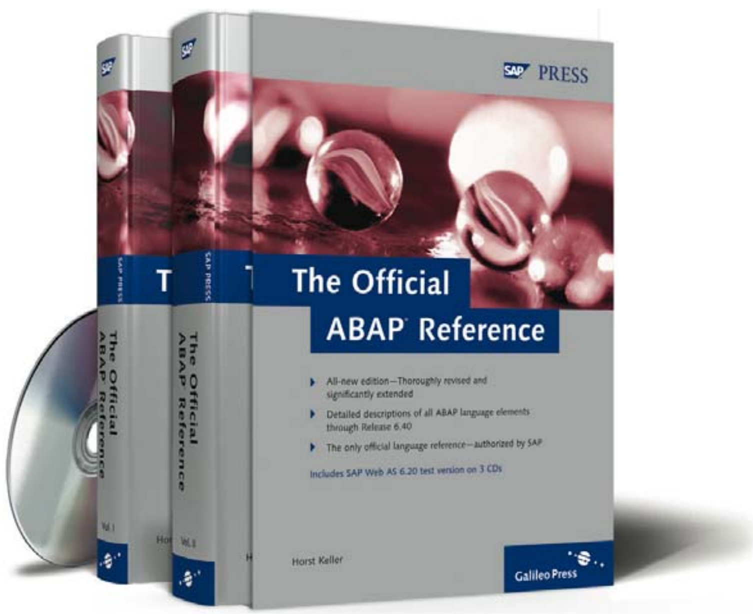
Horst Keller

The Official ABAP® Reference



SAP PRESS

Contents at a Glance

Contents

Part 3 Declarative Statements

Part 5 Calling and Exiting Program Units

Preface to the second edition

Barely two years have passed since the appearance of the first edition of the ABAP Reference. This new, second edition reflects the complete scope of ABAP elements produced for the current Release 6.40 and corrects and completely revises large sections of the previous text (see Section 1.8).

The most important innovations of Release 6.40 are the evolution of RTTI to RTTC and shared objects. In addition to the existing Run Time Type Information (RTTI), the Run Time Type Services (RTTS) now contain Run Time Type Creation (RTTC), which enables you to create and use any data types during the program runtime. This means that many ABAP developers have finally got what they wanted: They can create dynamic structures. The example in Section 9.2.6 shows, how this works. Shared objects are objects in the shared memory, which all programs of an application server can access. The related language elements and concepts are described in this book. Further important innovations in Release 6.40 are the introduction of simple transformations to serialize ABAP data to XML and vice versa, and the ABAP unit, which is a testing tool integrated in the language.

The SAP Web Application Server and the ABAP language has been a component of SAP NetWeaver since Release 6.30. SAP NetWeaver is an open integration and application platform which replaces and expands the preceding mySAP technology. In addition to the familiar ABAP environment, SAP Web Application Server—since Release 6.30—also contains an environment for the application development in Java. As this is not directly connected to ABAP language elements, this book will not explore the Java component of the Web Application Server.

As Mr Joachim Jacobitz, my co-author for the first edition, has since assumed different responsibilities within SAP, I am now the sole author of this book. In spite of this, however, Mr Jacobitz proofread each chapter with great enthusiasm and gave me many useful tips, for which I would like to sincerely thank him.

As the sole author of these voluminous two books, it is important for me to point out that this task would not have been possible without the "NetWeaver Developer Tools ABAP" group whose outstanding work I would like to emphasize at this point. As was the case in the first edition, the second edition was made possible by the constructive criticism and proofreading of many colleagues. In addition to the colleagues mentioned in the foreword to the first edition, the following people definitely deserve a mention in the second edition: Ulrich Brink, Christian Hansen, Ulrich Koch, Jürgen Lehmann, Mathias Müller, Helmut Prestel, Michael Redford, Christian Stork, Wolf Hagen Thümmel and Christoph Wedler. I would like to express my sincere gratitude to all other colleagues, who have not been named here, but whom I also bothered with queries or who gave me tips in one way or another.

On this occasion I would like to express my most sincere gratitude to my student assistants, Ms Agnieszka Chelminska, Ms Beata Kouchnir and Mr Christian Pretzsch, who enthusiastically and actively assisted me in creating and correcting the manuscript and the revision of the online documentation.

I would like to thank the involved people at Galileo Press and my editor Mr Florian Zimniak, for the fact that they enabled the rapid publication of the revised version of a book which I consider very important, and for their continuous pleasant cooperation.

For the English edition I would especially like to thank John Parker from UCG, Stefan Proksch from Galileo Press, as well as the students Kathrin Sturmhöfel, Marc Langen, and Dariusz Chelminski for their commitment during the technical editing of the text.

I owe special thanks to Ute for her patience and understanding for the fact that even after the appearance of the first edition, I continued to write this book.

Walldorf, Germany, May 2004
Horst Keller

Preface to the first edition

The introduction of ABAP Objects in Release 4.6 represented a significant step towards contemporary programming techniques for the ABAP programming language. Still, the next major challenge in the language's future development was already looming: to meet the demands of an international, Internet-driven programming environment, Web Application Server—as successor to the SAP Basis Module—had to be made Unicode-capable, and with it the ABAP programming language. This requirement was met in Release 6.10.

However, implementing Release 6.10 or 6.20 does not mean that you will be forced to modify your programs, as long as you do not convert the SAP system to Unicode. Nonetheless, regardless of whether a program will be used in a Unicode or non-Unicode system, the new features implemented together with Unicode represent an important step towards more stable, bug-free programming.

Accordingly, the introduction of Unicode was a particular impetus for writing this book. The online ABAP keyword documentation in the system grew with the language as new terms were added and existing terms remained. It has a section on "ABAP Objects" and—since Release 6.10—a section on "Unicode" as well, while the instructions for traditional reporting are still written as if every ABAP program were a report that is linked with a logical database. Because this type of documentation is acceptable for context-sensitive help—but no longer sufficient for full, integrated documentation—we resolved to write an ABAP Reference work to follow our introduction to SAP programming.

The goals we defined were to completely revise and standardize the ABAP documentation. This not only involved changing the outlines and summaries of the individual statements, but also their content and style. To this effect, we have introduced a new form of illustration, "pseudo syntax," to break down any statement from its basic form to its individual components. This book also marks the first time that the requirements for all operands that can appear in ABAP statements have been described exhaustively and uniformly in one go. Our intent is to give you a comprehensive description of all the ABAP language elements that are active in the current Release 6.20. This includes detailed descriptions of language elements that have been flagged as obsolete, but which are only prohibited in the context of ABAP Objects. This ABAP Reference will help you to analyze any existing ABAP program, as well as write your own. While it is not intended to be a "Best Practices" book, we hope that giving you precise descriptions of each statement will make their intent and purpose clearer than was previously the case.

Documenting a language boasting some 500 major language elements—while maintaining the same high quality and degree of detail throughout—was certainly no easy task, particularly since ABAP has been continually enhanced over the past 20 years and no element, once introduced, has ever been deleted (to ensure downward compatibility). This fact also explains the repeated delays in publishing this book, and we would like to thank all of you for waiting so patiently for its release. To compensate, the book you now hold in your hands documents the latest release of the ABAP language, and will remain valid for a long time to come. Despite our careful scrutiny of each and every chapter, we cannot discount the possibility that the book may still contain a few minor inconsistencies, and would be grateful to you for pointing out anything you might find. You can reach us at the SAP PRESS Web site under *www.sap-press.com*.

It would not have been possible to write this book in its current form without the direct and indirect help we enjoyed from a number of people during the authoring and proofreading processes. We would like to expressly thank Masoud Aghadavoodi, Thomas Bareiss, Adrian Goerler, Christian Jendel, Gerd Kluger, Björn Mielenhausen, Andreas Simon Schmitt and Christoph Stöck for their help. We thank Erhardt Vortanz for his assistance in writing the manuscript. We especially thank Andreas Blumenthal, Development Manager of the Business Programming Languages Group, for making this project possible in the first place and for more or less letting us decide what form it would take. We have the tireless Michael Demuth to thank for the fact that this book comes with an SAP system, this time as two CDs with SAP Web Application Server 6.10. Lastly, we would like to thank the staff of Galileo Press—particularly Iris Warkus and Florian Zimniak—for their help in proofreading the manuscript and for never giving up hope that we really would be finished at some point.

Horst Keller particularly thanks his wife Ute, who in the last few months nearly always found him hunched over his laptop, for her patience and willingness to sacrifice more of their already scarce free time together for yet another book project.

We hope you enjoy reading this ABAP Reference.

Walldorf, Germany, July 2002
Horst Keller and Joachim Jacobitz

Contents

Part 9 User Dialogs

Part 13 External Programming Interfaces

40 ABAP and XML

RELEASE 6.10

With the `CALL TRANSFORMATION` statement (see Section 40.4), you can convert ABAP data into the XML format and vice versa. Transformation programs of the following types are called: XSL transformations or Simple Transformations (as of Release 6.40).

40.1 XSL Transformations

An XSL transformation is a program in the repository that is written in XSLT (Extensible Stylesheet Language Transformation program) and used for the transformation of XML documents. When calling an XSL transformation using the `CALL TRANSFORMATION` statement, you can also directly convert ABAP data into XML and vice versa. For this purpose, a serialization or de-serialization is carried out implicitly.

In the case of transformations that use ABAP data as a source, the ABAP data is first serialized into a canonical XML representation (asXML, see 40.2), which then serves as the actual source for the XSL transformation. In the case of transformations that expect ABAP data as a result, the result of the XSL transformation is de-serialized into the ABAP data. As a prerequisite for de-serialization is the result must take the form of a canonical XML representation.

As of Release 6.10, the ABAP runtime environment contains an XSLT processor for executing the transformations. It supports almost all XSLT statements and provides enhancements (so-called extension instructions) such as the possibility to call ABAP methods from XSLT programs.

40.1.1 XSL Transformations in the Repository

XSL transformations that can be called with a `CALL TRANSFORMATION` statement must exist in the repository as XSLT programs. To create and edit XSLT programs in the Object Navigator in the ABAP Workbench, choose **Edit Object · More ... · Transformation** (or **XSLT program** before Release 6.40 and **XSL transformation** before Release 6.20) and choose XSLT program.

SAP delivers the identity transformation under the name ID. If you perform an identity transformation from XML to XML, the result is a copy of the source document. If you perform an identity transformation from ABAP to XML, this results in a canonical XML representation (asXML) of the ABAP data (explicit serialization). An identity transformation from

XML to ABAP transforms a canonical XML representation to ABAP data (explicit de-serialization).

40.2 Canonical XML Representation

The canonical XML representation is the format of an XML document that results from a serialization of ABAP data or that is required for a de-serialization. This format is also referred to as asXML (ABAP Serialization XML). The canonical XML representation supports all ABAP data types.

The asXML format is significant in the following cases:

- If you have written any XSL transformations of ABAP data into an XML format, the asXML format of the serialization result must be known.
- If you want to create external XML documents that can be de-serialized into ABAP data, they must be in an asXML format.

Note

The asXML format of serialized ABAP data or objects can be created and examined using the predefined identity transformation ID.

40.2.1 General asXML Format

The following lines show the general format of the canonical XML representation[1] without the XML header; line breaks and indents are included for clarification purposes only. A detailed example can be found in Section 40.5.

```
<asx:abap version = "1.0"
          xmlns:asx = "http://www.sap.com/abapxml">
  <asx:values>
    <bn1>...</bn1>
    ...
    <bnn>...</bnn>
  </asx:values>
  <asx:heap>
    ...
  </asx:heap>
</asx:abap>
```

The root element of an asXML documents is `abap` in the namespace (XML Namespace) `http://www.sap.com/abapxml`. The optional attribute `ver-`

1 The asXML format is a general format that cannot be completely defined with an XML pattern. The reason for this is that various ABAP types are referred to.

sion currently always has the value "1.0" and is intended for future enhancements of asXML. The root element abap must contain the sub-element values of the same namespace. The sub-elements bni of values represent the ABAP data objects that are specified as e1, e2, ... in the source addition to the CALL TRANSFORMATION statement or as f1, f2, ... in the result addition (see Section 40.4). The names of the elements bn1, bn2, ... are the names specified there in uppercase. The text contents of the elements <bn1>...</bn1> (or <bn1 ... />,), ... represent the contents of all named data objects with the exception of reference variables. The latter are represented by elements without text contents but with a special attribute (see Section 40.2.3.1). The optional element heap contains the contents of referenced anonymous data objects and objects (see Sections 40.2.3.2 and 40.2.3.3).

With the exception of the special cases in Table 40.1, the names of the element bn1, bn2, ... contain only capital letters. The names bn1, bn2, ... (or components of structures or objects, see Sections 40.2.2.2 and 40.2.2.3) specified in the source and result additions to the CALL TRANSFORMATION statement can only be used as (uppercase) names for XML elements if they consist solely of the characters "a" to "z", "A" to "Z", "0" to "9", or "_"; the first character must be a letter or "_". Other characters are replaced according to Table 40.1.

Character in the ABAP name	Replacement character in the XML name
ASCII character other than "a" to "z," "A" to "Z," "0" to "9," or "_" and character "0" to "9" as first character.	"_--hex(c)," where hex(c) is the two-digit hexadecimal representation of the ASCII code of the character c.
"/"	"_-"
"xml" as the first three characters in any combination of uppercase and lowercase	"x-ml" in a corresponding combination of uppercase and lowercase

Table 40.1 Replacement Rules for Charaters in ABAP Names

40.2.2 asXML Format for Named Data Objects with the Exception of Reference Variables

Named data objects, except for reference variables, are represented as the text contents of the elements <bn1>...</bn1>, ... The representation of named data objects in <bn1>...</bn1>, ... depends on the relevant ABAP data type.

40.2.2.1 Elementary Data Types

The asXML representation of elementary data objects with predefined ABAP types from Table 5.2 corresponds to the canonical representation of XML pattern data types (http://www.w3.org/TR/xmlschema-2/#built-in-datatypes, see Table 40.2), where date and time are represented according to ISO-8601, and binary data is represented using Base 64 encoding.

ABAP type	ABAP example	XML pattern type	XML example
`c`	`" Hi"`	`string`	`" Hi"`
`d`	`"20020204"`	`date`	`"2002-02-04"`
`f`	`-3.140...0E+02`	`double`	`"-3.14E2"`
`i, b, s`	`-123`	`int, unsignedByte, short`	`"-123"`
`n`	`"001234"`	`string (pattern [0-9]+)`	`"001234"`
`p`	`-1.23`	`decimal`	`"-1.23"`
`string`	`" Hello "`	`string`	`" Hello "`
`t`	`"201501"`	`time`	`"20:15:01"`
`x`	`"ABCDEF"`	`base64Binary`	`"q83v"`
`xstring`	`"456789AB"`	`base64Binary`	`"RweJqw=="`

Table 40.2 Canonical XML Representation (asXML) of Predefined ABAP Types

40.2.2.2 Structures

In asXML, the components of an ABAP structure are represented as a sequence of sub-elements of the structure element. The content of each sub-element corresponds to the canonical representation of the component value. The name of each sub-element is the name of the corresponding component. In the case of serialization, the sub-elements are represented in the order of the components in the structure. When the asXML representation of a structure is de-serialized, the order of the sub-elements is irrelevant and excess XML elements are ignored. Components of the structure for which there is no sub-element remain initial.

40.2.2.3 Internal Tables

In asXML, the rows of an internal table are represented as a a sequence of sub-elements of the table element. The content of each sub-element corresponds to the canonical representation of the row value. The name of a sub-element is irrelevant. If the canonical XML representation is created by serialization and the row type refers to the ABAP Dictionary, the name there is used; otherwise, the name `item` is used. Any table kind is allowed. During serialization, no information about the table kind is transferred to the XML document. If the target field of an XSL transformation is a sorted table, the rows are sorted accordingly during de-serialization.

40.2.3 asXML Format for Reference Variables and Referenced Objects

Anonymous data objects and instances of classes (objects) are addressed in ABAP exclusively by means of references in reference variables. The corresponding asXML format is made up of sub-elements of `values` for named reference variables (see Section 40.2.3.1) and of sub-elements of `heap` (see Sections 40.2.3.2 and 40.2.3.3) for the referenced objects. The link between the reference elements and the object elements is set up by means of an XML reference mechanism, whereby a referenced object in the same XML document is identified with a key. The dynamic type of the reference variables for the object elements under `heap` is specified when serialization takes place, so that de-serialization is unambiguous.

40.2.3.1 Named Reference Variables

A named reference variable is the only attribute of the corresponding sub-element of `values` that is displayed without textual content. An attribute of a reference variable has the name `href` and the content "#key", where `key` is the unique key of an object in the element `heap`. An element of an initial reference does not have a `href` attribute or any other content. During serialization, the ABAP runtime environment sets the key `key`; any key is possible for de-serialization.

40.2.3.2 Anonymous Data Objects

An anonymous data object that is a sub-element of `heap` is displayed as follows:

```
<asx:heap xmlns:nspace ...>
  <type id = "key" attr="...">...</type>
</asx:heap>
```

The value of a sub-element of this kind is displayed in the asXML display for named data objects (see Table 40.2) or for named reference variables (see Section 40.2.3.1). If the anonymous data object itself is a non-initial reference variable, it references a further element of `heap` according to the rules above. The element name `type` is the data type of the data object (or the dynamic type of the reference variables) specified as the XML schema type name from the name range `nspace` (see Table 40.3). Attributes `attr` may define technical characteristics of the type. The mandatory attribute `attri` contains the unique key `key` of the element, which is used to reference it from the display of the corresponding reference variables in `values` or `heap`.

The XML schema type name is constructed according to the following hierarchy:

1. If the data type of the data object is defined in the ABAP Dictionary, the XML schema type name is the name of the data type from the ABAP Dictionary in the corresponding name range (see Table 40.3).
2. If the data type is an elementary ABAP type, the XML schema type name is specified in Table 40.4.
3. If the data type is defined as a component of a global or local class or interface, the XML schema type name comprises the name of the class or interface and the name of the data type separated by a period (.). The corresponding name range (see Table 40.3) indicates whether the data type is a component of a global or local class or of an interface.
4. If the data type is a generic reference type defined with `REF TO data` or `REF TO object`, the XML schema type name is `refData` or `refObject`. Both of these have the name range `http://www.sap.com/abapxml/types/built-in`.
5. Otherwise, the XML schema type name is the name of a data type defined with `TYPES` and the corresponding name range (see Table 40.3) indicates where the data type is defined.

Before an XML schema type name can be constructed, the data type of the data object must have a name that can be used statically. If the data type only exists as a property of a data object and therefore only has a technical name (compare Section 6.1.4), a treatable exception takes place during serialization.

Table 40.3 indicates the name ranges for the XML schema type names; `types` in the first column stand for `http://www.sap.com/abapxml/types`. The name ranges indicate where a data type is defined. Characters other than "a" to "z," "A" to "Z," "0" to "9," "_" or "-" are displayed as "!hex(c)" in the names `prg`, `cpool`, `fpool`, `tpool`, `meth`, `func`, `form` and class, where hex(c) is the two-character hexadecimal display of the ASCII code for the character "c".

Name range	Location of definition
types/dictionary	ABAP Dictionary
types/program/prg	ABAP program prg
types/class-pool/cpool	Class pool cpool
types/type-pool/tpool	Type group tpool
types/function-pool/fpool	Function group fpool
types/function/func	Function module func
types/program.form/prg/frm	Subroutine frm in program prg
types/function-pool.form/fpool/frm	Subroutine frm in function group fpool
types/method/class/meth	Method meth of a global class class
types/program.method/prg/class/meth	Method meth of a local class class in program prg
types/class-pool.method/cpool/class/meth	Method meth of a local class class in a class pool cpool
types/function-pool.method/fpool/class/meth	Method meth of a local class class in function group fpool

Table 40.3 Name ranges for XML schema type names, whereby types in the first column stands for http://www.sap.com/abapxml/types

The following table lists the XML schema type names for elementary ABAP types. These are slightly different from the canonical XML schema data types from Table 40.2, since the data type of anonymous data objects must be specified in full. The name ranges nspace for the elementary ABAP types for anonymous data objects are either xsd="http://www.w3.org/2001/XMLSchema" for general schema types or abap="http://www.sap.com/abapxml/types/built-in" for special ABAP schema types for which some technical attributes must be specified.

ABAP type	XML schema type name	Attributes
c	abap:string	maxLength
d	abap:date	–
f	xsd:double	–

Table 40.4 XML Schema Type Names for Elementary ABAP Types

ABAP type	XML schema type name	Attributes
i, b, s	xsd:int, xsd:unsigned Byte, xsd:short	–
n	abap:digits	maxLength
p	abap:decimal	totalDigits, fractionDigits
string	xsd:string	–
t	abap:time	–
x	abap:base64Binary	maxLength
xstring	xsd:base64Binary	–

Table 40.4 XML Schema Type Names for Elementary ABAP Types (cont.)

The attribute `maxLength` defines the length for ABAP types with a generic length. The XML schema type `abap:digits` restricts the value range for an element to digits. The XML schema type `abap:decimal` specifies the length and fractional portions via the attributes `totalDigits` and `fractionDigits`. The length specification `totalDigits` defines the number of places between 1 and 31. In ABAP programs, the length of data objects of the type `p` is specified in bytes and the number of decimal places is calculated from 2×`len`-1 (see Table 5.2). This means that the value of `totalDigits` is always odd in serialization. During de-serialization, an even value of `totalDigits` is implicitly increased by one.

Instances of Classes

The instance of a class (object) as a sub-element of `heap` is displayed as follows:

```
<asx:heap xmlns:nspace ...>
    <class id = "key">
      <part classVersion = "...">
        <name>...</name>
      </part>
      ...
    </class>
  </asx:heap>
```

The element name `class` is the XML schema type name of the class for the object (or the dynamic type of the reference variables) from the name range `nspace` (see Table 40.5) in block capitals. The mandatory attribute `id` contains the unique key `key` of the element, which is used to reference it when the corresponding reference variables are displayed in `values`.

The sub-elements `<part>...</part>` contain the values of the instance attributes for individual object parts as sub-elements `<name>...</name>`. The individual object parts are defined by the classes in the current inheritance hierarchy that can be serialized (see below).

The name range for the class name indicates where the class is defined. Table 40.5 lists the possible name ranges; `classes` in the first column stands for `http://www.sap.com/apapxml/classes`. The substitution rule for the name ranges in Table 40.3 also applies to the names `prg`, `cpool` and `fpool`.

Name range	Location of definition
`classes/global`	Class library
`classes/program/prg`	Program `prg`
`classes/class-pool/cpool`	Class pool `cpool`
`classes/function-pool/fpool`	Function group `fpool`

Table 40.5 Name ranges for class names, whereby classes in the first column stands for http://www.sap.com/abapxml/classes

The values of the values of a class instance that can be serialized (instance attributes or output parameters for a special method, see below) are displayed as the content or as an attribute of `<name>...</name>` in the asXML display for named data objects (see Table 40.2) or for reference variables, wherein `name` is the name of an instance attribute or an output parameter in block capitals. If the object is an interface attribute, the name is preceded by the name of the interface separated by a period (.) in order to distinguish it from another class attribute of the same name. The substitution rules from Table 40.1 apply for the names.

The values of a class instance that can be serialized are defined by implementing the system interface IF_SERIALIZABLE_OBJECT in the class (see Section B.2). If the class does not implement the interface IF_SERIALIZABLE_OBJECT, the element `class` does not contain any sub-elements. All the instance attributes of a class in which the interface IF_SERIALIZABLE_OBJECT is implemented are serialized and de-serialized to the interface by default. You can change this behavior by declaring special utility methods (see below). Static attributes are neither taken into account during serialization nor during de-serialization (with the exception of the special constant SERIALIZABLE_CLASS_VERSION, see below).

Standard Behavior

If the class implements the interface IF_SERIALIZABLE_OBJECT, the element `<class>...</class>` contains at least one subelement `<part>...</part>`. These sub-elements correspond to individual object parts that can be serialized and contain the presentations of the instance attributes for the corresponding object part in an asXML format. An object part is defined by the class in which instance attributes are declared or in which an interface containing instance attributes is integrated. A object class that can be serialized contains an object part for itself as well as object parts for all superclasses in the current path in the inheritance tree, up to and including the class that implements the interface IF_SERIALIZABLE_OBJECT. The name `part` is the name of the class in question. If it is a local class, its name is preceded by the prefix `local` separated by a period (.) to distinguish it from a global class of the same name. Object parts of superclasses in which the interface IF_SERIALIZABLE_OBJECT is not implemented cannot be serialized and do not have a corresponding sub-object `part`. This means that a class in which the interface IF_SERIALIZABLE_OBJECT is not implemented (neither in the class itself nor in a superclass) creates a blank XML element `class` during serialization.

During serialization, the XML elements `part` of the object parts are created from the superclasses to the subclasses and the XML elements of the instance attributes are created as standard in the order in which they are declared in the class.

De-serialization creates an object in the corresponding class but the instance constructor is not executed. All instance attributes have their initial values or the start values specified with the `VALUE` addition for the `DATA` statement after the object creation. The values of the corresponding XML elements are entered in the instance attributes by default; the order of the object parts and attributes is irrelevant. Instance attributes without a corresponding XML element retain their value. Excess XML elements are ignored if they do not belong to a name range; otherwise, they create a treatable exception. When an element without `part` sub-elements is de-serialized, the system does not create an object but initializes the target reference variable.

If a class implements the interface IF_SERIALIZABLE_OBJECT, you can declare the private constant SERIALIZABLE_CLASS_VERSION of the type `i` in each object part; that is, each class involved in the inheritance tree. During serialization, the value of the constant is assigned to the attribute `classVersion` of the XML element `part`. A treatable exception is created during de-serialization by default if the value of the attribute does not match the value of the constant in the class specified. An object can only be de-serialized if the values match or if there is neither an attribute nor a constant. You can change this system behavior by declaring special utility methods.

Modified Behavior

By default, all the instance attributes for an object part are serialized as standard regardless of their visibility, and the version of the class is checked. To change this behavior, you can declare and implement the instance methods SERIALIZE_HELPER and DESERIALIZE_HELPER in the relevant class for each object part. These methods can only be declared as private instance methods in classes that implement the interface IF_SERIALIZABLE_OBJECT. If you declare one of the methods, you must also declare the other and the interface must be defined as follows for the syntax check:

- The method SERIALIZE_HELPER can only have output parameters, and the method DESERIALIZE_HELPER can only have input parameters with non-generic typing.
- There must be an input parameter of the method DESERIALIZE_HELPER with the same name for each output parameter of the method SERIALIZE_HELPER with the same typing. Additional input parameters for the method DESERIALIZE_HELPER must be optional.
- The method SERIALIZE_HELPER must not have an output parameter with the name SERIALIZABLE_CLASS_VERSION, and the method DESERIALIZE_HELPER can have an optional input parameter of this name that is of type `i`. This parameter is supplied with the value of the attribute `classVersion` of the element `part` during de-serialization, and the standard check on the version (see above) is skipped.

If the methods SERIALIZE_HELPER and DESERIALIZE_HELPER are declared in an object part, the instance attributes of the object part are not serialized and de-serialized. Instead, the method SERIALIZE_HELPER is executed during serialization and the values of all the output parameters are written in the asXML format as sub-elements to the corresponding element `part` in the specified order. Here, the name of a sub-element is the name of the corresponding output parameter in block capitals. The method DESERIALIZE_HELPER is called during de-serialization and the values of the sub-elements for the corresponding element `part` are transferred to the input parameters of the method with the same names. The order in which they appear is irrelevant and excess XML elements are ignored.

40.3 Simple Transformations

RELEASE 6.40

Simple Transformations (ST) is an SAP programming language for describing transformations between ABAP data and XML formats. ST is restricted to the two modes of serialization (ABAP to XML) and de-serialization (XML to ABAP) of ABAP data, which are most important for data integration. Like in the more general XSLT, transformations from ABAP to ABAP and XML to XML are not possible in ST.

In comparison with XSLT, the main advantages of ST programs are as follows:

- ST programs are declarative and thus easier to read.
- ST programs only have serial access to the XML data and are therefore very efficient even with large data volumes.
- ST programs describe serialization and de-serialization simultaneously; that is, ABAP data serialized in XML with ST can also be de-serialized with the same ST program.

Simple Transformations that can be called using `CALL TRANSFORMATION` must be in the repository. In the Object Navigator of the ABAP Workbench, you can create and edit ST programs by choosing **Edit Object · More · Transformation** followed by **Simple Transformation**.

A detailed description of the language ST goes beyond the scope of this ABAP reference. For more information, please turn to the corresponding online help. You can find an introductory example in Section 40.6.

40.4 Calling an XSL or ST Transformation

RELEASE 6.10

CALL TRANSFORMATION

Calling an XSLT or ST program.

Syntax

```
CALL TRANSFORMATION transformation
                    [PARAMETERS parameters]
                    [OBJECTS objects]
```

```
[OPTIONS options]
SOURCE source
RESULT result.
```

This statement calls the specified XSL transformation (XSLT) or a simple transformation (ST, as of Release 6.40). The source of the transformation is specified after `SOURCE`, and the result is stored as specified after `RESULT`. Use `PARAMETERS` and `OBJECTS` to pass parameters to the transformation. Possible transformation types are:

- From XML to XML (only for XSLT)
- From XML to ABAP (for XSLT and ST)
- From ABAP to XML (for XSLT and ST)
- From ABAP to ABAP (only for XSLT)

The last two types are available only as of Release 6.20.

Treatable Exceptions

The common superclass of all exception classes for CALL TRANSFORMATION is CX_TRANSFORMATION_ERROR (as of Release 6.40). The corresponding runtime errors cannot be caught. RELEASE 6.40

Exceptions with XSL-Transformations

All exception classes for XSL transformations are subclasses of CX_XSLT_EXCEPTION. RELEASE 6.20

If an error occurs when passing an XML document or if another error is reported by the XSLT processor, an exception defined by the class CX_XSLT_RUNTIME_ERROR is triggered. If the calling of an ABAP method from the XSLT program leads to an error, an exception defined by the class CX_XSLT_CALL_ERROR is triggered, whereby the attribute PREVIOUS points to the exception object of the original error.

If an XML document does not have the asXML format during the de-serialization, an exception defined by the class CX_XSLT_FORMAT_ERROR is triggered, where the attribute TREE_POSITION contains the error position. If, during serialization or de-serialization, invalid values or data types occur, exceptions defined by the classes CX_XSLT_SERIALIZATION_ERROR or CX_XSLT_DESERIALIZATION_ERROR are triggered, where the attribute PREVIOUS (if required) points to the exception object of the original error. The attribute TREE_POSITION contains the error position during de-serialization.

Exceptions with Simple Transformations
All exception classes for simple transformations are subclasses of CX_ST_ERROR.

40.4.1 Specifying the Transformation

Syntax of *transformation*

```
... trans | (name) ...
```

The name of the transformation can be specified either directly as `trans` or as a content of a character-type data object `name` in brackets. The specified transformation must exist as a XSLT program or as a simple transformation in the repository.

40.4.2 Transformation Source

Syntax of *source*

```
... { XML sxml }
  | {{bn1 = e1 bn2 = e2 ...}|(stab)} ...
```

40.4.2.1 Transformation of an XML Document

When you specify `XML sxml`, the XML document contained in `xsml` is transformed in such a way that `sxml` can have one of the following forms:

- Data object of type `string` and `xstring` or as a standard table with flat character-type row type
- Interface reference variable of type IF_IXML_ISTREAM which points to an iXML input stream (only for XSLT)
- Interface reference variable of type IF_IXML_NODE which points to an iXML nodeset (only for XSLT)
- Class reference variable of type CL_FX_READER, which points to an XML reader (only for ST)

Note
The interfaces IF_IXML_ISTREAM and IF_IXML_NODE are components of the "Stream" and "DOM" packages of the iXML Library delivered by SAP.

40.4.2.2 Transformation of ABAP Data

RELEASE 6.20

Use `bn1` = `e1`, `bn2` = `e2`, ... or `(stab)` to specify the ABAP data `e1`, `e2`, ... to be transformed.

- When calling an XSLT program, the ABAP data are serialized into the canonical XML representation, which is then used as source of the XSL transformation. Use `bn1`, `bn2`, ... to specify the names of the XML elements meant to represent the ABAP data objects in the canonical XML presentation.
- When calling a simple transformation, the names `bn1`, `bn2`, ... are used in the transformation to access the ABAP data in a written way.

Instead of using a static parameter list, you also can pass the data objects dynamically as value pairs in the columns of an internal table `stab` which has the type ABAP_TRANS_SRCBIND_TAB from the ABAP type group.

The following data objects cannot be serialized and trigger a treatable exception:

- Data objects of type `n`, whose current content does not exclusively consist of numbers.
- Data objects of type `p`, whose current content does not represent a valid packed number.
- Data objects of type `d` and `t`, whose current content contains leading or trailing blanks and at the same time uses the separators ("-" or ":") according to ISO-8601 for the presentation.
- Data reference variables pointing to data objects, whose data type has only a technical name (see Section 40.2.3).

Data reference variables pointing to data objects that were not created with `CREATE DATA` are treated as initial reference variables during the serialization.

40.4.3 Result of a Transformation

Syntax of *result*

```
... { XML rxml }
  | {{bn1 = f1 bn2 = f2 ...}|(rtab)} ...
```

40.4.3.1 Transformation into an XML Document

When you specify `XML rxml`, a transformation into an XML is executed; the document is then placed into `rxml`, where `rxml` can be one of the following:

- A data object of type `string` and `xstring` or a standard table with a flat, character-type row type.
- An interface reference variable of type IF_IXML_OSTREAM which points to an IXML output stream (only for XSLT).
- An interface reference variable of type IF_IXML_DOCUMENT which points to an IXML document (only for XSLT),
- A class reference variable of type CL_FX_WRITER, which points to an XML writer (only for ST).

Notes

- The interfaces IF_IXML_OSTREAM and IF_IXML_DOCUMENT are components of the "Stream" and "DOM" packages of the iXML Library delivered by SAP.
- If you use the data type `xstring` for `rxml`, then the result is stored in the UTF-8 character representation. This is helpful, if the resulting XML-document is to be stored in a file (see Chapter 32).

40.4.3.2 Transformation into ABAP Data

Use `bn1 = f1`, `bn2 = f2`, ... or `(rtab)` to specify the ABAP target fields `f1`, `f2`, ... into which you want the XML data to be transformed.

- When calling an XSLT program, the result of the XSL transformation into ABAP data objects is de-serialized, provided that it is a canonical XML representation. You should use `bn1`, `bn2`, ... to specify the names of the XML elements that represent the ABAP data objects in the canonical XML representation, and use `f1`, `f2`, ... to specify the ABAP data objects of the appropriate data type into which you want to de-serialize them.
- When calling a simple transformation, in the transformation, the names `bn1`, `bn2`, ... are used for write access to the ABAP data.

Instead of using a static parameter list, the data objects can also be passed dynamically as value pairs in the columns of the internal table `rtab`, which has the type ABAP_TRANS_RESBIND_TAB of the ABAP type group.

An XML element must be convertible into the respective ABAP data objects, where instead of the usual conversion rules (see Appendix A) the following restrictions apply:

- De-serialization into too short data objects of data types `c` or `n` must never lead to a loss of data, except when for data type `c` only leading and trailing blanks are concerned and for data type `n` only leading zeros.
- Data must never be lost due to the de-serialization into a data object of data type `p` with too few decimal places.
- Data must never be lost due to the de-serialization into a too short data object of data type `x`.
- Structures cannot be converted into elementary data objects.

If an XML element cannot be converted into the ABAP data object, a treatable exception is triggered.

When de-serializing into a reference variable, this variable must be the same as or more general than the dynamic type of the object stored in the XML document. The allocated ABAP objects or instances of a class are created during the de-serialization.

40.4.4 Parameters for an XSL Transformation

Syntax of *parameters*

```
... {p1 = e1 p2 = e2 ...}|(ptab) ...
```

Use this addition to pass ABAP data objects `e1`, `e2`, ... as parameters `p1`, `p2`, ... to an XSL transformation. In Release 6.10, the data objects `e1`, `e2`, ... must be character-type, as of Release 6.20 all elementary data objects and object references are allowed.

Instead of using a static parameter list, you also can pass the parameters dynamically as value pairs in the columns of the internal table `ptab` which has the type ABAP_TRANS_PARMBIND_TAB from the ABAP type group.

The specified parameters must be defined in the XSL transformation as input parameters as follows:

```
<xsl:param name="..." type="..."/>
```

For the attribute `name`, enter the parameter name in uppercase. For the optional attribute `type`, specify one of the type indicators `string`, `num-`

ber, boolean, xstring, nodeset or object(...), where you must enter the name of a global ABAP class in the brackets after object.

If no type is specified in the XSL transformation, the data types of elementary parameters are mapped to XSL types according to Table 40.6.

ABAP Data Type	XSL Parameter Type
c, d, n, string	string
i, s, b, f, p	number
x, xstring	string, where the content is presented to the base of 64

Table 40.6 Mapping of ABAP Data Types on XSL Parameter Types

If during the XSL transformation the XSL types shown in Table 40.6 are specified explicitly, you must enter the matching elementary ABAP parameters which can be converted into the XSL type:

- The XSL type boolean expects ABAP parameters of the type c with the length 1. A space is interpreted as "false" and a different character is interpreted as "true."
- The XSL type xstring expects ABAP parameters of the type x or xstring and the display of the content is hexadecimal.
- The XSL types nodeset and object expect an object reference variable pointing to a class instance. The type nodeset expects appropriate object properties.

If a parameter does not match the XSL type, an untreatable exception is triggered. If a parameter defined in the XSL transformation is not passed, it is set to the default value in the transformation. A specified parameter that is not defined in the XSL transformation is ignored.

Note

The XSL types string, number, boolean and nodeset are XSL standard types, whereas xstring and object are special SAP extensions. The type xstring allows a hexadecimal display of byte chains instead of the presentation to the base of 64. The type object enables you to call ABAP methods from the XSL program.

40.4.5 Passing External Objects to an XSL Transformation

Syntax of *objects*

```
... {o1 = e1 o2 = e2 ... }|(otab) ...
```

You can use this addition to pass object references `e1`, `e2`, ... as external objects `o1`, `o2`, ... to an XSL transformation where you can call their methods.

Instead of using a static parameter list, you can also pass the objects dynamically as value pairs in the columns of the internal table `otab` which has the type ABAP_TRANS_OBJBIND_TAB from the ABAP type group.

Note

RELEASE 6.20

As of Release 6.20, the addition `OBJECTS` is obsolete and external objects are treated as parameters. Therefore, object references should be passed with the addition `PARAMETERS` (see Section 40.4.4).

40.4.6 Controlling the Transformation

RELEASE 6.40

Syntax of *options*

```
... a1 = e1 a2 = e2 ...
```

You can use this addition to specify the values `e1`, `e2`, ... for additional control options `a1`, `a2`, ... of the transformation. The values `e1`, `e2`, ... must be of the type `c` or `string`.

For `a1`, `a2`, ... you can specify the following values:

- `XML_HEADER` to control the output of the XML header in case of a transformation to XML and in case of storage in a data object of the type `c`, `string` or in an internal table.

Possible values	Meaning
`no`	No output of an XML header
`without_encoding`	Output of an XML header without specification of the encodings
`full`	Default setting, output of an XML header with specification of the encoding

- `DATA_REFS` to control the output of data references in case of a transformation from ABAP to XML.

Possible values	Meaning
`no`	Default for ST, no output of data references

Possible values	Meaning
heap	Default for XSLT and only possible there; output of referenced data as sub-elements of the asXML elements <asx:heap>.
embedded	Output of referenced data with the reference

- INITIAL_COMPONENTS to control the output of initial structure components in case of a transformation from ABAP to XML.

Possible values	Meaning
include	Default setting, output of initial components of structures
suppress	No output of initial components of structures

40.5 Example of an XSL Transformation

This example shows the serialization of data objects in a string xmlstr using the identical transformation ID. A date field date, a time field time, and a data reference variable dref1 are serialized. The data reference variable points to an anonymous object reference variable, which in turn points to an object of the class c2. Objects serialized in this way can be stored persistently, for example in a data cluster. After the objects are imported from where they are stored, they are de-serialized into further data objects. Following de-serialization, dref2 points to another anonymous reference variable, such as dref1. This anonymous data object and the instance of the class c2 to which it points are generated during the de-serialization.

```
PROGRAM xmltst.

CLASS c1 DEFINITION.
  PUBLIC SECTION.
    INTERFACES if_serializable_object.
  PROTECTED SECTION.
    DATA carriers TYPE TABLE OF scarr.
ENDCLASS.

CLASS c2 DEFINITION INHERITING FROM c1.
  PUBLIC SECTION.
    METHODS constructor.
  PRIVATE SECTION.
    DATA lines TYPE i.
    METHODS: serialize_helper
```

```
                EXPORTING count TYPE i,
              deserialize_helper
                IMPORTING count TYPE i.
ENDCLASS.

CLASS c2 IMPLEMENTATION.
  METHOD constructor.
    super->constructor( ).
    SELECT * UP TO 2 ROWS
           FROM  scarr
           INTO  TABLE carriers.
  ENDMETHOD.
  METHOD serialize_helper.
    count = LINES( carriers ).
  ENDMETHOD.
  METHOD deserialize_helper.
    lines = count.
  ENDMETHOD.
ENDCLASS.

DATA: oref   TYPE REF TO object,
      dref1  LIKE REF TO oref,
      xmlstr TYPE string,
      date   TYPE d,
      time   TYPE t,
      dref2  LIKE dref1.

...

CREATE DATA dref1 LIKE oref.
CREATE OBJECT dref1->* TYPE c2.

CALL TRANSFORMATION id
                    SOURCE xmldat = sy-datum
                           xmltim = sy-uzeit
                           ref    = dref1
                    RESULT XML xmlstr.

EXPORT obj = xmlstr TO DATABASE indx(hk)
                    ID 'OBJECT'.

...

IMPORT obj = xmlstr FROM DATABASE indx(hk) ID 'OBJECT'.

CALL TRANSFORMATION id
                    SOURCE XML xmlstr
                    RESULT xmldat = date
```

```
                              xmltim = time
                              ref    = dref2.
```

The XML document generated in the serialization has the content described below. In this description, line breaks and indents have been added. The element `values` contains the asXML representations of the three transferred data objects (see Section 40.2). In the names `X-MLDAT` and `X-MLTIM`, "xml" has been replaced according to Table 40.1. The attribute `href` of the element `REF` uses the key "d1" to refer to the representation of the corresponding anonymous data object in the element `heap`. This uses the key "o3" to refer to the representation of the instance of the class `c2`, which is also in the element `heap`. This representation is divided into the object parts for the classes `c1` and `c2`. The object part for `c1` contains the representation of the double-line structured internal table `carriers`. The object part for `c2` contains the representation for the output parameter `count` of the method SERIALIZE_HELPER.


```
<?xml version="1.0" encoding="iso-8859-1" ?>
<asx:abap xmlns:asx="http://www.sap.com/
abapxml" version="1.0">
  <asx:values>
    <X-MLDAT>2003-04-15</X-MLDAT>
    <X-MLTIM>14:57:53</X-MLTIM>
    <REF href="#d1" />
  </asx:values>
  <asx:heap
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:abap="http://www.sap.com/abapxml/types/built-in"
      xmlns:cls="http://www.sap.com/abapxml/classes/global"
      xmlns:dic="http://www.sap.com/abapxml/types/dictionary">
    <abap:refObject href="#o3" id="d1" />
    <prg:C2
      xmlns:prg=http://www.sap.com/abapxml/classes/program/
        XMLTST id="o3">
      <local.C1>
        <CARRIERS>
          <SCARR>
            <MANDT>000</MANDT>
            <CARRID>AA</CARRID>
            <CARRNAME>American Airlines</CARRNAME>
            <CURRCODE>USD</CURRCODE>
            <URL>http://www.aa.com</URL>
          </SCARR>
          <SCARR>
```

```
            <MANDT>000</MANDT>
            <CARRID>AB</CARRID>
            <CARRNAME>Air Berlin</CARRNAME>
            <CURRCODE>DEM</CURRCODE>
            <URL>http://www.airberlin.de</URL>
          </SCARR>
        </CARRIERS>
      </local.C1>
      <local.C2>
        <COUNT>2</COUNT>
      </local.C2>
    </prg:C2>
  </asx:heap>
</asx:abap>
```

40.6 Simple Transformation Example

Serialization of a nested structure. In the following ABAP program section, a nested structure `struc1` is serialized to `xml_string` with the Simple Transformation ST_TRAFO and de-serialized with the same transformation.

```
DATA: BEGIN OF struc1,
        col1(10) TYPE c VALUE 'ABCDEFGHIJ',
        col2     TYPE i VALUE 111,
        BEGIN OF struc2,
          col1 TYPE d VALUE '20040126',
          col2 TYPE t VALUE '084000',
        END OF struc2,
      END OF struc1.

DATA: xml_string TYPE string,
      result LIKE struc1.

TRY.

    CALL TRANSFORMATION st_trafo
      SOURCE para = struc1
      RESULT XML xml_string.

    ...

    CALL TRANSFORMATION st_trafo
      SOURCE XML xml_string
      RESULT para = result.

  CATCH cx_st_error.
```

```
    ...
ENDTRY.
```

The Simple Transformation ST_TRAFO has the following form:

```
<?sap.transform simple?>
<tt:transform template="temp"
    xmlns:tt="http://www.sap.com/transformation-templates"
    version="0.1">

  <tt:root name="PARA"/>

  <tt:template name="temp">
    <X>
      <X1>
        <tt:value ref="PARA.COL1" />
      </X1>
      <X2>
        <tt:value ref="PARA.COL2" />
      </X2>
      <X3>
        <X1>
          <tt:value ref="PARA.STRUC2.COL1" />
        </X1>
        <X2>
          <tt:value ref="PARA.STRUC2.COL2" />
        </X2>
      </X3>
    </X>
  </tt:template>

</tt:transform>
```

The transformation consists of a template `temp` that defines the structure of the XML document and establishes relationships between value nodes and components of the structure. The result of the transformation is as follows (line breaks and indentations were inserted for clarification purposes):

```
<X>
  <X1>ABCDEFGHIJ</X1>
  <X2>111</X2>
  <X3>
    <X1>2004-01-26</X1>
    <X2>08:40:00</X2>
```

```
  </X3>
</X>
```

The conversion of the elementary data types is the same as for asXML (see Table 40.2). The reverse transformation generates the same content in the structure `result` as in `struc1`.

Index

B

C

D

G

H

I

U