ABAP 252: ABAP – XML Mapping
Learning Objectives

As a result of this workshop, you will:

■ learn about the role of XML in open system integration (e.g. in Web Services)
■ know the ABAP tools for basic XML processing
■ understand SAP’s approach to data / XML mapping (in relation to other approaches, e.g. JAXB)
■ be able to write mappings between ABAP data structures and XML …
  ◆ … using XSLT
  ◆ … using Simple Transformations
■ know when to use what
### Intro

- XML in Open Systems
- XML in ABAP
- ABAP – XML Mapping: Concepts
- ABAP – XML Mapping with XSLT
- Simple Transformations
- Summary
Intro

XML in Open Systems

XML in ABAP
ABAP – XML Mapping: Concepts
ABAP – XML Mapping with XSLT
Simple Transformations
Summary
From HTML to XML

The Web is evolving …

- … from **HTML** – a *fixed* language for human viewing
- … to **XML** – a *generic* language for machine processing

- **People** navigate through information in the form of **HTML** documents.
- **Machines** exchange *data and metadata* in the form of **XML** documents.

- If *information* is *data* with a *meaning* (interpreted data), then the enrichment of data with metadata in XML can be seen as an attempt to *approximate information*.

- Nevertheless, an XML document means nothing, regardless of the amount of metadata, without a data processing system interpreting it correctly.

- Hence, we are interested in uses of XML that support automated processing of data based on *application-specific* metadata.
Example: Slashdot.org

**RSS feed (news syndication)**

URI in:

http://slashdot.org/it.rss

XML out:

```xml
<item rdf:about="http://it.slashdot.org/article.pl?sid=04/03/21/204210"
     title="Microsoft Renovates Office Suite as a Web Service"
     url="http://it.slashdot.org/article.pl?sid=04/03/21/204210"
     description="According to an article in EcommerceTimes, Microsoft is</p>

... more effective viewing with **RSS aggregators**

- “RSS” stands for “RDF Site Summary” (or a few other meanings, depending on the version level). RDF, the Resource Description Framework, is an XML based language for metadata.

- An RSS reader, or aggregator, supports more effective ways of monitoring a personal set of RSS feeds than can be achieved by manually tracking web pages with a normal HTML browser.

- There is a large number of RSS aggregators, many of them freely available. A free reader for Windows is the Sharp Reader (http://www.sharpreader.net/index.html).

- The predominant application area of RSS at the moment is news and weblogs. For example, the **SAP Developer Network** has extensive RSS support – see https://weblogs.sdn.sap.com/pub/q/weblogs_rss?x-ver=1.0.

- However, the extensible nature of RSS 1.0 qualifies it for many kinds of summary/reporting tasks. Using BSP (Business Server Pages) or HTTP directly, you could export regular **system reports** (e.g., batch logs) from your WebAS in RSS format.
Example: Xignite Financial Web Services

Realtime currency exchange rates

URI in:

http://www.xignite.com/xCurrencies.asmx/GetRealTimeCrossRateTable?
Symbols=USD%2CEUR%2CJPY%2CGBP&InvokeButton=Invoke&
xMethod=GET

XML out:

```xml
<From>
  <Symbol>USD</Symbol>
  <Name>Dollar</Name>
</From>
<ExchangeRates>
  <ExchangeRate>
    <Outcome>Success</Outcome>
    <Delay>0</Delay>
    <To>
      <Symbol>EUR</Symbol>
      <Name>Euro</Name>
    </To>
    <Date>8/23/2004</Date>
    <Time>6:04:00 PM</Time>
    <Rate>0.823194996043398</Rate>
    <Text>1 Dollar = 0.82318 Euro</Text>
  </ExchangeRate>
</ExchangeRates>
```

Call a business service from ABAP ...

Or offer one.
Example: Amazon Web Services

Transactions on the Amazon marketplace

URI in:

http://xml.amazon.com/onca/xml3?locale=us&t=te&dev-t=te &KeywordSearch=ABAP&mode=books&sort=+daterank &offer=All&type=lite&page=1&f=xml

XML out:

```
<Details url="http://www.amazon.com/exec/obidos/A8IN/15922901 2#link_code=xm2">
  <ASIN>1592290111</ASIN>
  <ProductName>ABAP Objects: The Official Reference</ProductName>
  <Catalog>Book</Catalog>
  <Authors>
    <Author>Horst Keller</Author>
    <Author>Jouchoin Jacobitz</Author>
  </Authors>
  <ReleaseDate>05 October, 2001</ReleaseDate>
  <Manufacturer>SAP Press</Manufacturer>
</Details>
```

Amazon Web Services are offered both in RPC style (SOAP) and REST style. The REST (REpresentational State Transfer) style is simply the well-known interaction pattern of the web, based on URIs and XML (instead of HTML).
From a programmer’s perspective, XML is ...

- a **generic data model** ("XML Infoset") for trees with
  - ordered, structured nodes ("elements")
  - unordered, unstructured nodes ("attributes")

- a **generic syntax** ("markup") for representation of trees

- the basis for an open set of standards and tools
  - parsing: text ➔ tree; rendering: tree ➔ text
  - typing, validating: XSchema, RELAX NG
  - querying, transforming: XPath, XSLT

→ a **generic data structure concept** for programming languages
  - primitive: DOM, SAX
  - data binding: JAXB

For all kinds of processing except input/output, XML documents should be thought of in terms of the data model, not in terms of their textual representation. I.e., trees of nodes, not sequences of tags.

The XML Information Set is the data model defined along with XML itself. Unfortunately, there are slight variations in the data models used by other XML standards. The XPath data model is very similar, but not identical to the XML Infoset.

The DOM is not a data model, but an object-oriented *interface* to another variation of the XML Infoset.

Though XML is “generic”, it is not a completely general model of trees. In particular, it lacks the concept of unordered, structured nodes. (I.e., there are no structured attributes.)

JAXB is available as part of the Java Web Services Developer Pack (WSDP) 1.4 (http://java.sun.com/xml/jaxb/). Version 2.0 is being developed through the Java Community Process℠ program under JSR-222.
Reasons for using XML

- handle document-like data (e.g. forms)
- represent data under heavy schema evolution
  - extensibility, flexibility, persistence
- combine heterogeneous data
  - "framework" applications, middleware, ...
- platform-independent modeling
  - "repositories", e.g. SAP WebDynpro
- ignorance / abuse
  - too lazy to design & implement adequate object model
  ...
- open communication format

Document-like data typically have a heterogeneous, flexible structure which is hard to capture with an object model.

"Schema evolution" means that the structures of an application tend to change perpetually. This is also hard to capture with an object model, especially across multiple systems.

The mechanism of XML Namespaces supports the combination of data from several application areas. E.g., an XML based message middleware typically defines an "envelope" consisting of a few elements in its own namespace in which arbitrary content may be enclosed.

An application that uses XML extensively as a platform-independent modeling language is SAP WebDynpro: Both ABAP and Java classes are generated from the same XML source.
Using XML

- **Problems solved**
  - standard syntax
  - standard tools
  - can model almost anything

- **Problems gained**
  - performance drain
  - no semantics
  - unclear relationship with data structures (data conversion)
    → Part II (XML from/to ABAP)
  - schema inflation
    - "Business ML" by standards body X
    - "Business ML" by vendor Y
    - Proprietary format by application Z

Co-existence of different schemas for the same application area is inevitable. Therefore, XML-to-XML transformations must occur in communication middleware or at XML-based interfaces.

Data structures in application-language interfaces can also be considered "schemas", leading to XML-to-data / data-to-XML transformations.
Tree transformations with XSLT

(a) XML tree to XML tree (data-centric)
(b) XML tree to text (document-centric)

Data-centric tree transformations: XML ⇔ XML
- adapt different XML formats
- e.g. in B2B scenarios, “message brokering”

Document-centric tree transformations: XML ⇔ HTML / WML / Text
- e.g. produce HTML in Business Server Pages (BSP)
- e.g. generate program sources from an XML repository

In a data-centric application, both source and result represent data, usually not intended for direct viewing by a user. The goal of the transformation is to obtain data in a different format.

In a document-centric transformation, the source is either data or a document in an abstract format; the result is a document in a concrete format.

The development of XSL was originally motivated by document-centric applications, hence the term "stylesheet". Then it was recognized that the structure-transforming part of XSL was useful by itself, and it was separated as XSLT.
Transforming XML with XSLT (2)

**XSLT** is ...

- a high-level tree-transformation language
- with XML syntax
- declarative
  - no "state"
- rule-based
  - pattern matching against source tree
- functional
  - source tree navigation by XPath expressions
- compositional
  - result tree construction by XSLT instructions mixed with literal XML fragments
- "the SQL of the Web"

* "Extensible Stylesheet Language / Transformations"

XSLT (http://www.w3.org/TR/xslt) and XPath (http://www.w3.org/TR/xpath) are W3C recommendations, current version 1.0 from 11/1999. XSLT 2.0 (quite stable) and XPath 2.0 (not stable) are under development (Working Draft 07/2004).

“Declarative” means: Say what you want to do, not how you want to do it. The fundamental operation is function [“template”] invocation, not variable assignment.

“Rule-based” means: In any given context, the XSLT engine picks the appropriate transformation rule from the program (by matching the context against the rule’s pattern) and instantiates it, using the rule body as a template.

XSLT’s concrete syntax is in XML format. Clumsy to write, but this way

- it mixes conveniently with XML result fragments
- XSLT can be processed (esp.: generated) by XSLT

Unnecessarily, there is another XML query language, XQuery, under development by the W3C. Anything done in that language could also be done in XSLT / XPath or in natural extensions thereof.
XPath is a simple expression language with non-XML syntax.

The basic types of values on which XPath operates are: String, number, boolean, and nodeset. A nodeset is an unordered, duplicate-free collection of nodes from an XML document. XPath and XSLT deal only with the semantic values of XML documents (trees), not with their textual representations.

The most important type of XPath expression is the location path, which evaluates to a nodeset. A location path consists of a sequence of location steps. A location step consists of an axis (e.g. child, parent, following-sibling), a node test, and optional predicates. The default axis is child, so the expression book yields all children of the current node named “book”.

The location path in the example contains ten location steps, six of which are nested in predicates. It finds all authors of chapters in books edited by John Smith which contain a section whose heading contains the phrase “DOM”.

XPath example: tree navigation

```
book[editor[name/last='Smith' and name/first='John']] //section[contains(@head, 'DOM')]
/ancestor::chapter[1]/author
```

head = The Pain of DOM Programming
Intro
XML in Open Systems

XML in ABAP

ABAP – XML Mapping: Concepts
ABAP – XML Mapping with XSLT
Simple Transformations
Summary
The iXML package (since 4.6D): features

- implemented in kernel (C++)
- encapsulated in ABAP proxy classes
- general XML parser / renderer
- event-based parser (~ SAX)
- DOM (Document Object Model)
  - superset of DOM level 1 (incl. XML namespaces)
- Validation
  - DTD (since 6.10)
  - not XMLSchema

The iXML library is implemented in the SAP kernel and is made available in ABAP Objects by means of ABAP proxy classes. Parsing is fast because it is done completely in the kernel. ABAP objects corresponding to DOM nodes are not created until they are accessed from ABAP.
XML in ABAP: iXML Package (2)

iXML interfaces in ABAP Objects

- OO design: interfaces, inheritance, factories
- main factory class CL_IXML
- ~40 Interfaces
  - IF_IXML_PARSER, IF_IXML_NODE, ...
- documentation: SAP Library
  - “SAP NetWeaver Components”
  - “SAP Web Application Server”
  - “Basis Services / Communication”
  - “XML Library”
- used by many SAP applications

The design of the iXML package is completely object oriented. In particular, interfaces, inheritance and the factory pattern are frequently used. The main factory interface IF_IXML is the origin (direct or indirect) of all iXML objects.

The iXML main factory class implements the singleton pattern; i.e., when the call cl_ixml=>create( ) is done repeatedly, it yields the same object.

All iXML objects are integrated into the general ABAP Objects garbage collection mechanism.

The SAP Library is found in the Help menu. The iXML documentation is organized in several “packages” (DOM, Parser, Stream, etc.) and a “jumpstart” section. E.g., in the “DOM” package, all DOM-related interfaces (IF_IXML_ELEMENT, etc.) are explained. The jumpstart section should be referred to for an introduction to the concepts and services of the iXML library.
XML in ABAP: iXML Package (3)

iXML example programs in package SIXML_TEST

- parsing into a DOM: T_PARSING_DOM
- manipulating a DOM: T_DOM_MANIPULATE
- rendering a DOM: T_RENDERING_DOM
- validating with a DTD: T_DTD_VALIDATION

data element type ref to if_ixml_element.

```abap
element = document->create_element( name = 'date' ).
 element->set_attribute( name = 'format' value = 'yyyymmdd' ).
 element->set_value( value = ldate ).
```
XML in ABAP: XSLT Processor (1)

The SAP XSLT processor (since 6.10): features

- **performance**: implementation in SAP kernel (C++)
- **scalability**: optimization for server-side execution
- **interoperability**: with iXML package
- **conformance**: (except for justified omissions)
- **integration**: into language environment
  - call from ABAP, call back to ABAP
- **integration**: into development environment
  - workbench, transport

Scalability is essential: Performance must not degrade when the number of XSLT programs in parallel use increases. The SAP XSLT processor is scalable because it adopts a byte code / virtual machine approach (similar to Java and ABAP).

An XSLT program is parsed and analysed only at development time. It is then compiled into a byte code (developed by SAP specifically for XSLT) which is stored in the database. When an XSLT program is run, the byte code is loaded and executed on the virtual machine in the SAP kernel. An application-server-global buffer (the same which is used for ABAP programs) prevents frequent re-loading from the database.
XML in ABAP: XSLT Processor (2)

Unimplemented XSLT 1.0 features
- xsl:number
- Forwards-compatible processing, fallback processing
- Attribute sets
- Namespace aliasing

Implemented XSLT 2.0 features
- Grouping (xsl:for-each-group)
- User-defined XPath functions (xsl:function)
- Multiple results (xsl:result-document)

- ABAP Calls (by extension function or extension instruction)
- many extension functions
- XPath 2.0 features

Forwards-compatible processing is not supported because it is not expected that XSLT programs using a higher XSLT version will be maintained in a SAP system supporting a lower XSLT version.

Fallback is not supported because forwards-compatible processing is not supported, and it is not expected that XSLT programs using extension instructions of other XSLT processors will be maintained in the SAP repository.

Conversely, the SAP namespace need not be declared in extension-element-prefixes; SAP extensions are recognized automatically.

Stylesheet inclusion and import uses the sap:name attribute instead of the href attribute in order to refer to XSLT programs (programs are in the SAP repository).

Examples of extension functions: upper-case, lower-case, string-pad, ends-with, find-first, find-first-of, find-first-not-of, find-last, find-last-of, find-last-not-of, timestamp, otr-string, otr-text, line, column, max, min, abs, exp, log, parse-xpath.

Implemented XPath 2.0 features: generalized location paths A/ (B|C); generic namespace *:A; escaping quotes in string literals ("abc""def").

Conditional expression: [sap:if(e0,e1,e2)] = if [e0] then [e1] else [e2]
Variable binding: [sap:let(x,e1,e2)] = [e2] (x←[e1])
XML in ABAP: XSLT Processor (3)

Extension example: ABAP Call by extension function from XPath

At top level: Declare external function

```
<sap:external-function name="p:f1" method="METH">
  <sap:argument param="IP_1"/>
  <sap:argument param="IP_2"/>
  <sap:result param="EP" type="number"/>
</sap:external-function>
```

In XPath expression: Invoke instance method on external object

```
<xsl:value-of select = "p:f1($obj, 42,'foo') + 1"/>
```

- The function name must use a non-reserved namespace.
- The attribute kind of sap:external-function specifies whether it is a class or an instance method (default) or a constructor. For class methods and constructors, the ABAP class is specified in the class attribute.
- Invocations of instance methods require an external object as first function argument (as in the common Java binding mechanism). The object can come from a program parameter, a constructor call, or a preceding ABAP call.
- Appropriate conversions are applied between XPath and ABAP values. Nodesets correspond to iXML node collections.
- Multiple return values can be retrieved (less elegantly) by an extension instruction:

```
<sap:call-external name="obj" method="METH">
  <sap:callvalue param="IP" select="42"/>
  <sap:callvalue param="CP" select="'foo'"/>
  <sap:callvariable param="EP" name="v1" type="number"/>
  <sap:callvariable param="CP" name="v2"/>
</sap:call-external>
```

Here, XSLT variables v1 and v2 are bound to ABAP exporting / changing parameters EP and CP. The variable scope is the same as for xsl:variable.
This syntax example shows the case “source XML, result XML”. The syntax for “source ABAP” / “result ABAP” follows later.

Since ABAP is case insensitive, the case of the parameter names does not matter. In the XSLT program, however, they must be declared in upper case (because XSLT is case sensitive). Since the parameter names must be legal ABAP names, they cannot have a URI component.

A dynamic list of parameters is passed as a table of type `ABAP_TRANS_PARMBIND_TAB`.

Any ABAP type with a “natural” conversion to an XPath type can be passed as a parameter.

Both character and binary tables are allowed as the XML source and result.

An alternative way to execute XSLT programs is the class `CL_XSLT_PROCESSOR`. Its use is deprecated because it requires the application to take care of the lifetime of the processor object (which the ABAP instruction handles automatically). Basically there are two exceptions to this rule:

(i) *Separate evaluation of XPath expressions* (see class documentation).
(ii) Messages produced by the XSLT program are to be retrieved.
XML in ABAP: XSLT Development

Workbench integration

- **SE80**: 
  - *Edit object → More... → Transformation*

- **object tree, context menu**: 
  - *Create → More... → Transformation*

- **direct: transaction STRANS (6.20: XSLT_TOOL)**

- **testing: transaction XSLT**

- **programs must be activated before use**

- **check / activate triggers compilation**

```
XSLT Program XSLSLDEMCL_FLIGHTS_CONNECTIONS 19
unknown variable 'orders'
XSLT Program XSLSLDEMCL_FLIGHTS_CONNECTIONS 20
unsupported instruction 'xsl:sorts'
```

- **maintenance API: function XSLT_MAINTENANCE**

---

- XSLT programs are development objects, like ABAP programs. They are *not* in the MIME repository. From the workbench point of view, XSLT sources are primarily programs. That they are in XML format is secondary. This is in contrast with CSS stylesheets, which are mere content.

- XSLT programs are assigned to development classes / packages. Changes and transports are under the control of the transport system.

- An XSLT program can not be activated if it contains syntax errors. Only active programs can be included or imported by other programs. Consequently, only correct programs can be included / imported. However, programs which are correct in isolation may lead to errors in an including program (e.g. due to multiply declared templates).

- The XSLT tester loads the XML source from a frontend file. It offers four modes:
  - Transform directly to a string, display the string as a list.
  - Send result to a frontend file.
  - Transform to a DOM, then render the DOM with iXML methods; display as a list. In this case, XSLT output settings have no effect.
  - View result in HTML viewer.
Demo

- Transformations in the ABAP Workbench
  - Check – Activate – Test
- “Flights” example (SSTDEMO2):
  - XML to XML with XSLT
  - XML to HTML with XSLT
Intro
XML in Open Systems
XML in ABAP
ABAP – XML Mapping: Concepts
ABAP – XML Mapping with XSLT
Simple Transformations
Summary
Who needs (de)serialization?

- XML-based communication middleware
  - SAP Exchange Infrastructure (XI)
  - Web Services
- XML-based persistence
  - SAP Archiving
- XML repositories ↔ DB tables import / export
  - SAP WebDynpro metadata
- application-specific uses
  - XML via HTTP / in database / ...

In the Java data-binding community, the terms "marshalling" and "unmarshalling" are often used in place of "serialization" and "deserialization", respectively.
The general situation depicted here is that there is an ABAP system implementing certain functionality for which it offers an ABAP interface, and that there are other applications (maybe physically on the same server, but not in the ABAP system) offering an XML-based interface. The perspective is that of an ABAP developer, which leads to the "inside" / "outside" classification.
XML ↔ ABAP: Approaches

Which side is driving?

- **inside-out** approach
- **outside-in** approach
- **symmetric** approach (6.20)
  - canonical XML encoding of data structures
  - transformation with XSLT
- **symmetric** approach (6.40)
  - dedicated XML / data transformation language
  - no conceptual "canonical encoding" indirection

- "Inside-out": The ABAP structure determines the XML structure.
- "Outside-in": The XML structure determines the ABAP structure.
- "Symmetric" approaches retain given structures on each side. The two symmetric approaches presented here are original SAP developments.
- The first symmetric solution is based on an encoding of arbitrary ABAP data structures in XML: the canonical XML format asXML. This format is used as the source or result of XSLT transformations.
- The second symmetric solution works without a canonical XML format. It is based on a dedicated XML / data transformation language.
The inside-out approach is available with the 6.20 kernel: Using CALL TRANSFORMATION and the identity transformation, any ABAP data structure can be represented in a canonical XML encoding.

But for communication with external applications, this is usually not sufficient, as they will require different XML formats.

Consequently, XML-to-XML transformations will have to be performed. Note that such a transformation is not provided "inside" in this approach; it is the problem of the "outside" applications.

Even if the canonical format were accepted by the "outside", it would usually not be the best choice, because it exposes directly the "inside" data structures. For a communication format, modifications are often desirable, e.g. different XML names, element / attribute distinctions, or even structural changes.
Numerous data binding frameworks are available in Java, most notably JAXB. The basic idea is always that XML structures are somehow reflected in Java structures which are generated from an XML schema. Thus, every XML schema that a system must handle leads to the generation of proxy classes (about as many as node types in the schema).

In addition to the proxy classes, adapter code for given "inside" functionality may also be generated. Alternatively, it may be written by hand.

An even more radical outside-in approach would be to use the proxy classes directly as the interface and possibly even as the internal data model of the application. However, this approach is almost as inadequate as using the DOM as a data model: Data communication structures are hardly appropriate as data processing structures.
The symmetric approach (6.20)

- This picture omits a small piece of code that actually invokes the XSLT program with the XML document and ABAP data structure. This (trivial) adapter code could be located either in the application (if the corresponding XML format is implemented as part of its interface) or on top of it.

- The XSLT programs work on asXML, the canonical ABAP Serialization XML format (more details later).
The first step in the "outbound XML" case is the construction of the ABAP data structure's `asXML` representation as a DOM (implicitly done by the CALL TRANSFORMATION statement).

This is a prerequisite for the "outbound XSLT" program; more precisely, for the evaluation of the XPath expressions in it.

The DOM representation is the basis for powerful XPath navigation on the data structure. But of course, it is also the cause of considerable memory consumption.
The first step in the "inbound XML" case is the construction of the XML DOM (if it is received as text).

Note that the "inbound XSLT" program is different from the "outbound XSLT" program. It does not consume asXML, but produces it.

The construction of the asXML tree is only conceptual. In fact, the ABAP data structure is filled directly.

Thus, inbound and outbound XML have approximately the same complexity: DOM construction, XSLT execution, data structure traversal.
The ABAP / XML mapping engine presented here does not cover communication; i.e., it is left open how XML documents enter and leave the system. For example, communication could utilize the HTTP capabilities of the SAP Web AS.

Also, the mapping is not tied to any kind of RPC (Remote Procedure Call) mechanism. Any XML format can be processed; mapping is not limited to a "call / parameter" format.
Intro
XML in Open Systems
XML in ABAP
ABAP – XML Mapping: Concepts
**ABAP – XML Mapping with XSLT**
Simple Transformations
Summary
## XML ↔ ABAP: Invocation of XSLT

### Serialization

```
CALL TRANSFORMATION ... 
PARAMETERS ... 
OPTIONS  option_1 = string ... option_n = string 
SOURCE   XP_1 = my_var_1 ... XP_n = my_var_n 
          or (my_var_table) 
RESULT   XML    my_xml_result.
```

### Deserialization

```
CALL TRANSFORMATION ... 
PARAMETERS ... 
OPTIONS  option_1 = string ... option_n = string 
SOURCE   XML    my_xml_source 
RESULT   XP_1 = my_var_1 ... XP_n = my_var_n 
          or (my_var_table). 
```

- When the source of a transformation is a set of ABAP variables, XML bindings for these variables are specified after the keyword `SOURCE`. If the variables are the transformation result, then the bindings are specified after `RESULT`.

- An XML binding `x = v` associates an ABAP variable `v` with an XML name `x` in the asXML document. (Without this indirection, a transformation would be tied to a fixed set of variable names.)

- Since ABAP is case insensitive and normalizes parameter names to uppercase, the corresponding XML name is uppercase. Consequently, XSLT programs must specify these names in uppercase, since XML is case sensitive. In order to avoid confusion, XML names in ABAP should be specified in uppercase as well.

- By combining `SOURCE` and `RESULT` clauses, ABAP-to-ABAP is also possible.

- The option `XML_HEADER = 'NONE' | 'WITHOUT_ENCODING' | 'FULL'` controls output of the XML header `<?xml ... ?>`.

- The option `INITIAL_COMPONENTS = 'SUPPRESS'` suppresses ABAP fields with initial value during serialization.

- The option `VALUE_HANDLING = 'MOVE'` tolerates “illegal” values (e.g. non-digits in type N fields), corresponding to the behavior of the `MOVE` instruction.
Exercise

“Customers” example:
My first ABAP-XML Mapping (XSLT)
  - ABAP to XML
  - XML to ABAP
When the transformation is not the identity (the required result XML format is not 
asXML), it must process the asXML representation generated for the ABAP source 
variables.

- Data is written in the canonical (XML Schema) lexical representation.
When the transformation is not the identity (the required source XML format is not asXML), it must generate the asXML representation required for the ABAP result variables.

Data is accepted in any form that matches the lexical representation. Thus, deserialization is more liberal than serialization (e.g., leading and trailing spaces are not significant for most datatypes).

XML-to-ABAP example with dynamic parameter passing:

```
TYPE-POOLS: abap.
DATA bind TYPE abap_trans_resbind.
DATA itab TYPE abap_trans_resbind_tab.
DATA num TYPE P LENGTH 5 DECIMALS 2.
DATA txt TYPE STRING.

bind-name = 'FOO'. GET REFERENCE OF num INTO bind-value.
APPEND bind TO itab.
bind-name = 'BAR'. GET REFERENCE OF txt INTO bind-value.
APPEND bind TO itab.

CALL TRANSFORMATION ID
  SOURCE XML  my_xml_source
  RESULT      (itab).
```
ABAP Types and the asXML Format

- **ABAP → asXML**: type of ABAP source variable(s) determines generated asXML representation
- **asXML → ABAP**: type of ABAP result variable(s) determines required asXML representation

→ source / result ABAP variables must be typed

A Data Exchange Format

- **XML-Schema** datatypes
- "Human-readability" not a top priority
  - no dependence on customization
  - no dependence on locale settings
  - no use of conversion exits

The asXML format adopts XML Schema datatypes (http://www.w3.org/TR/xmlschema-2), on which many XML standards (e.g. SOAP) are based. (If asXML used a more ABAP-like format, XSLT programs would have to be used to do much more work than structural transformations, especially string operations.)
### asXML Format: Simple Types

#### Simple Types

<table>
<thead>
<tr>
<th>ABAP Type</th>
<th>ABAP Example*</th>
<th>XML Schema Type</th>
<th>XML Example</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>□Hi□</td>
<td>string</td>
<td>□Hi□</td>
<td>string of characters</td>
</tr>
<tr>
<td>C</td>
<td>□Hello□</td>
<td>string</td>
<td>□Hello□</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>001234</td>
<td>string ([0-9]+)</td>
<td>001234</td>
<td>string of digits</td>
</tr>
<tr>
<td>I</td>
<td>INT1, INT2</td>
<td>int</td>
<td>-123</td>
<td>number</td>
</tr>
<tr>
<td>P</td>
<td>1.23-</td>
<td>decimal</td>
<td>-1.23</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>-3.14E-2</td>
<td>double</td>
<td>-3.14E-2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>20010820</td>
<td>date</td>
<td>2001-08-20</td>
<td>ISO 8601 date/time</td>
</tr>
<tr>
<td>T</td>
<td>201501</td>
<td>time</td>
<td>20:15:01</td>
<td></td>
</tr>
<tr>
<td>XSTRING</td>
<td>456789AB</td>
<td>base64Binary</td>
<td>RW8Jqw==</td>
<td>base64-encoded binary data</td>
</tr>
<tr>
<td>X</td>
<td>ABCDEFG</td>
<td>base64Binary</td>
<td>q83v</td>
<td></td>
</tr>
</tbody>
</table>

* Internal string representation of ABAP datatypes

- STRING: Leading and trailing spaces are significant.
- C: Only leading spaces are significant.
- N: String of digits, i.e., leading 0s are significant (restriction of XML Schema type 'string').
- I, P, F: leading -, uppercase E (lowercase ok in input), no leading 0s (ok in input), no after-decimal-point trailing 0s (ok in input).
- D, T: No timezone, year in range 1-9999 (less than is allowed by XML Schema type), some invalid dates are allowed (e.g., year 0).
- XSTRING, X: Base-64 encoding is commonly used in an XML context (e.g. SOAP).
- Graphs of objects or data references need not be tree-shaped. Therefore, corresponding asXML documents contain in-document references to represent such graphs. Here, object and data references are not presented in further detail. For objects, the implementer of a class has the possibility to state whether its instances can be serialized and if so, which values constitute the serialized object.
- All types are allowed in components, including structures and internal tables.
- Program-local types can be used.
- Matching components are set as by MOVE-CORRESPONDING. Additional subelements are ignored in XML input, components without a corresponding XML element keep their value. In contrast with MOVE-CORRESPONDING, the "matching algorithm" is applied recursively.
- Components are serialized in the order in which they are defined in the structure. Order in input is insignificant; components are addressed by name.
- The DDict type PERSON is hypothetical.
Any kind of table (standard, sorted, hashed) and any line type (including non-structures) is allowed.

During deserialization, the name of the element representing a table line is irrelevant.

Table lines for sorted tables do not have to be sorted (will be sorted automatically).

The example XML fragments are pretty-printed only for presentation.
**asXML Format: Envelope**

The version attribute indicates the asXML version used in the document. Currently it must be set to 1.0.

The top-level element must be `<asx:abap>`, where the (arbitrary) prefix `asx` is bound to the asXML namespace `http://www.sap.com/abapxml`.

For each asXML representation of an ABAP variable bound to an XML name, there is an element with this XML name inside the `asx:values` element.
Exercise

“Customers” example:
Full version (XSLT)
Advantages of the XSLT solution

- **no restrictions**
  - arbitrary XML schemas
  - arbitrary data & object types
    - graphs of objects
  - arbitrarily complex structural transformations

- **no redundancy**
  - no generation of schemas from types (schema inflation)
  - no generation of types from schemas (type inflation)
  - each side retains its structures

- **high abstraction level**
  - no low-level XML handling in applications
  - separate, expressive transformation language

Data binding approaches are usually restricted to certain classes of XML schemas and offer only limited mapping capabilities. This is because they do not employ a programming language, but only a configuration mechanism.

By not generating types from schemas, the symmetric approach prevents the schema inflation in the "outside" XML world to cause a type inflation on the "inside". Controlling the amount of types in a system is generally a good thing.

With XSLT as a distinct programming language, mapping code is clearly separated from application code, and programmers are not tempted to abuse the DOM as a generic data structure.
Disadvantages of the XSLT solution

- learning XSLT
  - overkill for simple conversion tasks
  - no tool support
- asymmetric programs
  - one for XML $\rightarrow$ ABAP, one for ABAP $\rightarrow$ XML
- resource consumption (Time & Space)
  - $T / S$ : DOM construction (on source side)
  - $T$ : codepage conversions (internal encoding $\neq$ SAP CP)
  - $T$ : XSLT engine overhead (complex state, powerful operations)
- no static type checking

- asXML DOMs can easily become huge. Due to the OO overhead, a DOM typically requires about 10 times the memory of the XML text representation – which is already large compared to the data structure.

- iXML and the SAP XSLT processor work internally with encoding UTF-16. If this is not the SAP codepage (i.e. the system is non-Unicode), frequent conversions are necessary.

- Static type checking is impossible because XPath navigation on the asXML input (serialization) or control flow in the XSLT templates (deserialization) is not statically analyzable.
Intro
XML in Open Systems
XML in ABAP
ABAP – XML Mapping: Concepts
ABAP – XML Mapping with XSLT

Simple Transformations

Summary
XML ↔ ABAP without XSLT: Requirements

Requirements for a dedicated XML/ABAP mapping engine

- **time**: increase throughput by factor ≥ 10
- **space**: increase / eliminate upper limit on size of data
- **ease of use**
  - simple syntax & semantics, statically type-checked
  - tool support for creating mappings
  - one program for XML→ABAP and ABAP→XML

Deliberate trade-offs

- lower expressive power (but cover 90% of typical applications)
- not usable for XML ↔ XML

⇒ "Simple Transformations"

- The XSLT-based solution is used in XI 2.0.
- Simple Transformations are the basis for XI 3.0 and Web Services.
### Simple Transformations: Tree Access

#### Tree access in XSLT vs. Simple Transformations

<table>
<thead>
<tr>
<th></th>
<th>XSLT</th>
<th>Simple Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serialization</td>
<td><strong>ABAP source tree navigation</strong></td>
<td>random access</td>
</tr>
<tr>
<td></td>
<td>random access (XPath on canonical DOM)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>XML result tree construction</strong></td>
<td>linear</td>
</tr>
<tr>
<td></td>
<td><strong>XML source tree navigation</strong></td>
<td>linear (stream reader)</td>
</tr>
<tr>
<td></td>
<td>random access (XPath)</td>
<td></td>
</tr>
<tr>
<td>Deserialization</td>
<td><strong>ABAP result tree construction</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>linear (modulo component order)</td>
<td></td>
</tr>
</tbody>
</table>

- XSLT programs always produce the result in strictly linear fashion. However, order of ABAP structure components is irrelevant during deserialization.

- In contrast, Simple Transformations always access the XML document in linear order (even if it is on the source side) and the ABAP data structure in non-linear order (even if it is on the result side).
Simple Transformations: Expressive Power

Anything that can be done with ...
- accessing each node in the data tree any number of times
- accessing each node in the XML tree at most once, in document order (with "lookahead 1" on XML source)

... which includes (any combination of) ...
- renamings (e.g.: structure-component / element names)
- projections (omission of sub-trees)
- permutations (changes in sub-tree order)
- constants (e.g.: constant values, insertion of tree levels)
- defaults (for initial / special values)
- conditionals (e.g.: existence of sub-trees, value of nodes)
- value maps

→ covers most data mappings in practice
Simple Transformations: Key Features (1)

**Programs are XML templates**
- literal XML with interspersed instructions
- declarative, straightforward semantics

**Data tree access by node references**
- instructions access data by simple “reference expressions”
- all named children of a data node are accessible by name
- tables are accessible as a whole (all lines or none)

```
<Customers>
  <tt:loop ref="CUSTTAB">
    <LastName>
      <tt:value ref="NAME.LAST"/>
    </LastName>
  </tt:loop>
</Customers>
```

- The concept of literal XML with interspersed instructions is the same as in XSLT.
Simple Transformations: Key Features (2)

Programs are reversible

- **serialization**: write (produce) tokens to stream
- **deserialization**: match (consume) tokens from stream
- **invocation** determines direction
- **if no asymmetric construct is used:**
  
  - \[ D[\text{program}] ( S[\text{program}] (data) ) = \text{data} \]
  - \[ S[\text{program}] ( D[\text{program}] (document) ) \approx \text{document} \]

  (\(D:\) deserialization, \(S:\) serialization)

- Of course, the "reversibility" equations do not hold for programs that lose information, e.g. by not serializing a data component at all (projection).
From 6.20 to 6.40

“XSLT” is generalized to:

“Transformation” = “XSLT” or “Simple Transformation”

- workbench integration
  - same workbench / transport object type
  - same access paths
  - transformation type determined in creation dialog
  - XSLT_MAINTENANCE works for both types

- ABAP integration
  - CALL TRANSFORMATION works for both types
  - both types take same XML input / output types (e.g. iXML streams)
  - exception hierarchy generalized (new common root)

⇒ uniform appearance (easy to switch from XSLT solutions)
Simple Transformations: Implementation

Virtual machine (VM)
- compilation of programs to bytecode
- storage of bytecode in database / buffering in shared memory
- interpretation of bytecode in kernel (lean engine)
  ➔ scalable performance

XML stream reader / writer
- reader: efficient "token pull" discipline
- specialized for standard encoding UTF-8
- also usable directly from ABAP:
  classes CL_FX_READER, CL_FX_WRITER
  ➔ limit on data size lifted

Like the SAP XSLT processor, the Simple Transformations implementation re-uses the infrastructure for ABAP programs, in particular the load table and the program buffer.

However, execution takes place on a dedicated engine (neither ABAP nor XSLT). It works directly with UTF-8, which is the encoding that is used most frequently for XML communication.
“Flights” example (SSTDEMO2):

- ABAP to XML with ST
- XML to ABAP with ST
A program can have several named templates for the purpose of applying them several times (in the main template, or in an including program). In most cases a program will contain only one template. There is no concept of template selection by pattern matching, as in XSLT.

If there are several templates, then the main template is either specified by a `template="name"` attribute on the `tt:transform` element, or it is the (single) unnamed template.
ST Constructs: Value

Value

S  ref-node value to XML
D  XML value to ref-node

<tt:value
  [ref="ref"]
  [map="mapping-list"] />

example

copy value from/to field CHAR,
with special mappings:

S  map {*, +, -} to ~
D  map ~ to *

<tt:value
  ref="CHAR"
  map=""
  val('*') = xml('~'),
  val('+','-') > xml('~")" />
# ST Constructs: Literal Element / Attribute

## Literal element / attribute

| S | write literal content |
| D | match literal content  |
|   | (no match → error)    |

**Example**

```xml
<Time zone="CET" tt:ref="TIME">
  <tt:value/>
</Time>
```

**Alternative**

```xml
<Time zone="CET" tt:value-ref="TIME"/>
```

**XML**

```xml
<Time zone="CET">12:59:00</Time>
```
## ST Constructs: Literal Text

### Literal text

<table>
<thead>
<tr>
<th>S</th>
<th>write literal text</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>match literal text</td>
</tr>
</tbody>
</table>

### Example

```
<Time>
    <tt:value ref="TIME"/>
    <tt:text> CET</tt:text>
</Time>
```

### Alternative

```
<Time>
    <tt:value>
        ref="TIME"/
        CET
    </tt:value>
</Time>
```

### XML

```
<Time>12:59:00 CET</Time>
```
## ST Constructs: Attribute

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Write attribute</td>
</tr>
<tr>
<td>D</td>
<td>Match attribute</td>
</tr>
</tbody>
</table>

### Example

```xml
<Time tt:ref="ZTIME">
  <tt:attribute name="zone" value-ref="TZONE"/>
  <tt:value ref="TIME"/>
</Time>
```

### XML

```xml
<Time zone="CET">12:59:00</Time>
```
ST Constructs: Deep Copy

Deep Copy

S  copy sub-tree below ref-node to stream

D  copy sub-tree below current XML node to ref-node

example

XML

Note that tt:copy does not introduce an element for the root of the tree being copied (in this case, the internal table itself). Below the root node, elements are introduced according to the canonical XML representation of ABAP data structures. In this case, it is assumed that the line type of ZTIME_TAB is a dictionary type ZTIME with two components TZONE and TIME.
The name attribute on tt:loop serves to declare a symbolic name for the current table line which is in scope in the body of the loop. This is necessary if you want to refer to (components of) this line from within a nested loop: Having declared name="line-name", you can use the syntax $line-name as a reference expression in order to refer to the current line of this loop.
Exercise

“Customers” example:
My first Simple Transformation
Basic Conditional

S  [if data condition is true:]  
   evaluate template content

D  [if template content matches:]  
   evaluate template content  
   [establish assertions]  
   [evaluate check-clause]

example

S  if TZONE='CET' and 
   TIME = t > 000000:  
   write <CET>t</CET>

D  if <CET>t</CET>:  
   TIME:=t, TZONE:='CET',  
   check TIME > 000000

Using s-check / d-check instead of check causes the check-clause to be evaluated only during serialization / deserialization.

The “data condition” evaluated during serialization is the conjunction of the assertion-list and the check-clause. If there is no such condition, then the template content is evaluated unconditionally during serialization.

The template content of the conditional is a pattern if it is (i) a single literal element [with content], (ii) a tt:attribute instruction, (iii) a single non-empty literal text, (iv) the keyword <tt:empty/> (matching empty content), or (v) an application of a template whose body is a pattern, or (vi) a loop whose body is a pattern. If the body is not a pattern, then it is evaluated unconditionally during deserialization.

Deserialization fails if evaluation of the check-clause yields false or if a value different from the assertion value is assigned to a node with an assertion (during evaluation of the body).
The general form of the ref-node syntax for expressions is necessary because ABAP and the dictionary admit almost any character in names, which is impossible with the Simple Transformations expression parser (a reduced variant of the XPath parser).
Apart from the value assertions and check-clause, tt:cond may have an attribute using, containing a comma-separated list of node-existence and node-type checks.

This list of “using assertions” is evaluated first. If it fails (because a node does not exist or does not have the required type), then the condition body is not evaluated. If a using assertion fails during deserialization and the body is a pattern, then the matching part of the XML source (e.g. the XML element) is skipped.
ST Constructs: Composite Conditional

**Switch**

- **S** evaluate first case with true data condition
- **D** evaluate first case with matching pattern

```xml
<tt:switch>
  case*
</tt:switch>
```

**Group**

- **S** evaluate all cases with true data condition
- **D** evaluate all cases with matching pattern

```xml
<tt:group>
  case*
</tt:group>
```

- At most one case in a switch may have no data condition. If such a case exists, it is the serialization default.
- At most one case in a switch may have no pattern content. If such a case exists, it is the deserialization default.
ST Constructs: Modularization (1)

Variable / Parameter Declaration

```
<tt:root name="name"/>
<tt:variable name="name" val="constant"/>
<tt:parameter name="name" val="constant" kind="in"|"out"|"in/out"/>
```

- Context of main template: top-level declarations
- Default context of sub-template: one unnamed root
- initial / default value in val
- parameter direction (kind) independent of transformation direction

Each template works on a certain “context”. The context of the main template is the set of roots and variables / parameters declared at the top level of the transformation. A sub-template (i.e., non-main template) can have its own context declaration, in which it declares its roots, parameters, and variables. By default, sub-templates have a single unnamed root and no variables / parameters.

The treatment of root nodes (and all ref-nodes) depends on the transformation direction: Serialization reads them, deserialization writes them. They are either read-only or write-only. In contrast, variables can always be read and written. In particular, transformation parameters can be read during deserialization and written during serialization, which can’t be done with ref-nodes.
A sub-template without an explicit root declaration has an implicit unnamed root. This root is bound to the current node at the point of invocation (**tt:apply**).
Exercise

“Customers” example:
Full version (Simple Transformation)
ST Constructs: Variable [De]Serialization

**Read Variable**

S  no effect
D  XML value to variable

```xml
<tt:read var="name"
[type="C|D|F|I|N|P|T|X"]
[map="mapping-list"] />
```

**Write Variable**

S  variable value to XML
D  no effect

```xml
<tt:write var="name"
[map="mapping-list"] />
```
**ST Constructs: Assignment**

### Assign to Ref-Node

**S** no effect

**D** assign to ref-node

```xml
<tt:assign to-ref="ref"
  ( var="name" | val="constant" )/>
```

```xml
<tt:clear ref="ref"/>
```

### Assign to Variable

**S** assign to variable

**D** … except if ref (no effect)

```xml
<tt:assign to-var="name"
  [ var="name" | ref="ref" | val="constant" ] />
```

```xml
<tt:clear var="name"/>
```

**Example**

**same effect as**

```xml
<tt:value ref="A"/>
<tt:assign to-var="V" ref="A"/>
<tt:write var="V"/>
<tt:read var="V"/>
<tt:assign to-ref="A" var="V"/>
```

<table>
<thead>
<tr>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;tt:assign to-var=&quot;V&quot; ref=&quot;A&quot;/&gt;</td>
<td>&lt;tt:write var=&quot;V&quot;/&gt;</td>
</tr>
<tr>
<td>&lt;tt:clear var=&quot;name&quot;/&gt;</td>
<td>&lt;tt:clear var=&quot;name&quot;/&gt;</td>
</tr>
</tbody>
</table>
ST Constructs: Variable Conditional

Basic Variable Condition

```xml
<tt:cond-var check="check-clause"
  template content
</tt:cond-var>
```

Variable Switch

```xml
<tt:switch-var>
  case *
</tt:switch-var>
```

- check-clause uses variables (not ref-nodes)
- at most one case in `tt:switch-var` may have no condition (default)

The `tt:cond` and `tt:switch` constructs have direction-dependent semantics:
During deserialization, the condition is on the XML content, not on the values of ref-nodes. (As a matter of principle, it is impossible to let the values of ref-nodes influence the transformation result.)
In contrast, the `tt:cond-var` and `tt:switch-var` constructs are always conditions on the values of variables, even during deserialization. Consequently, the check-clause of `tt:cond-var` cannot use ref-nodes.
## ST Constructs: Directional

### Serialize

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>evaluate template content</td>
<td>no effect</td>
</tr>
</tbody>
</table>

```
<tt:serialize>
  template content
</tt:serialize>
```

### Deserialize

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>no effect</td>
<td>evaluate template content</td>
</tr>
</tbody>
</table>

```
<tt:deserialize>
  template content
</tt:deserialize>
```

### Skip

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>no effect</td>
<td>match &amp; skip [number of] nodes [named element-name]</td>
</tr>
</tbody>
</table>

```
<tt:skip [name="element-name"]
    [count="number" | "]>
  template content
</tt:skip>
```
ST Exception Hierarchy

CX_TRANSFORMATION_ERROR
  CX_ST_ERROR
    CX_ST_REF_ACCESS
    CX_ST_CONDITION
      CX_ST_COND_CHECK_FAIL
      CX_ST_GROUP_MISSING_CASE
      CX_ST_SWITCH_NO_CASE
    CX_ST_MATCH
      CX_ST_MATCH_NAMED
        CX_ST_MATCH_ATTRIBUTE
        CX_ST_MATCH_ELEMENT
      CX_ST_MATCH_TEXT
      CX_ST_MATCH_TYPE
    CX_ST_FORMAT_ERROR
    CX_ST_DESERIALIZATION_ERROR
    CX_ST_SERIALIZATION_ERROR
    CX_ST_INVALID_XML
    CX_ST_CALL_ERROR
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Summary**
Summary: ABAP – XML Mapping

ABAP Data

XML Doc

Simple Transformations

new in 6.40

HTML / Text

Network

DB

© SAP AG 2004, SAP TechEd / ABAP 252 / 80
Summary: Available Demos

Transactions in the Basis delivery

- (6.20) **SXSLTDEMO1** : XSLT

- (6.40) **SSTDEMO1** : ST

- (6.40 SP9) **SSTDEMO2** : XSLT and ST
Summary: Transformation Languages for ABAP

XSLT (since 6.10)
- works on canonical XML representation of ABAP data (asXML)
- builds DOM for source side
- arbitrarily complex transformations

Simple Transformations (since 6.40)
- only for ABAP ↔ XML
- only linear transformations (no DOM)
- speedup over XSLT: 10 – 30; “unlimited” size of data
- reversible (one program for both directions)

Both
- symmetric: no generation of ABAP code / XML schemas
- integrated in workbench (maintenance / transport)
- integrated in ABAP: CALL TRANSFORMATION

basis of: XI 2.0

basis of: XI 3.0 / Web Services
Summary: When To Use What

System Landscape Integration
- Exchange Infrastructure

RPC-Style Web Services
- ABAP Web Services

Direct XML Processing in ABAP
- REST-Style Web Services (URI in, XML out)
- Custom XML Persistence
- XML-Based Repositories
- ...

Simple mappings, high throughput
- Simple Transformations
Complex mappings, limited throughput
- XSLT
## Further Information

### Public Web:
SAP Developer Network: [www.sdn.sap.com](http://www.sdn.sap.com) ➔ Forums ➔ Web Application Server ➔ ABAP Programming

### SAP Professional Journal
- Jul/Aug 2002: “From XML to ABAP Data Structures and Back: Bridging the Gap with XSLT”
- Nov/Dec 2002: “Mastering the asXML Format to Leverage ABAP-XML Serialization”
- Coming up: “Simple Transformations between ABAP and XML”

### Related Workshops/Lectures at SAP TechEd 2004
- ABAP 256: “Utilizing Web Services with ABAP and SAP Web Application Server 6.40” (4h Hands-on)
- XI 252: “SAP Exchange Infrastructure - Integrating Heterogeneous Applications” (4h Hands-on)
Look for SAP TechEd '04 presentations and videos on the SAP Developer Network. Coming in November.

http://www.sdn.sap.com/
Please complete your session evaluation.

Be courteous — deposit your trash, and do not take the handouts for the following session.

Thank You!
I would suggest moving this slide before the guidelines, so people don't accidentally delete it with all the guideline slides