

Java Clients – Tutorial

Version 3.4

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA, Inc. The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with that agreement. No part of this guide may be reproduced or retransmitted in any form or by any means electronic, mechanical, or otherwise, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of BEA, Inc.

Copyright © 2001-2007 BEA, Inc. All rights reserved. All BEA Products are trademarks or registered trademarks of BEA, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

WRITING JAVA CLIENTS.....	4
USING SERVICE OBJECTS DEFINED IN DESIGNER.....	5
LOOKUPCONTEXT	6
TRANSFORMCONTEXT	7
GENERAL STEPS FOR PERFORMING A TRANSFORMATION.....	8
MESSAGE FLOW CLIENT	9
<i>Create Lookup Context</i>	<i>10</i>
<i>Create TransformContext</i>	<i>10</i>
<i>Setting Properties in TransformContext</i>	<i>10</i>
<i>Lookup Message Flow</i>	<i>11</i>
<i>Execute the Message flow</i>	<i>11</i>
RUN METHOD OVERLOADS.....	14
EXTERNAL MESSAGE CLIENT	16
<i>Create Lookup Context</i>	<i>16</i>
<i>Create TransformContext</i>	<i>17</i>
<i>Set Properties in TransformContext</i>	<i>17</i>
<i>Lookup External Message.....</i>	<i>17</i>
<i>Parse the Raw Message</i>	<i>17</i>
<i>Update Field in External Object.....</i>	<i>18</i>
<i>Serialize the External Object to CSV</i>	<i>18</i>
CLIENT CONTROLLED PROCESSING.....	20
<i>Create Lookup Context</i>	<i>21</i>
<i>Create TransformContext</i>	<i>21</i>
<i>Parse Raw Message</i>	<i>21</i>
<i>Map External Messages.....</i>	<i>22</i>
<i>Serialize the External Object</i>	<i>22</i>
<i>Extract field value from External Object.....</i>	<i>23</i>
REQUIRED JARS.....	24

Writing Java Clients

You can write Java clients that interact with cartridge entities deployed in a runtime environment. This tutorial helps you get started with writing Java clients that use the entities designed using Designer. Even though these samples are based on Simple Runtime environment, the clients for other Java runtime environments EJB and Allegro Server are similar to these clients.

Samples

a. Message Flow	<p>Illustrates invoking a Message flow from a Java client.</p> <p>Sample is under '<installation dir>\docs\Java\Clients\MessageFlow' directory.</p>
b. External Message	<p>Illustrates directly using the parse and write method of an External message.</p> <p>Sample is under '<installation dir>\docs\Java\Clients\ExternalMessage' directory.</p>
c. Client Controlled Processing	<p>This sample is similar to a) except that we perform all the activities of the message flow from the client itself. That is we don't use a Message flow here, but let the client coordinate the processing.</p> <p>Sample is under '<installation dir>\docs\Java\Clients\ClientControlled' directory.</p>

Though it is possible to access all artifacts that you designed in the Designer from Java client, we recommend that you use Message flow as a façade for other components. More specifically the examples b) and c) are just for demonstration purposes; if possible try to wrap your components with a Message flow and access it from your client. This way you need to learn about just the MessageFlow interface.

For more information on the classes and interfaces used refer to the [API documentation](#).

Please refer to the sample under the '<installation dir>\docs\Java\DeploymentAndConfiguration\EJB' directory for a sample application that illustrates deploying an application under EJB runtime environment.

Please refer to the sample under the '<installation dir>\docs\Java\DeploymentAndConfiguration\SimpleRuntime' directory for a sample application that illustrates deploying an application under Simple Runtime.

Please refer to the samples under the '<installation dir>\runtime\java\CommandProcessor' directory for sample applications that make use of the Command Processor client. The Command Processor is a canned client that can be used to exchange data between the runtime environment and various message sources like queues, files, FTP server, mail server, Web services, etc.

Please refer to the sample under the '<installation dir>\runtime\java\RepairReferenceImpl' directory for a sample application that illustrates how to use WebForm tags for interacting with cartridge entities deployed in the runtime environment from a JSP.

caveat:

- The focus of this tutorial is on writing Java client programs and not on designing messages or message flows. For more information on design time aspects refer to Designer.doc or MessageFlow.doc.
- It also assumes that you have generated code for the cartridge and deployed it in the runtime environment of your choice (Simple/Allegro/ EJB). The clients that are discussed here are independent of the target runtime. This tutorial focuses on writing a client program which uses the deployed entities and not on how to deploy the generated code in a specific runtime/server.

See Also:

[Using Service Objects defined in Designer](#)
[Required Jars](#)

Using Service Objects defined in Designer

The following service objects defined in Designer are available at runtime

- External Message
- Internal Message
- Message Flow

- Message Mapping

During code generation the service objects are converted to platform specific code. The generated service objects can be accessed at runtime using the LookupContext. These service objects can be obtained using the LookupContext. The recommended approach is to use the message flow as façade for other components. That is, the client program should interact only with the Message flow component. The message flow itself would invoke or make use of other components listed above.

See Also:

[LookupContext](#)

[TransformContext](#)

[General Steps for Performing a Transformation](#)

[Writing Java Clients](#)

LookupContext

The LookupContext provides access to other service objects executing in the runtime environment. Using the lookup context you can access other generated components such as input format, business transaction etc.

Obtaining Lookup Context:

The 'LookupContext' context can be obtained from the LookupContextFactory as shown below.

```
LookupContext lcxt = LookupContextFactory.getLookupContext();
```

The Lookup Context that is returned will differ depending on the environment in which it is obtained. That is why instead of instantiating the Lookup Context we use LookupContextFactory to obtain it.

Methods in Lookup Context:

The following are the methods that are most commonly used during transformation. These are not the complete set of methods available.

lookupMessage(String name)	Looks up a message and returns it. The name to be looked up should be the name of an external message or a internal message defined in Designer.
----------------------------	--

lookupExternalMessage(String name)	Looks up an external message and returns it. The name to be looked up should be the name of the external message defined in Designer.
lookupInternalMessage(String name)	Look up a Internal message (formerly Business Transaction) and returns it. The name to be looked up should be the name of the Internal message defined in Designer.
lookupMessageFlow(String name)	Looks up Message flow and returns it. The name to be looked up should be the name of a message flow defined in Designer.
lookupMessageMapping(String name)	Looks up Message Mapping and returns it. The name to be looked up should be the name of a mapping defined in Designer.
lookupDataSource(String name)	Looks up a data source. This is mostly used internally and is not relevant for clients

See Also:

[Using Service Objects defined in Designer](#)

TransformContext

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 components (input format, business transaction, etc.) that take part in processing.

Methods in Transform Context:

The following are the methods that are most commonly used during transformation. These are not the complete set of methods available.

setProperty(String prop, Object value)	This method can be used set any property related to the current transformation.
--	---

Note that TransformContext is an interface. You can use the class TransformContextImpl, class to create an instance.

See Also:

[Using Service Objects defined in Designer](#)

General Steps for Performing a Transformation

1. Create lookup context. The lookup context provides access to other components executing in the runtime environment.
2. Create transform context. This defines the context in which the transformation occurs.
3. Set relevant properties in TransformContext. In most cases you wouldn't need to set any properties.
4. Lookup the service object you are interested in. It can be an external message, a message flow etc. The names of the external message or message flow to be looked up should be defined in the cartridge.
5. Invoke the key method in service object you looked up in the previous step. This method depends on the service object and the operation you want to perform. In case of MessageFlow you would use the *run* method. In case of an ExternalMessage you would invoke the *parse* or *write* method.

These steps apply all type of clients and can be performed irrespective of whether you are using simple runtime, or whether your application is deployed in Allegro server or EJB.

See Also:

[Using Service Objects defined in Designer](#)

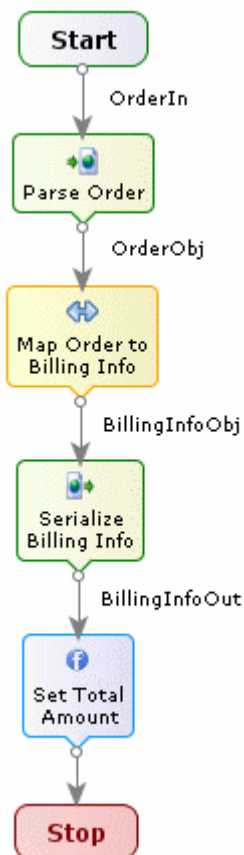
Message Flow Client

This sample illustrates looking up and invoking a message flow from a Java client.

Assumptions:

- The section assumes that you already have a cartridge and have defined a message flow in it. For details of how to compose a message definition and design a message flow refer to Designer.doc or MessageFlow.doc under <installation dir>\docs\Designer folder.
- It also assumes that you have generated code for the cartridge and deployed it in the runtime environment of your choice (Simple/Allegro/ EJB). This section focuses on writing a client program which uses the deployed entities.






For illustration purposes we use the MessageFlow cartridge which has a PurchaseOrderFlow defined in it. This sample is under the folder '<installation dir>\docs\Java\Clients\MessageFlow'.



The message flow in the sample cartridge takes a Order CSV as input and extracts the billing information from it and returns Bill Info in CSV form as output.

For illustration purposes it also computes the total amount in the order and returns it also as the output.

The input and output variables of the message flow are as shown below.

Field Name	Scope	Type
 OrderIn	INPUT	Binary
 OrderObj	LOCAL	Order
 BillingInfoObj	LOCAL	BillingInfo
 BillingInfoOut	OUTPUT	Binary
 TotalAmount	OUTPUT	Double

The following steps have to be performed for invoking a Message flow.

1. [Create Lookup Context](#)
2. [Create TransformContext](#)
3. [Setting Properties in TransformContext](#)
4. [Lookup Message Flow](#)
5. [Execute the Message flow](#)

Create Lookup Context

```
LookupContext lcxt = LookupContextFactory.getLookupContext();
```

The Lookup Context provides access to other components executing in the runtime environment. For more information regarding Lookup Context please refer [LookupContext](#).

Create TransformContext

TransformContext defines the context in which the transformation occurs. This context is passed to all the activities of the message flow. For more information regarding Transform Context please refer [TransformContext](#).

```
TransformContext tcxt = new TransformContextImpl();
```

Setting Properties in TransformContext

The properties of the TransformContext are used to pass additional options to the Message flow and its activities. As of now very few properties are supported; the TransformContext is mainly there to facilitate future enhancements. In this example we don't need to set any properties.

See Also:

[Message Flow Client](#)

Lookup Message Flow

A message flow with this name should have been defined in the cartridge.

```
MessageFlow messageFlow = lcxt.lookupMessageFlow("PurchaseOrderFlow");
```

If the message flow to be looked up is not deployed or the name specified is incorrect it will result in `javax.naming.NamingException`.

See Also:

[Message Flow Client](#)






Execute the Message flow

The Message flow interface provides a generic *run* method which executes the message flow.

```
Object[] run(Object[] messageFlowArgs, TransformContext cxt)
```

The run method takes an `Object[]` as parameter and returns a `Object[]` as output. Note that the actual parameters and return values of a Message Flow are defined by the user who designs a message flow. This generic signature accommodates any number of input parameters and return values.

Consider `PurchaseOrderFlow` which has the following variables (as shown in the picture)

Field Name	Scope	Type
 OrderIn	INPUT	Binary
 OrderObj	LOCAL	Order
 BillingInfoObj	LOCAL	BillingInfo
 BillingInfoOut	OUTPUT	Binary
 TotalAmount	OUTPUT	Double

The input variables are,

- OrderIn (Binary – raw message)

The output variables are,

- BillingInfoOut (Binary – raw message)
- TotalAmount. (Double)

When we call the run method, we need to pass a raw message as input (byte[]) and we will get a transformed raw message as output along with another output of type double.

1. Construct the input Object[]. We know that the message flow takes one argument as input. Get the raw message from some source (say file) and create a Object[] of length 1, as given below.

```
byte[] orderIn = new FileInputStream(fileName).getAsBytes();
Object[] messageFlowArgs = new Object[] { orderIn };
```

2. Execute the message flow by invoking the run method.

```
Object[] output = messageFlow.run(messageFlowArgs, tcxt);
```

3. We know that the message flow returns two outputs. The outputs are in the returned Object[] at the appropriate index (same order as defined in Message flow design). Extract the outputs and use them.

```
byte[] billingInfoOut = (byte[])output[0];
Double totalAmount = (Double)output[1];
```

Note that the order of the output in the array is same as that of the output variables specified during Message Flow design.

Source Code:

```
import com.tplus.transform.runtime.*;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        String fileName = "data.txt";
        if(args.length > 0) {
            fileName = args[0];
        }
        try {
            // Get the lookup context for the current environment
            LookupContext lcxt = LookupContextFactory.getLookupContext();

            // Lookup purchase order message flow
            MessageFlow messageFlow =
                lcxt.lookupMessageFlow("PurchaseOrderFlow");
```

```

//Create a TransformContext. We have no special properties
TransformContext cxt = new TransformContextImpl();

//Prepare the input for the message flow
//The message flow defined in the cartridge takes
//only a raw message as input
byte[] orderIn = new FileInputStream(fileName).getAsBytes();
Object[] messageFlowArgs = new Object[] { orderIn };

// Execute the message flow.
Object[] output = messageFlow.run(messageFlowArgs, cxt);

// Extract the outputs
byte[] billingInfoOut = (byte[])output[0];
Double totalAmount = (Double)output[1];

System.out.println("BillingInfoOut:");
System.out.write(rawOut);
System.out.println("TotalAmount: " + totalAmount);
}
catch(TransformException e) {
    System.err.println(e.toXMLString());
}
catch(javax.naming.NamingException e) {
    e.printStackTrace();
}
catch(java.io.IOException e) {
    e.printStackTrace();
}
}
}

```

See Also:

[run method overloads](#)
[Message Flow Client](#)

run method overloads

The MessageFlow interface provides two of variants of the run method.

- run - returns the output and throws an exception in case of failure.
- run2 - returns the output and any exceptions as a Result object. The output generated is a snapshot of output variables in the message flow at the time of the exception. This method is useful in a 'Repair' like application where incorrect input should not be treated as fatal. This method is for advanced users and this document does not discuss it in detail.

For each variant it provides number of overloaded run methods. These overloads are for the convenience of the client and are functionally equivalent. The general idea of invoking the message flow remains the same.

One such overloaded run method has the following signature,

```
DataObject run(DataObject input, TransformContext cxt);
```

Note that the first argument and the return value has changed from Object[] to a DataObject. This method is equivalent in functionality to the one described earlier. Since DataObjects are type safe objects, this method has better type safety compared to the one that uses Object[].

The DataObject that you pass as input, should be created using the createInputDataObject() in the MessageFlow interface.

1. Construct the input Object[]. We know that the message flow takes one argument as input.
 - Get the raw message from some source (say file)
 - create a DataObject using the createInputDataObject
 - Set input parameter's value

```
byte[] orderIn = new FileInputStream(fileName).getAsBytes();  
DataObject inputDataObj = messageFlow.createInputDataObject();  
inputDataObj.setField("OrderIn", orderIn);
```

2. Execute the message flow by invoking the run method.

```
Result result = messageFlow.run(inputDataObj, cxt);
```

3. Extract the output from the Result object.

```
DataObject outputDataObj = (DataObject)result.getOutput();
```

4. We know that the message flow returns two outputs. The outputs are in the returned DataObject. Extract the outputs using the names specified in the Designer.

```
byte[] billingInfoOut =  
    (byte[])outputDataObj.getField("BillingInfoOut");  
Double totalAmount = (Double)outputDataObj.getField("TotalAmount");
```

```
public class Main {  
    public static void main(String[] args) {  
  
        // .... as in previous sample  
  
        // Prepare the input for the message flow  
        // The message flow defined in the cartridge takes  
        // only a raw message as input  
        DataObject inputDataObj = messageFlow.createInputDataObject();  
        inputDataObj.setField("OrderIn",  
            new FileInputStream(fileName).getAsBytes());  
  
        // Execute the message flow  
        Result result = messageFlow.run2(inputDataObj, cxt);  
  
        // Extract the output  
        DataObject outputDataObj = (DataObject)result.getOutput();  
  
        if (outputDataObj != null) {  
            byte[] billingInfoOut =  
                (byte[])outputDataObj.getField("BillingInfoOut");  
            Double totalAmount =  
                (Double)outputDataObj.getField("TotalAmount");  
  
            // Write output  
            if(billingInfoOut != null) {  
                System.out.println("BillingInfoOut:");  
                System.out.write(billingInfoOut);  
            }  
            if (totalAmount != null) {  
                System.out.println("TotalAmount: " + totalAmount);  
            }  
        }  
    }  
    // Write exception
```

```

        if (result.hasException()) {
            java.util.List exceptions = result.getExceptions();
            StringBuffer errors = new StringBuffer();
            for (int i = 0; i < exceptions.size(); ++i) {
                errors.append(((ExceptionObject)
                            exceptions.get(i)).toXMLString());
            }
            System.out.println("Exceptions:");
            System.out.println(errors);
        }
    } // handle exceptions

```

See Also:

[Message Flow Client](#)

External Message Client

This section illustrates using an ExternalMessage object to parse a raw message, modify it, and serialize it back to raw message.

The sample cartridge is under the folder '<installation dir>\docs\Java\Clients\ExternalMessage'.

- Parse the Order CSV.
- Change the order date in the Order object to today.
- Write back the order object as CSV.

The following steps have to be performed for using an External Message.

1. [Create Lookup Context](#)
2. [Create TransformContext](#)
3. [Set Properties in TransformContext](#)
4. [Lookup External Message](#)
5. [Parse the raw message](#)
6. [Update Field in External Object](#)
7. [Serialize the External Object to CSV](#)

Create Lookup Context

```
LookupContext lcxt = LookupContextFactory.getLookupContext();
```

The Lookup Context provides access to other components executing in the runtime environment. For more information regarding Lookup Context please refer [LookupContext](#). We will use this to lookup the external message.

Create TransformContext

TransformContext defines the context in which the transformation occurs. This context is passed to all the components (input format, business transaction, trigger etc.)

```
TransformContext tcxt = new TransformContextImpl();
```

For more information regarding Transform Context please refer [TransformContext](#)

Set Properties in TransformContext

The properties of the TransformContext are used to pass additional options to the Message flow and its activities. As of now very few properties are supported the TransformContext is mainly there to facilitate future enhancements. In this example we don't need to set any properties.

See Also:

[External Message Client](#)

Lookup External Message

An external message with this name has been defined in the cartridge.

```
ExternalMessage orderMessage = lcxt.lookupExternalMessage("Order");
```

If this external message is not deployed or the name specified is incorrect it will result in `javax.naming.NamingException`.

See Also:

[External Message Client](#)

Parse the Raw Message

The external message interface has a parse method that parses a raw message and returns its object representation.

```
DataObject parse(InputSource source, TransformContext cxt)
```

The parse method takes an `InputSource` as parameter. There are number of concrete implementations of `InputSource` such as `FileInputSource`, `ByteArrayInputSource` etc.

```
FileInputSource inputSource = new FileInputSource(fileName);
```

```
DataObject orderObject = orderMessage.parse(inputSource, cxt);
```









Here, we use the `FileInputStream` to read contents of a file and pass it to the `parse` method, which returns an `ExternalObject`. The structure of the `ExternalMessage` was defined in the Designer.

See Also:

[External Message Client](#)

Update Field in External Object

In this example we will change the order date in the order to today's date. The field we want to change is in the Header part of the External Object.

Field Name	Type
 orderDate	DateOnly
 billToCountry	String
 billToName	String
 billToStreet	String
 billToCity	String
 billToState	String
 billToZIP	String
 comment	String

```
DataObject headerPart = orderObject.getHeader();  
headerPart.setField ("orderDate", new Date());
```

See Also:

[External Message Client](#)

Serialize the External Object to CSV

To serialize the `ExternalObject` we need to use the `serialize` method in the `ExternalMessage` interface.

```
byte[] serialize(DataObject obj, TransformContext cxt)
```

The `serialize` method takes the external object as parameter and returns the serialized (in this case as CSV) data as output.

```
byte[] output = (byte[])orderMessage.serialize(orderObject, cxt);
```

Note:

There is another variant of parse and write methods, parse2 and write2. The primary difference is that the latter continue to process even if there are errors. Also they may return the output, exceptions encountered or both.

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

Source Code:

```
import com.tplus.transform.runtime.*;
import java.io.IOException;
import com.tplus.transform.runtime.formula.DateFunctions;
public class Main {
    public static void main(String[] args) {
        com.tplus.transform.util.LoggingUtil.enableLogging("log.xml");
        String fileName = "data.txt";
        if(args.length > 0) {
            fileName = args[0];
        }
        try {
            // Get the lookup context for the current environment
            LookupContext lcxt = LookupContextFactory.getLookupContext();

            // Lookup order message (defined in the cartridge).
            ExternalMessage orderMessage =
                lcxt.lookupExternalMessage("Order");

            // Create a TransformContext.
            TransformContext cxt = new TransformContextImpl();

            //Prepare the input message to be parsed
            FileInputSource inputSource = new FileInputSource(fileName);

            // Invoke the parse() method on the Order message object.
            ExternalObject orderObject =
                (ExternalObject)orderMessage.parse(inputSource, cxt);

            //Modify the orderDate field with the current date value.
            orderObject.getHeader().setField("orderDate", new java.util.Date());

            //Write the modified Order message object.
            byte[] output = orderMessage.serialize(orderObject, cxt);

            System.out.println("Modified Order:");
            System.out.write(output);
        }
    }
}
```

```

    }
    catch(TransformException e)
    {
        System.err.println(e.toXMLString());
    }
    catch(javax.xml.naming.NamingException e)
    {
        e.printStackTrace();
    }
    catch(java.io.IOException e)
    {
        e.printStackTrace();
    }
}
}

```

See Also:

[External Message Client](#)

Client Controlled Processing

In this sample the client code invokes number of service elements one after the other, passing output of one to the next. That is, we execute all the activities that are typically used as part of a message flow from the client itself. We don't use a message flow here, but let the client coordinate the processing.

This sample is functionally same as sample a) where an Order CSV was transformed to Billing Info CSV. The main difference is that we don't use a message flow here but let the client wire the activities.

As mentioned earlier, this approach is not recommended. Prefer a message flow façade if possible.

The sample cartridge and source are under the folder '<installation dir>\docs\Java\Clients\ClientControlled'.

The following steps have to be performed for using Client Controlled Processing.

1. [Create Lookup Context](#)
2. [Create TransformContext](#)
3. [Parse Raw Message](#)
4. [Map External Messages](#)
5. [Serialize the External Object](#)

6. [Extract field value from External Object](#)

Create Lookup Context

```
LookupContext lcxt = LookupContextFactory.getLookupContext();
```

The Lookup Context provides access to other components executing in the runtime environment. For more information regarding Lookup Context please refer [LookupContext](#).

Create TransformContext

TransformContext defines the context in which the transformation occurs. This context is passed to all the activities of the message flow. For more information regarding Transform Context please refer [TransformContext](#)

```
TransformContext tcxt = new TransformContextImpl();
```

The properties of the TransformContext are used to pass additional options to activities. As of now very few properties are supported; the TransformContext is mainly there to facilitate future enhancements. In this example we don't need to set any properties.

Parse Raw Message

- First we look up the order external message.

```
ExternalMessage orderMessage = lcxt.lookupExternalMessage("Order");
```

- The external message interface has a parse method that parses a raw message and returns its object representation.

```
DataObject parse(InputSource source, TransformContext cxt)
```

- The parse method takes an InputSource as parameter. There are number of concrete implementations of InputSource such as FileInputSource, ByteArrayInputSource etc.

```
FileInputSource inputSource = new FileInputSource(fileName);  
DataObject orderObject = orderMessage.parse(inputSource, cxt);
```

Here, we use the FileInputSource to read contents of a file and pass it to the parse method, which returns an ExternalObject. The structure of the ExternalMessage was defined in the Designer.

See Also:

[Client Controlled Processing](#)

Map External Messages

Here we map/transform a Purchase order object to a Billing Info object using the mapping defined in the cartridge.

- First, we lookup the Message Mapping component as shown below.

```
MessageMapping mapping = lcxt.lookupMessageMapping("OrderToBillingInfo");
```

The MessageMapping interface has a map method with the following signature.

```
void map(DataObject src, DataObject dest, TransformContext cxt)
```

It maps from a source object to an uninitialized destination object.

We have the source object ready, the one we got by invoking the parse method.

We need a raw destination object; that is, an object of type BillingInfo.

- We create an uninitialized destination object by looking up BillingInfo external message and creating a raw instance.

```
ExternalMessage billingInfoMessage =  
    cxt.lookupExternalMessage("BillingInfo");  
DataObject billingInfoObject= billingInfoMessage.createObject();
```

- Then we map the source object to the destination object.

```
mapping.map(orderObject, billingInfoObject, cxt);
```

See Also:

[Client Controlled Processing](#)

Serialize the External Object

We then serialize the Billing Info object (obtained from mapping) to raw form. We use the serialize method in the ExternalMessage interface for this purpose.

```
byte[] rawOut = billingInfoMessage.serialize(billingInfoObject, cxt);
```

See Also:

[Client Controlled Processing](#)

Extract field value from External Object

Let's say that we want to get the total value (price) of the Purchase order in the client code. We have added this information to the trailer of the Billing info during mapping. We can extract this value from the billingInfoObject as shown below.

```
DataObject trailer = billingInfoObject.getTrailer();
Double totalAmount = (Double)trailer.getField("TotalAmount");
```

Source Code:

```
import com.tplus.transform.runtime.*;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        com.tplus.transform.util.LoggingUtil.enableLogging("log.xml");
        String fileName = "data.txt";
        if(args.length > 0) {
            fileName = args[0];
        }
        try {
            // Get the lookup context for the current environment
            LookupContext lcxt = LookupContextFactory.getLookupContext();

            // Create a TransformContext.
            TransformContext cxt = new TransformContextImpl();

            // Lookup order message (defined in the cartridge)
            ExternalMessage orderMessage =
                lcxt.lookupExternalMessage("Order");

            //Prepare the input message to be parsed
            FileInputSource inputSource = new FileInputSource(fileName);

            // Parse the input message and create the 'Order' object.
            DataObject orderObject = orderMessage.parse(inputSource, cxt);

            // Lookup 'OrderToBillingInfo' mapping
            MessageMapping mapping =
                lcxt.lookupMessageMapping("OrderToBillingInfo");

            // Lookup 'BillingInfo' external message
            ExternalMessage billingInfoMessage =
                lcxt.lookupExternalMessage("BillingInfo");
```

```

        // Create 'BillingInfo' object
        ExternalObject billingInfoObject =
            (ExternalObject)billingInfoMessage.createObject();

        // Map 'Order' object to 'BillingInfo' object
        mapping.map(orderObject, billingInfoObject, cxt);

        // Write the 'BillingInfo' object
        byte[] rawOut = billingInfoMessage.serialize(
            billingInfoObject, cxt);

        System.out.println("Output Message:");
        System.out.write(rawOut);

        // Retrieve and write 'TotalAmount'
        DataObject trailer = billingInfoObject.getTrailer();
        Double totalAmount = (Double)trailer.getField("TotalAmount");
        System.out.println("TotalAmount: " + totalAmount);
    }
    // handle exception ...
    catch(TransformException e) {
        System.err.println(e.toXMLString());
    }
    catch(javax.xml.naming.NamingException e) {
        e.printStackTrace();
    }
    catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
    catch(java.io.IOException e) {
        e.printStackTrace();
    }
}
}

```

See Also:

[Client Controlled Processing](#)

Required Jars

Given below is the list of libraries that are required when deploying the application under Simple Runtime. Please refer to '<Designer

Home>\docs\Java\Volante-Jars.html' for details of Runtime Jars and their dependencies.

- cp.jar (to be used if your application uses the Command Processor).
- simplert.jar
- hsqldb.jar or the file that contains the JDBC driver class for connecting to your database, if required.
- the Jar files generated for your cartridge(s).
- the Jar files specified in the 'Manifest Entries' code generation property of the cartridge.
- Plug-in runtime Jar files (such as swifttrt.jar) corresponding to the plug-ins used in the cartridge and the Jar files specified in their manifest entries (such as swiftresources.jar in case of swifttrt.jar).
- transformrt.jar, generaltutils.jar and sqlutils.jar (to be used with all runtime applications).
- commons-logging.jar and log4j.jar required for logging. If you are using JDK1.4 or above and would like to use JDK logging instead of log4j, log4j.jar is not needed.
- j2ee.jar and resourcemanager.jar specified in the manifest entry of simplert.jar.
- xml-apis.jar and xercesImpl.jar specified in the manifest entry of cp.jar (not needed if you are using JRE1.4 or above).
- jakarta-regexp.jar if your cartridge uses regex formula functions. Also needed if you use XML plug-in with pattern facets.

See Also:

[Writing Java Clients](#)