Implementing Java integration components for IBM Business Process Manager V7.5.1

Ricardo Olivieri

June 13, 2012

Learn how to write Java integration components for business process applications in IBM® Business Process Manager V7.5.1. Depending on the functional or non-functional requirements of the business process application that you are implementing, you may find that none of the out-of-the-box connectors provide the functionality you need. For such cases, you may want to consider writing a Java™ integration component.

Introduction

In this article, we will show you how to write Java integration components for business process applications in IBM Business Process Manager. Depending on the functional or non-functional requirements of the business process application that you are implementing, you may find that none of the out-of-the-box connectors provide the functionality you need. For such cases, you may want to consider writing a Java integration component.

Assumptions

A deep and thorough understanding of IBM Business Process Manager (BPM) V7.5.1 is not required to understand the topics discussed in this article. However, you should be familiar with the product's basic function and with general concepts of business process management. Also, knowledge of Java, JavaScript, integrated development environments (such as Eclipse), and programming languages in general are assumed.

Java integration

Try the Workflow service

Create long-running, stateful workflows that orchestrate tasks and services with synchronous or asynchronous event-driven interactions with the Workflow service from Bluemix. Try it for free!

The Java integration capability allows you to use Java code to extend the functionality of your business process application in IBM BPM. For instance, let's say you wanted to use the power of a templating engine, such as Velocity or FreeMarker, in your process application to generate text. You can write a Java integration component to do so.

© Copyright IBM Corporation 2012

An integration service that uses Java components

Advanced integration services

An alternative to writing a Java integration component is to implement an advanced integration service (AIS) in IBM Integration Designer. However, doing so is beyond the scope of this article. Instead, we will focus on the Java integration option, which is available in Process Designer (an option that was inherited from the WebSphere® Lombardi Edition product). Depending on the complexity of the functionality you may need in your business process application, you can decide on the option best suited for you.

A sample integration service named "Service With Java Integration Components" was implemented in IBM BPM and is used throughout this article to show how integration with a Java component is achieved. A TWX file that contains this sample integration service is included in the Download section of this article so you can import it into your IBM BPM Process Center.

This TWX file also includes the Java JAR file that contains the Java code that was written to support the implementation of the integration components in the sample service. Figure 1 shows the implementation diagram for the sample integration service, Service With Java Integration Components. Figure 2 shows the different variables that are used in this service.

Figure 1. Service with Java integration components

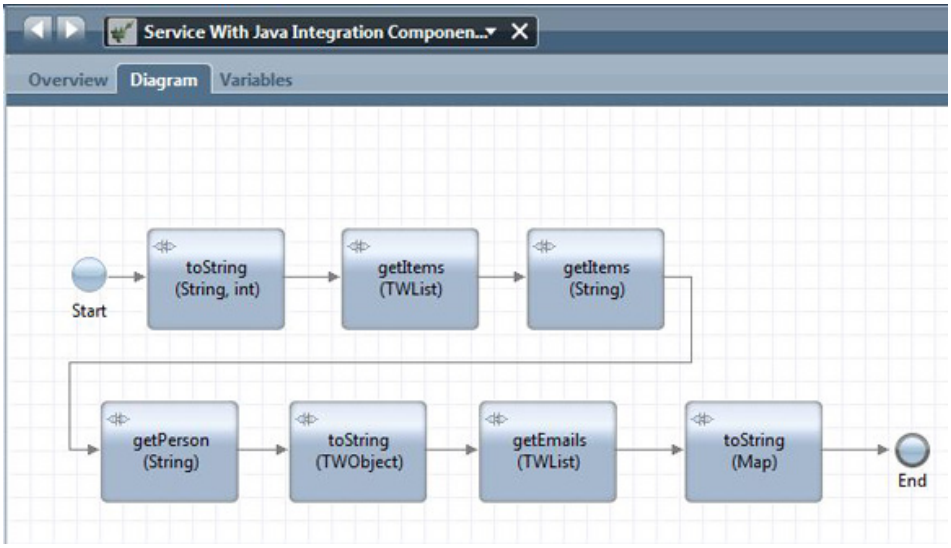
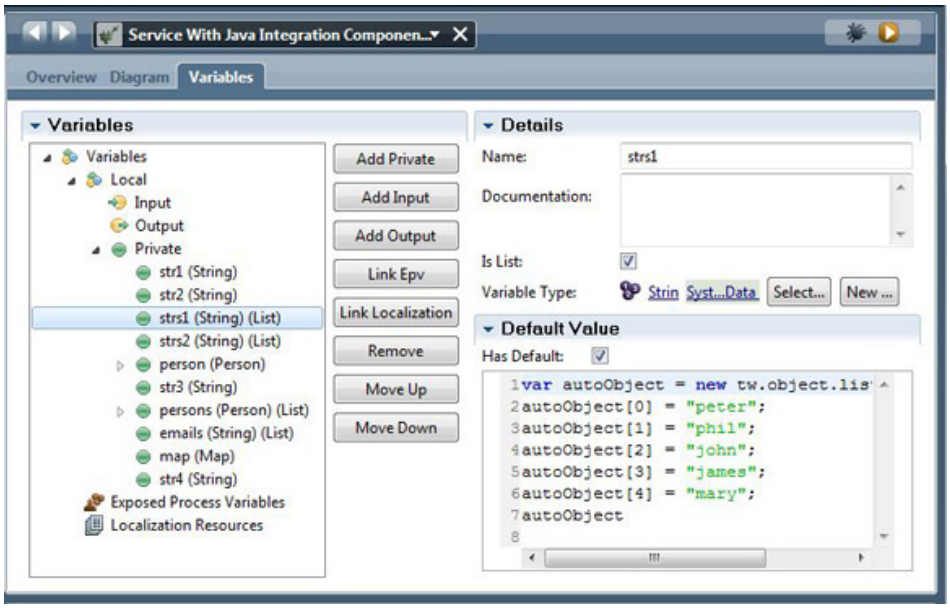


Figure 2. Variables used in the sample integration service



The sample integration service consists of a series of steps that are implemented in Java code. Each one of the boxes shown in the diagram represents a Java method invocation.

Note that each one of these boxes was created by dragging the "Java Integration" component from the service palette on Process Designer (see Figure 3) and dropping it on the service canvas. Once a Java integration component is placed on the canvas, you can associate it with the Java class and method that will be invoked. We will go over each one of the Java methods used in this service and describe how they are implemented.

Figure 3. Java integration component in the service palette

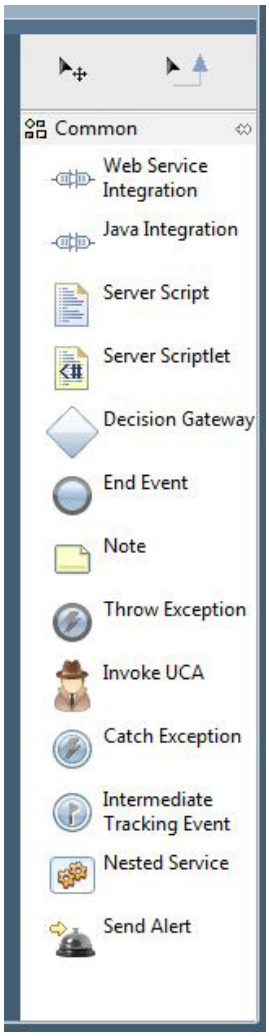
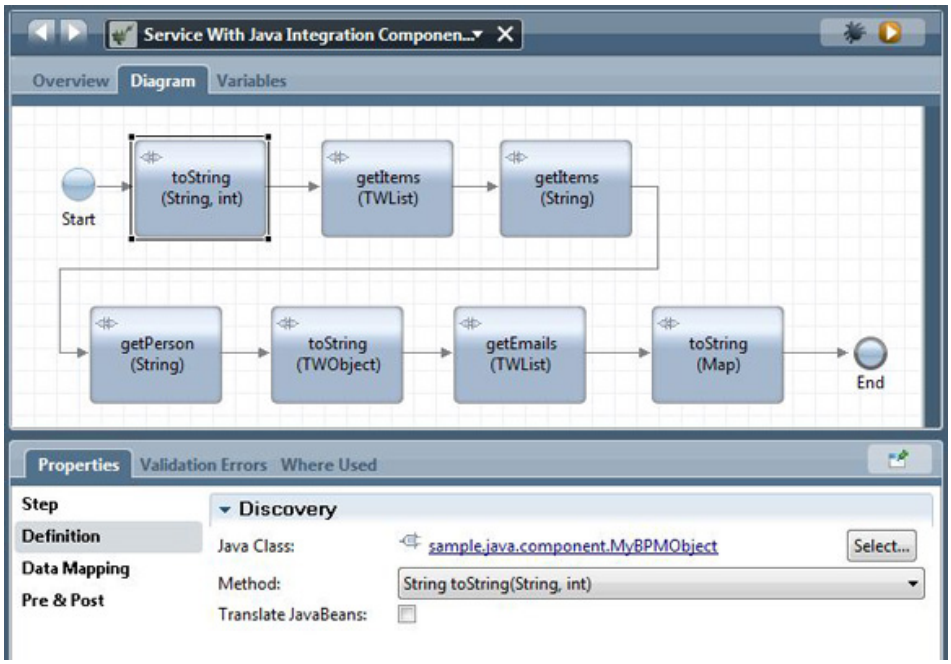


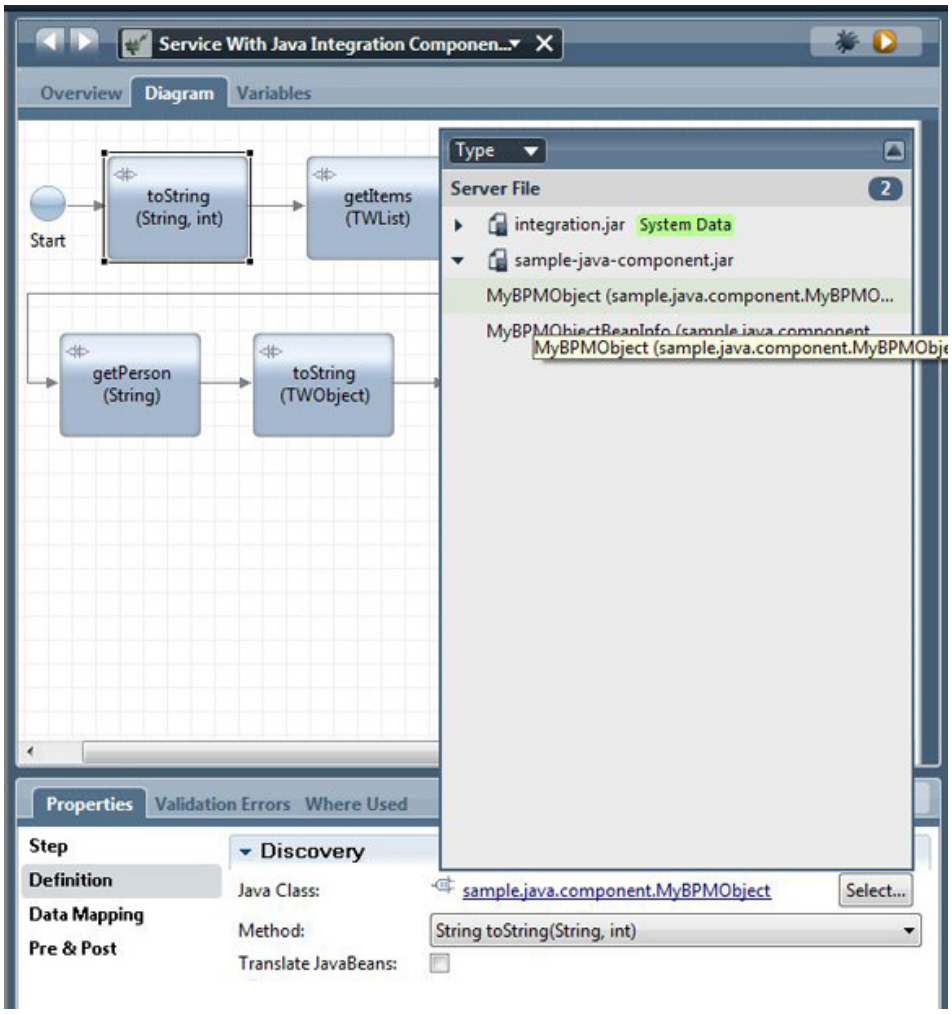
Figure 4 shows the Definition section in the Properties tab for the first integration step in the service diagram. You can see that the Java method invocation corresponds to the `toString(String, int)` method and that it belongs to the `sample.java.component.MyBPMObject` class. It is worth noting now that all method invocations in the sample service correspond to methods contained in the `sample.java.component.MyBPMObject` class.

Figure 4. Java integration definition for the first step



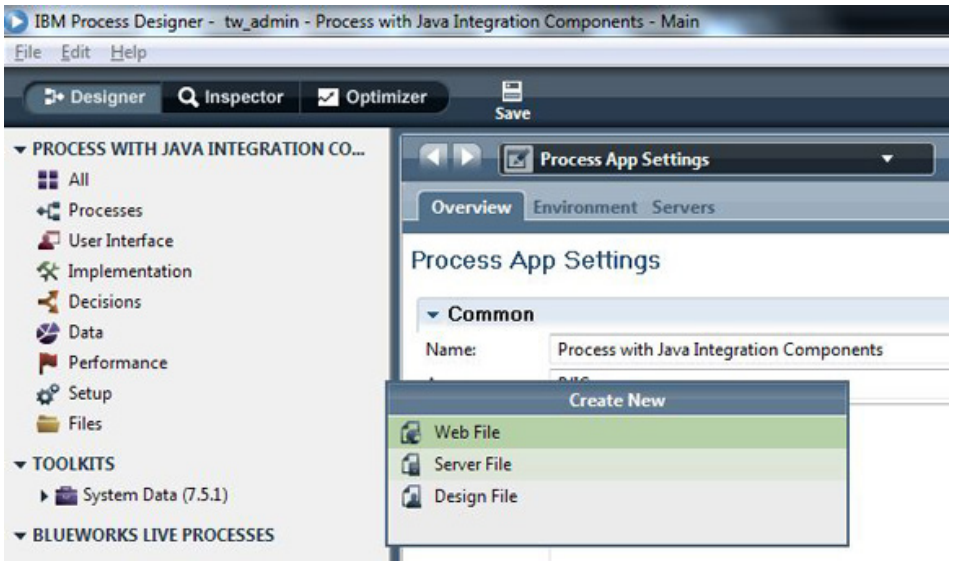
Looking at the method signature shown in the Definition section, you can see that it returns a string and receives two parameters, a string and an integer. If you click on the **Select** button shown next to the class name (`sample.java.component.MyBPMObject`), you get a list of all the JAR files that are available for the business process application (see Figure 5). If you expand the JAR file named **sample-java-component.jar**, you will find that this is the JAR file that contains the `sample.java.component.MyBPMObject` class.

Figure 5. JAR file containing the MyBPMObject class



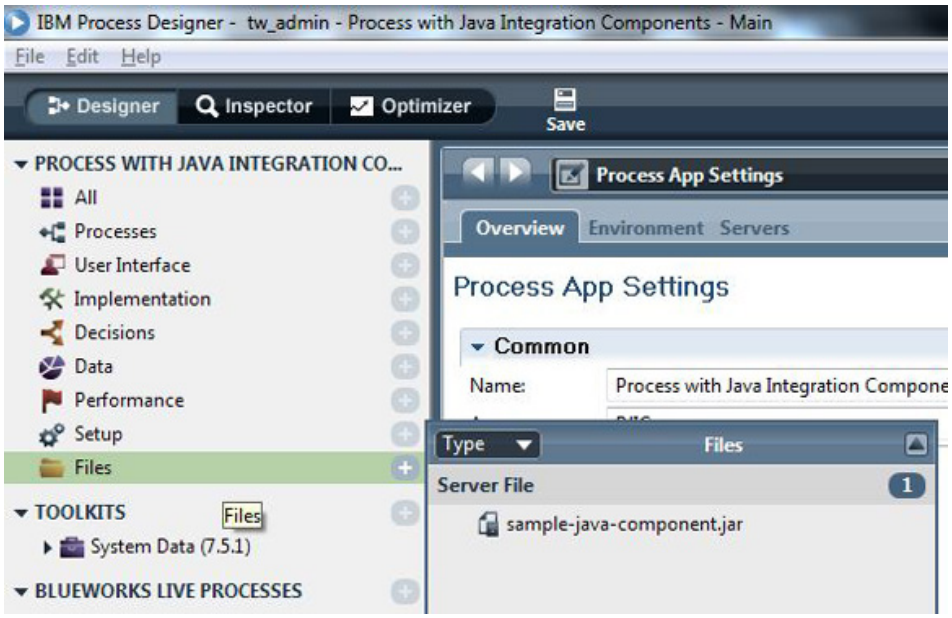
To upload a Java JAR file into your process application, simply add the JAR file as a server file as shown in Figure 6.

Figure 6. Uploading a JAR file into a process application



Once the JAR file has been added, it will show up as an available File resource to your process application, as shown in Figure 7. This makes the content of the JAR file available as an option when defining a Java integration component.

Figure 7. JAR file as a file resource in a process application



Let's now look at the implementation of the `toString(String, int)` method, which is shown in Listing 1. The signature of this method matches the definition you saw earlier for the first Java integration step in the sample service. The primitive types, such as strings and integers, used for scripting in IBM BPM can be mapped directly to the primitive types in Java. The `toString(String, int)` method simply returns a string containing the string and integer values that were passed to it. The string value returned from this method is mapped to the `str1` string variable in IBM BPM

as shown in Figure 8. Figure 8 shows the data mapping for the first integration component in the sample service. Note that the data mapping matches the method's signature.

Listing 1. Implementation of the toString(String) method

```
public String toString(String str, int num) {
 return "String: '" + str + "', Integer: '" + num + '\'';
}
```

Figure 8. Data mapping for the first integration step

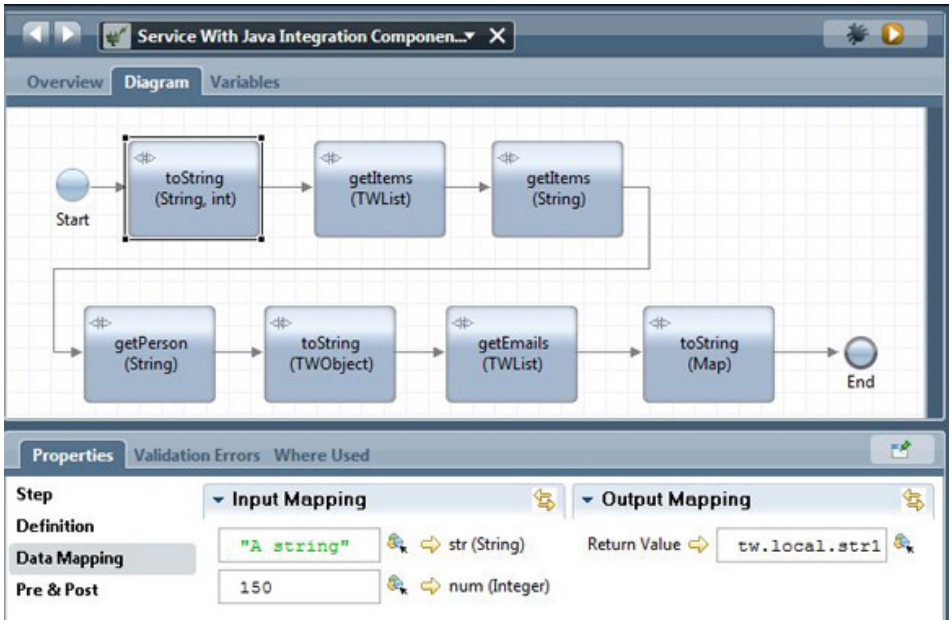


Figure 9 shows the Definition section for the second box in the integration service diagram, while Figure 10 shows its data mapping. This integration step invokes the `getItems(TWList)` method of the `MyBPMObject` class. If you have a list of objects in IBM BPM and you want to pass it to a method in a Java integration component, then you define a method in Java that receives a `teamworks.TWList` object as its argument. You can think of the `TWList` object as the Java entity that IBM BPM uses to represent an array of items. Note that items in an array or list in IBM BPM are either simple types or complex types.

Figure 9. Java integration definition for the second step

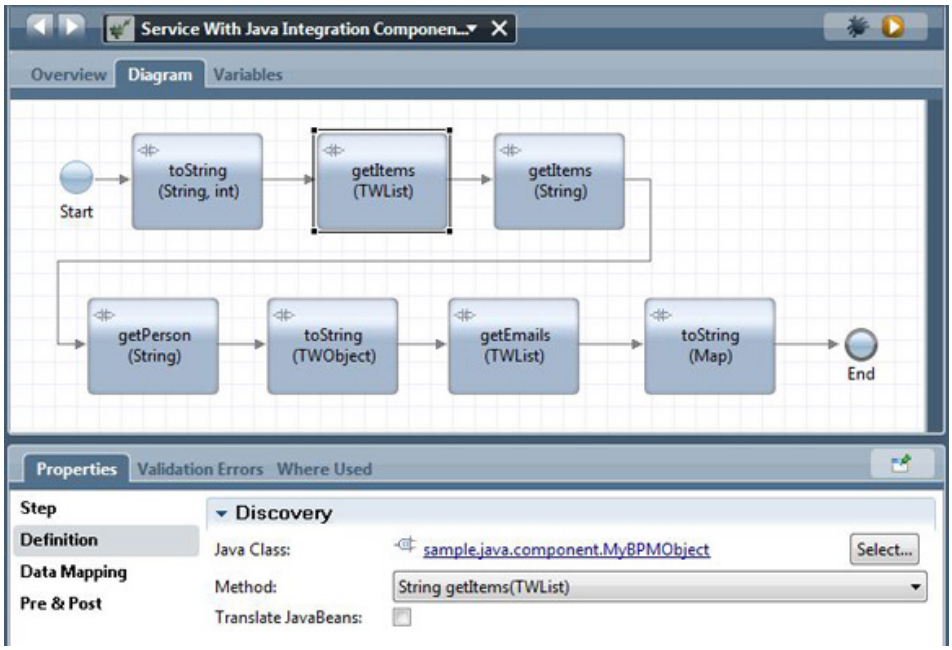
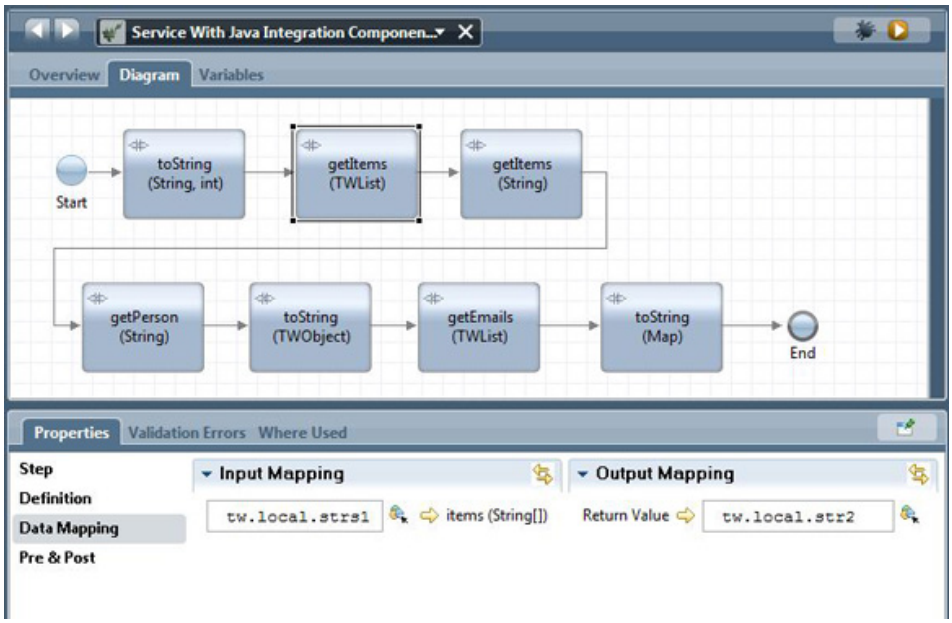


Figure 10. Data mapping for the second integration step



The `getItems(TWList)` method receives a `teamworks.TWList` object, iterates over its elements, and creates a comma delimited string that contains each one of the array elements. Since different TWList instances can hold objects of different types, the value returned from the invocation of the `TWList.getArrayData(int)` method must be cast to its corresponding type. Listing 2 shows the implementation for this method.

Listing 2. Implementation of the getItems(TWList) method

```
public String getItems(TWList items) {
  StringBuffer buffer = new StringBuffer();
  int size = items.getArraySize();
  for (int i = 0; i < size; i++) {
    String member = (String) items.getArrayData(i);
    buffer.append(member);
    if (i < (size - 1)) {
      buffer.append(',');
    }
  }
  return buffer.toString();
}
```

Figure 11 shows the Definition section for the third integration step in the sample service, while Figure 12 shows its data mapping. The `getItems(String)` method shows how to create a `teamworks.TWList` instance and how to add `TWObjectFactory.createList()` static method and elements are added to it by invoking its `addArrayData(Object)` instance method. As mentioned earlier, TWList is what IBM BPM uses to represent an array or list of elements. The `getItems(String)` method receives a comma delimited string and returns a list of the strings that are separated by a comma in the input string (see Listing 3).

Figure 11. Java integration definition for the third step

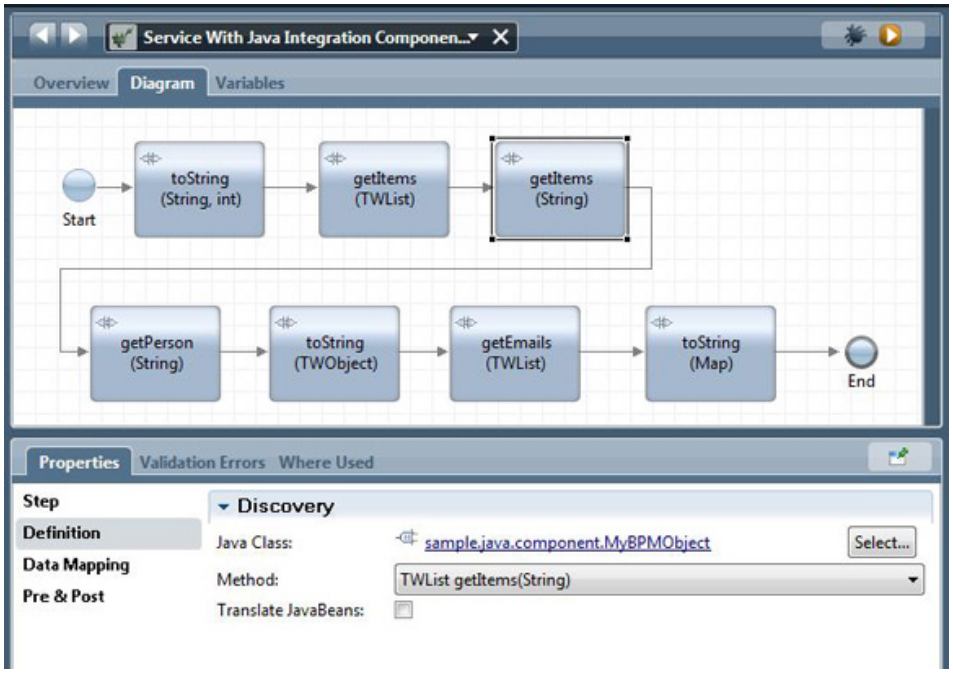
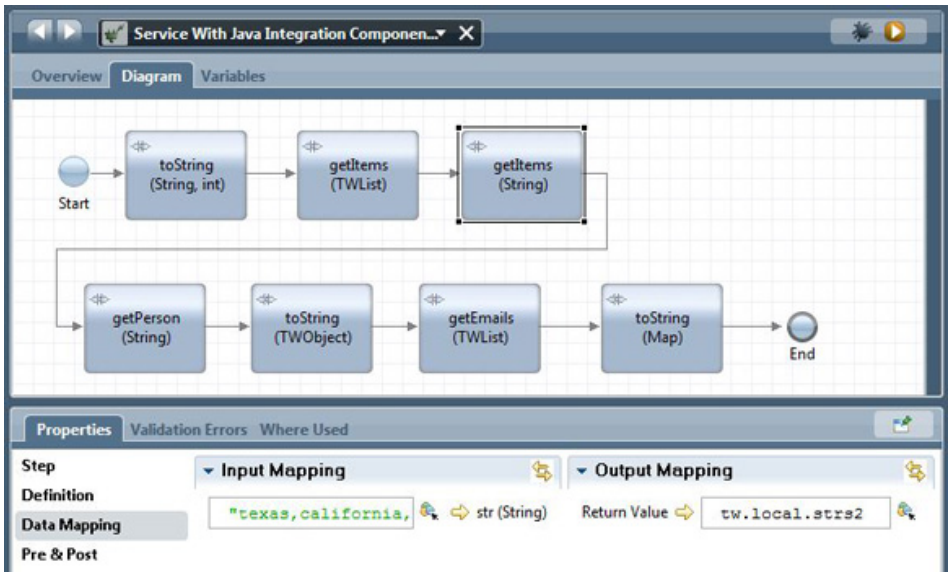


Figure 12. Data mapping for the third integration step



Listing 3. Implementation of the getItems(String) method

```
public TWList getItems(String str) throws Exception {
 String tokens[] = str.split(",");
 TWList items = TWObjectFactory.createList();
  for (String member : tokens) {
   items.addArrayData(member);
  }
 return items;
}
```

Purpose of the MyBPMObject class

Note that the same functionality provided in the sample `MyBPMObject` class can be implemented using the JavaScript scripting in IBM BPM. The intent of the sample `MyBPMObject` class is to show how IBM BPM interfaces with code written in Java and *not* to show how to concatenate strings or how to create a comma delimited string in Java.

Figure 13 shows the Definition section for the fourth box in the service diagram, while Figure 14 shows its data mapping. The `getPerson(String)` method returns an object that matches the structure of the Person complex type defined in the IBM BPM process application (see Figure 15). It is a structure made up of the strings and integer simple types. Listing 4 shows the implementation for the `getPerson(String)` method. `teamworks.TWObject` is the Java entity that IBM BPM uses to represent complex data types defined in your business process application. Hence, this method creates an instance of the `teamworks.TWObject` type, sets the corresponding property values on it, and returns it. The `TWObjectFactory.createObject()` creates a `TWObject` instance. The `setPropertyValue(String,Object)` instance method on the `TWObject` object is used to set its properties.

Figure 13. Java integration definition for the fourth step

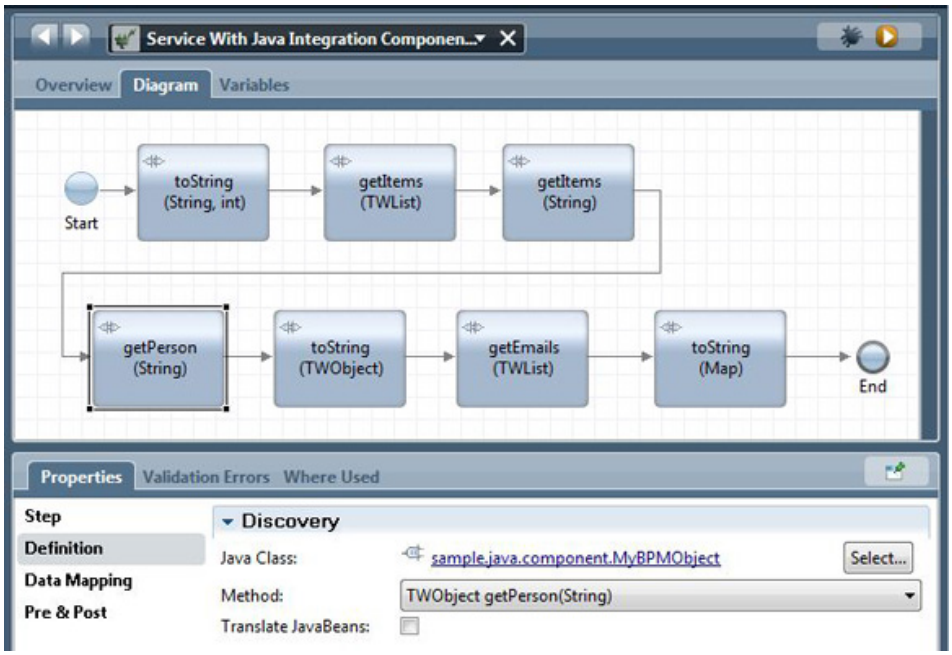


Figure 14. Data mapping for the fourth integration step

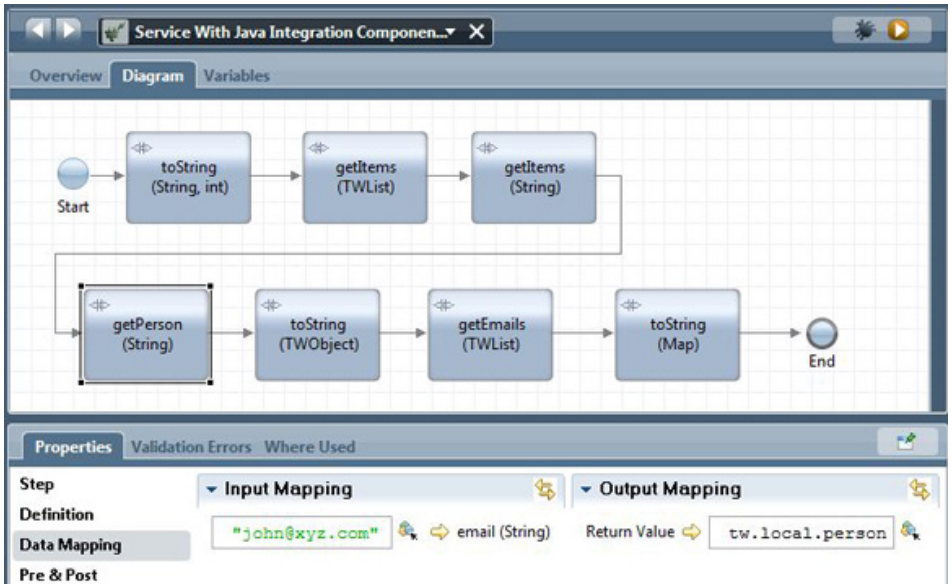
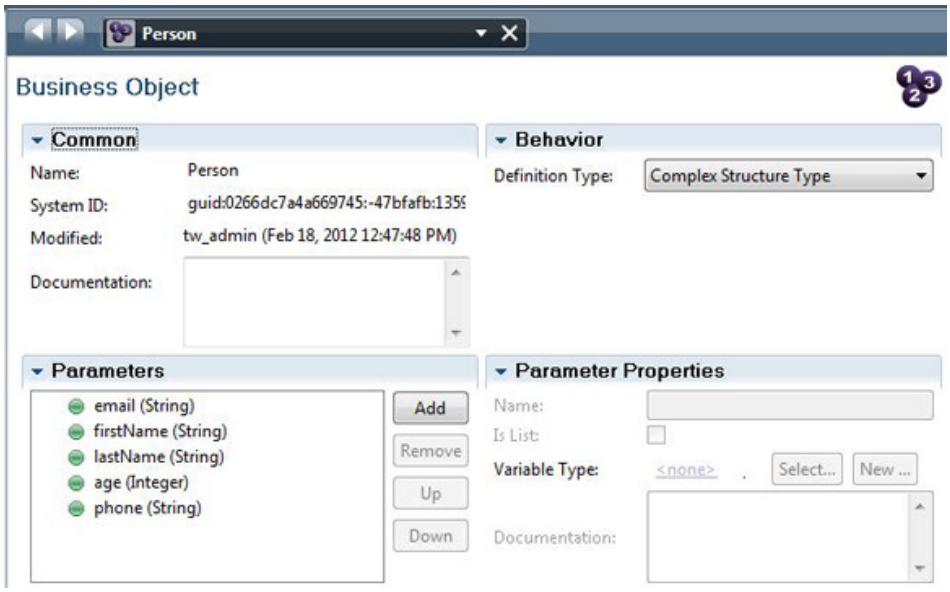


Figure 15. Definition of the Person complex type



Listing 4. Implementation of the getPerson(String) method

```
public TWObject getPerson(String email) throws Exception {
 // Create twobject person
 TWObject twPerson = TWObjectFactory.createObject();
 twPerson.setPropertyValue("email", email);
 twPerson.setPropertyValue("firstName", "John");
 twPerson.setPropertyValue("lastName", "Doe");
 twPerson.setPropertyValue("age", 25);
 twPerson.setPropertyValue("phone", "512-777-9999");
 return twPerson;
}
```

Figure 16 shows the Definition section for the fifth integration step in the sample integration service, while Figure 17 shows its data mapping. The `toString(TWObject)` receives a `Person` structure and returns a string representation of it. The `TWObject.getPropertyValue(String)` instance method is used in this method to obtain the property values on the `Person` data type instance (see Listing 5) that is received as an argument.

Figure 16. Java integration definition for the fifth step

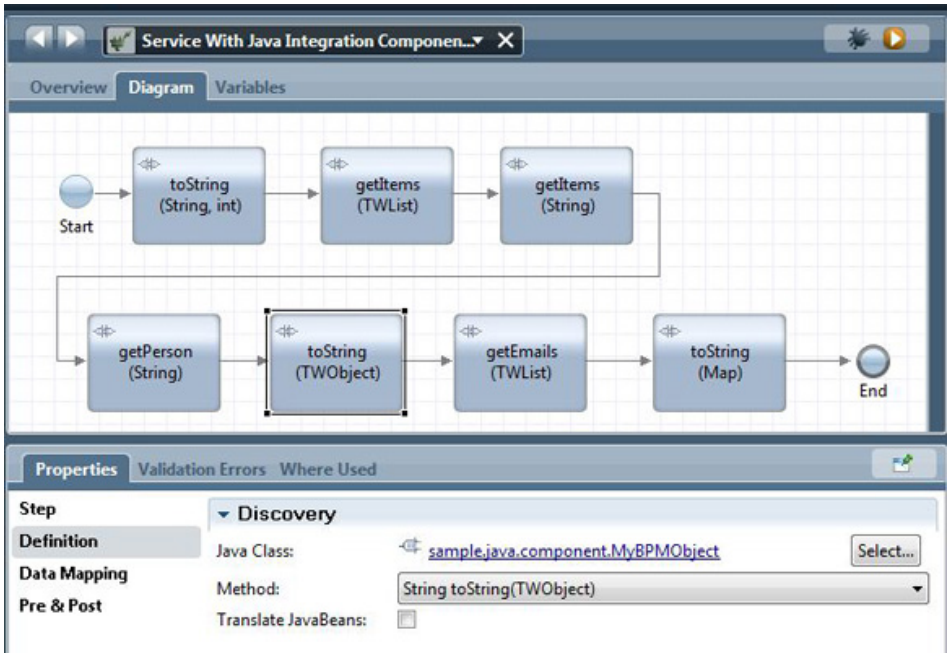
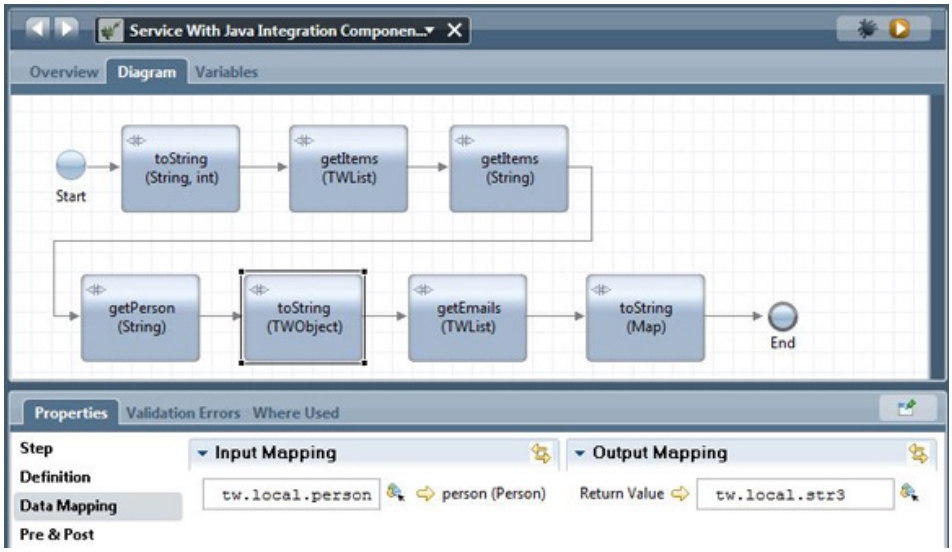


Figure 17. Data mapping for the fifth integration step



Listing 5. Implementation of the toString(TWObject) method

```
public String toString(TWObject twPerson) {
 String email = (String) twPerson.getPropertyValue("email");
 int age = (Integer) twPerson.getPropertyValue("age");
 String firstName = (String) twPerson.getPropertyValue("firstName");
 String lastName = (String) twPerson.getPropertyValue("lastName");
 String phone = (String) twPerson.getPropertyValue("phone");
 StringBuffer buffer = new StringBuffer();
 buffer.append("Email: ");
 buffer.append(email);
 buffer.append(", ");
 buffer.append("Age: ");
 buffer.append(age);
 buffer.append(", ");
 buffer.append("First name: ");
```

```
 buffer.append(firstName);
 buffer.append(", ");
 buffer.append("Last name: ");
 buffer.append(lastName);
 buffer.append(", ");
 buffer.append("Phone: ");
 buffer.append(phone);
 return buffer.toString();
}
```

Figure 18 shows the Definition section for the sixth box in the service diagram, while Figure 19 shows its data mapping. The `getEmails(TWList)` method receives a list of `Person` instances and creates a list of strings that contains the emails of each one of the `Person` instances. Listing 6 shows the implementation for this method.

Figure 18. Java integration definition for the sixth step

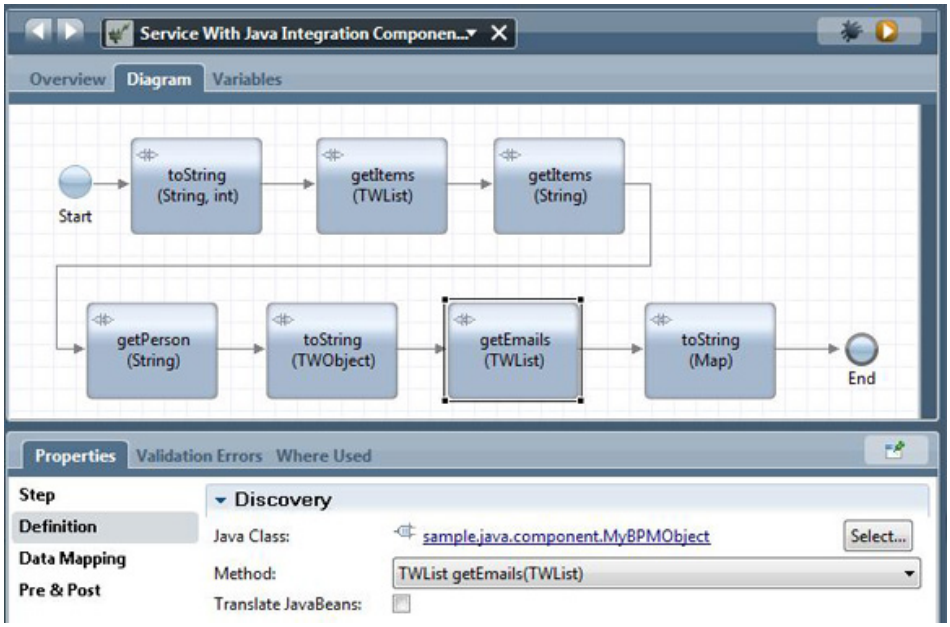
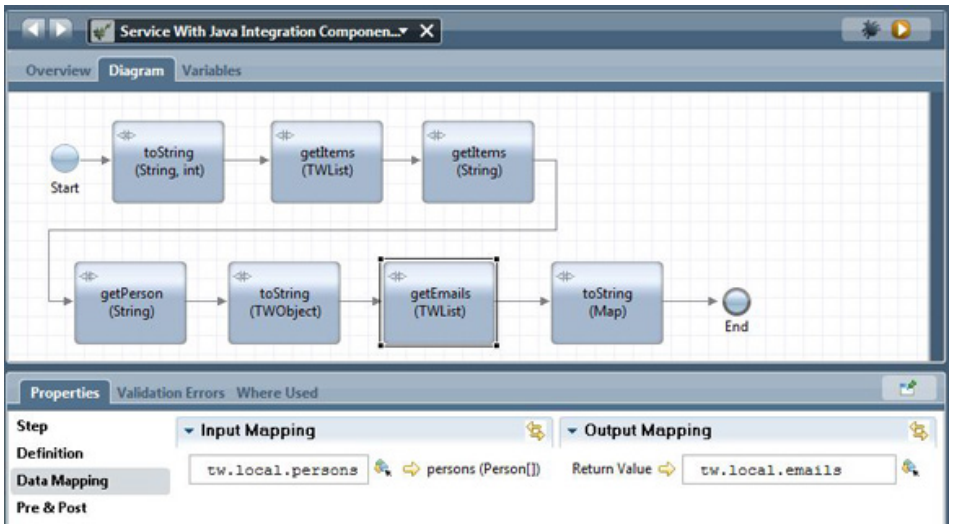


Figure 19. Data mapping for the sixth integration step



Listing 6. Implementation of the getEmails(TWlist) method

```
public TWList getEmails(TWList persons) throws Exception {
 TWList emails = TWObjectFactory.createList();
 int size = persons.getArraySize();
 for (int i = 0; i < size; i++) {
   TWObject person = (TWObject) persons.getArrayData(i);
   String email = (String) person.getPropertyValue("email");
   emails.addArrayData(email);
 }
 return emails;
}
```

Last but not least, Figure 20 shows the Definition section for the seventh and last integration step in the sample service, while Figure 21 shows its data mapping. The `toString(Map)` method receives a `Map` object and returns its string implementation. As seen in Listing 7, the `Map` data type in IBM BPM maps directly to the `java.util.Map` data type in Java.

Figure 20. Java integration definition for the seventh step

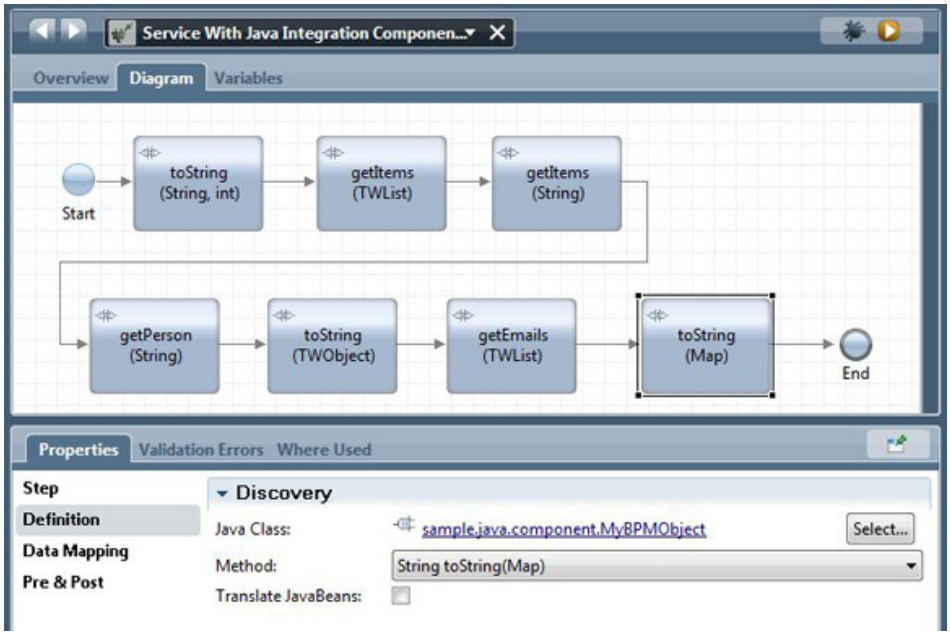
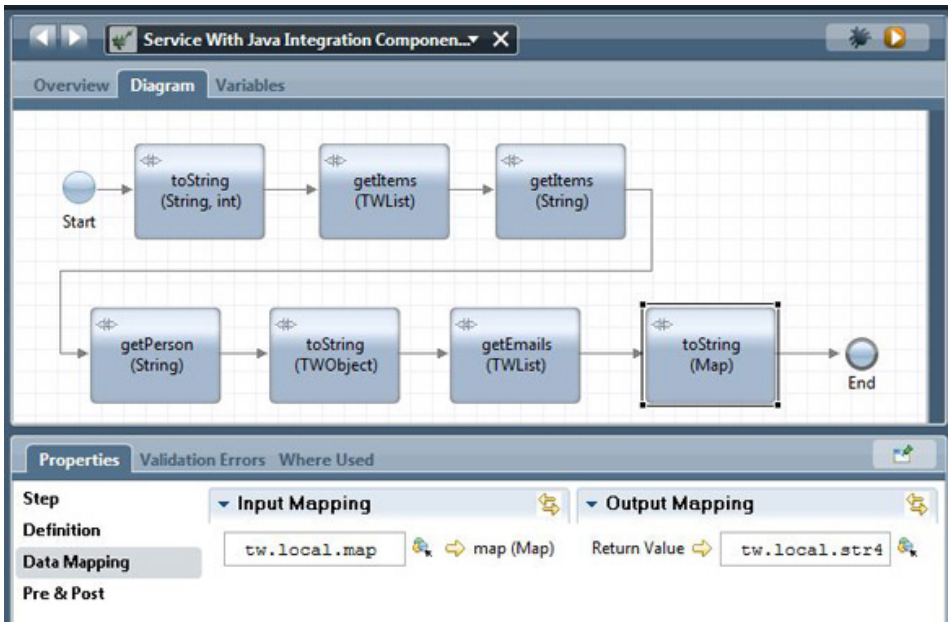


Figure 21. Data mapping for the seventh integration step



Listing 7. Implementation of the toString(Map) method

```
public String toString(@SuppressWarnings("rawtypes") Map map) {
 return map.toString();
}
```

Development of Java integration components

The actual code implementation in Java to support the Java integration components in the sample service that were discussed earlier required a minimum of two Java classes. The first class defines the implementation for the object that will be invoked from the business process application in IBM BPM. You can think of this as the target class. In our sample scenario, this is the `sample.java.component.MyBPMObject` class. Note that when an *instance* method on the target class is invoked, IBM BPM creates an instance of the target class before invoking the method. Hence, a no-argument constructor is needed in the target class definition (see Listing 8). If the method to be invoked on the target class is a *static* method, then there is no need to create an instance of the class.

Listing 8. No-agument constructor of the MyBPMObject class

```
public MyBPMObject() {
 super();
}
```

The second class (though optional) is used to specify the parameter names and types that are shown to a user for each one of the available methods in the object to be invoked from the business process application (for example, `MyBPMObject`). You can think of this second class as a descriptor class. In our case, this second class corresponds to the `sample.java.component.MyBPMObjectBeanInfo` class, which is also contained in the `sample-java-component.jar` file. IBM BPM uses this class to determine what variable names and types to show on the `Data Mapping` section of the `Properties` tab (see Figure 8). The `getMethodDescriptors()`

method shown in Listing 9 shows how the parameter names and types are mapped to each one of the methods defined in the `sample.java.component.MyBPMObject` class. These descriptor classes are known as `BeanInfo` classes and must follow this naming convention:

```
<name of the class Java class to describe>BeanInfo
```

Hence, `MyBPMObjectBeanInfo` is the class name for the descriptor class. To learn more about BeanInfo classes, see Adding a BeanInfo Class for NervousText and BeanInfo classes and introspection.

Listing 9. Implementation of the getMethodDescriptors() method

```
public MethodDescriptor[] getMethodDescriptors() {
 try {
    MethodDescriptor methodDescriptor1 = getMethodDescription(
       "toString", new String[] { "str (String)", "num (Integer)" },
       new Class[] { String.class, int.class });

    MethodDescriptor methodDescriptor2 = getMethodDescription(
       "getItems", new String[] { "items (String[])" },
       new Class[] { TWList.class });

    MethodDescriptor methodDescriptor3 = getMethodDescription(
       "getItems", new String[] { "str (String)" },
       new Class[] { String.class });

    MethodDescriptor methodDescriptor4 = getMethodDescription(
       "getPerson", new String[] { "email (String)" },
       new Class[] { String.class });

    MethodDescriptor methodDescriptor5 = getMethodDescription(
       "toString", new String[] { "person (Person)" },
       new Class[] { TWObject.class });

    MethodDescriptor methodDescriptor6 = getMethodDescription(
       "getEmails", new String[] { "persons (Person[])" },
       new Class[] { TWList.class });

    MethodDescriptor methodDescriptor7 = getMethodDescription(
       "toString", new String[] { "map (Map)" },
       new Class[] { Map.class });

    return new MethodDescriptor[] { methodDescriptor1,
       methodDescriptor2, methodDescriptor3, methodDescriptor4,
       methodDescriptor5, methodDescriptor6, methodDescriptor7 };
 } catch (Exception e) {
     return super.getMethodDescriptors();
 }
}
```

To compile the supporting classes for a Java integration component for IBM BPM, you need to reference in your build *classpath* the `pscInt.jar` file that is shipped with IBM BPM. This JAR file is found in the following folder:

```
<BPM Installation folder>BPM/Lombardi/lib
```

For instance, on a Windows® system, this JAR file is more than likely found in the following folder:

```
C:\IBM\BPM\v7.5\BPM\Lombardi\lib
```

Figure 22 shows the Java project in Rational® Application Developer that contains the supporting code for the Java integration components discussed in this article. Note that instead of Rational Application Developer, you can use Eclipse or your IDE of choice to develop the supporting code for a Java integration component. Figure 22 shows the build path for the Java project. Note that the build path includes the `pscInt.jar` file.

Figure 22. Java project in Rational Application Developer

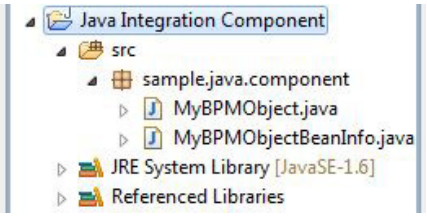
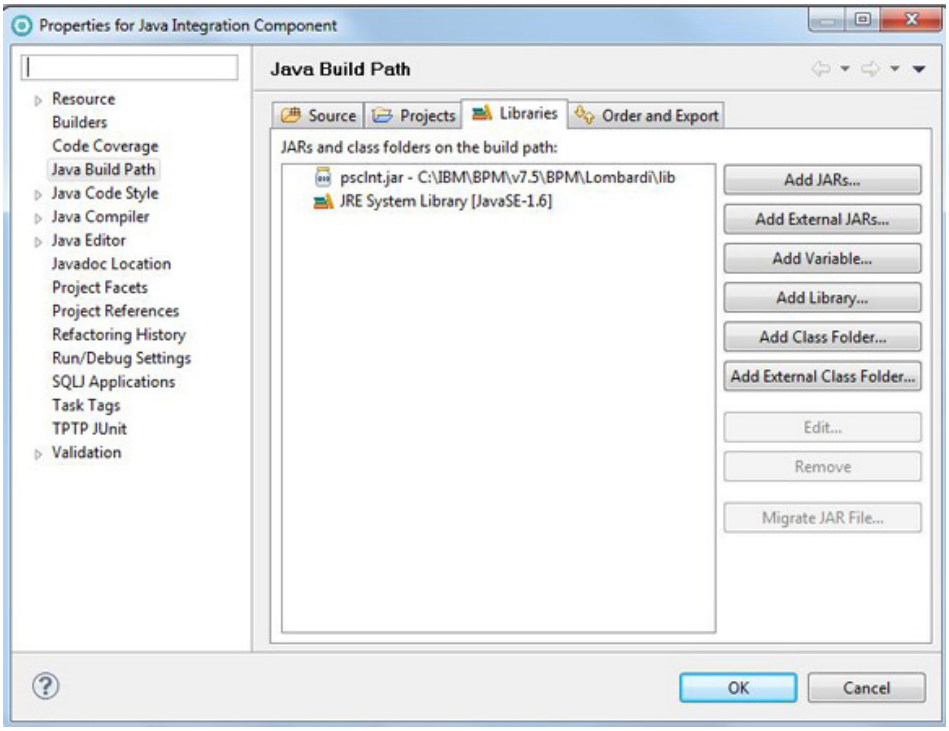


Figure 23. Java build path



Conclusion

This article discussed the necessary building blocks for understanding what it takes to build and implement a Java integration component. You learned how variables defined in an IBM BPM process application (such as integers, strings, maps, lists, and complex data types) are mapped to variables defined in Java. These are passed to a Java method and returned from a Java method to the process application in IBM BPM. The article also discussed the minimum set of classes required to develop the supporting code for a Java integration component for IBM BPM and the required dependencies to successfully compile your code. Using a Java integration component allows you to extend the functionality of your process application without too much effort.

Downloadable resources

Description	Name	Size
Java integration sample files	java_integration_files.zip	475KB

Related topics

- Java Beans Tutorial, Part 4: Adding a BeanInfo Class for NervousText
- Information Center: BeanInfo classes and introspection
- IBM Business Process Manager V7.5.1 Information Center
- IBM Business Process Management zone on developerWorks

© Copyright IBM Corporation 2012
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)