Mastering the asXML Format to Leverage ABAP-XML Serialization

Stefan Bresch, Christian Stork, and Christoph Wedler

Stefan Bresch, Business Programming Languages Group, SAP AG

Christian Stork, Business Programming Languages Group, SAP AG

Christoph Wedler, Business Programming Languages Group, SAP AG

(complete bios appear on page 52)

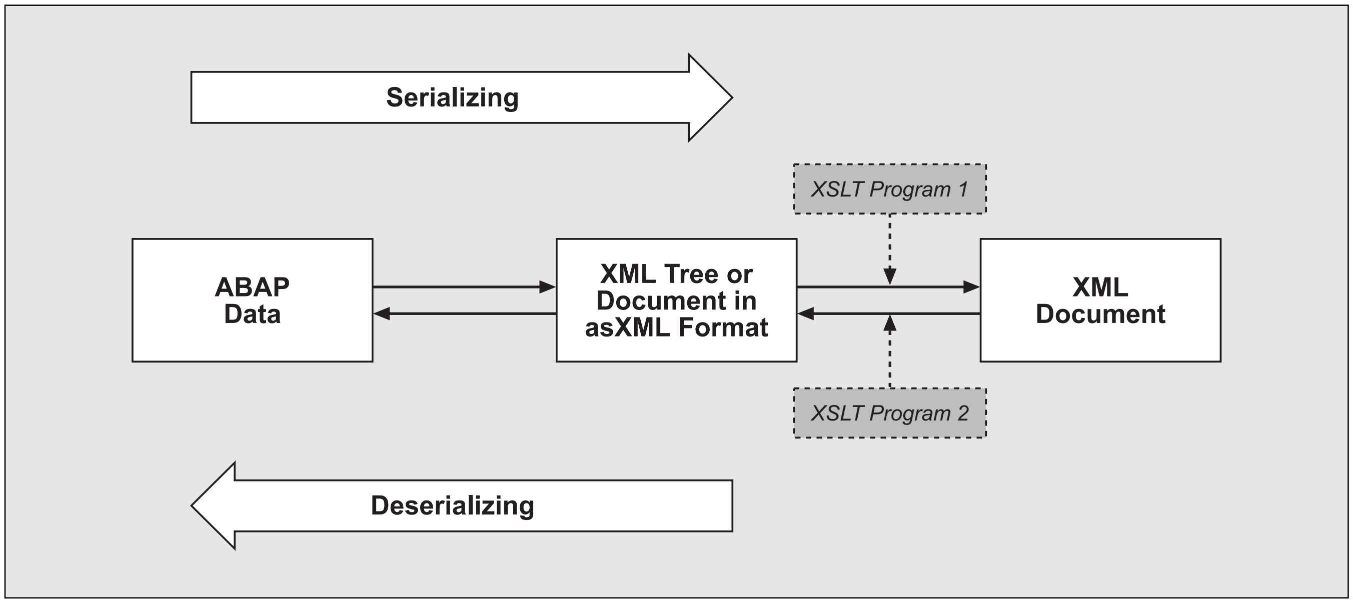
As every SAP developer knows, exchanging data with other applications and systems is simply a fact of everyday life. But implementation can be complex, even for experienced ABAP programmers. You need to *serialize* (or *write*) your application data into a stream (e.g., a string), in a format that another application can recognize. You also need to *deserialize* (or *read*) a stream from another application into data in your application. In the past, your serialization[1] options were limited to what you could accomplish via the ABAP statements `EXPORT` and `IMPORT`. Beginning with Release 6.20,[2] ABAP-XML serialization offers ABAP programmers, ABAP class designers, and XSLT programmers a more full-featured alternative:

- It supports all data types, including data and object references. The `EXPORT`/`IMPORT` method supports only elementary data types, structures, and internal tables.
- Class designers can control whether objects of a class can be serialized, and how.
- It uses an XML-based format called *asXML* (ABAP Serialization XML), which provides a common intermediate format for exchanging data between SAP and other systems in both directions. You can use XSLT (the XML transformation language) to transform documents easily between asXML and other XML-based formats.

[1] For simplicity, we use the collective term *serialization* to represent the process in both directions (i.e., serialization and deserialization). Where we specifically mean one direction only, we use the verb form (i.e., *serialize* or *deserialize*).

[2] ABAP-XML deserialization was initially implemented in Release 6.10, but with significant restrictions (such as lack of support for deserializing objects).

Figure 1 How ABAP-XML Serialization Works



(Please note that we do not cover XSLT here, since the topic was covered in detail in a previous *SAP Professional Journal* article.[3])

Clearly, mastering the asXML format is your key to ABAP-XML serialization success. In this article, we describe the asXML format for all data types and examine object serialization in detail.

Understanding ABAP-XML Serialization

Let's start with a brief review of how ABAP-XML serialization works (see **Figure 1**). XML (eXtensible Markup Language) has become the web standard for generic representation of tree-structured data. It provides a generic data model and syntax ("markup") for writing down ("serializing") the trees. Like other programming languages, ABAP provides its own model for structured data. Serializing ABAP data to XML documents consists of first converting the ABAP data into an equivalent XML tree, then serializing that XML tree using the generic XML markup. Thus serialization creates XML trees and documents in asXML format from ABAP data. Optionally, an XSLT program can transform these trees or documents into documents of a standard XML format. It is common to use the term *XML serialization* even if the second step (serialization of the XML tree) is omitted for a particular situation. The opposite process, "deserializing" XML documents and trees to ABAP data, creates ABAP data elements from XML.

[3] For more information about using XSLT in this context, see the article "From XML to ABAP Data Structures and Back: Bridging the Gap with XSLT" (July/August 2002).

The Role of the ABAP Data Type

The ABAP data type always determines how data is represented in asXML format:

- When serializing ABAP data, the data type of the ABAP source data determines the format of the data in the generated asXML document.
- When deserializing from XML, the data type of the ABAP data being constructed determines the expected format. The ABAP data type is the static type of the field that will contain the constructed data, the type of component (deduced

Possible Errors During Serialization

Errors that occur during the serialization process typically result from data type or format problems, including corruption. The following table describes the associated exceptions* that will be thrown:

Exception Class	Cause	Example	Class Design Notes
If You Are Serializing…			
CX_XSLT_SERIALIZATION_ERROR	• The ABAP data is corrupted. • Part of the ABAP data can't be reasonably represented.	The data contains an illegal decimal number, where any arithmetic operation performed on it would lead to the short dump BCD_BADDATA.	You can define when an exception will be thrown (e.g., if the object cannot be reasonably represented).
If You Are Deserializing…			
CX_XSLT_FORMAT_ERROR	• The XML document structure doesn't match the ABAP data type being constructed.	You expected a number, but the XML document contains elements that correspond to the components of a structure.	N/A
CX_XSLT_DESERIALIZATION_ERROR	• The character data doesn't have the stated character format. • Deserialization leads to either ambiguities or illegal data.	An XML text node contains letters, but a number is being constructed.	You can define when an exception will be thrown (e.g., if attribute values are not legal).

* These exceptions are *class-based exceptions*, which were introduced with Release 6.10. For more information, see the article "A Programmer's Guide to the New Exception-Handling Concept in ABAP" (September/October 2002).

from the structure type), the type of table line (deduced from the table type), etc.

During serialization and deserialization, format checks are automatically performed to validate the data. These checks are typically more forgiving when deserializing than when serializing. For example, you can deserialize XML documents even if the data does not *exactly* match the expected format (such as numbers with leading or trailing spaces). However, if the data in the XML document does not meet even this lax format interpretation, the system typically throws an exception (see the sidebar above).

Invoking Serialization from ABAP

You transform ABAP data to and from XML with the ABAP statement `CALL TRANSFORMATION`, which combines both serialization and XSLT transformation. To keep the examples focused on asXML, we always use the predefined XSLT transformation `ID`, which does nothing. In other words, `CALL TRANSFORMATION ID` works like serialization without an XSLT transformation.

So, to *serialize* ABAP data, use the following `CALL TRANSFORMATION` syntax:

```
CALL TRANSFORMATION ID
  SOURCE BN_1 = e_1 … BN_n = e_n
  RESULT XML asx_doc.
```

The parameters `e_i` (where $1 \leq i \leq n$) are the ABAP data elements to be serialized. The parameters `BN_i` (where $1 \leq i \leq n$) are the corresponding binding names (the symbolic names in the XML representation) to which the data elements are bound. The result of the serialization is assigned to the field `asx_doc`, which can be of type `STRING` or `XSTRING`, or can be a reference to an iXML DOM[4] or an iXML output stream.[5] A `STRING` is a stream of characters of the system code page, so it must be converted to the operating system code page when written to the file system.[6] Since an XML document also contains XML format-encoding information (which could be converted inaccurately), use the result type `XSTRING` when writing an XML document to the file system.

To *deserialize* XML documents, use the following `CALL TRANSFORMATION` syntax:

```
CALL TRANSFORMATION ID
  SOURCE XML asx_doc
  RESULT BN_1 = v_1 … BN_n = v_n.
```

The parameters `v_i` (where $1 \leq i \leq n$) are the ABAP fields where the deserialized data is to be stored. The parameters `BN_i` (where $1 \leq i \leq n$) are the corresponding binding names (the symbolic names in the XML representation) to which the data elements are bound. The source of the deserialization is the field `asx_doc`, which can be of type `STRING` or `XSTRING`, or a reference to an iXML node set, or an iXML input stream.[7] As with serialization, use the source type `XSTRING` when reading an XML document from the file system.

Alternatively, you can provide the `CALL TRANSFORMATION` parameters dynamically as an internal table that contains pairs of binding names and references to data elements. For example, the following code fragment shows how to use an internal table when serializing:

```
CALL TRANSFORMATION ID
  SOURCE (itabsrc)
  RESULT XML asx_doc
```

This next code fragment shows how to use an internal table when deserializing:

```
CALL TRANSFORMATION ID
  SOURCE XML asx_doc
  RESULT (itabres).
```

The internal tables `itabsrc` (the source table) and `itabres` (the result table) are of types `ABAP_TRANS_SRCBIND_TAB` and `ABAP_TRANS_RESBIND_TAB`, respectively.

Introducing the asXML Format

The asXML format defines the representation of ABAP data in XML. As in **Listing 1**, you can identify documents in asXML format by the fixed "envelope" with the root element `abap` (i.e., `asx:abap`) in the namespace `http://www.sap.com/abapxml`. The root element includes the mandatory subelement `values` (in the same namespace), which in turn contains the elements `BN_i` (where $1 \leq i \leq n$) (without an XML namespace[8]). These element names are the binding names from the ABAP statement `CALL TRANSFORMATION`.

The root element `abap` also includes an optional `version` attribute with an unsigned number and a maximum of one decimal place. This attribute enables SAP to extend the asXML format in the

[4] If you're new to XML in the SAP context or you haven't read the article in the July/August 2002 issue, the "i" stands for integrated (into the kernel). DOM stands for Document Object Model.

[5] You can also use an internal table of type `C`, although it's not recommended for performance reasons.

[6] For more on code pages, see the article "Globalizing Applications Part 1: Pre-Unicode Solutions" (September/October 2001).

[7] As with the output stream, you can also use an internal table of type `C`, although it's not recommended for performance reasons.

[8] For more about XML namespaces, see the XML specification at **www.w3.org/TR/REC-xml-names/**.

Listing 1: The asXML Document Format

```
<asx:abap version = "1.0" xmlns:asx = "http://www.sap.com/abapxml">
  <asx:values>
    <BN_1>…</BN_1>

    …

    <BN_n>…</BN_n>
  </asx:values>
  <asx:heap>
    <!-- sequence of referenced objects and data objects -->
  </asx:heap>
</asx:abap>
```

future without impacting existing asXML documents. When serializing, the `version` attribute is always automatically assigned the default value `1.0`.[9]

The optional element `heap` contains referenced objects and data objects (which are covered in detail later on, when we discuss references).

A Basic Example: Serializing a String

As a starting point, let's look at an ABAP program fragment that shows how to serialize a simple string into a document in asXML format:

```
DATA: hello TYPE string VALUE
      'hello'.
DATA: result TYPE string.
CALL TRANSFORMATION ID
  SOURCE GREETING = hello
  RESULT XML result.
```

This code generates the string `result` in asXML format with the following contents (as viewed with the debugger):

```
#<?xml version="1.0" encoding=
     "utf-16"?>#<asx:abap xmlns:asx=
     "http://www.sap.com/abapxml"
     version="1.0"><asx:values>
     <GREETING>hello</GREETING>
     </asx:values></asx:abap>
```

Note the following characteristics of the string:

- It begins with an XML header, which was automatically added during serialization. It includes a BOM (byte order mark) at the beginning of the header (only on Unicode systems[10]) and a line break at the end; both appear as a hashmark (#) in the example. The value of the `encoding` element depends on your R/3 system (this example reflects a Unicode system).
- There are no indents or additional line breaks.
- Notice the asXML envelope, which was also automatically added during serialization.
- This example does not contain the optional element `heap`, since no references were serialized.
- The `values` element contains one element with the binding name `GREETING`, which contains the asXML representation of the ABAP field `hello`.

[9] When deserializing, you either do not provide the attribute `version`, or you specify it with a value between `0.0` and `1.9`. Specifying a value outside that range will cause an exception of class `CX_XSLT_FORMAT_ERROR`. When deserializing, accepting a value between `1.1` and `1.9` ensures restricted forward compatibility of the current kernel (Release 6.20) with future format extensions.

[10] See "Looking Forward to the Unicode Advantage: Internationalization and Integration" (January/February 2002) for more on Unicode.

✓ ***Note!***

For simplicity and clarity, the remaining examples in this article omit the XML header, but include line indents and breaks. Unused XML namespace declarations are also omitted.

Time to get working with the data itself. Before we get started, see the sidebar on the next page for a review of how ABAP names are used as element names in XML documents.

Representing Data Values in asXML

Given an understanding of the basics of ABAP-XML serialization, we can now turn to representing data in asXML. First we will look at how to represent elementary ABAP data types (such as strings or numbers), structures, and internal tables. Then in the next section, we'll move on to the more complex task of representing references, referenced data objects, and referenced objects. Along the way, we will point out some of the format rules to keep in mind.

Representing Elementary Data Types

✓ ***asXML Format Rule: Elementary Data Types***

ABAP data of an elementary data type is represented by characters using the canonical lexical representation[11] *of the corresponding XML Schema data type.*

When working with elementary data types, the standard XML character-encoding rules are automatically applied (`&`, `<`, and `>` are mapped respectively to `&amp;`, `&lt;`, and `&gt;`). Some control characters (such as formfeed) are not valid XML characters.[12] If the ABAP runtime must produce one of these characters when serializing, it will throw an exception of class `CX_XSLT_SERIALIZATION_ERROR`. When deserializing, the XML parser will reject documents containing these characters and throw an exception of class `CX_XSLT_RUNTIME_ERROR`.

By adopting the XML Schema representation rules (especially for built-in data types[13]), the asXML format ensures that simple values are represented the same as in XML-based standards. This consistency simplifies the XSLT programming task of transforming documents between standard XML formats and asXML. For many data types, the representation is the same as the usual ABAP character representation. However, for some data types, the representation is somewhat different. (See **Figure 2**.)

ABAP data of all data types except `STRING` and `C` will be represented in asXML without leading and trailing spaces. Data of type `C` is also represented without trailing spaces. When serializing, such spaces are ignored.

If errors occur, an exception of class `CX_XSLT_SERIALIZATION_ERROR` will be thrown when serializing, and an exception of class `CX_XSLT_DESERIALIZATION_ERROR` when deserializing (with an exception representing the cause in the attribute `PREVIOUS`).

The asXML representation is not always identical to the built-in data type of the XML Schema shown in Figure 2. However, except for types `D` and `T` (more on this in a moment), all generated values are legal according to the corresponding XML Schema type. SAP's design goal was to be able to represent most valid[14] ABAP values, including data

[11] See **www.w3.org/TR/xmlschema-2/#canonical-lexical-representation**.

[12] They cannot even be encoded using character references. For example, using `&#12;` instead of formfeed is also not valid.

[13] See **www.w3.org/TR/xmlschema-2/#built-in-datatypes**.

[14] A value is considered valid if you can use the ABAP statement `MOVE` to set a field of the corresponding ABAP type to that value. This excludes the use of `ASSIGN...CASTING` to directly change the internal representation of the value, which would, for example, be necessary to produce data of type `N` that doesn't contain digits.

Using ABAP Names as XML Element Names

Let's review how binding names and other ABAP names (such as component names) are used as element names in XML documents. Except for character or string literals, ABAP names are case-*insensitive* and normalized to uppercase. Therefore, these names always appear in uppercase in the generated XML document. However, remember that XML is case-*sensitive*. Providing lowercase binding names in asXML documents will usually* prevent the corresponding elements from being deserialized.

Additionally, ABAP names can contain characters that are illegal for XML element names. Consequently, these characters are automatically replaced with standard escape sequences during serialization. For example, the ABAP name /CRM/FOO is automatically mapped to _-CRM_-FOO. The table on the next page describes the rules for mapping ABAP names to XML element names. (Note that these rules are compatible with the XRFC format of the SAP Business Connector.)

* You can serialize or deserialize these elements if you provide the CALL TRANSFORMATION parameters dynamically. Providing the binding names as string contents prevents them from being normalized to uppercase.

(continued on next page)

Figure 2 Representation of ABAP Data Types According to XML Schema Type

ABAP Type	ABAP Example	XML Schema Type	XML Example	Comment
STRING	_Hello_	string	_Hello_	string of characters
C	_Hi	string	_Hi	
N	001234	string (pattern [0-9]+)	001234	string of digits
I (INT1), (INT2)	123-	int unsignedByte, short	-123	number
P	1.23-	decimal	-1.23	
F	-3.1400000000000000E+02	double	-3.14E2	
D	20020204	date	2002-02-04	ISO8601 date/time
T	201501	time	20:15:01	
XSTRING	456789AB	base64Binary	RweJqw==	Base64-encoded binary data
X	ABCDEF	base64Binary	q83v	

(continued from previous page)

Character in ABAP Name	Mapped Character in XML Element Name
xml as the first three characters (in any combination of uppercase and lowercase letters)	x-ml (in the corresponding combination of uppercase and lowercase letters). This prefix is reserved in the XML standard.
A to Z a to z _ (underscore)	No change (the character remains in its original form).
0 to 9, unless the first character of a name (see below)	No change (the character remains in its original form).
/ (forward slash)	_- (note that ABAP namespaces are not mapped to XML namespaces because the concepts are different).
Any other ASCII character, including 0 to 9 as the first character of a name	_--hex(c), where hex(c) is the two-character hexadecimal representation of the ASCII code of the character c.

of types `D` and `T`, in asXML. Let's take a closer look at the details of Figure 2:

- In asXML, ABAP data of types `STRING`, `C`, and `N` is represented as is. For data of type `N`, characters other than digits are not allowed.[15] Deserializing data into a field of type `C` or `N` that is longer than the data in the document will cause the rest of the field to be filled with spaces for type `C`, or the beginning of the field to be filled with zeros for type `N` (as the ABAP statement `MOVE` does). If some data loss occurs — i.e., if the field is too short[16] — an exception will be thrown.

- ABAP data of types `I`[17], `P`, and `F` is represented as a decimal number with an optional leading character for the sign, using the scientific notation for type `F`. When serializing, no positive sign is produced, either as a plus sign (+) or a space. When deserializing, the ABAP notation for data of types `I` and `P` (trailing sign) is also accepted. However, you might get an exception due to the usual errors when reading a number. When serializing a corrupted number of type `P`[18], an exception will be thrown. When deserializing a number into a field of type `P` with too few decimal places, an exception will be thrown.

- ABAP data of type `D` or `T` is represented according to ISO8601, with restrictions (e.g., the year must be in the range 0-9999) and extensions. A date value is represented with four characters for the year, two characters for the month, and two

[15] If anything other than a digit appears, an exception will be thrown. When deserializing, leading and trailing spaces are deleted before the digit check.

[16] The length is checked after leading and trailing spaces have been deleted, as well as leading zeros for type `N`.

[17] Including the restricted integer types represented by the dictionary types `INT1` and `INT2`.

[18] If a corrupted number of type `P` is used in any arithmetic operation, a short dump `BCD_BADDATA` occurs.

Listing 2: Serializing a Date

```
DATA: today TYPE d VALUE '20020816'.
DATA: result TYPE string.
CALL TRANSFORMATION ID
  SOURCE TODAY = today
  RESULT XML result.
```

Listing 3: asXML Result of a Serialized Date

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <TODAY>2002-08-16</TODAY>
  </asx:values>
</asx:abap>
```

characters for the day, separated by hyphens (-). A time value is represented with two characters for the hour, two characters for the minute, and two characters for the second, separated by colons (:). Unfortunately, in ABAP, any characters are allowed for values of types `D` and `T`, so the above-mentioned characters do not have to be numbers. No further checks are performed, so, for example, the date `0000-00-00` (which is not a valid date according to ISO8601) *is* allowed in ABAP. If the first or the last character in the value is a space, the value cannot contain a hyphen for values of type `D`, and cannot contain a colon for values of type `T`.[19]

[19] To understand the reason for this format restriction, suppose you have two values: `'  :::::'` and `':::::  '` of type `T`. Since ABAP data of type `T` is represented without leading and trailing spaces, both values are represented as `'::::::::'`. When deserializing, the ABAP runtime cannot correctly create two different values from the same representation. To overcome this problem, SAP could have dropped the rule to ignore leading and trailing spaces for types `D` and `T`, but chose not to since this is not what users expect and the restriction affects only fairly uncommon values for data of types `D` and `T`.

- In asXML, ABAP data of type `XSTRING` or `X` is represented using Base64 encoding (as defined in RFC 2045[20]), which is the standard encoding for binary data in XML documents (probably because it uses 33% fewer characters than hexadecimal encoding). Deserializing data into a field of type `X` that is longer than the data in the document will cause the rest of the field to be filled with binary zeros (as the ABAP statement `MOVE` does). If some data loss occurs — i.e., if the field is too short — an exception will be thrown.

Now let's look at a simple serialization example. **Listing 2** demonstrates how to serialize an elementary data type (a date).

In **Listing 3**, you see the asXML generated from the `result` string in Listing 2.

[20] For more information about RFC (Request for Comments) 2045, go to **www.ietf.org/rfc/rfc2045.txt**.

Listing 4: Serializing a Structure

```
TYPES: BEGIN OF struc_type,
         /abap/s TYPE string,
         i TYPE i,
       END OF struc_type.
DATA: struc TYPE struc_type.
DATA: result TYPE string.
struc-/abap/s = 'the answer is'.
struc-i = 42.
CALL TRANSFORMATION ID
  SOURCE STRUCTURE = struc
  RESULT XML result.
```

Listing 5: asXML Result of a Serialized Structure

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <STRUCTURE>
      <_-ABAP_-S>the answer is</_-ABAP_-S>
      <I>42</I>
    </STRUCTURE>
  </asx:values>
</asx:abap>
```

Representing Structures

✓ asXML Format Rule: Structures

A structure is represented by a sequence of elements that represent the components of the structure. The name of each element is the name of the component (using character-mapping rules for element names, if necessary). The content of the element is the asXML representation of the component's value.

While serializing an elementary data type involves a single element, serializing a structure consists of representing multiple elements. When serializing, all components of a structure are serialized in the order defined in the structure. When deserializing, the order of XML elements corresponding to components does not matter. Additional XML elements will be ignored,[21] and components without a corresponding XML element retain their values.

Let's examine a more complex serialization example. The code fragment in **Listing 4** shows how to serialize a structure. **Listing 5** shows the generated asXML. In particular, note the mapping from the ABAP component name `/abap/s` (Listing 4) to the XML element name `_-ABAP_-S` (Listing 5).

[21] Element names with a non-empty XML namespace will cause an exception of type `CX_XSLT_FORMAT_ERROR`.

Listing 6: Serializing an Internal Table

```
TYPES: tab_type TYPE STANDARD TABLE OF i WITH DEFAULT KEY.
DATA: tab TYPE tab_type.
DATA: result TYPE string.
APPEND 6 TO tab.
APPEND 7 TO tab.
APPEND 42 TO tab.
CALL TRANSFORMATION ID
  SOURCE ITAB = tab
  RESULT XML result.
```

Listing 7: asXML Result of a Serialized Internal Table

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <ITAB>
      <item>6</item>
      <item>7</item>
      <item>42</item>
    </ITAB>
  </asx:values>
</asx:abap>
```

Representing Internal Tables

✓ asXML Format Rule: Internal Tables

An internal table is represented by a sequence of elements that represent the lines of the internal table. The content of each element is the asXML representation of the corresponding table line.

Like structures, internal tables consist of multiple elements — in this case, a variable number of table lines. When serializing an internal table, the name of the line element is the name of the table line type in the ABAP Dictionary. If the line type is not defined there, the name of the line element is `item`. When deserializing, the element name is irrelevant. Any type of table (standard table, sorted table, hash table) is allowed. Tables will be sorted automatically, if required by the result type.

Now let's look at an example of how to handle internal tables. **Listing 6** shows how to serialize a simple internal table with three lines.

In **Listing 7**, you see the generated asXML from the string `result`. Note that the element name for the table line is `item`, because the table line type is not an ABAP Dictionary type.

Representing References, Referenced Data Objects, and Referenced Objects

As you have just seen, ABAP data that contains no references is straightforward to represent in asXML. However, this is not the case for ABAP data that *does* contain references (either to data objects or objects). The sidebar on the next page outlines a few of the problems associated with representing references in asXML, and hints at how these problems are addressed. As you can see, none of these challenges is insurmountable.

In this section, we will delve into the details of representing references and the data objects or objects that they reference.

Representing References

✓ *asXML Format Rule: References*

A reference is represented as an attribute "href" with a value that looks like "#key," where "key" identifies the referenced data object or object. If the reference is initial or is considered initial (see below), it is represented by nothing (that is, the element that contains the reference does not have the attribute "href" or any other content).

Until now, you have seen how serialization works when you need to handle data values only. When serializing references, you need to handle both the reference and the value to which it refers. For an element that represents a referenced data object or object, the value of the attribute `id` must be the key `key` (without the hashmark). A referenced data object or object is only serialized if the reference pointing to it will also be serialized.

Note that specific non-initial references to data objects are considered initial for the serialization, which we discuss in more detail shortly.

✓ *Value vs. Object Distinction*

As ABAP programmers, when we talk about data objects, we usually say something like, "The data object is an integer." But as described in the sidebar on the next page, you must represent a data object with its identity (represented by the key) and its dynamic type. Therefore, we distinguish between the value of a data object (which can be an integer and whose asXML representation is known, as described in the previous section) and the referenced data object itself. The latter is represented in asXML as an XML element with its key and dynamic type, and the asXML representation of the value as content. Similarly, we also distinguish between the value of an object and the referenced object itself.

Listing 8 shows a fragment of an ABAP program that serializes the data reference `ref_to_data`, together with the dynamically created data object.

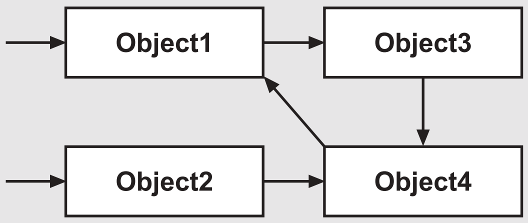
In **Listing 9**, you see the generated asXML in the string `result`. It also shows how the referenced data value is represented, which we discuss in the next section. For now, note that the attribute `id` defines the key `d1` for the referenced data object.

Now that you understand how to represent references in asXML, let's look at how to handle referenced data objects and referenced objects.

Listing 8: Serializing a Data Reference

```
DATA: ref_to_i TYPE REF TO i,
      ref_to_data TYPE REF TO data,
      result TYPE string.
CREATE DATA ref_to_i.
ref_to_i->* = 42.
ref_to_data = ref_to_i.
CALL TRANSFORMATION ID
  SOURCE REFERENCE = ref_to_data
  RESULT XML result.
```

The Challenges of Representing References in asXML



ABAP data that contains no references is tree-like, which fits the XML data model. However, ABAP data *with* references can have an arbitrary structure, like the cyclical graph shown in the diagram to the right.

Since two or more references could point to the same object, you cannot embed the asXML representation of the *object* in the asXML representation of each *reference* pointing to that object. If you did, you would lose the unique identity of the object. When deserializing, the XML parser would not know whether to produce two objects with the same value, or just one. To overcome this problem, you use the XML reference mechanism, where a key uniquely identifies the referenced object in the XML document. When serializing, the ABAP runtime automatically chooses a name for the key. When deserializing, the key can be an arbitrary name.* A reference to such a data object or object is represented by the key. Note that a reference can only use a key that is defined by the representation of a data object or object.

Another problem with representing references in asXML is that ABAP references are generally polymorphic. Therefore, the ABAP runtime cannot deduce the dynamic type or class of the referenced (data) object from the static type of the reference. For example, a reference to an integer could be statically typed as REF TO DATA, while its dynamic type is REF TO I. In order for the ABAP runtime to create the object when deserializing, the asXML representation of the referenced data object or object must also include its type or class name.

You are probably now asking, "If we cannot take an object's asXML representation and embed it in the asXML representation of the reference pointing to it, then where *do* we insert it?" Remember the optional "heap" section of an asXML document, which we introduced at the beginning of the article? The XML element that represents the referenced object is represented as an element in the "heap" section — i.e., as a subelement of "heap" in the namespace http://www.sap.com/abapxml.

* Well, not *completely* arbitrary. The name must consist of letters, digits, underscores (_), hyphens (-), and periods (.), and must start with either a letter or an underscore. This is in accordance with the production rule *Name* in the XML specification (see **www.w3.org/TR/REC-xml**). The values of the XML Schema types ID and IDREF must also follow the *Name* production rule.

Listing 9: asXML Result of a Serialized Data Reference

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <REFERENCE href="#d1"/>
  </asx:values>
  <asx:heap xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:int id="d1">42</xsd:int>
  </asx:heap>
</asx:abap>
```

Representing Referenced Data Objects

✓ *asXML Format Rule: Referenced Data Objects*

A dynamically created data object is represented as an element in the "heap" section. The element name is the XML Schema type name for the data object's type, the value of the attribute "id" is the data object's identity (i.e., defines its key), and the element content is the asXML representation of the data object's value.

A data object is *dynamically created* when created with the ABAP statement `CREATE DATA`. Referenced data objects that are not dynamically created are not serialized,[22] and references pointing to them are considered to be initial for serialization. For example, references to global or local variables will be represented as initial references.

The value of a data object is always an elementary data type, a structure, an internal table, or a reference. Thus you already know its asXML representation, as described in the previous sections. But the interesting aspect of a referenced data object is the XML Schema type name for the data object type. It follows these naming rules:

- If the type is an ABAP Dictionary type, the XML Schema type name is the name of the dictionary type with a special XML namespace for dictionary types. (See the appendix to this article for more information about XML namespaces.)
- If the type is identical to an ABAP elementary data type, the XML Schema type name is based on the values shown in Figure 2.
- If the type is defined in a global or local class (outside of methods), the XML Schema type name is the class name plus the type name, concatenated with a period (.) as the separator. An XML namespace differentiates between global and local classes with the same name.
- If the type is defined somewhere else (such as in a program or a form), the XML Schema type name is the type name. An XML namespace differentiates between different types with the same name.
- If the type does not have a name[23] and does not match an ABAP elementary data type, an exception of class `CX_XSLT_SERIALIZATION_ERROR` will be thrown when serializing.

As we discussed earlier, the asXML representation of elementary data values does not always exactly match the representation of the corresponding built-in data type of the XML Schema (as shown in Figure 2). Since for a referenced data object the XML Schema name is included in the asXML representation, it must precisely correspond to the ABAP type. Therefore, we must use XML Schema type names other than the built-in ones for some ABAP types. The generic ABAP types even require additional attributes, as shown in **Figure 3**.

The following points explain Figure 3 (where the type of the referenced data object is identical to an ABAP elementary data type) in more detail:

- ABAP types `STRING`, `I`[24], `F`, and `XSTRING` have the same representation rules as the corresponding built-in data types of the XML Schema. Therefore, the XML Schema type names for these ABAP types are the same as the built-in data types of the XML Schema.
- ABAP types `D` and `T` have representation rules that are similar to those for the built-in XML Schema data types `date` and `time`. Therefore, the XML Schema type names for these ABAP types are also `date` and `time`, but in the

[22] Here are two examples to help you understand why these values are not serialized: (1) referenced data objects that are not dynamically created would have to be deserialized as dynamically created data objects in specific situations; and (2) the combination of the ABAP statements `GET REFERENCE` and `ASSIGN…CASTING` might lead to a situation where the same piece of data could be viewed as differently typed by different references.

[23] If you create a data object with `CREATE DATA ref1 TYPE TABLE OF I`, the type of the referenced data object doesn't have a name, and it is not an elementary data type.

[24] As before, including the restricted integer types represented by the dictionary types `INT1` and `INT2`.

Figure 3 *XML Schema Type Names for Elementary ABAP Types*

ABAP Type	XML Schema Type Name	Additional Attributes
STRING	xsd:string	—
C	abap:string	maxLength
N	abap:digits	maxLength
I (INT1) (INT2)	xsd:int xsd:unsignedByte xsd:short	—
P	abap:decimal	totalDigits, (fractionDigits)
F	xsd:double	—
D, T	abap:date, abap:time	—
XSTRING	xsd:base64Binary	—
X	abap:base64Binary	maxLength
The namespace prefix “xsd” stands for the namespace **http://www.w3.org/2001/XMLSchema**. The namespace prefix “abap” stands for **http://www.sap.com/abapxml/types/built-in**.		

namespace `http://www.sap.com/abapxml/types/built-in`.

- ABAP types `C`, `N`, and `X` are generic. You make them concrete by providing the length (which cannot always be deduced by checking the content) as the value of the attribute `maxLength`.[25] The XML Schema data type `abap:digits` restricts the value space of the built-in XML Schema type `string` to a sequence of digits.
- ABAP type `P` is generic. You make it concrete by providing the total number of digits and (optionally) the number of decimal places (defaults to 0) as the values of the attributes `totalDigits` and `fractionDigits`. Allowed ranges for these values are 1 to 31 and 0 to 14, respectively. When serializing, the number of total digits is always odd.[26] When deserializing, the value of `totalDigits` (in this direction, always an even number) is incremented by one, and the result used as the number of total digits.

[25] The same attribute name is used in the XML Schema to constrain the value space of data types.

[26] In ABAP, you provide the number of total digits by providing the byte length necessary to store the decimal number. Since one byte can store two digits and half a byte is used for the sign, the total number of digits is always odd.

If the referenced data object is a reference itself, we consider it to have an elementary ABAP type (in order to find a corresponding XML Schema type name) in the following two cases: the type is `REF TO DATA` or `REF TO OBJECT`. In the former case, the XML Schema type name is `refData`; in the latter, it is `refObject` (both are in the namespace `http://www.sap.com/abapxml/types/built-in`).

Listing 10 shows an example of how to construct and serialize a reference to a decimal number. The XML Schema type name reflects the naming rules presented in Figure 3.

Listing 10: Serializing a Reference to a Decimal Number

```
TYPES: p_7_2 TYPE p LENGTH 4
       DECIMALS 2.
DATA: ref    TYPE REF TO p_7_2,
      result TYPE string.
CREATE DATA ref.
ref->* = '5320.15'.
CALL TRANSFORMATION id
  SOURCE REF = ref
  RESULT XML result.
```

Listing 11: asXML Result of a Serialized Reference to a Decimal Number

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <REF href="#d1"/>
  </asx:values>
  <asx:heap xmlns:abap="http://www.sap.com/abapxml/types/built-in">
    <abap:decimal totalDigits="7" fractionDigits="2" id="d1">
      5320.15
    </abap:decimal>
  </asx:heap>
</asx:abap>
```

In **Listing 11**, you can see the generated asXML in the string `result`.

Representing Referenced Objects

✓ ***asXML Format Rule: Referenced Objects***

A referenced object is represented as an element in the "heap" section. The element name is the XML Schema type name for the object's class name, the value of the attribute "id" is the object's identity (i.e., defines its key), and the content is the asXML representation of the object's value as children. Non-serializable objects are represented without any children[27]; references pointing to these objects are constructed as initial references during deserialization.

The ability to serialize/deserialize referenced objects is based on their class. Objects are serializable if and only if their class (directly or indirectly) implements the interface `IF_SERIALIZABLE_OBJECT`. If you are a class designer, we strongly recommend that you consider implementing this interface to give users the option of serializing the objects. We discuss this concept in more detail in the next section.

The XML Schema type name for referenced objects is the class name plus an XML namespace used to differentiate between global and local classes with the same name. (See the appendix to this article for more information about XML namespaces.) We discuss the asXML representation of object values in the next section.

Representing Values of Serializable Objects

At this point, you now have a fairly good understanding of the basics of asXML representation of referenced data objects and objects. There is still one element missing, however. How do you represent the *value* of the object, and how can class designers influence that representation? In this section, we will show you.

[27] Might be used for further serialization/deserialization extensions by SAP.

As you have learned, objects are only serializable if their class is serializable,[28] which means that the class directly or indirectly implements the interface `IF_SERIALIZABLE_OBJECT`. Class designers must explicitly declare a class to work under serialization. In making this decision for any class, be sure to consider the following points:

- Remember that all attributes determine the object *value*, also called the object *state*. By default, all attributes will be serialized, including private ones that could store hidden data. Thus the class designer must consider whether it is appropriate to use the default serialization or whether to customize it (more on this in a moment).
- Not all attribute values are legal. If you need to check any attributes when serializing or deserializing, you must also write the code to perform the corresponding test.
- Some objects (such as active handles) deal with information that only makes sense in the current context of the runtime environment. Thus it does not make sense to serialize these types of objects.

If you're an ABAP class designer, don't let these points discourage you from implementing the interface `IF_SERIALIZABLE_OBJECT`. By declaring your classes as serializable, you make your classes much more usable. So, this task needs to be simple and straightforward — and it is. The ABAP-XML serialization mechanism is:

- ☑ *Easy to use* — In simple cases, you just need to implement the associated interface.
- ☑ *Generic* — There is no restriction on the type of classes that can be serialized.
- ☑ *Safe* — A versioning mechanism is introduced to protect against class design changes.
- ☑ *Flexible* — You can control how class serialization and deserialization is performed. You can define whether a class uses the default serialization (which serializes/deserializes all attributes), or your own custom serialization (which serializes/deserializes specified attributes only).

[28] Actually, classes will never be serialized. Calling a class serializable simply means that its instances are serializable.

Now let's look at the details of representing object values, including how to define default versus custom serialization for the corresponding class.

Representing Object Values by Working with Object Parts

As a class designer, suppose you want to customize the serialization of your classes. You don't want to have to consider any private attributes that are defined in superclasses of your class — the class designers should have already taken care of the correct serialization of their attributes. Therefore, you really need to modify the serialization only for the attributes defined by your class.

For this reason, the object state is divided into *parts*, and class designers can decide, for each part, whether to use default serialization (which imposes no additional programming) or to customize the serialization. (We discuss both options in this section.) For each serializable class in the class hierarchy of the object, the corresponding *part* contains the instance attributes defined by that class, but not the inherited ones. Interface attributes belong to the part corresponding to the topmost class in the class hierarchy that implements the interface.

✓ *asXML Format Rule: Object Values*

The value of a serializable object is represented by a sequence of elements that represent the serializable parts of the object. The name of each element is the class name corresponding to the part. The element's content is the (default or customized) asXML representation of the part. If the part has a version, the element has an attribute "classVersion" with the version number as its value.

A class hierarchy can contain both a global class and a local class with the same name (although no class hierarchy can contain two local classes with the same name). Therefore, if the corresponding class is local, the name of the element representing the object part is preceded by `local` followed by a period (.).

The term *serializable* applies to parts as well. A part is serializable if, and only if, the corresponding class (directly or indirectly) implements the interface `IF_SERIALIZABLE_OBJECT`. Be aware that only the serializable parts of an object will be serialized!

When serializing, the order of serializable parts of the object is top-down. The part corresponding to the topmost serializable class is first, and the part corresponding to the class itself is last. When deserializing, the ABAP runtime first creates the object with initial values for all attributes (the object's constructor is not called[29]), and then sets the object's attributes by deserializing the parts. The order of the parts is not relevant.

In addition, an object part is said to have a *version* if the corresponding class defines a private class constant `SERIALIZABLE_CLASS_VERSION`, which, if defined, must be of type `I`. Class designers can increase the value of that constant if a change in the class could lead to incorrect deserialization of XML documents that were generated by serialization using a previous version of the class.

When deserializing, if the version number in the XML document doesn't match the version number in the class constant, an exception of class `CX_XSLT_DESERIALIZATION_ERROR` will be thrown. If both versions do not exist, or if both exist and the values are the same, then the versions are considered matching.

[29] If the ABAP runtime did call the constructor, it would need to do so without parameters. Since ABAP does not allow method overloading, serializable classes would need to have a constructor without parameters (or only optional ones). This restriction would clearly be severe enough to prevent many classes from being declared as serializable.

Default Representation of an Object Part

Declaring classes as serializable is a simple task that makes classes much more usable. You have already learned that in order to do so, you first have to implement the interface `IF_SERIALIZABLE_OBJECT`, which is an easy exercise because you don't need to implement any additional methods.

What else do you need to do? If you are satisfied with the default representation of the part (which should be sufficient for many classes), the answer is nothing. Here you see another reason for dividing the object state into parts. The class designers of the superclasses of your class could have chosen to use a custom representation of their parts. However, you are free to use the default representation, because you do not need to care about the correct serialization of any attributes defined in superclasses.

✓ *asXML Format Rule: Default Representation*

By default, a serializable object part is represented by a sequence of elements representing the instance attributes belonging to the part. The name of each element is the name of the attribute, and the element content is the asXML representation of the value of the attribute.

Let's examine the default representation. If the attribute is an interface attribute, the element name is the interface name plus the attribute name, concatenated with a period (.) as the separator. When serializing, all attributes are serialized in order, as defined in the class. When deserializing, the order of XML elements corresponding to attributes does not matter. Additional XML elements will be ignored,[30] and attributes without a corresponding XML element retain their values.

[30] Element names with a non-empty XML namespace will cause an exception of type `CX_XSLT_FORMAT_ERROR`.

Listing 12: Defining a Class and Serializing an Object with Default Part Representation

```
REPORT ZSPJ.

INTERFACE lif_1.
  DATA: a TYPE REF TO lif_1.
ENDINTERFACE.

CLASS lcl_1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES: if_serializable_object.
  PRIVATE SECTION.
    DATA: a TYPE i VALUE 1.
    CONSTANTS: serializable_class_version TYPE i VALUE 7.
ENDCLASS.

CLASS lcl_1 IMPLEMENTATION.
ENDCLASS.

CLASS lcl_2 DEFINITION INHERITING FROM lcl_1.
  PUBLIC SECTION.
    INTERFACES: lif_1.
  PRIVATE SECTION.
    DATA: a TYPE i VALUE 2.
ENDCLASS.

CLASS lcl_2 IMPLEMENTATION.
ENDCLASS.

DATA: result TYPE string.
DATA: object TYPE REF TO lcl_2.
CREATE OBJECT object.
object->lif_1~a = object.

CALL TRANSFORMATION ID
  SOURCE OBJECT_REF = object
  RESULT XML result.
```

In **Listing 12**, you see a class hierarchy definition and how to serialize the object reference.

The class `LCL_2` in program `ZSPJ` has two serializable parts, because the class `LCL_2` inherits from the serializable class `LCL_1`. The class `LCL_1` is serializable because it declares the interface `IF_SERIALIZABLE_OBJECT`. Because the interface `LIF_1` is declared in the class `LCL_2` (and not in any of its superclasses), the interface attributes are added to the object part corresponding to the class `LCL_2`. The interface attribute `LIF_1~A` holds a reference to the object itself. Note that all instance attributes of the classes are serialized, including the private attributes of superclasses.

Listing 13: asXML Result of an Object Serialized with Default Part Representation

```
<asx:abap xmlns:asx="http://www.sap.com/abapxml" version="1.0">
  <asx:values>
    <OBJECT_REF href="#o11"/>
  </asx:values>
  <asx:heap>
    <prg:LCL_2 id="o11"
         xmlns:prg="http://www.sap.com/abapxml/classes/program/ZSPJ">
      <local.LCL_1 classVersion="7">
        <A>1</A>
      </local.LCL_1>
      <local.LCL_2>
        <A>2</A>
        <LIF_1.A href="#o11"/>
      </local.LCL_2>
    </prg:LCL_2>
  </asx:heap>
</asx:abap>
```

In **Listing 13**, you see the generated asXML contained in the string `result`. The key for the object is `o11`. The element name representing the local class `LCL_2` in the program `ZSPJ` is `LCL_2` with the XML namespace `http://www.sap.com/abapxml/classes/program/ZSPJ`.

Custom Representation of an Object Part

> ✓ ***asXML Format Rule: Custom Representation***
>
> *A serializable object part corresponding to a class C is serialized to (or deserialized from) a sequence of elements representing the exporting (or importing) parameters of the private instance method SERIALIZE_HELPER (or DESERIALIZE_HELPER) if the method is defined in class C. The ABAP runtime invokes the method on the object being serialized (or deserialized). The name of each element is the name of the attribute, and the element content is the asXML representation of the parameter's value.*

While the default part representation is the easiest to use, it may not always be the most appropriate. You might want to use a custom representation of object parts if:

- The part is the topmost serializable part of the object, and you want the XML representation of the object to include attributes that belong to parts corresponding to non-serializable superclasses.
- You want to exclude some attributes from the XML representation.
- During deserialization, you want to test whether attribute values of the newly created object are legal.
- You want to completely change the way that specific attributes are represented.
- You want to support older asXML representations (i.e., you want to be able to deserialize from XML documents that were produced with a previous version of the class implementation).

- An interface attribute belongs to the part. The default representation would include elements that no `DESERIALIZE_HELPER` method (which you might want to implement in future versions) could deserialize.

When serializing, the exporting parameters are serialized in the defined order. When deserializing, the order of XML elements corresponding to parameters does not matter. Additional XML elements will be ignored,[31] and parameters without a corresponding XML element are considered to be not supplied.

To use a custom representation of an object part, the following conditions must be met (as enforced by a syntax check):

- Both methods `SERIALIZE_HELPER` and `DESERIALIZE_HELPER` must be defined.[32] If they are not defined, the default representation will be used.

- Both methods must be private instance methods.

[31] Element names with a non-empty XML namespace will cause an exception of type `CX_XSLT_FORMAT_ERROR`.

[32] SAP reserves the method names `SERIALIZE_REPLACE` and `DESERIALIZE_REPLACE` for future extensions.

- The method `SERIALIZE_HELPER` must only have `EXPORTING` parameters. The method `DESERIALIZE_HELPER` must only have `IMPORTING` parameters.

- Each parameter in the method `SERIALIZE_HELPER` must have a corresponding parameter in the method `DESERIALIZE_HELPER`. Additional parameters in the method `DESERIALIZE_HELPER` must be optional.

- The method `SERIALIZE_HELPER` must not have the parameter `SERIALIZABLE_CLASS_VERSION`. The method `DESERIALIZE_HELPER` may have the optional parameter `SERIALIZABLE_CLASS_VERSION`. If defined, it must be of type `I`.

If the method `DESERIALIZE_HELPER` defines a parameter `SERIALIZABLE_CLASS_VERSION`, the parameter will be supplied with the version number in the generated asXML (the value of the attribute `classVersion` in the XML element for the part), if present, and the standard version number check will not be performed.

Listing 14 shows how class designers can specify custom part representation for object serialization.

Listing 14: Defining Custom Part Representation for Object Serialization

```
CLASS lcl_1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES: if_serializable_object.
    DATA: pfli TYPE REF TO cl_spfli_persistent.
  PRIVATE SECTION.
    METHODS:
      serialize_helper
        EXPORTING
          value(carrid) TYPE s_carr_id
          value(connid) TYPE s_conn_id,
```

(continued on next page)

(continued from previous page)

```
      deserialize_helper
        IMPORTING
          value(carrid) TYPE s_carr_id
          value(connid) TYPE s_conn_id
        RAISING cx_os_object_not_found.
ENDCLASS.

CLASS lcl_1 IMPLEMENTATION.
  METHOD serialize_helper.
    IF pfli IS NOT INITIAL.
      carrid = pfli->get_carrid( ).
      connid = pfli->get_connid( ).
    ENDIF.
  ENDMETHOD.
  METHOD deserialize_helper.
    DATA: ca TYPE REF TO ca_spfli_persistent.
    ca = ca_spfli_persistent=>agent.
    pfli = ca->get_persistent( i_carrid = carrid
                               i_connid = connid ).
  ENDMETHOD.
ENDCLASS.

DATA: result TYPE string.
DATA: object TYPE REF TO lcl_1.
DATA: ca TYPE REF TO ca_spfli_persistent.

START-OF-SELECTION.

CREATE OBJECT object.
ca = ca_spfli_persistent=>agent.
object->pfli = ca->get_persistent( i_carrid = 'LH'
                                   i_connid = '2402' ).

CALL TRANSFORMATION ID
  SOURCE OBJECT_REF = object
  RESULT XML result.
```

The serializable class `LCL_1` has an attribute `PFLI` containing a reference to the persistent class[33] `CL_SPFLI_PERSISTENT`. Because the persistent class `CL_SPFLI_PERSISTENT` is not serializable, it is appropriate to include the business key of the persistent object in the XML representation instead. The business key consists of the attributes `CARRID` and `CONNID`, which are also the formal parameters of the `SERIALIZE_HELPER` and `DESERIALIZE_HELPER` methods.

[33] For more on persistent classes in ABAP, see the article "Write Smarter ABAP Programs with Less Effort: Manage Persistent Objects and Transactions with Object Services" (January/February 2002).

Listing 15: asXML Result of an Object Serialized with Custom Part Representation

```
<prg:LCL_1 id="o56"
     xmlns:prg="http://www.sap.com/abapxml/classes/program/ZSPJ">
  <local.LCL_1>
    <CARRID>LH</CARRID>
    <CONNID>2402</CONNID>
  </local.LCL_1>
</prg:LCL_1>
```

When serializing, the business keys are retrieved from the persistent object using the appropriate getter methods.

When deserializing, the business key is provided to the factory method `GET_PERSISTENT` of the instance manager `CA_SPFLI_PERSISTENT`. This method then returns the reference to the persistent object.

If the persistent object is not already in the persistent objects cache, it is loaded from the database. If the object is not found, an exception of class `CX_OS_OBJECT_NOT_FOUND` will be thrown, and the ABAP runtime will wrap this exception in an exception of class `CX_XSLT_DESERIALIZATION_ERROR`.

In **Listing 15**, you see the generated asXML of the referenced object (assuming the identity in the XML document is `o56`). Note that the part for the class consists of the formal parameters `CARRID` and `CONNID`.

Helpful Hints

Here are a few helpful hints to keep in mind as you set out to use ABAP-XML serialization in your own environment:

☑ Remember to use uppercase XML element names, if these names correspond to ABAP names such as component names.

☑ If you get an error during deserialization, check the attribute `TREE_POSITION` of the exception object to see which XML node has caused the error.

☑ Make your classes serializable whenever possible by implementing the interface `IF_SERIALIZABLE_OBJECT`.

Conclusion

The SAP Web Application Server now comes with a powerful feature that provides support for open, standards-based data exchange. ABAP-XML serialization enables the persistence and exchange of any SAP data, including data that contains object references. Its power derives in part from the underlying asXML format, which is the basis for representing ABAP data in XML. Together with the XSLT processor, you can now easily generate and read documents or communication data based on any XML-based standard.

In this article, we examined what you need to know to take advantage of the asXML format and

ABAP-XML serialization, focusing mainly on the following areas:

- ☑ An overview of the ABAP-XML serialization mechanism.
- ☑ How to invoke and control ABAP-XML serialization in your ABAP programs.
- ☑ asXML format rules for serializing and deserializing data types, objects, and references.
- ☑ Design options for controlling class serialization.

Regardless of your role as an SAP developer, we believe this information will simplify and improve how you implement data exchange for your SAP R/3 system. Class designers can define more usable classes by allowing serialization. ABAP programmers can serialize and deserialize data to and from XML with a single ABAP statement. And, because of the asXML format, XSLT programmers can now easily write even the most complicated XML transformations to exchange data to and from SAP R/3.

We hope this knowledge gives you the confidence to put the power of the asXML format and ABAP-XML serialization to work for you in your own SAP environment.

Stefan Bresch received his diploma in computer science from the University of Karlsruhe, Germany. Stefan joined SAP in 2000 as a member of the Business Programming Languages Group in Walldorf. Since then, he has been working on object persistence for ABAP (Object Services) and on XML serialization of ABAP data. Currently, he is working on the JDO (Java Data Objects) implementation (as part of the SAP J2EE Engine). He can be reached at stefan.bresch@sap.com.

Christian Stork studied mathematics and computer science at the Westfälische Wilhelms-University of Münster, Germany. He joined SAP in 1995 and worked for two years as a trainer, then returned to the university for his doctorate, specializing in algebraic geometry. In 2000, Christian rejoined SAP and became a member of the Business Programming Languages Group, where he works as a kernel developer. Since then, he has been responsible for the implementation and maintenance of the Object Services and the ABAP call of methods from XSLT. Currently, he is working on the JDO (Java Data Objects) implementation as part of the SAP J2EE Engine. He can be reached at christian.stork@sap.com.

Christoph Wedler received a degree in computer science from the University of Erlangen, Germany, in 1993. He joined SAP in 1999 and is now a member of the Business Programming Languages Group. Christoph is responsible for the integration of XML into the ABAP language and different parts of the ABAP runtime environment. He can be reached at christoph.wedler@sap.com.

Appendix: XML Namespaces

As introduced in the article "Mastering the asXML Format to Leverage ABAP-XML Serialization," XML Schema type names determine the dynamic type or class of a referenced data object or object. You use XML namespaces to differentiate between global and local ABAP types or classes with the same name. This appendix provides a reference for using XML namespaces for serialization.

The following table shows XML namespaces for ABAP types:

Type Defined in	XML Namespace
Dictionary	http://www.sap.com/abapxml/types/dictionary
Program <PRG>	http://www.sap.com/abapxml/types/program/<PRG>
Class pool <CPOOL>	http://www.sap.com/abapxml/types/class-pool/<CPOOL>
Type pool <TPOOL>	http://www.sap.com/abapxml/types/type-pool/<TPOOL>
Function pool <FPOOL>	http://www.sap.com/abapxml/types/function-pool/<FPOOL>
Function module <FUNC>	http://www.sap.com/abapxml/types/function/<FUNC>
Form routine <FORM> in program <PRG>	http://www.sap.com/abapxml/types/program.form/<PRG>/<FORM>
Form routine <FORM> in function pool <FPOOL>	http://www.sap.com/abapxml/types/function-pool.form/<FPOOL>/<FORM>
Method <METH> of global class <GCLS>	http://www.sap.com/abapxml/types/method/<GCLS>/<METH>
Method <METH> of local class <LCLS> in program <PRG>	http://www.sap.com/abapxml/types/program.method/<PRG>/<LCLS>/<METH>
Method <METH> of local class <LCLS> in class pool <CPOOL>	http://www.sap.com/abapxml/types/class-pool.method/<CPOOL>/<LCLS>/<METH>
Method <METH> of local class <LCLS> in function pool <FPOOL>	http://www.sap.com/abapxml/types/function-pool.method/<FPOOL>/<LCLS>/<METH>

This next table shows XML namespaces for ABAP classes:

Class Defined in	XML Namespace
Global	http://www.sap.com/abapxml/classes/global
Program <PRG>	http://www.sap.com/abapxml/classes/program/<PRG>
Class pool <CPOOL>	http://www.sap.com/abapxml/classes/class-pool/<CPOOL>
Function pool <FPOOL>	http://www.sap.com/abapxml/classes/function-pool/<FPOOL>

To allow XML namespaces to be used as URLs, the following mapping rules apply to the variable namespace parts (`<PRG>`, `<CPOOL>`, `<TPOOL>`, `<FPOOL>`, `<FUNC>`, `<FORM>`, `<METH>`, `<GCLS>`, and `<LCLS>`):

- Characters `a` through `z`, `A` through `Z`, `0` through `9`, - (hyphen), and _ (underscore) are never mapped. They retain their original form.
- Any other character `c` is mapped to `!hex(c)`, where `hex(c)` is the two-character representation of ASCII code of the character `c`.