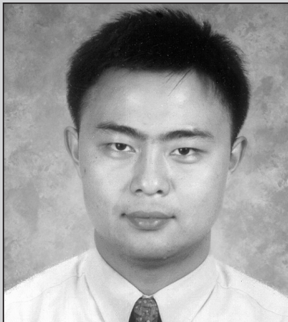
Introducing the SAP Java IDoc Class Library Part 2: An Easier Way to Write Java Programs That Receive and Process IDocs from an SAP System

Robert Chu



Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs, where he focuses on the SAP integration technologies. Robert has been regularly teaching classes in this area at SAP training centers and is the main author of the BIT531 training course and a few other internal workshops.

(complete bio appears on page 100)

Interface programming can be a time-consuming and labor-intensive task, and creating and processing Intermediate Documents (IDocs) for SAP data exchange has been no different — until now. The new SAP Java IDoc Class Library, a free add-on to the SAP Java Connector (JCo), reduces the complexity, time, and expense associated with writing programs to handle IDoc exchange with other systems. It provides a Java API that simplifies navigating, reading, filling, and modifying IDocs, as well as a runtime library of prebuilt functions for creating, sending, and receiving IDocs. With the SAP Java IDoc Class Library, which works with all supported versions of SAP R/3 and any other SAP applications, you no longer need to expend the considerable resources required in the past to develop and maintain complex custom code or license expensive third-party middleware.

This is the second article in a two-part series that discusses the SAP Java IDoc Class Library and how SAP integration architects, integration developers, technical consultants, and Java developers can use it to easily and efficiently exchange IDocs with an SAP system. In the first article, I discussed the basic concepts of the SAP IDoc technology, introduced the SAP Java IDoc Class Library, and showed you how to write efficient and reliable Java programs that compose and send IDocs to an SAP system — i.e., how to write *client* programs. I also demonstrated how to implement robust transaction ID (TID) management in IDoc client programs in order to track IDoc processing. Here in this second installment, I will discuss how to use the SAP Java IDoc Class Library to write efficient and reliable Java programs that receive, track, and process IDocs coming from an SAP system — i.e., how to write IDoc *server* programs. As in the first article, I will provide many working examples to illustrate the concepts under discussion.

SAP IDoc Technology and the SAP Java IDoc Class Library

The SAP IDoc interface is an SAP-proprietary integration technology that is used for asynchronous messaging among SAP systems, and between SAP and non-SAP systems. An *IDoc* (Intermediate Document) is a concrete instance of an *IDoc type*, which defines the data segments and fields of the IDoc. The IDoc acts as a data container, carrying the data of the business message from the sender system to the receiver system.

> ✓ ***Note!***
>
> *For a detailed discussion of the SAP IDoc technology, please refer to the first article in this two-part series.*

An IDoc consists of:

- A single **control record**, which contains the routing and processing information for the IDoc
- **Status records**, which contain information on the IDoc's processing status
- **Data records**, which contain the field data of an IDoc data segment

A data segment contains many data fields and may optionally contain child data segments (i.e., segments one level below the specified segment). All the data segments of an IDoc put together form a hierarchical tree known as a *segment tree*.

IDocs are mostly transmitted using the transactional Remote Function Call (tRFC) protocol. Using this protocol, the sender system calls the receiver system's IDOC_INBOUND_ASYNCHRONOUS or INBOUND_IDOC_PROCESS method, depending on the IDoc version used (version 3 for R/3 4.x systems and version 2 for R/3 3.x systems, respectively), passing the IDoc data in a concatenated flat format as parameters of the tRFC call.

To fully benefit from the discussions in this article, be sure to read the first article of this two-part series (see page 49 of this issue), at least up to the section "Introducing the SAP Java IDoc Class Library" (see the sidebar above for a brief refresher on the key technologies covered there). Here in this article, we turn our attention to IDoc server programming. The remainder of this article will focus on the following aspects of writing IDoc server programs with Java:

- Configuring the SAP system to send IDocs
- Writing an external IDoc listener to receive IDocs from the SAP system
- Traversing the received IDocs and accessing the desired data
- Managing the TID for the IDoc listener
- Starting the IDoc listener

> ✓ ***Note!***
>
> *This article series discusses the most important aspects of using the SAP Java IDoc Class Library APIs; if you are interested in the complete API details, please refer to the respective JavaDoc included with the downloads (go to* ***www.service.sap.com/connectors*** *and select SAP Java Connector → Tools and Services → SAP Java IDoc Class Library).*

Configuring the SAP System to Send IDocs

Before you start to actually write an IDoc server program, you need to understand how the SAP system is configured to send out IDocs.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

The SAP Java IDoc Class Library

In the past, it was very difficult to write programs for sending or receiving IDocs. Retrieving and interpreting the IDoc metadata and using the tRFC protocol correctly are complicated tasks, and prior to the SAP Java IDoc Class Library, there were only a few options available to help you, none of which were easy to use or cost-effective. You could write lots of custom code using one of the existing SAP connectivity toolkits, but some of them have become obsolete, some have no API, and others are simply not optimized to the task, and in the end you would have a lot of custom code to maintain. Another option is to use the SAP Exchange Infrastructure (XI) or a third-party middleware, but the financial and infrastructural investments required in these cases are likely to be far too large to justify for the task at hand.

> ✓ ***Note!***
>
> *For a detailed discussion of the challenges involved in writing IDoc-handling programs without the SAP Java IDoc Class Library, please refer to the first article in this two-part series.*

The SAP Java IDoc Class Library, on the other hand, supplies you with a user-friendly set of predefined general base classes and programming interfaces that enable you to easily write programs that efficiently exchange IDocs with the SAP system without expending too many resources or getting mired in the technical details. What's more, it is available free-of-charge as an add-on to the SAP Java Connector (JCo). To download the SAP Java IDoc Class Library, go to **www.service.sap.com/connectors** and select *SAP Java Connector → Tools and Services → SAP Java IDoc Class Library*. Note that the download has two parts — the SAP Java Base IDoc Class Library (which contains the classes and interfaces) and the SAP Java Connector IDoc Class Library (which implements them) — and requires JCo 2.1 or later and JDK (Java Development Kit) or JRE (Java Runtime Environment) 1.3 or later. For installation details, see the documentation included with the downloads.

From the SAP IDoc transmission perspective, the external server program is a logical system, so configuring the SAP system to exchange IDocs with the external server program is no different from configuring SAP to exchange IDocs with other systems in an ALE (Application Link Enabling) scenario.[1]

The following ALE configuration tasks must be completed in order to send IDocs from the SAP system to the external server program:

- An RFC (Remote Function Call) destination that points to the external server program must be defined inside the SAP system (using transaction *SM59*).
- The external server program must be defined as a logical system. In order to generate partner profiles[2] automatically, the logical system name needs to be the same name defined for the RFC destination name.
- An ALE distribution model view must be created.[3] The desired IDoc message type, sender system, and receiver system need to be defined in the view.
- A partner profile must be generated for the distribution model view and the partner logical system.

[1] The details of setting up ALE distribution are beyond the scope of this article. For more information, please refer to the corresponding SAP documentation and related training courses. For your convenience, I provide a high-level overview here.

[2] A partner profile defines the processing parameters of a partner (e.g., a logical system).

[3] The distribution model defines the ALE message flow between logical systems. It consists of separate views that define the sender/receiver systems and message types of your distributed business processes.

Configuring the SAP System to Create IDocs

The way in which the SAP system is configured to create an IDoc depends on the type of IDoc and the application module involved:

- For master data IDocs (e.g., MATMAS), a change pointer can be activated to record all changes to the master data. Later, the change pointers can be processed to generate the IDocs.
- There are also SAP transactions available to manually generate master data IDocs. These transactions are available under *Tools → ALE → Master Data Distribution*. They are useful for initial data synchronization and testing.
- For MM (Materials Management) and SD (Sales and Distribution) IDocs (e.g., ORDERS), the message control (a.k.a. output determination) mechanism is used to generate the IDoc automatically once the corresponding application document (e.g., a sales order or purchase order) is posted.
- For some other applications, such as FI (Financial Accounting), special transactions or reports can be executed to generate the IDoc.

> ✓ ***Note!***
>
> *For further details on using change pointers and message control, see the article "Real-Time, Outbound Interfaces to Non-R/3 Systems Made Simple with Change Pointers, Message Control, and Workflow" in the Premiere Issue of this publication.*

This ALE configuration provides information to the ALE layer on how to distribute an IDoc once it is created (see the sidebar above for more on creating IDocs).

Once the SAP system is set up to create and send IDocs, you need to write your Java server program to act as an "external tRFC server" for the RFC destination defined during the ALE configuration so that it can listen for and receive the IDocs sent from the SAP system. We will look at this task next.

Writing an External IDoc Listener to Receive IDocs from the SAP System

It used to be a difficult task to write an external IDoc server program. You had to write a transactional RFC (tRFC) server that would handle the method calls IDOC_INBOUND_ASYNCHRONOUS (used for version 3 IDocs in R/3 4.x systems) and INBOUND_IDOC_PROCESS (used for version 2 IDocs in R/3 3.x systems) directly, retrieve the IDoc metadata from the SAP system, and then use that metadata to parse the IDoc data sent as tRFC call parameters and reconstruct the logical segment tree. With the help of the SAP Java IDoc Class Library,[4] writing an IDoc server program is now a much easier task. First, let's look at **Figure 1**, which shows a simple IDoc server program capable of listening for and receiving IDocs from the SAP system.

In the example program, we define a static inner class called *MyIDocListener*,[5] which extends the IDoc

[4] For simplicity, I'll refer to the SAP Java IDoc Class Library as the "IDoc library" for the remainder of this article.

[5] A normal Java class can be used as well. I use a static inner class here to simplify the organization of the code. For ease of reference, I call the *MyIDocListener* subclass of the *JCoIDoc.Server* class the "IDoc listener."

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 1 ***A Simple IDoc Server Program That Listens for and Receives IDocs***

```
 1 package com.spj.idocserver;
 2
 3 import com.sap.mw.jco.*;
 4 import com.sap.mw.idoc.*;
 5 import com.sap.mw.idoc.jco.*;
 6 import java.io.*;
 7 import java.util.*;
 8
 9 public class IDocServer1 {
10   public static class MyIDocListener extends JCoIDoc.Server {
11     // Overridden constructors of JCoIDoc.Server.  Choose the
          constructors that you would like to make public.
12     public MyIDocListener(String gwhost, String gwserv, String
         progid, IRepository jcoRepository, IDoc.Repository
         idocRepository) {
13       super(gwhost, gwserv, progid, jcoRepository, idocRepository);
14     }
15     public MyIDocListener(Properties properties, IRepository
         jcoRepository, IDoc.Repository idocRepository) {
16       super(properties, jcoRepository, idocRepository);
17     }
18
19     // Overridden method of JCoIDoc.Server.  Function requests that
          contain IDocs will be handled here.
20     protected void handleRequest(IDoc.DocumentList documentList) {
21       System.out.println("Number of incoming IDocs received: " +
           documentList.getNumDocuments());
22       IDoc.DocumentIterator iterator =
           documentList.iterator();            // get the iterator
23       IDoc.Document doc = null;
24       while (iterator.hasNext()) {          // iterate through all
                                                  the IDocs in the list
25         doc = iterator.nextDocument();      // get the next IDoc in
                                                  the list
26
27         // access the IDoc control data
28         String idocNumber = doc.getIDocNumber();
29         String msgType = doc.getMessageType();
30         String idocType = doc.getIDocTypeExtension();
31         if (idocType.length() == 0)
32           idocType = doc.getIDocType();
33         String sender = doc.getSenderPartnerNumber();
34
35         System.out.println("IDoc number: " + idocNumber + ", message
             type: " + msgType + ", IDoc type: " + idocType
36             + ", sender: " + sender);
37
38         if (msgType.equals("MATMAS")) {
```

(continued on next page)

Figure 1 *(continued)*

```
39            System.out.println("Further processing IDoc of message
                type MATMAS...");
40            processMATMAS(doc);
41          } else {
42            System.out.println("IDoc of message type '" +
                doc.getMessageType() + "' won't be further processed.");
43          }
44        }
45      }
46
47      public static void processMATMAS(IDoc.Document idoc) {
48        // process IDocs of message type MATMAS here ...
49      }
50
51      // tRFC server TID management functions omitted ...
52    }
53
54    // code to instantiate MyIDocListener and start the listening
         omitted ...
55  }
```

library's *JCoIDoc.Server* interface (line 10). The *JCoIDoc.Server* class defines several "protected" constructors — for the IDoc listener, we need to expose at least one (and optionally more) of them as "public." In the following code sample, we expose two constructors, which are shown in bold:

```
public MyIDocListener(String
  gwhost, String gwserv,
  String progid, IRepository
  jcoRepository, IDoc.Repository
  idocRepository)

public MyIDocListener(Properties
  properties, IRepository
  jcoRepository, IDoc.Repository
  idocRepository)
```

The arguments of the constructors must provide the following information to the IDoc listener:

- **The SAP system's gateway host machine:** This is typically the SAP application server's host name, because by default there is a gateway process running on each application server. It can also be a standalone gateway's host name, if a standalone gateway is used to offload RFC traffic from the application servers.

- **Gateway service name:** This is usually provided in the form *sapgw<nn>*, where *<nn>* is the two-digit SAP application server instance number or the standalone gateway's instance number, if it is used.

- **Program ID used for external RFC server registration:** This is a case-sensitive arbitrary string (without spaces) used by the external RFC server to register itself to the SAP gateway; during startup, the external RFC server establishes a connection to the SAP gateway and identifies itself using the program ID. In the SAP system, the RFC destination needs to be defined accordingly in transaction *SM59*, using "Registration" mode and the exact same program ID.

©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

- **A *JCO.Repository* instance:** The *JCO.Repository* instance is used by the IDoc library to retrieve IDoc metadata from the RFC-enabled function modules (RFMs) that enable programmatic retrieval of IDoc metadata (the data segments and fields of the IDoc).

- **An *IDoc.Repository* instance:** The *IDoc.Repository* instance caches the retrieved IDoc metadata, which is used by the IDoc listener to interpret the received IDocs.

The gateway host name, gateway service name, and program ID can be provided as constructor arguments directly, as in the first constructor from the previous code sample (this particular constructor's arguments are shown in bold):

```
public MyIDocListener(String
  gwhost, String gwserv,
  String progid, IRepository
  jcoRepository, IDoc.Repository
  idocRepository)
```

The constructor arguments can also be provided as entries in a Java *Properties* object, as in the second constructor from the previous code sample (again, the arguments are shown in bold):

```
public MyIDocListener(Properties
  properties, IRepository
  jcoRepository, IDoc.Repository
  idocRepository)
```

Next, we need to provide implementation code to handle the IDoc transmission tRFC calls. To do that, we override the *handleRequest()* method defined in *JCoIDoc.Server* (line 20 in Figure 1):

```
protected void
  handleRequest(IDoc.DocumentList
  documentList)
```

When the SAP system initiates IDoc transmission to the external server program — by calling function module IDOC_INBOUND_ASYNCHRONOUS for version 3 IDocs (used for R/3 4.x systems) or INBOUND_IDOC_PROCESS for version 2 IDocs (used for R/3 3.x systems) against the external server program using the tRFC protocol — the IDoc library runtime automatically detects the transmission and marshals the tRFC call parameter data into an *IDoc.DocumentList* object, which serves as a container for storing and managing the received documents. The IDoc library runtime then passes this list object to the *handleRequest()* method implementation defined in the IDoc listener class.

So, essentially, the only things you need to do to write an IDoc listener program are to extend the *JCoIDoc.Server* class provided by the IDoc library, expose at least one constructor, and provide the specific implementation code to the *handleRequest()* method. The IDoc library takes care of the rest, and the IDoc listener can now automatically receive an instance of *IDoc.DocumentList* (i.e., a "document list") containing the IDocs sent by the SAP system. Isn't that easy? (We will examine how to start the IDoc listener threads later, in the section "Starting the IDoc Listener.")

While we are now ready to receive IDocs, we still need to write the code required to process them — i.e., traverse them and access the desired data segments and fields — once they are received. We will look at these tasks next.

Traversing the Received IDocs and Accessing the Desired Data

In the code of the *handleRequest()* method, or the methods invoked from within *handleRequest()*, you will typically need to perform the following tasks in order to traverse the IDocs in the received list and access the desired data:

- Access the individual IDocs in the received IDoc list

- Access an IDoc's control record fields

- Access an IDoc's data segments and fields

Access the Individual IDocs in the Received IDoc List

Since the SAP system may send either a single IDoc or multiple IDocs in an IDoc packet, the *handleRequest()* method is designed to have as its argument an *IDoc.DocumentList* instance, which can contain either a single IDoc or a packet of IDocs.

The *IDoc.DocumentList* interface contains many methods that allow you to obtain the number of IDocs contained in the list of received documents and access individual IDocs in the list:

- *boolean isEmpty()* — Checks if the list is empty (i.e., does not contain any documents).
- *int getNumDocuments(), int size()* — Returns the number of documents in the list.
- *IDoc.Document first()* — Returns the first document in the list.
- *IDoc.Document last()* — Returns the last document in the list.
- *IDoc.Document get(int index)* — Returns the document at a specified index position in the list (the index count starts from 0).
- *IDoc.Document getNext(IDoc.Document document)* — Returns the document that follows a specified document in the list.
- *IDoc.Document getPrevious(IDoc.Document document)* — Returns the document that precedes a specified document in the list.
- *int indexOf(IDoc.Document document)* — Searches for the first occurrence of a given document, testing for a match using the *equals()* method.
- *int indexOf(IDoc.Document document, int startIndex)* — Searches for the first occurrence of a given document, beginning the search at a specified index position and testing for a match using the *equals()* method.
- *int lastIndexOf(IDoc.Document document)* — Searches for the last occurrence of a given document, testing for a match using the *equals()* method.

In the example program shown in Figure 1, the *getNumDocuments()* method is used to obtain the number of documents in the list (line 21).

While these *IDoc.DocumentList* methods are useful when searching for specific documents in the list, the easiest way to access all the individual IDocs in the list is to use the *IDoc.DocumentIterator* class, which iterates over the document list sequentially. *IDoc.DocumentIterator* can be instantiated using the following *IDoc.DocumentList* method:

- *IDoc.DocumentIterator iterator()* — Returns an iterator that runs through all the documents in the list in sequential order.

The *IDoc.DocumentIterator* class implements the *java.util.Iterator* interface and provides the following easy-to-use methods:

- *boolean hasNext()* — Checks if the iteration has at least one more document.
- *IDoc.Document nextDocument()* — Returns the next document in the iteration process.

The following code pattern illustrates how to iterate through all the IDocs in the document list using the iterator:

```
IDoc.DocumentIterator iterator =
  documentList.iterator();
IDoc.Document doc = null;
while (iterator.hasNext()) {
  doc = iterator.nextDocument();
  // process the IDoc ...
}
```

In Figure 1, this iterator pattern is used in lines 22-25.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Access an IDoc's Control Record Fields

With an *IDoc.Document* instance, to access the IDoc's control record field values, you can use the getter methods (described in detail in the previous article) as shown in lines 27-33 in Figure 1.

Access an IDoc's Data Segments and Fields

To access an IDoc's data segments, you first need to get the "virtual" root segment[6] of the IDoc by using the *getRootSegment()* method of the *IDoc.Document* interface. With the root segment, you can then use the many methods of the *IDoc.Segment* interface to navigate and traverse through the fields of the specified data segment tree and access the values of the individual data fields. The code segment in **Figure 2** illustrates accessing the data segments and fields of a *MATMAS* IDoc.

The following is a categorized list of the *IDoc.Segment* interface's methods for traversing a data segment tree, organized into related groupings for ease of reference.

- **Methods for getting information about the currently specified data segment:**

[6] The root segment is not an actual data segment — it is a general entry point used to simplify navigation through the IDoc segment tree. Each IDoc instance has one virtual root segment that is the parent of the "real" top-level data segments. This concept was discussed in detail in the first article of this two-part series.

Figure 2 *Accessing IDoc Segments and Fields*

```
 1 // process IDocs of message type MATMAS
 2 public static void processMATMAS(IDoc.Document idoc) {
 3   IDoc.Segment root = idoc.getRootSegment();     // get the virtual
                                                      root segment
 4   IDoc.Segment segE1MARAM = root.getFirstChild("E1MARAM");
 5   System.out.println("\tMaterial number: " +
       segE1MARAM.getString("MATNR"));
 6
 7   // locate a segment of name "E1MAKTM" whose "SPRAS" field has
        value "E"
 8   IDoc.Segment segE1MAKTM = segE1MARAM.getFirstChild("E1MAKTM",
       "SPRAS", "E");
 9   System.out.println("\tMaterial description in English: " +
       segE1MAKTM.getString("MAKTX"));
10
11   // use segment iterator
12   IDoc.SegmentIterator iterator =
       segE1MARAM.getChildrenIterator("E1MARCM");
13   StringBuffer sbPlants = new StringBuffer();
14   while (iterator.hasNext()) {
15     IDoc.Segment segE1MARCM = iterator.nextSegment();
16     String plant = segE1MARCM.getString("WERKS");
17     sbPlants.append(plant + " ");
18   }
19   System.out.println("\tMaterial maintained in plants: " +
       sbPlants.toString());
20 }
```

- *String getType()* — Gets the current segment's type.

- *String getDefinition()* — Gets the current segment's definition name.

- *String getDescription()* — Gets the current segment's description.

- *int getHierarchyLevel()* — Gets the current segment's hierarchy level.

- *boolean isRoot()* — Determines whether the current segment is the root segment of the segment tree.

- *boolean isParent()* — Determines whether the current segment is a parent of any child segments in the segment tree.

- *boolean isLeaf()* — Determines whether the current segment is a leaf segment (i.e., a segment that has no segments below it).

- *IDoc.Segment getParent()* — Gets the current segment's parent segment.

- *IDoc.Document getDocument()* — Gets the IDoc containing the current segment.

- **Methods for retrieving the next or previous sibling segments of the currently specified segment (i.e., a segment that has the same parent as the specified segment):**

  - *IDoc.Segment getNextSibling()* — Gets the current segment's next sibling segment.

  - *IDoc.Segment getPreviousSibling()* — Gets the current segment's previous sibling segment.

  - *IDoc.Segment getNextSibling(String segmentType)* — Gets the current segment's next sibling segment of a certain type.

  - *IDoc.Segment getPreviousSibling(String segmentType)* — Gets the current segment's previous sibling segment of a certain type.

  - *IDoc.Segment getNextSibling(String segmentType, int fieldIndex / String*[7] *fieldName, String fieldValue)* — Gets the current segment's next sibling segment of a certain type and containing a certain field value.

  - *IDoc.Segment getPreviousSibling(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the current segment's previous sibling segment of a certain type and containing a certain field value.

- **Methods for retrieving the segment next to (determined by the "depth-first" traversal order) or previous to (determined by the "reverse-depth-first" traversal order) the currently specified segment:**

  - *IDoc.Segment getNext()* — Gets the segment next to the current segment.

  - *IDoc.Segment getPrevious()* — Gets the segment previous to the current segment.

  - *IDoc.Segment getNext(String segmentType)* — Gets the segment next to the current segment that is of a certain type.

  - *IDoc.Segment getPrevious(String segmentType)* — Gets the segment previous to the current segment that is of a certain type.

  - *IDoc.Segment getNext(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the segment next to the current segment that is of a certain type and has a certain field value.

[7] To save some space, I use the *int / String* notation to indicate that the argument can be either an integer or a string.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

- *IDoc.Segment getPrevious(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the segment previous to the current segment that is of a certain type and has a certain field value.

- **Methods for determining the currently specified segment's number of direct child segments (dependent segments one level below the specified segment) or the number of descendant segments (subtree segments of the specified segment's child segments):**

  - *int getNumChildren()* — Gets the current segment's number of direct child segments.

  - *int getNumDescendants()* — Gets the current segment's number of descendant segments.

  - *int getNumChildren(String segmentType)* — Gets the current segment's number of direct child segments of a certain type.

  - *int getNumDescendants(String segmentType)* — Gets the current segment's number of descendant segments of a certain type.

- **Methods for retrieving a specific child segment or retrieving an array of child segments of the currently specified segment:**

  - *Doc.Segment getChild(int index)* — Gets a specific child segment of the current segment by index value.

  - *IDoc.Segment[] getChildren()* — Gets an array containing all of the current segment's child segments.

  - *IDoc.Segment[] getChildren(String segmentType)* — Gets an array containing all of the current segment's child segments of a certain type.

- **Methods for retrieving the first or last direct child segment of the currently specified segment:**

  - *IDoc.Segment getFirstChild()* — Gets the current segment's first direct child segment (see line 4 in Figure 2).

  - *IDoc.Segment getLastChild()* — Gets the current segment's last direct child segment.

  - *IDoc.Segment getFirstChild(String segmentType)* — Gets the current segment's first direct child segment of a certain type.

  - *IDoc.Segment getLastChild(String segmentType)* — Gets the current segment's last direct child segment of a certain type.

  - *IDoc.Segment getFirstChild(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the current segment's first direct child segment of a certain type and containing a certain field value (see line 8 in Figure 2).

  - *IDoc.Segment getLastChild(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the current segment's last direct child segment of a certain type and containing a certain field value.

- **Methods for retrieving the first descendant segment (determined by the "breadth-first" traversal order) of the currently specified segment:**

  - *IDoc.Segment getFirstDescendant(String segmentType)* — Gets the current segment's first descendant segment of a certain type.

  - *IDoc.Segment getFirstDescendant(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the current segment's first descendant segment

of a certain type and containing a certain field value.

- **Methods for retrieving the next descendant segment (determined by the "depth-first" traversal order) of the currently specified segment after calling *getFirstDescendant()*:**

  - *IDoc.Segment getNextDescendant(String segmentType)* — Gets the current segment's next descendant segment of a certain type.

  - *IDoc.Segment getNextDescendant(String segmentType, int fieldIndex / String fieldName, String fieldValue)* — Gets the current segment's next descendant segment of a certain type and containing a certain field value.

- **Methods for getting the segment iterator for the direct child segments or the segment iterator for the descendant segments of the currently specified segment:**

  - *IDoc.SegmentIterator getChildrenIterator()* — Gets the segment iterator for the current segment's direct child segments (see line 15 in Figure 2).

  - *IDoc.SegmentIterator getChildrenIterator(String segmentType)* — Gets the segment iterator for the current segment's direct child segments of a certain type.

  - *IDoc.SegmentIterator getDescendantsIterator()* — Gets the segment iterator for the current segment's descendant segments (in "depth-first" traversal order).

With these methods, you can easily traverse the segment tree and locate the segments of interest. The following are some common segment tree navigation patterns:

- **Accessing a child segment directly:** If you want to access a child segment of an existing segment, and you know the child segment occurs only once (or you are just interested in the first occurrence of a child segment), you can access it directly (see line 4 in Figure 2):

```
IDoc.Segment segE1MARAM =
  root.getFirstChild("E1MARAM");
```

- **Locating a child segment with a certain field value:** If you want to access an existing segment's child segment that has a certain field value,[8] you can use another overloaded version of the *getFirstChild()* method (see line 8 in Figure 2):

```
IDoc.Segment segE1MAKTM =
  segE1MARAM.getFirstChild
  ("E1MAKTM", "SPRAS", "E");
```

- **Looping over multiple occurrences of a child segment:** If you want to access multiple occurrences of a child segment of an existing segment, you can use the children segment iterator (see lines 12-18 in Figure 2):

```
IDoc.SegmentIterator iterator =
  segE1MARAM.getChildrenIterator
    ("E1MARCM");
  ...
while (iterator.hasNext()) {
  IDoc.Segment segE1MARCM =
    iterator.nextSegment();
  ...
}
```

Once the segment of interest is located, you can then access the segment's field values using one of the following getter methods of the *IDoc.Segment* interface, specifying the field's index value or name as an argument:

- *java.math.BigDecimal getBigDecimal(int index / String name)*

[8] Many IDoc segments have the so-called "qualifier" field, where different values indicate different types of information contained in the segment. Often you will need to look for a segment with a certain value in the qualifier field.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

- *java.math.BigInteger getBigInteger(int index / String name)*
- *byte[] getByteArray(int index / String name)*
- *char getChar(int index / String name)*
- *java.util.Date getDate(int index / String name)*
- *double getDouble(int index / String name)*
- *int getInt(int index / String name)*
- *long getLong(int index / String name)*
- *short getShort(int index / String name)*
- *String getString(int index / String name)*
- *java.util.Date getTime(int index / String name)*
- *java.lang.Object getValue(int index / String name)*

Among these getter methods, *getString()* is used most often because the majority of the IDoc fields have a character string type. Other getter methods are provided for convenience.

Managing the TID for the IDoc Listener

If a transmission fails, the SAP system will automatically resend the IDoc using the exact same TID; therefore, it is crucial for the external IDoc listener to handle the TID correctly in order to avoid unnecessary, and potentially harmful, duplicate processing of the IDoc.

In order to track the incoming TID's state and be able to survive a program crash or system restart, a persistent storage mechanism, such as a relational database table, must be used.

In the following sections, we will walk through an example of a robust IDoc listener TID management implementation. You will learn how to:

- Define a database table (*TidIn*) to store the state of the incoming TID
- Write a Java helper class to enable the IDoc server program to access the *TidIn* table
- Add code to the IDoc server program to implement the TID management functions in the IDoc listener

Define a Database Table (TidIn) to Store the State of the Incoming TID

We can define a *TidIn* table with the columns shown in **Figure 3**.

Write a Java Helper Class to Enable the IDoc Server Program to Access the TidIn Table

To facilitate access to the *TidIn* table by the TID

Figure 3 *TidIn Table Definition*

Column Name	Description	Data Type, Length	Key Column?
TID	Transaction ID (TID)	Character, 24	Yes
STATE	State of the incoming TID	Integer	—

Figure 4 ***The TidInDAO.java Helper Class***

```
 1 package com.spj.idocserver;
 2
 3 import java.sql.*;
 4
 5 // DAO class for the TidIn table
 6 public class TidInDAO {
 7   public static final int IDOC_IN_NEW = 21;
 8   public static final int IDOC_IN_PROCESSING = 22;
 9   public static final int IDOC_IN_ROLLBACKED = 23;
10   public static final int IDOC_IN_COMMITTED = 24;
11   public static final int IDOC_IN_CONFIRMED = 25;
12   public static final int IDOC_IN_NOTEXIST = 29;
13
14   private Connection sqlConn = null;
15   private PreparedStatement selectTidStmt = null;
16   private PreparedStatement writeTidStmt = null;
17   private PreparedStatement updateTidStmt = null;
18
19   public TidInDAO() {
20     try {
21       Class.forName("com.microsoft.jdbc.sqlserver.SQLServerDriver");
22       sqlConn = DriverManager.getConnection(
          "jdbc:microsoft:sqlserver://localhost:1433;DatabaseName=TidStates",
          "sa", "admin");
23       writeTidStmt = sqlConn.prepareStatement(
           "INSERT INTO tidin (tid, state) VALUES (?, ?)");
24       selectTidStmt = sqlConn.prepareStatement(
           "SELECT state FROM tidin WHERE tid = ?");
25       updateTidStmt = sqlConn.prepareStatement(
           "UPDATE tidin SET state = ? WHERE tid = ?");
26     } catch (ClassNotFoundException cnfex) {
27       cnfex.printStackTrace();
28       System.exit(1);
29     } catch (SQLException sqlex) {
30       sqlex.printStackTrace();
31       System.exit(1);
32     }
33   }
34
35   public int writeNewTid(String tid) throws SQLException {
36     int rowsUpdated = -1;
37
38     writeTidStmt.setString(1, tid);
39     // the state is NEW
40     writeTidStmt.setInt(2, IDOC_IN_NEW);
```

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 4 *(continued)*

```
41     rowsUpdated = writeTidStmt.executeUpdate();
42
43     return rowsUpdated;
44   }
45
46   public int getState(String tid) throws SQLException {
47     int idocState = IDOC_IN_NOTEXIST;
48     selectTidStmt.setString(1, tid);
49     ResultSet rs = selectTidStmt.executeQuery();
50     if (rs.next()) {
51       idocState = rs.getInt("state");
52     }
53     return idocState;
54   }
55
56   public int updateState(String tid, int state) throws SQLException {
57     updateTidStmt.setInt(1, state);
58     updateTidStmt.setString(2, tid);
59     int rowsUpdated = updateTidStmt.executeUpdate();
60     return rowsUpdated;
61   }
62 }
```

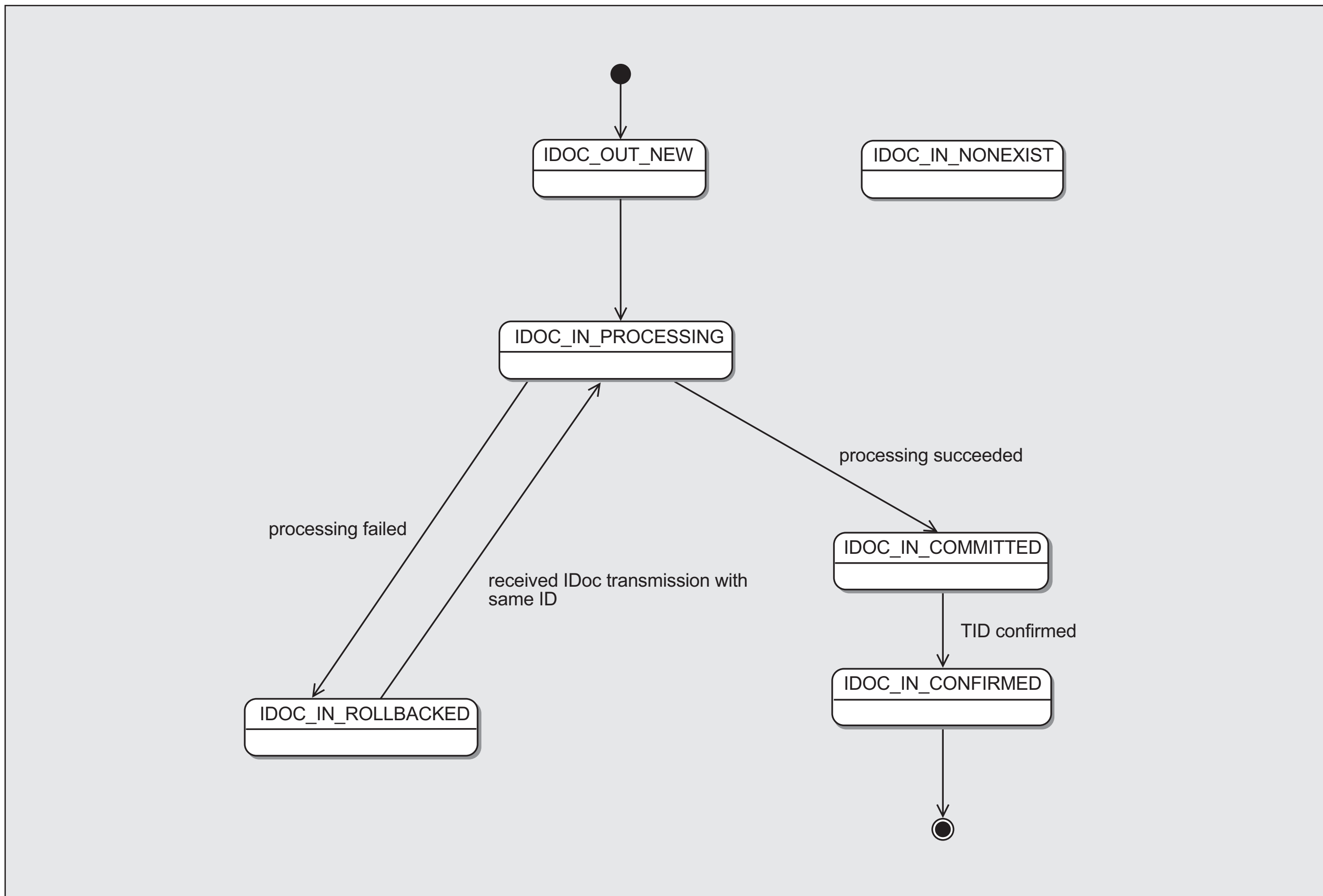
management functions in the IDoc server program, we create a Java helper class called *TidInDAO*, as shown in **Figure 4**.

In the beginning of the *TidInDAO* class (lines 7-12), we define several constants for the state of the incoming TID (see **Figure 5**).

Figure 5 ***Constants for Incoming IDoc TID State***

Constant	Meaning
IDOC_IN_NEW	The incoming IDoc (from the SAP system) has just been received.
IDOC_IN_PROCESSING	The incoming IDoc is currently being processed.
IDOC_IN_ROLLBACKED	The processing of the IDoc failed; all local changes rolled back.
IDOC_IN_COMMITTED	The processing of the IDoc succeeded; all local changes committed.
IDOC_IN_CONFIRMED	The SAP system requested confirmation of the TID; TID confirmed.
IDOC_IN_NOTEXIST	The queried TID does not exist in the TidIn table.

Figure 6 **UML State Transition Diagram of an Incoming TID**



The diagram in **Figure 6** illustrates the state transitions of a TID as it is received and processed by the IDoc server program.

In the *TidInDAO* class outlined in Figure 4, we define the following methods to create, read, and update the TID state records:

- *public int writeNewTid(String tid) throws SQLException* — This method (shown in lines 35-44) writes the TID to the *TidIn* table with state *IDOC_IN_NEW*.
- *public int getState(String tid) throws SQLException* — This method (shown in lines 46-54) retrieves the state of the given TID and returns *IDOC_IN_NONEXIST* if no matching record is found in the *TidIn* table.
- *public int updateState(String tid, int state) throws SQLException* — This method (shown in lines 56-61) updates the state associated with the given TID to the state value given.

✔ ***Note!***

The implementation of these methods uses the prepared JDBC (Java Database Connectivity) statements initialized in the TidInDAO() constructor to improve the performance of repetitive database access. The sample code in Figure 4 uses the Microsoft SQL Server JDBC driver; you may need to change the name of the JDBC driver class as well as the JDBC connection string according to your own system settings.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Add Code to the IDoc Server Program to Implement the TID Management Functions in the IDoc Listener

As I explained in the first article of this two-part series, when the tRFC server receives a tRFC call, it must first check the received TID against its own log of processed TIDs. If the TID is new, the server logs it and processes the call. If the TID has been previously processed, the server instead skips the processing.

With the *TidInDAO* helper class, we can now implement in our IDoc server program the TID management methods that make this TID handling possible, as shown in **Figure 7**. The *JCoIDoc.Server* class defines four TID management methods to which the IDoc listener class must provide implementation code:

- *protected boolean onCheckTID(String tid)* — This TID management method (shown in lines 15-46) is the first method called when the IDoc library runtime receives an incoming IDoc transmission tRFC call. Use this method to check the incoming TID against the TID table to see whether it's a new TID or if it has already been processed:
  - If it's a new TID, write it to the TID table and return *true* for the method (lines 22-32). The *handleRequest()* method will be subsequently called by the IDoc library runtime.
  - If the TID has already been successfully processed, return *false* (lines 33-38). The *handleRequest()* method will be skipped, and the SAP system will subsequently request confirmation of the TID, thus *onConfirmTID()* will be the next method called.
- *protected void onCommit(String tid)* — This TID management method (shown in lines 62-72) is called immediately after *handleRequest()* is executed successfully (i.e., without a Java exception being thrown). Commit all the local changes (if any) in this method. The SAP system will subsequently request confirmation of the TID, thus *onConfirmTID()* will be the next method called.
- *protected void onRollback(String tid)* — This TID management method (shown in lines 74-84) is called immediately after the *handleRequest()* method runs into a problem (i.e., a Java exception is thrown). Roll back all the local changes (if any) in this method. The SAP system will automatically schedule a retransmission of the IDoc.
- *protected void onConfirmTID(String tid)* — This TID management method (shown in lines

Figure 7 ***Incoming TID Management***

```
 1 public static class MyIDocListener extends JCoIDoc.Server {
 2   private TidInDAO tidDAO = null;
 3   private String tid = null;
 4
 5   // Overridden constructor of JCoIDoc.Server.  Choose the
        constructors that you would like to make public.
 6   public MyIDocListener(String gwhost, String gwserv, String progid,
         IRepository jcoRepository, IDoc.Repository idocRepository) {
 7     super(gwhost, gwserv, progid, jcoRepository, idocRepository);
 8     tidDAO = new TidInDAO();
 9   }
```

(continued on next page)

Figure 7 *(continued)*

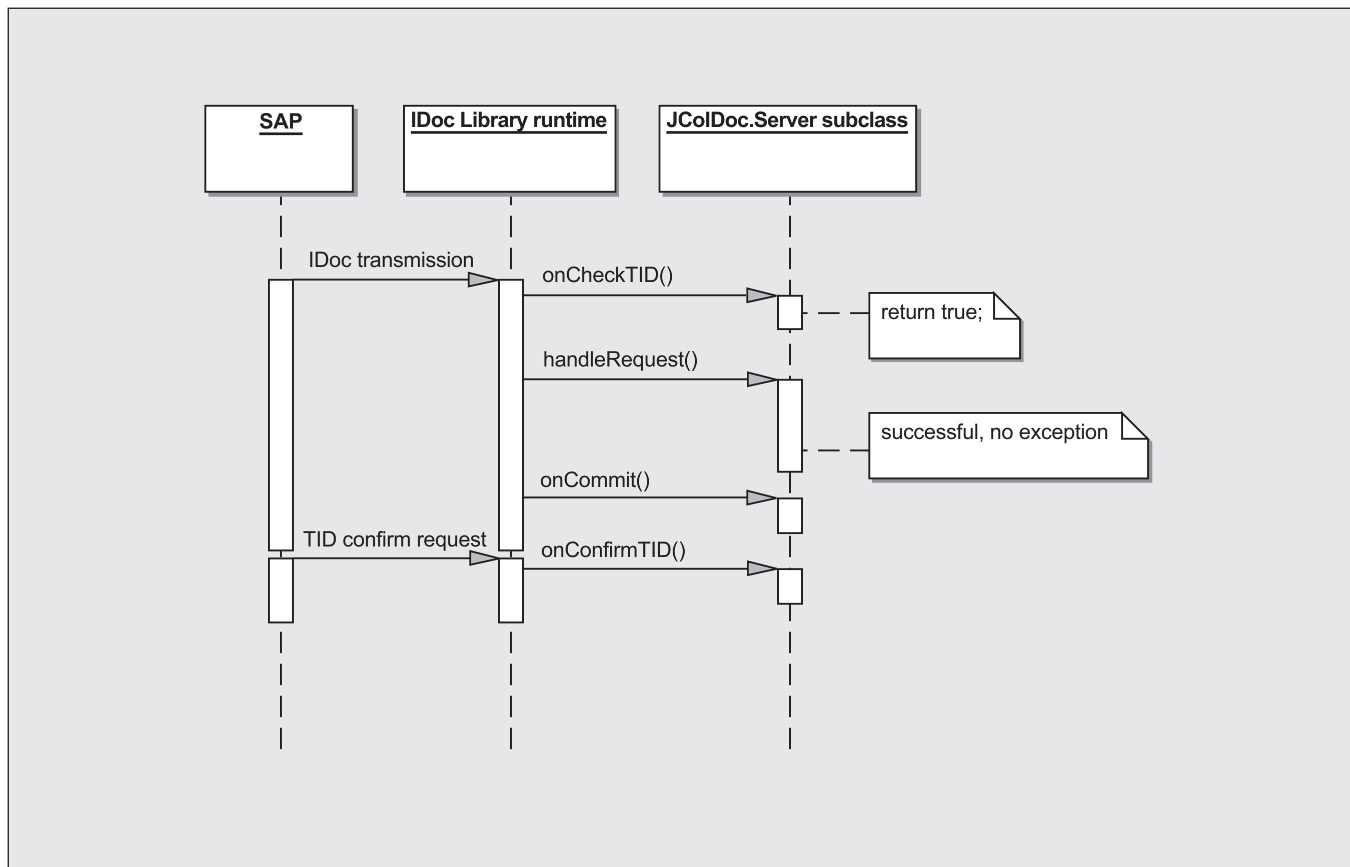
```
10   public MyIDocListener(Properties properties, IRepository
         jcoRepository, IDoc.Repository idocRepository) {
11     super(properties, jcoRepository, idocRepository);
12     tidDAO = new TidInDAO();
13   }
14
15   protected boolean onCheckTID(String tid) {
16     System.out.println("\nonCheckTID() called for TID " + tid);
17     this.tid = tid;
18     try {
19       int currentTidState;
20       currentTidState = tidDAO.getState(tid);
21       switch (currentTidState) {
22         case TidInDAO.IDOC_IN_NOTEXIST:
23           int rowsWritten = tidDAO.writeNewTid(tid);
24           return true;
25         case TidInDAO.IDOC_IN_NEW:
26           // shouldn't really happen -- let's reprocess it anyway
27           return true;
28         case TidInDAO.IDOC_IN_PROCESSING:
29           // shouldn't really happen -- let's reprocess it anyway
30           return true;
31         case TidInDAO.IDOC_IN_ROLLBACKED:
32           return true;
33         case TidInDAO.IDOC_IN_COMMITTED:
34           // already processed and committed, skip processing
35           return false;
36         case TidInDAO.IDOC_IN_CONFIRMED:
37         // already processed, committed and confirmed, skip processing
38         return false;
39       }
40     } catch (SQLException sqlex) {
41       sqlex.printStackTrace();
42       System.exit(1);
43     }
44     // shouldn't reach here -- just to satisfy the compiler
45     return true;
46   }
47
48   // Overridden method of JCoIDoc.Server.  Function requests that
        contain IDocs will be handled here.
49   protected void handleRequest(IDoc.DocumentList documentList)
         throws Exception {
50     System.out.println("handleRequest(IDoc.DocumentList) called for
           TID " + this.tid);
51     try {
52       tidDAO.updateState(this.tid, TidInDAO.IDOC_IN_PROCESSING);
53     } catch (SQLException sqlex) {
```

 www.SAPpro.com ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 7 *(continued)*

```
54        sqlex.printStackTrace();
55        System.exit(1);
56      }
57
58      // process the IDocs received; throw Java exceptions
           if run into problems ...
59
60    }
61
62    protected void onCommit(String tid) {
63      System.out.println("onCommit() called for TID " + tid);
64      // commit whatever local changes ...
65      // then set the TID state to COMMITTED
66      try {
67        tidDAO.updateState(this.tid, TidInDAO.IDOC_IN_COMMITTED);
68      } catch (SQLException sqlex) {
69        sqlex.printStackTrace();
70        System.exit(1);
71      }
72    }
73
74    protected void onRollback(String tid) {
75      System.out.println("onRollback() called for TID " + tid);
76      // rollback whatever local changes ...
77      // then set the TID state to ROLLBACKED
78      try {
79        tidDAO.updateState(this.tid, TidInDAO.IDOC_IN_ROLLBACKED);
80      } catch (SQLException sqlex) {
81        sqlex.printStackTrace();
82        System.exit(1);
83      }
84    }
85
86    protected void onConfirmTID(String tid) {
87      System.out.println("onConfirmTID() called for TID " + tid);
88      try {
89        // because of the strange behavior of the SAP system,
             we must only change state to CONFIRMED if the previous state
             is COMMITTED
90        if (tidDAO.getState(tid) == TidInDAO.IDOC_IN_COMMITTED) {
91          tidDAO.updateState(tid, TidInDAO.IDOC_IN_CONFIRMED);
92        }
93      } catch (SQLException sqlex) {
94        sqlex.printStackTrace();
95        // no harm done here
96      }
97    }
98  }
```

Figure 8 UML Sequence Diagram for Processing a New Incoming IDoc



86-97) is called if the SAP system requests confirmation of the TID. Set the TID state to "confirmed" in the TID table.[9]

The diagram in **Figure 8** illustrates the normal sequence of the TID management method invocations for processing a new incoming IDoc.

The diagram in **Figure 9** illustrates the sequence of the TID management method invocations for an IDoc transmission that has been identified as a duplicate.

[9] Alternatively, you can delete the TID from the table if you don't want to keep a history. Due to a known problem with the SAP system, if the tRFC server has multiple listener threads, the external tRFC server may receive a TID confirmation request from the SAP system even if the TID is assigned a rollback status. Set the TID status to "confirmed" only if the TID is already committed.

The diagram in **Figure 10** illustrates the sequence of the TID management method invocations when the processing of an incoming IDoc fails with a Java exception.

You've now seen how to write your IDoc listener class (*MyIDocListener*) as a *JCoIDoc.Server* subclass and implement the necessary TID management methods. The only thing left to do is start the IDoc listener so that it is ready to listen for and receive IDocs from the SAP system.

Starting the IDoc Listener

To start the IDoc listener in our server program (refer to line 54 in Figure 1), you first need to provide the

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Figure 9 UML Sequence Diagram for a Duplicate Incoming IDoc

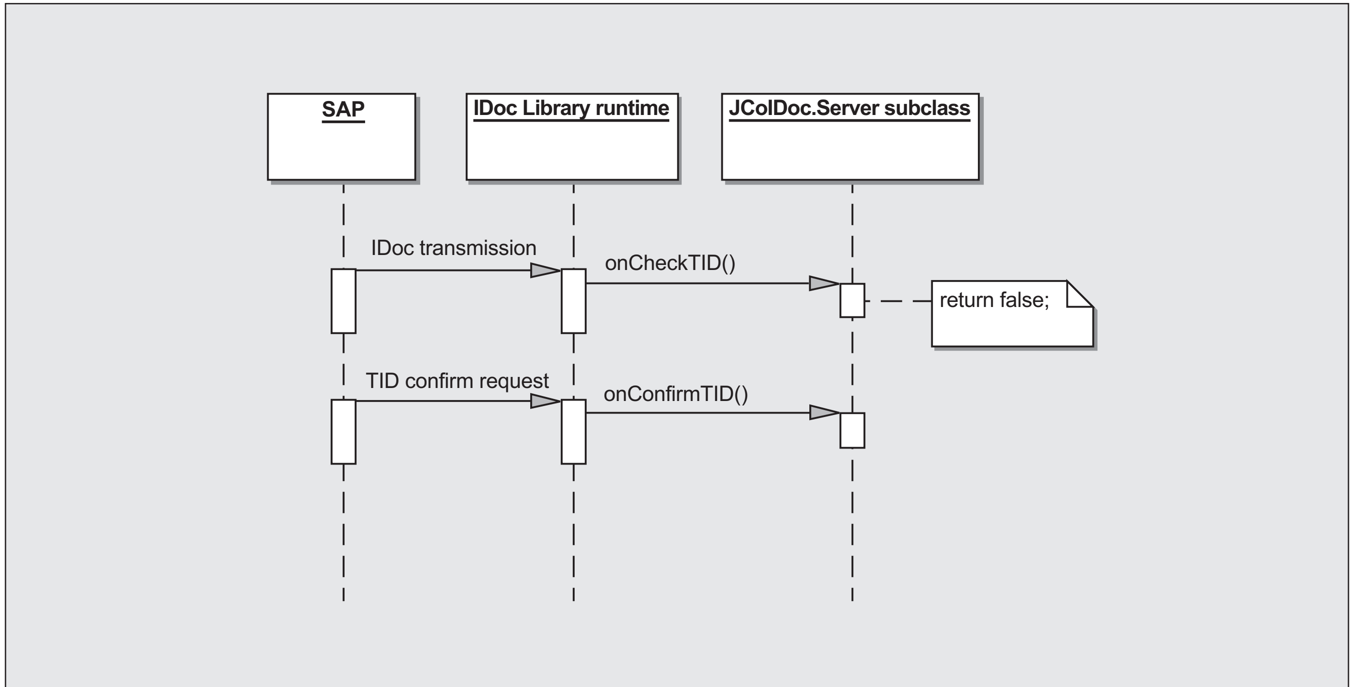


Figure 10 UML Sequence Diagram for Failed Processing of an Incoming IDoc

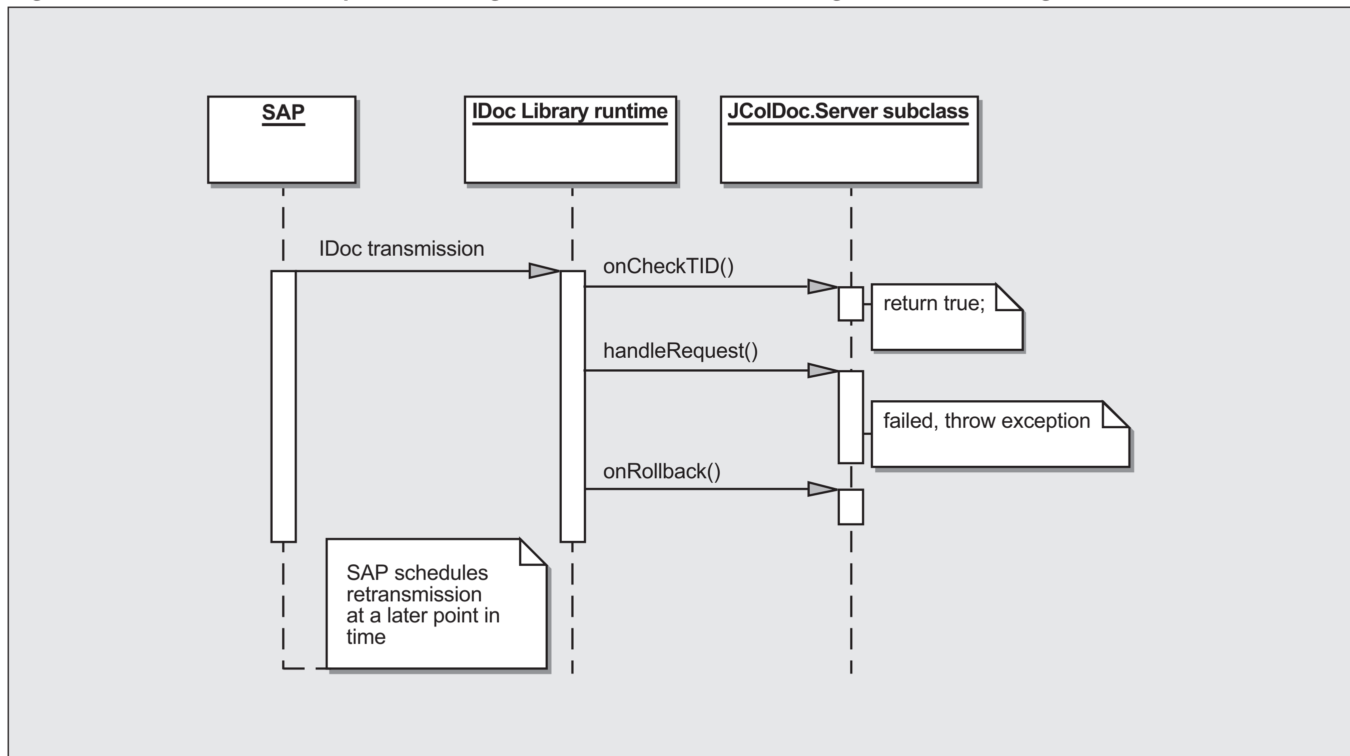


Figure 11 *Starting Multiple IDoc Listener Threads*

```
 1 int NUM_LISTENERS = 3;
 2   JCO.Repository jcoRepository =
       JCO.createRepository("MyJCoRepository", POOL);
 3   IDoc.Repository idocRepository =
       JCoIDoc.createRepository("MyIDocRepository", POOL);
 4
 5   Properties srvProps = new Properties();
 6   srvProps.put("jco.server.gwhost", "mygatewayhost");
 7   srvProps.put("jco.server.gwserv", "sapgw00");
 8   srvProps.put("jco.server.progid", "myprogid");
 9
10   for (int i=0; i<NUM_LISTENERS; i++) {    // start multiple IDoc
                                                 listener threads
11     // instantiate the IDoc server
12     listeners[i] = new MyIDocListener(srvProps, jcoRepository,
         idocRepository);
13     try {
14       listeners[i].start();                // start the listener
                                                 thread
15     } catch (Exception ex) {
16       System.out.println("Could not start the listener thread!\n" + ex);
17       System.exit(1);
18     }
```

following information to the constructor of your IDoc listener to instantiate an instance of the listener (see the example shown in **Figure 11**):

- An instance of *JCO.Repository* (line 2)
- An instance of *IDoc.Repository* (line 3)
- The gateway host (line 6)
- The gateway service (line 7)
- The program ID (line 8)

To avoid hard-coding the gateway and program ID, you can store them in a property file and retrieve them dynamically at runtime.

Once the listener instance is created, you can then call its *start()* method to start the listener thread (line 14). The external RFC server thread will then connect to the SAP gateway, register itself with the given program ID, and be ready to receive IDocs from the SAP system.

To improve the performance and scalability of your IDoc server program, you may want to start multiple IDoc listener threads. The IDoc library makes this very easy to do[10]: you can simply create multiple instances of your IDoc listener class and call each instance's *start()* method. These listener threads will all register at the SAP gateway and be able to process concurrent IDoc transmissions from the SAP system in parallel. Lines 10-18 of Figure 11 illustrate this.

[10] By utilizing the RFC server multithread support in the underlying SAP Java Connector (JCo).

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.

Limited XML Capability of the SAP Java IDoc Class Library

The SAP Java IDoc Class Library offers some limited XML capability. Both the IDoc.DocumentList and IDoc.Document interface provide the following XML-producing methods:

- String toXML() — Returns the document list (for IDoc.DocumentList) or the document (for IDoc.Document) in the default IDoc-XML* format.
- String toXML(String release) — Returns the document list (for IDoc.DocumentList) or the document (for IDoc.Document) in a specified version of the IDoc-XML format.
- void writeXML(String filename) — Dumps the document list (for IDoc.DocumentList) or the document (for IDoc.Document) to the specified XML file using UTF-8 character encoding.
- void writeXML(String filename, String charEncoding) — Dumps the document list (for IDoc.DocumentList) or the document (for IDoc.Document) to the specified XML file using the specified character encoding.
- void writeXML(java.io.OutputStreamWriter writer) — Dumps the document list (for IDoc.DocumentList) or the document (for IDoc.Document) to the specified output stream writer as XML.
- void writeXML(java.io.Writer writer) — Dumps the document list (for IDoc.DocumentList) or the document (for IDoc.Document) to the specified writer as XML.

These methods are handy for debugging purposes. You may even use them to write lightweight XML-generation programs to convert the received IDocs to the SAP IDoc-XML document format. However, there is a serious limitation to the XML capabilities of the SAP Java IDoc Class Library: it can only produce XML from received IDocs; it cannot create IDocs from XML documents. If you need complete XML capabilities, you must use the SAP Business Connector** or the SAP Exchange Infrastructure.

---

* This is SAP's XML vocabulary for IDocs. For details, please see **http://ifr.sap.com/home/Documents/XML_Kap3_E.htm#323**.

** For details on using the SAP Business Connector for XML messaging, please read my articles in the March/April 2003 issue of this publication.

Helpful Hints

The following is a quick reference of useful tips presented in this article.

☑ Extend the *JCoIDoc.Server* class to implement your IDoc listener subclass.

☑ Implement the *handleRequest(IDoc.DocumentList idocs)* method to process received IDocs and use an iterator to iterate through all the IDocs in the received documents list.

☑ Navigate the data segment tree of an IDoc using the various methods of the IDoc library's *IDoc.Segment* interface.

☑ To provide TID management, you must implement

the *onCheckTID()*, *onCommit()*, *onRollback()*, and *onConfirmTID()* methods in your IDoc server program.

☑ Use a database table to track the state of incoming TIDs, and write a Java helper class to facilitate access to the TID table by the IDoc server program.

☑ If necessary, start multiple threads of the IDoc listener to improve performance and scalability.

Conclusion

The new SAP Java IDoc Class Library has made the complicated and tedious task of writing Java programs that receive and process IDocs much easier and more efficient.

In the first part of this two-part series, I discussed the basic concepts of the SAP IDoc technology, introduced the IDoc library, and showed you how to write efficient Java programs that compose and send IDocs to the SAP system as well as implement robust TID management in the IDoc client programs for monitoring IDoc processing. Here in this second installment, I discussed how to use the IDoc library to write efficient Java programs that receive and process IDocs from the SAP system as well as how to implement robust TID management in IDoc server programs.

I hope this article series has convinced you that the SAP Java IDoc Class Library makes it significantly easier for you to write Java programs that create and send IDocs, as well as receive, process, and track them, and has provided you with the knowledge you need to start using it in your own environment. Have fun!

Robert Chu joined SAP at the end of 1996. He currently works for the Integration and Certification Center at SAP Labs in Palo Alto, California. Previously, he worked in technical consulting and training at SAP America and SAP Asia. Robert's current focus is SAP's integration technologies. He has been regularly teaching classes in this area at SAP training centers and is the main author of the BIT531 training course, as well as a few other internal workshops. Robert has spoken at the past three SAP TechEd events. In addition to his SAP expertise, he is also an SCEA (Sun Certified Enterprise Architect) for J2EE, an MCSD (Microsoft Certified Solution Developer), and an MCSE (Microsoft Certified System Engineer). Robert can be reached at robert.chu@sap.com.

 ©2003 SAP Professional Journal. Reproduction prohibited. All rights reserved.