interface. For further details on the `IFieldValidation` interface, refer to the API documentation.

It is recommended that the field validation class is available under a child directory of the cartridge.

2. Compile the field validation class.
3. Open the cartridge in Designer and select the 'Validation Rules' node of the message that contains the field to be validated.
4. In the field-wise validations view, select the field to be validated and click on the 'Add New Custom Validation' button .
5. Type in the reference name of the field validation class in the 'Validation' column.
6. Add the field validation class to the code generation settings of the cartridge. See the section [Adding External Java Classes](#) for more information.
7. Bind the reference name of the field validation class to the actual Java class name. See the section [Binding a Custom Class Reference to a Concrete Java Class](#) for more information.

See Also:

[Defining Validation Classes](#)

How to write your own field validation class?

The AgeValidation class given below is an example of a field validation class.

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

public class AgeValidation implements IFieldValidation{
    public void validate(String fldName, Object value, DataObject obj)
        throws FieldValidationException {

        int age = ((Integer)value).intValue();

        try {
            Object desObj = obj.getField("designation");
            if(desObj != null){
                String designation = (String)desObj;
                if(designation.equalsIgnoreCase("Engineer"))
                    if(age<22 || age>35)
                        throw new FieldValidationException(
                            "age should be between 22 and 35 for Engineer");
            }
        }
        catch(FieldNotFoundException fnf) {
            throw new FieldValidationException("Field designation Not found");
        }
    }
}
```

The AgeValidation class implements the IFieldValidation interface shown below by defining the validate() method.

```
public interface IFieldValidation {
    public void validate(String fldName, Object value, DataObject obj)
        throws FieldValidationException;
}
```

The first argument of the validate() method is the name of the field to be validated. The second argument is the current value of the field. The third argument allows access to the other fields. Please note that none of these arguments should be modified within the validate() method.

The getField() method of the DataObject class can be used to read values of the other fields. The following statement retrieves the value of the 'designation' field.

```
Object desObj = obj.getField("designation");
```

Note:

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

See Also:

[Defining Validation Classes](#)

Defining Processing Classes

There are two types of custom processing classes:

1. custom message processing classes
2. custom field processing classes

A custom message processing class can manipulate the fields of an internal message; that is, all the fields of an internal message are accessible from a custom processing class. To customize pre/post-processing of an internal message, you have to [write a class](#) that implements the IProcessing interface and [plug it to the required internal message](#).

To customize pre/post processing of an internal message field/section, you have to [write a class](#) that implements the IFieldProcessing interface and [plug it to the required internal message field/section](#). Please refer to the API documentation for more information on the IFieldProcessing interface.

Note

Be careful when the value of a field is set in both message level processing and field level processing. As field level processing is carried out after message level processing, the value set to a field during message level processing will be replaced with the value returned by the specified field processing. This applies to both pre processing and post processing of an internal message.

See Also:

[How to plug in your own pre/post processing class?](#)

[How to write your own processing class?](#)

[How to plug in your own field processing class?](#)

[How to write your own field processing class?](#)

[Defining Java Custom Classes](#)

How to plug in your own pre/post processing class?

Follow the steps given below to associate a custom processing class for the pre/post processing of an internal message.

1. [Write the processing class](#). The processing class must implement the `com.tplus.transform.runtime.handler.IProcessing` interface given below.

```
public interface IProcessing {  
    public void process(DataObject obj) throws ValidationException;  
}
```

It is recommended that the processing class is available under a child directory of the cartridge.

2. Compile the processing class.
3. Open the cartridge in Designer and select the 'Processing Rules' node of the internal message that needs to be processed using the custom class.
4. Type in the reference name of the processing class in the 'Custom Processing' text field of the Pre/Post-Processing tab.
5. Add the processing class to the code generation settings of the cartridge. See the section [Adding External Java Classes](#) for more information.
6. Bind the reference name of the processing class to the actual Java class name. See the section [Binding a Custom Class Reference to a Concrete Java Class](#) for more information.

See Also:

[Defining Processing Classes](#)

How to write your own processing class?

The IncrementSal class given below is an example of a processing class.

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

public class IncrementSal implements IProcessing {
    public IncrementSal() {
    }
    public void process(DataObject obj) throws ValidationException {
        try {
            int age = ((Integer)obj.getField("age")).intValue();
            if (age > 35) {
                double sal = ((Double)obj.getField("salary")).doubleValue();
                obj.setField("salary", new Double(sal+1000));
            }
        }
        catch(FieldNotFoundException fnf) {
            throw new ValidationException("Field age or salary not found");
        }
        catch(FieldTypeMismatchException fme) {
            throw new ValidationException("Type mismatch in setting salary");
        }
    }
}
```

The IncrementSal class implements the IProcessing interface by defining the process() method.

Note that the class also has defined a default constructor as specified in the API documentation of the IProcessing interface.

The getField() method of the DataObject class can be used to retrieve the value of a field as shown in the following statement.

```
int age = ((Integer)obj.getField("age")).intValue();
```

The setField() method of the DataObject class can be used to set the value of a field as shown below.

```
obj.setField("salary", new Double(sal+1000));
```


Note:

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

See Also:

[Defining Processing Classes](#)

How to plug in your own field processing class?

Follow the steps given below to associate a custom field processing class for the pre/post processing of an internal message field.

1. Write the field processing class. The processing class must implement the `com.tplus.transform.runtime.handler.IFieldProcessing` interface given below.

```
public interface IFieldProcessing {  
    public Object processField(String fldName, DataObject obj)  
        throws FieldValidationException;  
}
```

It is recommended that the field processing class is available under a child directory of the cartridge.

2. Compile the field processing class.
3. Open the cartridge in Designer and select the 'Processing Rules' node of the internal message whose field needs to be processed using the custom class.
4. Type in the reference name of the field processing class in the 'Custom' column of the row corresponding to the required field.
5. Add the field processing class to the code generation settings of the cartridge. See the section [Adding External Java Classes](#) for more information.
6. Bind the reference name of the field processing class to the actual Java class name. See the section [Binding a Custom Class Reference to a Concrete Java Class](#) for more information.

See Also:

[Defining Processing Classes](#)

How to write your own field processing class?

The InsertMark class given below is an example of a field processing class.

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

public class InsertMark implements IFieldProcessing {
    public InsertMark() {
    }
    public Object processField(String fldName, DataObject obj)
        throws FieldValidationException {
        try {
            String val = (String)obj.getField(fldName);
            return "ABC " + val;
        }
        catch(FieldNotFoundException fnf) {
            throw new FieldValidationException("Field " + fldName +
                                                " not found");
        }
    }
}
```

The InsertMark class implements the IFieldProcessing interface by defining the processField() method.

Note that the InsertMark class defines a default constructor as required by the base class IFieldProcessing.

The first argument of the processField() method is the name of the field to be processed. The second argument should only be used to read the values of other fields.

The value of the current field can be modified by returning the new value from the processField() method, not by setting it through the DataObject.

Note:

For an easy way of creating a custom field processing class please refer the section 'New File from Template' in **Designer Guide** documentation.

See Also:

[Defining Processing Classes](#)

Defining Mapping Classes

A mapping class allows you to define mapping between two messages when the standard mechanism provided by Designer is not sufficient. For example, a custom mapping class can be used to calculate the value of a top-level field based on multiple occurrences of a nested field. It can also be used to generate summary information for trailer fields based on all the records of a batch.

There are four types of mappings and the custom class must implement the corresponding interface as explained below.

1. Mapping from an external message to an internal message is called input mapping. A custom class that defines this type of mapping should implement the `IInputMapping` interface.
2. Mapping from an internal message to an external message is called output mapping. A custom class that defines this type of mapping should implement the `IOutputMapping` interface.
3. Mapping from one internal message to another internal message is called internal message mapping or normalized object mapping. A custom class that defines this type of mapping should implement the `INormalizedObjectMapping` interface.
4. Mapping from one external message to another external message is called external message mapping. A custom class that defines this type of mapping should implement the `IExternalMapping` interface.

Please refer to the API documentation for more information on the mapping interfaces.

See Also:

[How to plug in your own mapping class?](#)

[How to write your own mapping class?](#)

[Defining Java Custom Classes](#)

How to plug in your own mapping class?

Follow the steps given below to associate a custom mapping class for the mapping between two messages.

1. [Write the custom mapping class.](#)

It is recommended that the custom mapping class is available under a child directory of the cartridge.

2. Compile the mapping class.
3. Open the cartridge in Designer and select the 'Mapping Rules' child node of mapping design element for which you want to specify the custom class.
4. In the Mapping Rules UI, click on the 'Custom Mapping' button.

It displays the 'Custom Mapping' dialog.

5. Type in the reference name of the custom mapping class in the 'Custom Mapping Class' text field of the 'Custom Mapping' dialog and click on the OK button to close the dialog.
6. Add the mapping class to the code generation settings of the cartridge. See the section [Adding External Java Classes](#) for more information.
7. Bind the reference name of the mapping class to the actual Java class name. See the section [Binding a Custom Class Reference to a Concrete Java Class](#) for more information.

See Also:

[Defining Mapping Classes](#)

How to write your own mapping class?

Example 1

The SummaryCalc class given below is an example of the output mapping class.

```
import com.tplus.transform.runtime.handler.*;
import com.tplus.transform.runtime.*;
import java.util.*;

public class SummaryCalc implements IOutputMapping {

    public SummaryCalc() {
    }

    public void mapHeader(NormalizedObject nf, DataObject header,
                        TransformContext cxt) throws ValidationException {
    }

    public void mapData(NormalizedObject nf, DataObject record,
                      TransformContext cxt) throws ValidationException {
        try {
            BatchContext bcxt = (BatchContext)
                cxt.getProperty(TransformContext.INPUT_BATCH_CONTEXT_PROPERTY);
            Integer recNo = (Integer)bcxt.getBatchProperty("RecordNumber");
            if(recNo == null) {
                bcxt.setBatchProperty("RecordNumber", new Integer(1));
            }
            else {
                bcxt.setBatchProperty("RecordNumber",
                                    new Integer(recNo.intValue() + 1));
            }

            DataObjectSection itemSec =
                (DataObjectSection)record.getField("Item");
            if(itemSec != null) {
                if(itemSec.getElementCount() > 0) {
                    int qtyIdx = itemSec.getElement(0).getFieldIndex("Qty");
                    int priceIdx = itemSec.getElement(0).getFieldIndex("Price");
                    int costIdx = itemSec.getElement(0).getFieldIndex("Cost");
                    double totalCost = 0.0;
                    for(int i = 0; i < itemSec.getElementCount(); i++) {
                        DataObject dataObj = itemSec.getElement(i);
                        Integer qty = (Integer)dataObj.getField(qtyIdx);
```

```

        Double price = (Double)dataObj.getField(priceIdx);
        double cost = qty.intValue() * price.doubleValue();
        dataObj.setField(costIdx, new Double(cost));
        totalCost += cost;
    }
    record.setField("TotalCost", new Double(totalCost));
}
}
}
catch(Exception e) {
    e.printStackTrace();
}
}

public void mapTrailer(DataObject record, DataObject trailer,
                      TransformContext cxt) throws ValidationException {
    try {
        BatchContext bcxt = (BatchContext)
            cxt.getProperty(TransformContext.INPUT_BATCH_CONTEXT_PROPERTY);
        Integer recNo = (Integer)bcxt.getBatchProperty("RecordNumber");
        HashSet itemIDSet = null;
        double totalCost = 0.0;
        Double totalCostObj = (Double)record.getField("TotalCost");
        if(totalCostObj != null) {
            totalCost = totalCostObj.doubleValue();
        }
        double netCost = 0.0;
        if(recNo.intValue() == 1) {
            itemIDSet = new HashSet();
        }
        else {
            itemIDSet = (HashSet)bcxt.getBatchProperty("ItemIDSet");
            Double netCostObj = (Double)bcxt.getBatchProperty("RunningNetCost");
            netCost = netCostObj.doubleValue();
        }
        DataObjectSection itemSec =
            (DataObjectSection)record.getField("Item");
        if(itemSec != null) {
            if(itemSec.getElementCount() > 0) {
                int itemIDIdx = itemSec.getElement(0).getFieldIndex("ItemID");
                for(int i = 0; i < itemSec.getElementCount(); i++) {
                    DataObject dataObj = itemSec.getElement(i);
                    if(dataObj != null &&
                       dataObj.getField(itemIDIdx) instanceof String) {
                        String itemID = (String)dataObj.getField(itemIDIdx);
                        itemIDSet.add(itemID);
                    }
                }
            }
        }
    }
}

```

```

        }
    }
}
trailer.setField("NetCost", new Double(totalCost + netCost));
trailer.setField("UniqueItemCount", new Integer(itemIDSet.size()));
bcxt.setBatchProperty("RunningNetCost",
    new Double(totalCost + netCost));
bcxt.setBatchProperty("ItemIDSet", itemIDSet);
}
catch(Exception e) {
    e.printStackTrace();
}
}
}

```

The SummaryCalc custom mapping class implements the **IOutputMapping** interface.

In the corresponding cartridge, the Data section of the output message consists of the 'Item' section and 'TotalCost' as top-level items. The 'Item' section consists of the fields 'ItemID', 'Qty', 'Price' and 'Cost'. The Trailer section of the output message consists of the 'NetCost' and 'UniqueItemCount' fields.

One of the requirements for the cartridge is to calculate the value of the 'TotalCost' field. It must be the sum of all the occurrences of the 'Cost' field, which is a nested field of the 'Item' section. This cannot be done using one-to-one mapping and formula mapping supported by Designer without using the SecSum() formula function. But it can be done by the custom mapping class.

The IoutputMapping interface is shown below for your convenience.

```

public interface IOutputMapping {
    public void mapHeader(NormalizedObject nf, DataObject header,
        TransformContext cxt) throws ValidationException;
    public void mapData(NormalizedObject nf, DataObject record,
        TransformContext cxt) throws ValidationException;
    public void mapTrailer(DataObject record, DataObject trailer,
        TransformContext cxt) throws ValidationException;
}

```

The mapHeader() method is invoked only once when the first record of the batch arrives. The other two methods are invoked for each record of a batch.

The following statement finds the index of the 'Qty' field within the 'Item' section. The index is found here because, index based accessing is more efficient than name

based access. The first element (element with index 0) of the 'Item' section is used to find the indices. Remember that a section is an array of elements and each element contains the fields defined for that section. In the case of the 'Item' section, each element of the 'Item' section contains the 'Qty', 'Price' and 'Cost' fields.

```
int qtyIdx = itemSec.getElement(0).getFieldIndex("Qty");
```

A 'for' loop is used to iterate through the elements of the Item section to calculate the value of the 'TotalCost'.

The following statement retrieves an element of the 'Item' section.

```
DataObject dataObj = itemSec.getElement(i);
```

Remember that an element of a section is of DataObject type and it contains a set of fields. A field of DataObject can be accessed by the setField() and getField() methods. The following statement retrieves the value of the 'Qty' field. Since 'Qty' is a simple field, the value is returned as the wrapper class object corresponding to the data type of that field. If one of the fields of a DataObject is a section, it is returned as of DataObjectSection type.

```
Integer qty=(Integer)dataObj.getField(qtyIdx);
```

The following statement sets the value of the 'Cost' field.

```
dataObj.setField(costIdx, new Double(cost));
```

The following statement accumulates total cost for each element of the 'Item' section.

```
totalCost+=cost;
```

The following statement sets the accumulated total cost to the 'TotalCost' field.

```
record.setField("TotalCost", new Double(totalCost));
```

This example also calculates the number of the unique items encountered in the whole batch of records. The number of unique items encountered in a record can be easily calculated by accessing the 'Item' section. But calculating the number of unique items encountered in the whole batch of records is bit tricky. The key is how to define a storage that can be accessed whenever a new record of a particular batch arrives. The common storage can be defined using the BatchContext object.

The following statement gets the reference of the BatchContext object from the TransformContext object. Please note that the TransformContext object is an input parameter of all the methods specified by the IOutputMapping interface.


```
BatchContext bcxt =
    BatchContext)cxt.getProperty(TransformContext.INPUT_BATCH_CONTEXT_PROPERTY);
```

The following 'if' statement creates a Set object **itemIDSet** that stores the unique ItemID values, when the first record of the batch arrives.

```
if (recNo.intValue() == 1) {
    itemIDSet= new HashSet();
}
```

It will be stored into the BatchContext object at the end of record processing as shown below.

```
bcxt.setBatchProperty("ItemIDSet", itemIDSet);
```

The following statement retrieves **itemIDSet** from the BatchContext object for the second and subsequent records.

```
itemIDSet = (HashSet)bcxt.getBatchProperty("ItemIDSet");
```

The following statement retrieves the 'Item' section of a record.

```
DataObjectSection itemSec=(DataObjectSection)record.getField("Item");
```

The following 'for' loop adds the value of the 'ItemID' field retrieved from each element of the 'Item' section to **itemIDSet**, which maintains only the unique values.

```
for (int i=0; i<itemSec.getElementCount(); i++) {
    DataObject dataObj = itemSec.getElement(i);
    if (dataObj != null &&
        dataObj.getField(itemIDIdx) instanceof String) {
        String itemID=(String)dataObj.getField(itemIDIdx);
        itemIDSet.add(itemID);
    }
}
```

The following statement sets the 'UniqueItemCount' field of the Trailer section, the number of unique items in the whole batch of records.

```
trailer.setField("UniqueItemCount", new Integer(itemIDSet.size()));
```

Example 2

The CustomExternalMapping class given below is an example of the external mapping class.

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

public class CustomExternalMapping implements IExternalMapping {

    public CustomExternalMapping() {
    }

    public void map(ExternalObject input, ExternalObject output,
        TransformContext cxt) throws ValidationException {
        try {
            DataObject inputDataObj = input.getData();
            DataObject outputDataObj = output.getData();
            outputDataObj.setField("InvoiceDate",
                inputDataObj.getField("InvoiceDate"));
            outputDataObj.setField("ClientID",
                inputDataObj.getField("ClientID"));
            DataObjectSection inItemSec = inputDataObj.getSection("Item");
            DataObjectSection outItemSec = outputDataObj.getSection("Item");
            double totalCost = 0.0;
            if(inItemSec.getElementCount() > 0) {
                for(int i = 0; i < inItemSec.getElementCount(); i++) {
                    DataObject inItemElm = inItemSec.getElement(i);
                    String itemID = (String)inItemElm.getField("ItemID");
                    Integer qty = (Integer)inItemElm.getField("Qty");
                    Double price = (Double)inItemElm.getField("Price");
                    double cost = qty.intValue() * price.doubleValue();
                    totalCost += cost;

                    DataObject outItemElm = outItemSec.createElement();
                    outItemElm.setField("ItemID", itemID);
                    outItemElm.setField("Qty", qty);
                    outItemElm.setField("Price", price);
                    outItemElm.setField("Cost", new Double(cost));
                    outItemSec.addElement(outItemElm);
                }
            }
            outputDataObj.setField("TotalCost", new Double(totalCost));
        }
        catch(Exception e) {
```

```

        throw new ValidationException("Unexpected error while mapping.");
    }
}

```

The CustomExternalMapping class implements the IExternalMapping interface by implementing the map () method with the following signature and this method is invoked whenever the associated mapping needs to be carried out.

```

public void map(ExternalObject input, ExternalObject output,
               TransformContext cxt) throws ValidationException;

```

Here, the first parameter 'input' represents the source external message, the second parameter 'output' represents the target external message and the last parameter 'cxt' represents the context in which the current transformation occurs.

The following statement retrieves the Data section of the source external message.

```
DataObject inputDataObj = input.getData();
```

The following statement retrieves the Data section of the target external message.

```
DataObject outputDataObj = output.getData();
```

The following statement retrieves the value of the source field named 'InvoiceDate' and sets it to a similarly named field in the target external message.

```
outputDataObj.setField("InvoiceDate", inputDataObj.getField("InvoiceDate"));
```

Note:

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

See Also:

[Defining Mapping Classes](#)

Defining I Invokable Classes

You can execute platform specific custom code from a function definition. For this you have to [write a class](#) that implements the IInvokable interface and [plug it to the function definition](#). An invokable external class thus adds flexibility to the function definition.

See Also:

[Defining Java Custom Classes](#)

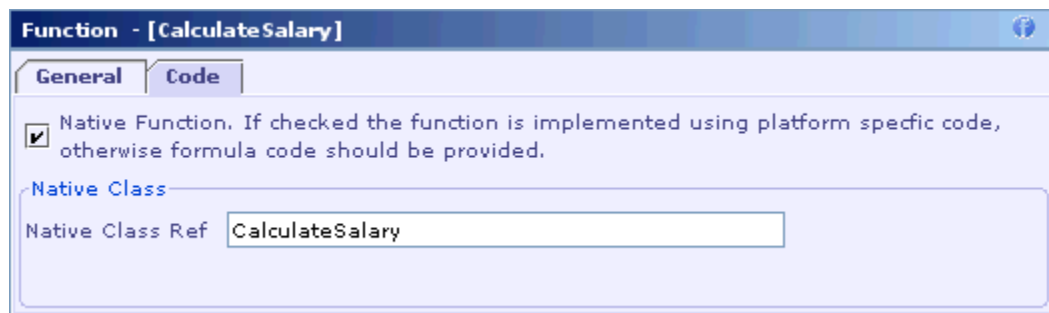
How to plug in your own IInvokable class to a function definition?

Follow the steps given below to associate an invokable external class to a function definition.

1. [Write the IInvokable class.](#)

It is recommended that the custom class is available under a child directory of the cartridge.

2. Compile the invokable external class.
3. Open the cartridge in Designer and select required the function definition node.
4. In the Function Definition UI, select the Code tab.



5. Select the 'Native Function' check box and specify a reference name in the 'Native Class Ref' text field.
6. Add the invokable external class to the code generation settings of the cartridge. See the section [Adding External Java Classes](#) for more information.
7. Bind the reference name of the invokable external class to the actual Java class name. See the section [Binding a Custom Class Reference to a Concrete Java Class](#) for more information.

See Also:

[Defining IInvokable Classes](#)

How to write your own IInvokable class?

The API for the `com.tplus.transform.runtime.handler.IInvokable` interface is given below for your convenience.

Interface IInvokable

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

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.

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 corresponding 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 AddPrefix class given below is an example of the IInvokable class.

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

public class AddPrefix implements IInvokable{
    public AddPrefix() {
    }
    public Object run(Object[] args, TransformContext cxt)
                                throws TransformException {
        if (args.length > 0) {
            return "Out:" + args[0];
        }
        return "";
    }
}
```

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

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

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

The following 'if' statement first checks for the number of input arguments passed and if the number of arguments is greater than zero, it returns the value obtained by concatenating the string "Out:" with the input argument at index 0.

```
if (args.length > 0) {
    return "Out:" + args[0];
}
```

Note:

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

See Also:

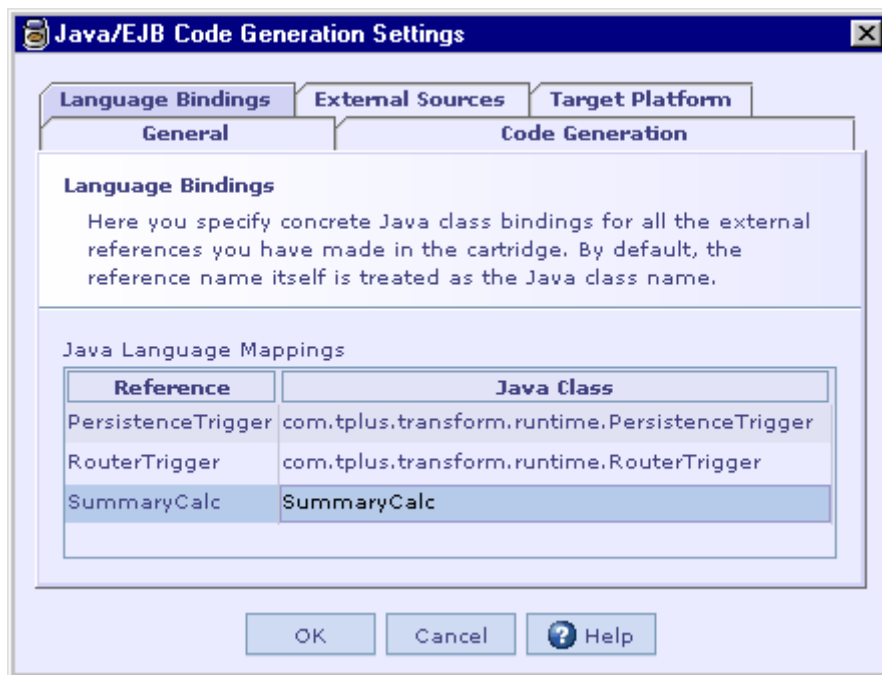
[Defining IInvokable Classes](#)

Binding a Custom Class Reference to a Concrete Java Class

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

Follow the steps given below to bind a custom 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:

[Defining Java Custom Classes](#)

Adding External Java Classes

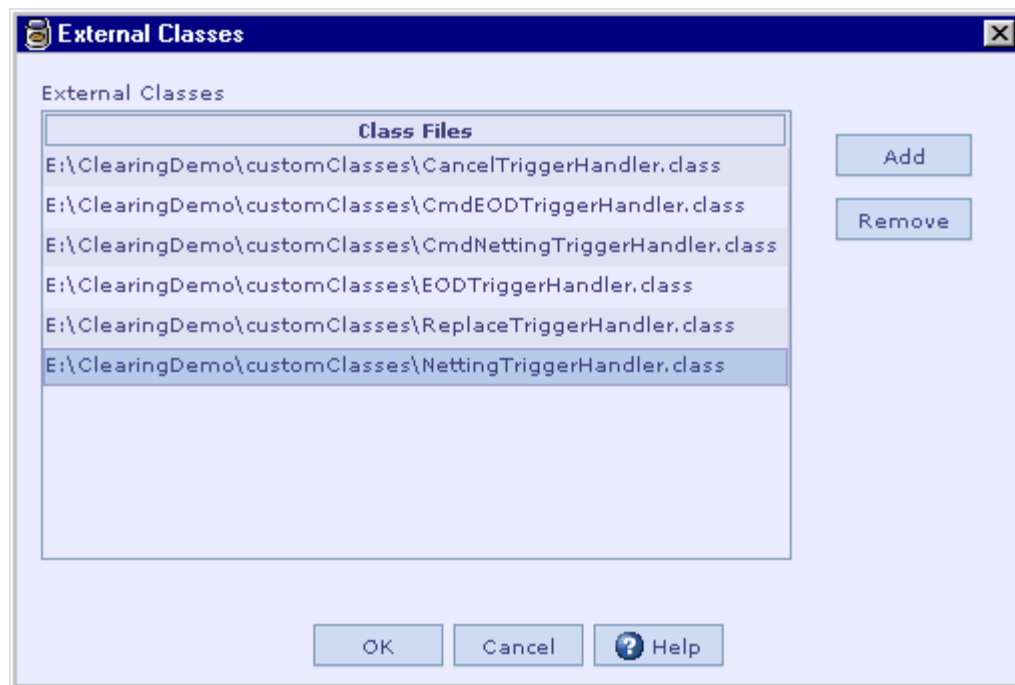
You can specify external Java class files (not Java source files) to be bundled with the component generated for a design element by following the steps given below.

Steps to add external Java class files to the code generation settings of a cartridge:

1. Compile the Java source files into Java class files.

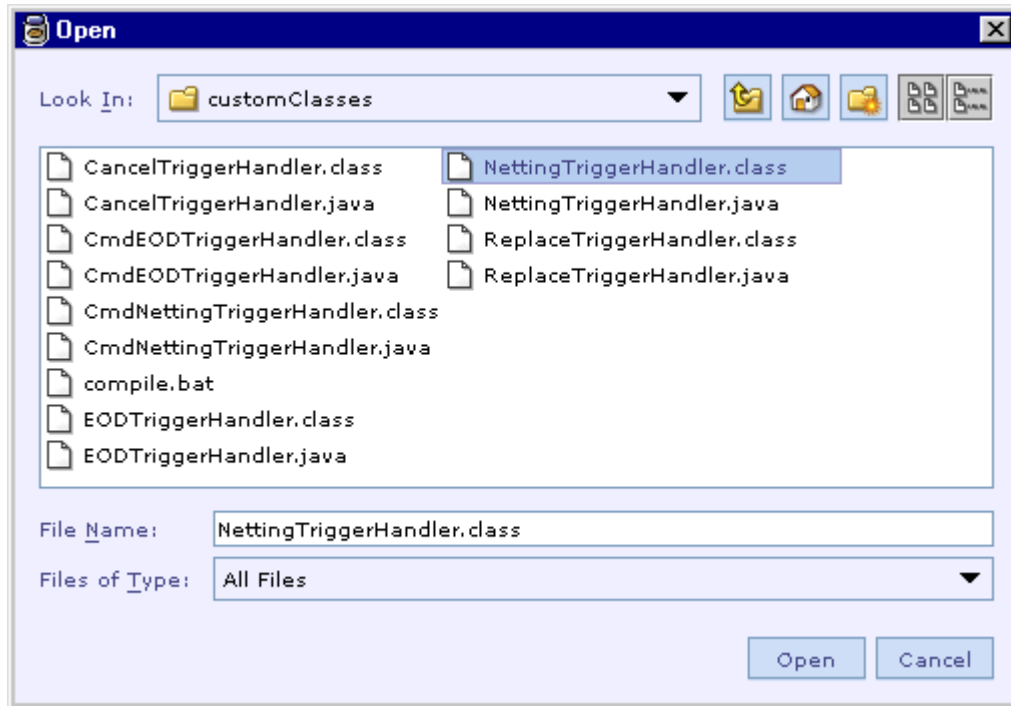
It is recommended that the compiled Java class files are available under a child directory of the cartridge directory.

2. In the 'Code Generation' tab of the 'Java/EJB Code Generation Settings' dialog, select the required design element in the tree structure displayed on left pane.
3. Click on the 'External Classes' button at the bottom of the right pane.
4. The 'External Classes' dialog will be displayed.



5. Click on the 'Add' button.

The file Open dialog will be displayed.



6. Navigate to the directory that contains the required Java class files and select them.
7. Click the 'Open' button.

The files will be included in the 'Class Files' list of the 'External Classes' dialog.

Note:

To remove class files from the 'Class Files' list, select them from the list and then click on the 'Remove' button of the 'External Classes' dialog.

See Also:

[Defining Java Custom Classes](#)

Appendix

The following table describes interfaces and classes that are used in writing the above described custom classes.

Handler Interfaces

Name	Description
------	-------------

com.tplus.transform.runtime.handler.IFieldValidation	Interface for external classes to validate a field of an internal/external message. The implementing class should check the value of the field and throw a FieldValidationException if it is not valid.
com.tplus.transform.runtime.handler.IProcessing	Interface for external classes to process (manipulate) the internal message. The implementing method can access and modify the fields of the internal message. This same interface is used for both pre and post processing.
com.tplus.transform.runtime.handler.IFieldProcessing	Interface for external classes to set the value of a field in an internal message. The implementing method should return a value (or null). This value will be set as the value of the field. This same interface is used for both pre and post processing of internal message fields.
com.tplus.transform.runtime.handler.IInputMapping	Interface for external classes to map the fields of input object to normalized object.
com.tplus.transform.runtime.handler.IOutputMapping	Interface for external classes to map the fields of normalized object to output object.
com.tplus.transform.runtime.handler.INormalizedObjectMapping	Interface for external classes to map the fields of one normalized object to another normalized object.
com.tplus.transform.runtime.handler.IExternalMapping	Interface for external classes to map the fields of one external message object to another external message object.

Context objects

Name	Description
com.tplus.transform.runtime.LookupContext	<p>The LookupContext provides access to other components executing in the runtime environment. Using the LookupContext you can</p> <ul style="list-style-type: none"> access other generated components, such as a external/internal message, message mapping etc. access application specific components. For instance,

	application specific bean if it is running under EJB environment.
com.tplus.transform.runtime.TransformContext	<p>Defines the context in which the current transformation occurs. This context object contains a set of properties (name-value) pairs related to the current transformation. This object is passed to all the activities (external message, mapping etc.) that take part in processing. The external entity (for instance CP) that sends input to the runtime sets properties related to input and output formats. Along the way, a component processing input may set more properties to be used by rest of components. These properties contain processing options for an activity and processing instructions such as what is the output format, and how it should be sent out (protocol). For a concrete implementation of this interface use TransformContextImpl. For an implementation that wraps another context use TransformContextWrapper.</p> <p>The value corresponding to the property INPUT_BATCH_CONTEXT_PROPERTY contains a BatchContext object, if the input is sent in batch mode. You can use BatchContext to store batch specific values.</p>

Data Objects

Name	Description
com.tplus.transform.runtime.DataObject	Generic object that encapsulates a data structure. Fields of the data structure can be accessed by name or by index. The index parameter is the 0 based index of field as defined in Designer. The class also provides meta-info about the data structure. The names of the fields can be obtained using the method getFieldname(index).
com.tplus.transform.runtime.NormalizedObject	Generic object that encapsulates an internal message. Fields of the data structure can be accessed by name or by index. The index parameter is the 0 based index of field as defined in Designer. The class also provides meta-info about the data structure. All the methods related to the NormalizedObject are defined in the DataObject interface.
com.tplus.transform.runtime.ExternalObject	Object representation of an external message. This object contains three parts, the header, data and the trailer. Fields in these three components can be accessed using methods in the DataObject interface. The structure of this object corresponds to

	the External Message defined using Designer. The External Object has a close relationship with External Message interface.
--	--