Semistructured Data & XML
(Summer Term 2020)

XML Software Lab
(Winter Term 2020/2021)

(c) Prof Dr. Wolfgang May

Universität Göttingen, Germany

may@informatik.uni-goettingen.de

SSD&XML: Advanced Course in Informatics; 3+1 hrs/week, 6 ECTS Credit Points
XML Lab: Advanced Lab Course in Informatics; 2+2 hrs/week, 6 ECTS Credit Points

A comprehensive German-English dictionary can e.g. be found at

`http://dict.leo.org/`

Tasks in Informatics

1. Implementing a proposed solution: a job
   requires: good knowledge of common tools
2. Designing solutions: an interesting task
   requires: solid knowledge of up-to-date concepts
3. Development of concepts: a fascination
   requires: deep understanding and analysis of existing concepts

XML is a good example for all of them.

Aims of the Course

- knowledge of the concepts of the XML-World, practical experiences
  ⇒ application-oriented
  (requires also to work on your own)
- backgrounds why XML developed, and why it is as it is
  ⇒ understanding of concepts und developments
- underlying meta-concepts
  ⇒ as an example of “Informatics” as a whole

Overview

- first: other talk “Introduction to XML” ...
- note: table of contents at the end of the slide set (lecture + lab course)

Chapter 1
Introduction

Context and Overview

- Databases are used in many areas ... economics, administration, research ...
- originally: storage of information
  late 60s: Network Data Model, Hierarchical Model
  70s: Relational model, SQL – Lecture "Introduction to Databases"
- evolution: information systems, combination of databases and applications, distributed databases, federated databases, interoperability, data integration
- today: Web-based information systems, electronic data interchange
  → new challenges, semistructured data, XML
- tomorrow: Semantic Web etc.

1.1 Data Models

A *data model* defines the modeling constructs that can be used for modeling the application domain.

- *conceptual modeling:* application-oriented model
  - Entity-Relationship-Model (1976, only static concepts: entities and relationships, graphical)
    Lecture: Introduction to Databases
  - Unified Modeling Language (UML 1.0: 1996)
    comprehensive graphical formalism for modeling of processes, based on the object-oriented idea: classes and objects, properties, behavior (states and actions).
    Lecture: Software Engineering
- *logical data models:* (e.g. relational model)
  serve as *abstract data types* for implementations:
  - definitions of operations and their semantics, e.g. relational algebra
  - corresponding languages (as *application programming interfaces*): e.g. SQL
- *physical data models:* the implemented structures.

Data Model: Database Schema and Database State

Usually, for a database (for both, conceptual and logical models), its *schema* and its *state* are considered:

Database schema: the schema contains the *metadata* about the database, i.e., it describes the structure (in terms of the concepts of the data model).

The set of legal states is also described in metadata (e.g., by integrity constraints).

Database state: the state of a database is given as the currently stored information. It describes all objects and relationships that exist in the application at a given *timepoint*.

The database state changes over the time (representing changes in the real word), whereas the database schema is in general unchanged.

Logically spoken, the database state is an *interpretation* of the structure that is determined by the metadata.

Languages for Logical Data Models: In general, a language for operating on a data model consists of

- **D**ata **D**efinition **L**anguage (DDL) for schema definitions,
- **D**ata **M**anipulation **L**anguage (DML) for manipulating and querying database states.

Logical/Implementation Data Models

... there are many different data models.

Basically, all database approaches are grounded on the concept of a "data item" (german: "Datensatz").

- logical data models and implementation models
  - network data model (IDS (General Electric) 1964; CODASYL Standard 1971), hierarchical data model (IMS (IBM) 1965); data records,
  - relational model (Codd 1970), SQL (IBM System R 1973; products since 1979 (Oracle), ISO SQL Standard 1986); tuples
  - object-oriented model (ODMG 1993; OQL); objects
- document-data model (SGML)
- semistructured data models, XML; nodes: elements, attributes, text
  - why?
  - evolution and current situation

1.2 Relational Model

- *relational model* by E.F. Codd (1970, IBM San Jose): mathematical foundation: set theory
- only a single structural concept: *relation* for entities/objects and relationship types (note that the notions “entity” and “relationship” from the ER model [1976] were not yet defined!)
- properties of entities/objects and relationship types are represented by *attributes*
- a relation schema consists of a name and a set of attributes
  Continent: Name, Area
- each attribute is associated with a *domain* that contains all legal values of the attribute. Attributes can also have *null values*:
  Continent: Name: VARCHAR(25), Area: NUMBER
- a **(relational) database schema** is given by a (finite) set of (relation)schemata:
  Continent: . . . ; Country: . . . ; City: . . . ; encompasses: . . .

RELATIONS

- a *(database) state* associates a *relation* with each *relation schema*.
- the elements of a relation are called *tuples*.
  Each tuple represents an object or a relationship:
  (Name: Asia, area: 4.5E7)

Example:

Continent	
Name	**Area**
VARCHAR(20)	NUMBER
Europe	9562489.6
Africa	3.02547e+07
Asia	4.50953e+07
America	3.9872e+07
Australia	8503474.56

Relations: Example

Continent	
Name	**Area**
Europe	9562489.6
Africa	3.02547e+07
Asia	4.50953e+07
America	3.9872e+07
Australia	8503474.56

Country				
Name	**code**	**Population**	**Capital**	**...**
Germany	D	83536115	Berlin	
Sweden	S	8900954	Stockholm	
Canada	CDN	28820671	Ottawa	
Poland	PL	38642565	Warsaw	
Bolivia	BOL	7165257	La Paz	
..	..	..	..	

encompasses		
Country	**Continent**	**Percent**
VARCHAR(4)	VARCHAR(20)	NUMBER
R	Europe	20
R	Asia	80
D	Europe	100
. . .	. . .	. . .

- ... with referential integrity constraints
- abstract datatype for this model: relational algebra
- application interface: SQL

QUERY LANGUAGE: SQL

- Since 1973 “SEQUEL – Structured English Query Language” in IBM System R (E.F. Codd (Turing Award 1981), D. Chamberlin (2001: co-designer of XQuery)) etc.; Research-only (IBM continued to sell only IMS until SQL/DS (1980), DB2 (1983))
  Stories: `http://www.mcjones.org/System_R/SQL_Reunion_95/`
  `http://www.nap.edu/readingroom/books/far/ch6.html`
- 1974 INGRES (UC Berkeley, M. Stonebraker; NSF funding), QUEL language, open-source.
  Led to the products INGRES (“Relational Technology Inc.” 1980, QUEL; since 1986 with SQL), INFORMIX (1981; since 1984 with SQL), SYBASE (1984, since 1987 with SQL)
- Oracle: founded in 1977 as “Relational Software” (L. Ellison worked before on a consultant project for CIA that wanted to use SEQUEL), 1983 renamed to “Oracle”. Product: 1979 Oracle V2 (SQL), first commercial relational DB system.
- Standard SQL: 1986 ANSI/ISO (least common denominator of existing products); SQL-1 1989 (Foreign Keys, ...); SQL-2 1992 (multiple result tuples in subqueries, SFW in FROM, JOIN syntaxes, ...); SQL-3 1999 (PL/SQL etc) ...
- 1995: 80% of all databases use the relational model and SQL

Query Language: SQL

SELECT name, percent
FROM country, encompasses
WHERE country.code = encompasses.country
AND encompasses.continent = 'Europe';

- intuitive to understand,
- *clause-based*, *declarative* language,
- *set-oriented*, *closed*: result of (nearly) each expression is again a relation,
- *orthogonal constructs*, can be nested (nearly) arbitrarily,
- *functional programming paradigm*: each SFW query is a function that maps relations to another relation. Such functions can be nested.

... so far the things you have learnt in "Databases" about the relational model and SQL.

1.3 Concepts and Notions

- the relational model is a *data model*.
- (relational) databases follow a 3-level architecture:
  - *physical level/schema:* actual storage of tables in files, as sequenced records, with length indicators etc; additional index files, and allocation tables.
  - *logical level/schema: user level.*
    Relational model (*logical data model*) with given database schema (table names, attributes, keys, foreign keys etc), relational algebra, SQL (*database language*).
    Abstract, *declarative*, *set-oriented* language, distinguished notions of schema and state.
    Internal: mapping to physical schema. Admin can change the physical schema and adapt the mapping without effecting the logical schema.
  - *external level (optional):* possible views, given by SQL queries.
    A *view* is (any kind of) a mapping from underlying "base" data to derived information.
- note: SQL is the only language with which users work on relational data. Relational data exists only inside databases.

Concepts: Preview

- network data model: mainly a physical data model; "logical" model on a very low level of abstraction.
  No database language, only some data-management-oriented operations extending a common programming language.
- relational model: abstract/logical data model, relational algebra, declarative, set-oriented query+update language.
- early semistructured data models (OEM, F-Logic etc.): not comparable, separate experiments how to extend functionality without losing the advantages from relational databases and SQL.
- for XML there are several languages ("views" can also be defined in several ways), and XML exists also as a data structure used in non-database tools.

1.4 Aside: Really Declarative Languages ...

SQL is already called "declarative": express what, not how.

But there is an even more declarative language family: *logic-based* languages.

Relational Calculus, Datalog

- Facts (tuples) are called "atoms":
  country("Germany", "D", 83536115, 356910, "Berlin", "Berlin"),
  city("Berlin", "Berlin", "D", 3472009), etc.
- queries are given as "patterns" with free variables:
  ?- country(N,C,Pop,Area,CapProv,Capital).
  yields a set of *answer bindings* for the variables N,C,Pop,Area,CapProv,Capital.
- Projection via *don't care* variables:
  ?- country(N,_C,Pop,_Area,_,_).
  yields a set of *answer bindings* for the variables N and Pop.
- Selection: ?- country("Germany", "D", Pop, Area,_,_). binds only Pop and Area.

Relational Calculus (cont'd)

- Selection as Conjunction:
  ?- country(N, C, Pop, _,_,_), Pop > 1.000.000. binds N, C, Pop
  ?- country(N, _, _Pop, _,_,_), _Pop > 1.000.000. returns only the set of names of countries with more than 1000000 inhabitants.
- Joins as conjunctions:
  ?- country(N,_C,_,_,Area,_,_), encompassed(_C,Cont,Perc), continent(Cont, ContArea).
  ?- country(_, "D",_,_,_,CapProv,Capital), city(Capital, CapProv, "D", Pop, _, _)

Datalog

- Views as "derived/virtual relations":
  ancestor(X,Y) :- parent(X,Y).
  ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
  ?- ancestor(X,Y).
  can express e.g. transitive closure (recursive rules and fixpoint semantics).
- flows_transitive(Name, Sea) :- river(Name, _, Sea, _).
  flows_transitive(Name, Sea) :- river(Name, River, _ , _), flows_transitive(River, Sea).
  ?- flows_transitive(River, Sea).

[see Datalog/tcRivers.P]

REMARK

The term "Database" does not only mean the relational model & SQL, but is a general notion:

- persistent storage
- mass data
- multiuser concepts
  - access control/safety
- transaction concepts
  - correctness/consistency
  - safety: rollback, recovery

... all these are *general concepts* that apply for the network data model, the relational model, the object-oriented model, and also for XML databases.

Chapter 2
Database Concepts and Extensions

- The notion of "semistructured data (SSD)" has mainly been coined by the "TSIMMIS" project (The Stanford-IBM Manager of Multiple Information Sources, 1995-2000; persons: J. Ullman, H. Garcia-Molina, J. Widom, Y. Papakonstantinou).
- The problem has already been investigated before in several areas and projects.

WHY SEMISTRUCTURED DATA?

... mainly two requirements:

1. *data integration* from different sources
   (late 1980s/early 1990s):
   - increasing networks
   - combination of contents of several databases
     * multi-database-systems
     * federated database systems
     * different schemata
     * mostly only different relational schemata,
       · partially also under the aspect of integration of metadata into the DML – this aspect is originally independent from semistructured data.
     * sometimes different data models ("legacy"-databases according to earlier data models)
     * since mid-90s increasingly data from the Web

Why Semistructured Data?

2. storage of "unregular" data:
   no fixed/homogeneous/known schema, many null values (e.g. biochemistry)
   - data exchange (B2B); standard formats e.g. for suppliers in automobile industry
   - partly also full-text portions,
   - management of document content
     * coarsely structured
       SGML (special form: HTML)
   - annotated binaries (graphics, audio, video, etc.)
   - mixed forms between databases and documents
     * collections: (tax) laws, partially in SGML
     * health care and clinical information systems

The Evolution towards XML

The evolution in the area of semistructured data and XML combined concepts, experiences and developments from many previous approaches:

- network data model, hierarchical model ("legacy"-databases),
- relational databases,
- object-oriented databases,
- distributed and federated databases,
- data integration (purely relational environments, or mixed ones),
- document management.

Different lines of evolution have been brought together with XML & friends:

⇒ (nearly) nothing new, but a perfect combination!

Textbook on "Databases" in general (but without document management and XML):

- R.Elmasri, S.Navathe: "Foundations of databases"/"Grundlagen von Datenbanksystemen". Pearson Studium, 3rd edition, 2002.

2.1 Early Databases: the Network Data Model

Situation 1960: first primitive "high-level" programming languages for "calculations"

- FORTRAN 1957: "formula translator"
- COBOL 1959: "common business-oriented language"

Goal: somehow store and organize lots of data:

- first development in the database system *IDS (Integrated Data Store)* at *General Electric* (Bachman & Williams, 1964; Turing-Award 1973)
- specification of the "Conference on Data Systems Languages Data Base Task Group (CODASYL)", 1971.
- products: e.g. VAX-DBMS (Digital Equipment)

Network Data Model

- data is stored in *data records*,
- classified by *data record types*, with attributes (name and datatype to be specified).

Country				
Name	Code	Population	Area	...

City		
Name	Population	...

- Sample data records:

"Germany"	"D"	83536115	356910	...

"Berlin"	"3472009"	...

- So far, the same as the mapping of entity types in the relational model.
- difference: the *organization* of the records (and their relationships) in the database ...

Relationships: Set Types

Relationships are represented as sets: “all B that are in a given relationship with a certain A”
(E.g. all cities in a given country)

Definition of set types:

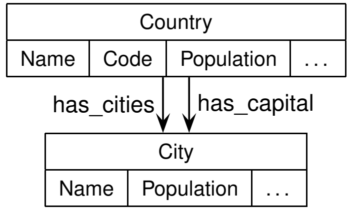
- name of the set type
- owner record type (“owner”; “where the relationship starts from”)
- member record type (“member”)

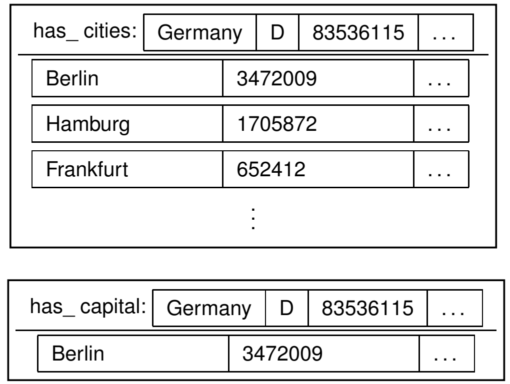
A set instance then represents the relationship for exact one owner of type “A”. Each instance of a set consists of

- a data record of the owner record type
- an *ordered* set of data records of the member record type
- comparison with XML: parent-children relationship, ordered children, but all of the same type
- intuitition: not as a set, but as a wire that is fixed at the owner and then pulled through all members.

Relationships: Set Types

Graphical representation:
“Bachman Diagram”





- similar sets for France//Paris/Lyon/Marseille/... and France//Paris
- a member record can belong to only one instance of a set of *each* set type (thus, only 1:N-relationships can be modeled directly)
- n:m relationships: later

Entry Points

- system-owned instances of a set serve as entry points
  (e.g. an instance of a set “countries” whose members are the country-data records)

Access Operations

Access to (and navigation through) the database only via sets.

Actually, this is again an *abstract datatype*:

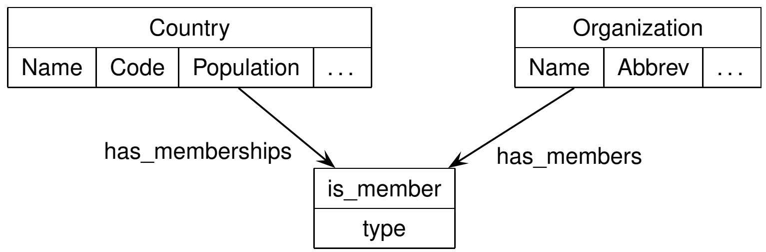
- access to the attribute values of a record,
- an *iterator* (first, next) for traversing the relationships ,
- a selector “find_owner” for inverse relationships.

⇒ the user does not explicitly work with pointers or identifiers, but already uses the semantic notions of the data model.

N:M-Relationships

Cannot be represented by a single set type (analogously for attributed relationships).

- split into a 1:M and an inverse N:1-relationship
  Problem: consistency maintenance (symmetry!)
- introduce an auxiliary data record type that represents the relationship, and two set types:



- later, there is a mapping from the ER model (1976) to the network model.

Organisation of the Set Types

Each data record contains reference entries for each set type where it belongs to (either as owner or as member):

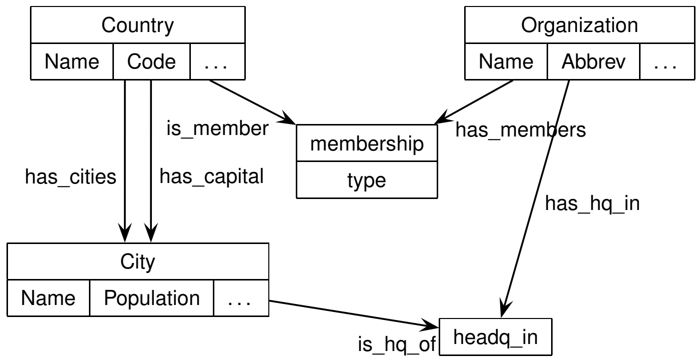
- as owner: a “first”-reference, labeled with the name of the set type, pointing to the first member record
- as member:
  - a “next”-reference, labeled with the name of the set type, pointing to the next member record
  - additionally a labeled backwards pointer to the owner of the set instance
  - a labeled null pointer if there exists no first/next element.

Exercise 2.1

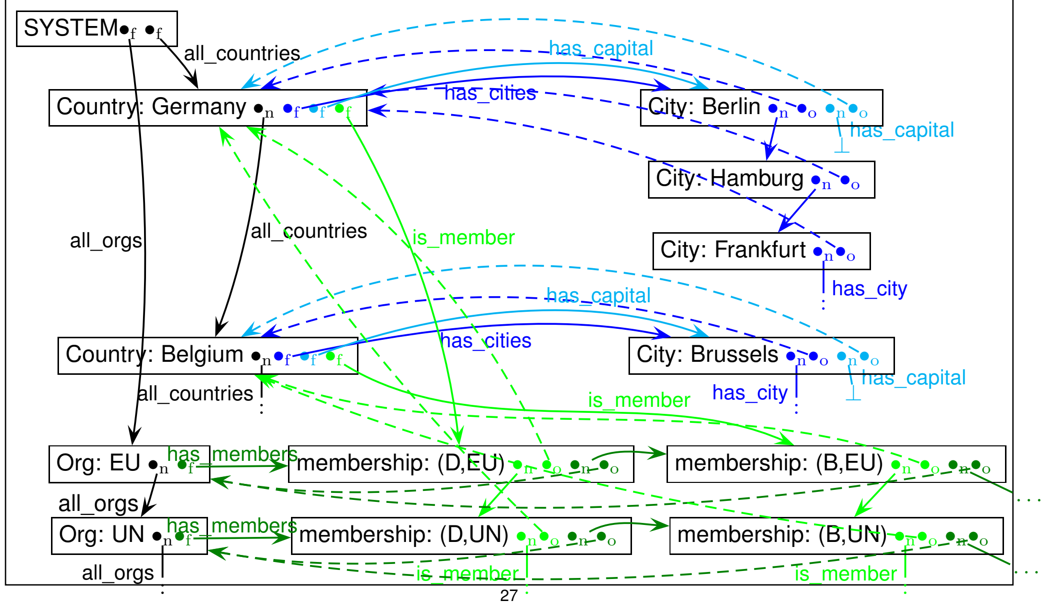
a) Visualize the model by drawing some country, city and organization data records.

b) Consider the “has_headq”-relationship that describes that organizations have their headquarters in a city. □

Solution: Network Schema Diagram



Solution: Instance Level



Data Definition Language: Example

RECORD NAME IS country
  DUPLICATES ARE NOT ALLOWED FOR Code
    Name TYPE IS CHARACTER 20
    Code TYPE IS CHARACTER 4
    Population TYPE IS NUMERIC INTEGER
    Area TYPE IS NUMERIC INTEGER

RECORD NAME IS city
    Name TYPE IS CHARACTER 25
    Population TYPE IS NUMERIC INTEGER

SET NAME IS all_countries
  OWNER IS SYSTEM
  MEMBER IS country

SET NAME IS has_cities
  OWNER IS country
  MEMBER IS city

Query and Data Manipulation Language

- record-at-a-time DML
- based on *iterators* (common design pattern/interface, e.g. in Java!) over sets
  - commands for navigation, access and data manipulation
  - embedded into a host language (COBOL, PL/I, later ... Pascal, C)
- “Current of” (cf. PL/SQL: “cursor”) that points to an instance of a record/set type in the DB
  - current of each record type
  - current of each set type (pointing on either the owner or one of the member records)
  - current of run unit (CRU): the record most recently accessed – any record type
- UWA (User Work Area) in the programming language runtime environment
  - one variable for each record type (auto-defined from the schema)
  - current of ... can be “fetched” into the corresponding UWA record

Retrieval and Navigation Commands

Query answering consists of stepwise navigation, carefully tracing currency indicators, and fetching tuples to the UWA:

- Retrieval: move the CRU into the corresponding UWA record,
- Navigation: navigate by using iterators and currency indicators to specific records and set owners/members.

Search for a Record of a Record Type

- FIND ANY <data record type> [USING <UWA.field.list>]
- FIND DUPLICATE <data record type> [USING <UWA.field.list>]
- tests/loops can be programmed by `IF/WHILE DBSTATUS=0` // 0: successfully found
- FIND sets all current of record/set type in which the record participates to that record. Can be avoided with RETAINING clause.

```
UWA.city.name = “Santiago”;
FIND ANY city USING name;
// sets also current of city indicator
while DBSTATUS=0 do begin
  GET city      // fetches data record into UWA.city
  if UWA.city.population > 1.000.000 then writeln (UWA.city.name|UWA.city.population);
  FIND DUPLICATE city USING name;
end;
```

- How to print out the city name and the country where it is located? Needs the “owner” of the city wrt. “has_cities”.

Search for a Record in a Set Type

- FIND (FIRST | NEXT | PRIOR | LAST) WITHIN <set type> [USING <UWA.field.list>]
- FIND OWNER WITHIN <set type>
- starts always from the current of this set (which is implicitly set when the CRU points to a suitable record type)

```
UWA.country.name = “Belgium”;
FIND ANY country USING name;
FIND FIRST city WITHIN has_capital
GET city      // fetches data record (Brussels) into UWA.city
writeln (UWA.city.name);
FIND OWNER WITHIN in_province
GET province      // fetches data record (Brabant) into UWA.province
writeln (UWA.province.name);
```

- Joins are only possible via navigation and loops in the host language.

Exercise 2.2
Write a program that outputs all organizations that have their headquarter in the capital of one of their member countries. Compare with the equivalent SQL query against Mondial. □

Updates

Updates on Data Records

STORE, ERASE, MODIFY (of the current data record)

Updates on Sets

CONNECT, DISCONNECT, RECONNECT (for the current data record wrt. a set)

Hierarchical Data Model

- In general very similar: parent-child-relationships define a tree structure; additionally, “virtual” parent-child-relationships.
- Systems: IMS (IBM & Rockwell International, 1969 for NASA Apollo), Adabas (Software AG, 1969), etc ...

Solution

```
// not tested
find any organization // sets current of has_headq, current of has_members
while ok do
{ get organization // current organization into UWA
  find first headq_in within has_headq_in // auxiliary record hq(org,cty)
  find owner within is_headq_of  // is a city
  find owner within has_capital // is a country
  if ok then // city is a capital
  { get country // UWA.country now holds this country
    found = 0;
    find first membership within has_members
        // starts from the organization
        // points to an auxiliary membership record m(org,c)
    while ok & not found do
    { find owner within is_member using code // UWA.country.code
       // check if the owner country is the same as in UWA
      if ok then { println(UWA.organization.name); found = 1;}
      find next membership within has_members
    }
  }
  find duplicate organization // next organization
}
```


The same in SQL

```
SELECT name
FROM organization org
WHERE (city,country) IN (SELECT capital, code
                         FROM country
                         WHERE code IN (SELECT country
                                        FROM is_member
                                        WHERE organization = org.abbreviation))
SELECT organization.name
FROM organization, is_member, country
WHERE organization.abbreviation = is_member.organization
  AND is_member.country = country.code
  AND organization.city = country.capital
  AND organization.country = country.code

SELECT organization.name
FROM   organization, country
WHERE  organization.city = country.capital
  AND  organization.country = country.code
  AND (abbreviation, code) IN (SELECT organization, country
                               FROM is_member)
```


Conclusion

- importance decreased rapidly since SQL came up (1979), in the meantime it is only present in “legacy systems”.
- no underlying theory (required as a base for normalization and optimization)
- only procedural, (data-model-level) navigation- and record-oriented query language, non-declarative, needs to be embedded into a host language (COBOL, PL/I, Pascal, C).
- not possible to state ad-hoc queries.
  Error-prone due to behavior of currency indicators.
- nevertheless, the idea of navigation and parent-child-relationships between data records is elegant (no problems with referential integrity).
  These concepts came up again in later approaches ... with high-level navigation!
- graph data model, “node + edge-labeled”
- expecially, ordered “child data records” are used again in XML. Then, there is
  - the DOM as an abstract datatype (stepwise, record-oriented),
  - XPath/XQuery as a *declarative*, set-oriented high-level language.

2.2 Object-Oriented Databases

Mid-80s: Object-orientation

- object-oriented design and modeling (UML)
- object-oriented programming (C++)

Application programs are developed and programmed in an object-oriented way.

- "impedance mismatch" between tuple-based SQL databases and the object-oriented data structures of the programming languages.

Goals:

- make objects of the application programs persistent
- bring object-orientation into the DBMS
  - class hierarchy and inheritance, polymorphism
  - implementation and encapsulation of behavior

Further Influences

- Networks: Internet and Intranets
- Interoperability and data exchange
- CORBA (1989) "Common Object Request Broker Architecture" (standardized by OMG – Object Management Group; predecessor of Web Services):
  - central ORB bus where services can connect
  - service registry (predecessor of WSDL and UDDI ideas)
  - description of service interfaces in object-oriented style (IDL - interface description language, similar to C++ declarations)
  - exchanging objects between services

⇒ requires a format for exchanging data:
Object interchange format - OIF (a predecessor of XML and of JSON (2006: RFC 4627; ECMA standard since 2013))

In this lecture, OODBS are only discussed shortly to sketch the central ideas.
An extended lecture can be found in "Information Systems", available at
`http://user.informatik.uni-goettingen.de/~may/Lectures`.

Lifetime of Objects

- Object-oriented programming language: Objects are created during runtime of an application program, and they are destroyed when the program terminates.

Objects in OO Database Systems

- persistent: objects that are created by an activity, and then they are stored in the database system and survive also the termination of the activity that created it (until they are explicitly destroyed by another activity)
- transient: objects that are only needed temporarily for executing an activity. They exist only as long as the application is actually active, and they are only managed by the runtime environment of the programming language.

Lifetime of Objects

- Relational DBMS: all SQL types have only persistent instances that are stored in the DBMS. All non-SQL types (i.e., types of the host language) have only transient instances, these are destroyed with the termination of the application-program (= when the host language is left).

  Persistent objects can only be manipulated/used by SQL, while transient objects can only be manipulated/used by the host language.

  ⇒ "impedance mismatch".

- ODBMS: object types of the DBMS and of the application coincide. They can have persistent and transient instances at the same time.

  For persistent and transient objects the same programming language and the same operations are used.

- comparison with XML: XML nodes can also be processed uniformly in the runtime environment and stored in a database. The DOM-API can be used in both cases.

Object-Oriented Data Model

- from the point of view as a *data model*, only the *(database) state* (*attributes, relationships, class membership and class hierarchy*) are relevant, not the behavior;
- representation of the current state of the application-domain,
- corresponding conceptual modeling language: UML (see Software Engineering)
- more expressive than the relational model/ER-model
- (behavior of objects is integrated into the data manipulation language)

OO-DBMS

Standardization activities similar to the standardization of relational databases:

Success of the relational database systems:

- not only by the simple, high-level data model,
- but also due to the standardization: SQL (at least after some time)
  - portability
  - interoperability

ODMG: Object Database Management Group

- founded 1991
- Architecture of OODBMS, DDL, query language (OQL), data formats
- ODMG-1.0 standard (1993)
- ODMG-2.0 standard (1997)
- ODMG-3.0 standard (2000); incremental changes

Literature: Cattell et al; Object Database Management (ODMG, 1993/1997/2001)

ODMG: Object Database Management Group

- Voting members: organizations/companies, who commercially work at an ODBMS, among others JavaSoft, Windward Solutions, Lucent Technologies, Unidata, GemStone, ObjectDesign, Versant, ...
  Reviewer members: Organizations who have a material interest in the work of ODMG.
- not the goal to define identical products, but to obtain source code portability (cf. Java, SQL, later also XML).
- enough freedom to define own properties and targets of products:
  - performance, optimization, (price)
  - support of certain programming languages,
  - functionality dedicated to special application areas (multimedia, CAD, ...), predefined types
  - integrated programming environments, design tools ...

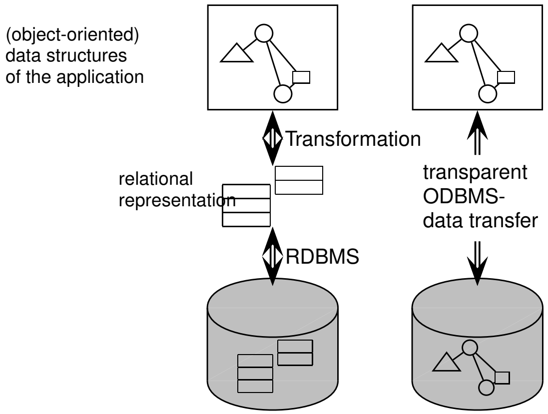
Architecture of ODBMS

- Different from “classical” relational DBMS:
  SQL: high-level language for data manipulation,
  applications are then written in other programming languages (cf. embedded approaches).
- ODBMS/ODM: transparent integration of DBMS functionality (persistence, multiuser, recovery) into application programming language (cf. Persistent Java).
  The objects of the application are simply stored in the database.
- no separate DML necessary. The application-level programming language *is* the DML.
- There is also a *set-oriented, declarative query language*
  (the impedance mismatch between variable-orientation and set-orientation remains): OQL
- no transformation between the (logical) database representation and the representation in the programming language (cf. datatype conversion in JDBC).

Architectures

ODMG is concerned with two types of products:

- Object Database Management Systems (ODBMSs) store the objects directly,
- Object-to-Database Mappings (ODMs) convert objects and store them in a relational (or any other) representation.



Remark:
There are similar approaches for XML databases.

ODMG-Standard

A standard that consists of several languages for implementation-level specification of object-oriented systems.

Components of the ODMG Standard

- Object specification languages/data model
  - Object Definition Language (ODL)
  - Object Interchange Format (OIF)
- Object Query Language (OQL) – based on SQL
- C++/Smalltalk/Java Language Binding
  specifies how to work with persistent objects in the target languages.

2.2.1 ODL: Object Definition Language

- Data definition language for object types:
- not a programming language, but only a language for definition of object specifications,
- characterizes object types (class hierarchy, properties and relationships)
- extends IDL (Interface Definition Language) from the OMG/CORBA (1989/1990) standard (which is in course closely related to the declaration commands in Java)

Data Types: Literals

Literals are only values, they have no *object identity*.

Atomic literals

- long, short, unsigned long/short, float, boolean, char, string,
- enumeration {...} (“type generator”)
  Z.B. enum Weekday {Sunday, Monday, . . . , Saturday}

Structured Literals

- predefined types: date, interval, time, timestamp
  (additionally to *actual object types* Date, Interval, Time, Timestamp)
- user-defined structural types, e.g. address or
  struct geoCoord { real latitude;
  real longitude; }

Collection literals

- set<t>, bag<t>, list<t>, array<t>, dictionary<t> – these are immutable “write once” (additionally to the *actual collection class types* Set, Bag, List, Array, Dictionary whose contents can be changed)

CLASSES

... are used to define and categorize complex object types.

Classes define the *signature* of their instances (the implementation does not belong to the object model):

class <name> { <attribute-defs>;
<relationship-defs>;
<operation-defs>;
}

<attribute-def> ::= attribute <domain-type> <attribute-name>

```
class City { attribute string name;      % attributes ...
             attribute number population;
             attribute geoCoord coordinates;
             relationship Country in_country;    % ... and relationships
           }
```


RELATIONSHIPS

- relationships are defined in course of the definition of classes.
- in UML and ODMG, only *binary* relationships are allowed.
- *bidirectional* and *inverse* relationships can be specified. Inverse relationships exist in UML, and later again in the Semantic Web languages (OWL).
- one-to-one / one-to-many / many-to-many-relationships.

class <name> {
<attribute-defs>;
<relationship-defs>;
<operation-defs>; }

<relationship-def>::= relationship <target_of_path> <relationship-name>
inverse <domain-type> :: <relationship-name'>
<target_of_path>::= <domain-type> |
<collection type> <<domain-type>>

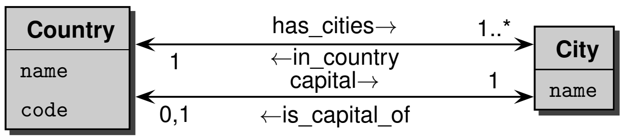
- <collection type> for -to-many-relationships

RELATIONSHIPS

class <name> { <attribute-defs>;
<relationship-defs>;
<operation-defs>; }

<relationship-def>::= relationship <target_of_path> <relationship-name>
inverse <domain-type> :: <relationship-name>
<target_of_path>::= <domain-type> |
<collection type> <<domain-type>>

```
class Country { attribute string name;
                relationship City capital   inverse City::is_capital_of;
                relationship set<City> has_cities   inverse City::in_country; }
class City { attribute string name;}
```



RELATIONSHIPS

- the instance level can be represented as a graph:
  - nodes: objects; nodes have labels (names of the object types) and an ID
  - edges: relationships; edges have labels (names of relationships)
- ODBMS is responsible for maintaining referential integrity:
  If an object is deleted, all relationships with/to it must also be deleted.
- relationships define *access paths*, e.g. Germany.capital for navigation through the graph.
- graph-data model, "node + edge-labeled"
- The set<...> is very similar to the set-oriented representation of set-valued relationships from the network data model (→ handled by iterators)
- the query language OQL solves this problem SQL-like in a declarative way (see later).

Exercise 2.3
Visualize an excerpt of the Mondial database as an object graph. □

2.2.2 Object Interchange Format (OIF)

- serialize into a charecter stream for exchanging one or more objects with another application,
- serialize fo a file: dump the database state to one or more files (cf. *export* in ORACLE),
- specification language for persistent objects and their states,
- OIF output contains for each object its type, its attributes and values, and its relationships to other objects,
- the database schema (class definitions *and* class hierarchy) is *not* represented in OIF!

Object Interchange Format (OIF)

- Simplest form: only the class membership

  <object> <class> {}
  Germany Country {}
  Berlin City {}

- attribute values are enumerated in braces:

  Germany Country {name "Germany", area 356910, ...}

- structured attributes: nested brace structures

```
struct geoCoord { real latitude;
                  real longitude; }

class City { attribute string name;
             attribute geoCoord coordinates;
             relationship Country in_country; }

Berlin City {name "Berlin", in_country Germany,
             coordinates {latitude 52.45, longitude 13.3} }
```


Object Interchange Format (OIF)

- Collections, set-valued relationships:

```
class Country {
      attribute string name;
      attribute integer area;
      City capital;
      relationship set<City> has_cities; }

Germany Country
    {name "Germany", area 356910,
     capital Berlin,
     has_cities {Berlin, Frankfurt, Freiburg, ...} }
```

- cyclic references: no problem.
- attributed relationships (e.g. border) cannot be represented directly

⇒ OIF is already a self-describing data format!

2.2.3 OQL (overview)

- Query language of the ODMG standards (Object Query Language)
- similar to SQL:
  SELECT - FROM - WHERE - clause, extended by complex objects, object-identity, path expressions, polymorphism, operation calls and late binding.
- but: functional language (like SQL), fully orthogonal (in SQL not completely)
- no explicit UPDATE statement: instead, object methods are used
- not Turing complete (cf. SQL/transitive closure)
- OQL can be embedded into suitable object-oriented programming languages (C++, Java, Smalltalk). Results of queries (collections!) are then processed by iterators.

EXTENTS

SQL: SELECT ... FROM <relation> ...

What corresponds to a *relation* in an ODBMS ?

⇒ *Extension*: set of all instances of a class (similar to system-owned sets in NWDBMS).
Extensions are defined in ODL together with the class declaration:

```
class <name> (extent <extent_name>)
{  <attribute-def>;
   <relationship-def>;
   <operations-def>; }
```

```
class Country (extent Countries)
{  attribute string name;
   relationship City capital;
   set<string> languages;
   ...}
```


QUERIES

Queries against the database are expressed with the SELECT statement, with the same simple basic structure as in SQL:

```
SELECT  <expression>
FROM    <extents>
WHERE   ...
```

```
SELECT c.population
FROM Countries c
WHERE c.name = 'Germany'
```

- with an iterator variable (here: c) – cf. SQL Aliasing

Similar to SQL:

- DISTINCT, aggregate functions: COUNT, SUM, . . . , set functions: UNION, INTERSECT, EXCEPT (MINUS)

QUERIES

- SQL: all results of queries of the form

  SELECT a,b,c FROM ...

  are virtual "relations" (i.e. sets of tuples),
- OQL: the result is a virtual set of objects,
- in most cases an (implicit) collection.

```
SELECT c.capital
FROM Countries c
```

Result is of the type

collection <City>

⇒ queries can be nested arbitrarily (like in SQL)

QUERIES

in case that the result has more than one attribute (e.g. with SELECT *), a

bag <struct{...}>

is automatically generated:

```
SELECT c.name, c.population
FROM Countries c
WHERE c.area > 100000
```

Result is of the type

bag <struct {string name; number population}>

COMPLEX RESULTS

- bags (here: set-valued relationship) can be handled as a whole,
- by explicit generation of a struct, the properties of the result can be renamed:

```
SELECT struct(name: c.name,
              cities: c.has_cities)
FROM Countries c
```

result is of the type

```
collection <struct {string name;
                    collection<City> cities}>
```

How can something *in the collection* be selected?

... straightforwardly: apply a SELECT statement to the collection:

```
SELECT struct(name: c.name,
              cities: (SELECT cty
                       FROM c.has_cities as cty
                       WHERE cty.population>1000000))
FROM Countries c
```

- Traversing the relationship *has_cities* by a *path expression* in the query
- nested SELECT in the SELECT statement: the inner SELECT ranges over the (virtual) set *c.has_cities* of instances of type *set*<*City*>.
- the inner SELECT is evaluated separately for every result (i.e. for each instance c) of the outer SELECT.

PATH EXPRESSIONS

for navigation along *scalar* relationships:

```
SELECT name: c.name,
       cpp: c.capital.province.population
FROM Countries c
```


SELECT IN THE FROM-CLAUSE

Navigation along *set-valued* relationships:

not allowed:

```
SELECT name: c.has_cities.name,
       pop: c.has_cities.population
FROM Countries c
WHERE c.name = "Germany"
```

has_cities is a *set of cities*, thus, the method *population* cannot be applied (to the set).

This can be done e.g. by a SELECT statement in the FROM-clause:

```
SELECT name: cty.name,
       pop: cty.population
FROM (SELECT c.has_cities
      FROM Countries c
      WHERE c.name = "Germany") as cty
```

Correlated Joins

... do the above example even better:

SELECT name: cty.name,
    pop: cty.population
FROM Countries c, c.has_cities cty
WHERE c.name = “Germany”

This would be a nice feature also in SQL ... the right side of the join is computed dependent on the left one.

⇒ asymmetric joins that express nested iteration in a declarative way

⇒ not aligned with the relational algebra

OQL: Functional Language Concept

SQL:

- declarative, relational algebra as theoretical base,
- somewhat ad-hoc language (around SELECT – FROM – WHERE),
- not completely orthogonal composition (aggregate functions, method applications)

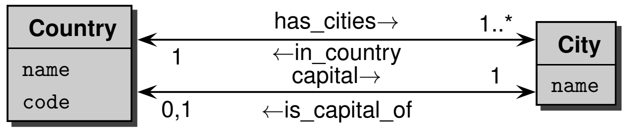
OQL:

- orthogonal composition rules: operators can be nested as long as the type system is not violated
- functional concept, includes the *simple* queries in SQL syntax.
- result of a query is always a
  collection()
- can be processed in the same way as an extension (*intensional* part of the database).

Conclusion

- Object-oriented databases have not been accepted by the market.
- Products: ObjectStore, Adabas, O2, GemStone, Poet, ...
  Some of them served as the base for the first commercial XML database systems (Excelon, Tamino [Software AG]).
- Object-relational extensions to SQL and relational systems (SQL-3-Standard): evolutionary instead of revolutionary development.
- graph data model, “node + edge-labeled”
- set-oriented (extents similar to relations) and navigation-based access, integrated in a declarative language.
  Problems with navigating along set-valued properties.
- OQL as a functional language with fully orthogonal constructs and the possibility to generate structures in the SELECT-clause.
  The XML-Query language XQuery will be very similar ...
- OIF as self-describing character-based data exchange format (usually, ISO 8859-1, Latin), but still with a fixed schema.

2.2.4 Analysis: 1:n-Relationships

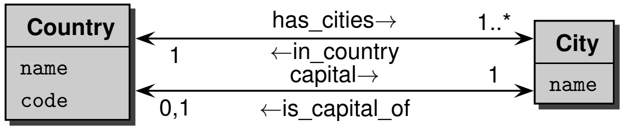


class Country { attribute string name;
    relationship City capital   inverse City::is_capital_of;
    relationship set<City> has_cities   inverse City::in_country; }

class City { attribute string name;}

- correct: germany.capital.name
- not correct: germany.has_cities.name
- translation to set<City> “country is in relation with a set of cities” is a tribute to programming language influence: must be something that exists in programming languages and that can be bound to a single variable.
  “set-valued” – one answer which is a set.
- applying “.name” to a set is obviously not correct.

Alternative Translation

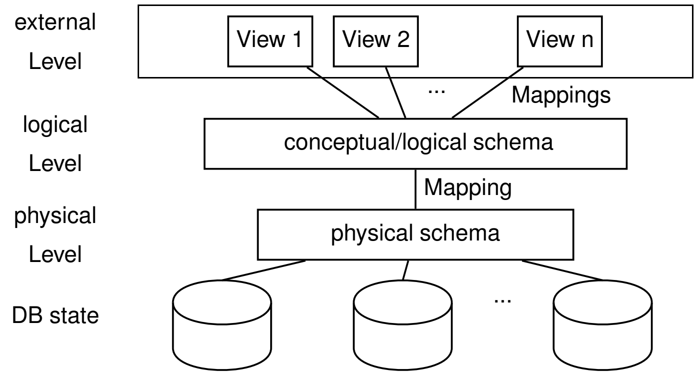


- database style: "country is in relation with multiple cities"
  "multi-valued" – a set of *answers*, each of them is a city,
- "set of answers" is a meta-concept of the query language, not of the underlying programming language,
- applying ".name" to a set of answers can be defined by the semantics of the query language!
- "Modern" query languages change to multivalued semantics:
  – F-Logic (1989, see later): germany.has_cities.name,
  – XPath (for XML, 1998): //country[name="Germany"]/province/city/name,
  – semantics of path expressions stays within the semantics of the query language.

2.3 Data Integration and Metadata Queries: SchemaSQL

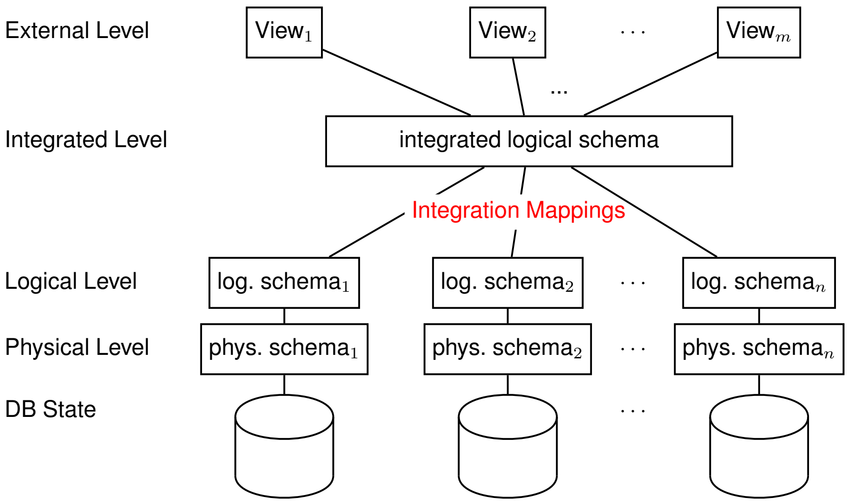
2.3.1 Introduction

- So far: single databases
- according to the classical 3-level architecture



Multidatabase Systems and Federated Database Systems

- providing a common, integrated view over several databases



Data Integration and Metadata Queries in SQL: SchemaSQL

SchemaSQL (Lakshmanan et al. 1996; non-commercial academic system) extends SQL:

- combination of relations and attributes of different (federated) databases.
- uniform handling of data and metadata (by SchemaSQL variables).
- possible domains of variables are the names of the components of a federation, names of the relations of a database, names of the attributes of a relation, tuples of a relation, and values of a column of a relation.
- additionally to the "vertical" aggregations over columns, also "horizontal" aggregations over relations or even tables are possible.

Example

univ-A

salInfo

category	dept	salary
Prof	CS	65,000
Assoc Prof	CS	50,000
Prof	Math	60,000
Assoc Prof	Math	55,000

univ-C

CS

category	salary
Prof	60,000
Assoc Prof	55,000

Math

category	salary
Prof	70,000
Assoc Prof	60,000

univ-B

salInfo

category	CS	Math
Prof	55,000	65,000
Assoc Prof	50,000	55,000

univ-D

salInfo

dept	Prof	Assoc Prof
CS	75,000	60,000
Math	60,000	45,000

2.3.2 Declaration of Variables

... as known from SQL in the FROM-clause: FROM <range> <var>

SQL: FROM <table> <var>

SELECT cty.name, cty.population
FROM City cty

$$< \text{range} > \in \{\rightarrow, \text{db} \rightarrow, \text{db} :: \text{rel}, \text{db} :: \text{rel} \rightarrow, \text{db} :: \text{rel.attr}\} .$$

- $\rightarrow$: names of the databases of the federation
- $db \rightarrow$: names of the relations of the database db.
- $db :: rel$: tuples of the relation rel of the database db [as in SQL].
- $db :: rel \rightarrow$: names of the attributes of the schema of the relation rel of the database db.
- $db :: rel.attr$: values of the column of the attribute $attr$ of the relation rel of the database db.
- SchemaSQL: iterated declarations of variables are allowed!
  $\Rightarrow$ joins not longer symmetrical (cf. OQL).

Declaration of Variables: Tuple- and Domain Variables

- tuple variables as known from SQL:
  $db :: rel$ ranges over the set of tuples of the relation rel of the database db.

  SELECT tuple.category, tuple.salary
  FROM univ-C::CS tuple

category	salary
“Prof”	60000
“AssocProf”	55000

- Domain-Variables:
  $db :: rel.attr$ ranges over the set of values of the attribute $attr$ of the relation rel of the database db

  SELECT cat
  FROM univ-A::salInfo.category cat

cat
“Prof”
“AssocProf”

Note: SQL-style SELECT category FROM univ-A::salInfo yields the same result – but does not allow to bind the values to a variable (that can be used somewhere else)

Declaration of Variables: Metadata Variables

- $db \rightarrow$ ranges over the relation names of the database db.

  SELECT relname FROM univ-C $\rightarrow$ relname

relname
“CS”
“math”

- Nested declarations: Second variable depends on the first one:

  SELECT dept, tuple.category, tuple.salary
  FROM univ-C $\rightarrow$ dept, univ-C::dept tuple

dept	category	salary
“CS”	“Prof”	60000
“CS”	“AssocProf”	55000
“Math”	“Prof”	70000
“Math”	“AssocProf”	60000

... *integrates* both tables from univ-C in one.

Declaration of Variables

- Variables over names of attributes:
  $db :: rel \to$ ranges over the set of attribute names of the schema of the relation rel of the database db.

  SELECT attrname
  FROM univ-C::CS → attrname

attrname
"category"
"Salary"

- `SELECT C`: *name* of the attribute,
  `SELECT T.C`: *value* of the respective attribute of the current tuple.

  SELECT attrname, univ-C::CS.attrname
  FROM univ-C::CS → attrname

"category"	"Prof"
"category"	"AssocProf"
"Salary"	60000
"Salary"	55000

Declaration of Variables

- → ranges over the names of the databases of the federation.

  SELECT dbname FROM → dbname

dbname
"univ-a"
"univ-b"
"univ-c"
"univ-d"

- SELECT dbname, relname
  FROM → dbname, dbname → relname

dbname	relname
"univ-A"	"SalInfo"
"univ-B"	"SalInfo"
"univ-C"	"CS"
"univ-C"	"math"
"univ-D"	"SalInfo"

2.3.3 Queries

All departments of Univ-A that pay a higher salary to their professors than the corresponding departments of Univ-B:

```
select A.dept
   - all variables are independent
from   univ-A::salInfo A, univ-B::salInfo B,
       univ-B::SalInfo-> AttB
where     AttB <> "category" and
          A.dept = AttB and
          A.category = "Prof" and
          B.category = "Prof" and
          A.salary > B.AttB.
```


Queries (Cont'd)

Same for C/D:

```
select RelC
  - C depends on RelC
from    univ-C-> RelC, univ-C::RelC C,
        univ-D::salInfo D
where   RelC = D.dept and
        C.category = "Prof" and
        C.salary > D.Prof
```

Aggregation

Similar to SQL, there can be aggregation over a variable.

⇒ here also *horizontal* and *blockwise* aggregation possible.

Average salary for each kind of professors over all departments of Univ-B:

```
select T.category, avg(T.D)
from univ-B::salInfo→D,  univ-B::salInfo T
where D <> "category"
group by T.category
```

- select the values for D,
- compute the cartesian product with univ-B::salInfo T
- include column T.D
- evaluate, do the grouping, compute the aggregate

D	category	CS	Math	T.D
category	Prof	55,000	65,000	55,000
CS	Prof	55,000	65,000	55,000
math	Prof	55,000	65,000	65,000
category	Assoc Prof	50,000	55,000	50,000
CS	Assoc Prof	50,000	55,000	50,000
math	Assoc Prof	50,000	55,000	55,000

Aggregation

Average salary for each kind of professors over all departments of Univ-C:

```
select T.category, avg(T.salary)
from univ-C→D, univ-C::D T
group by T.category
```

- compute values for D,
- join with tuple variable D T

D	category	salary
CS	Prof	60,000
CS	Assoc Prof	55,000
math	Prof	70,000
math	Assoc Prof	60,000

- grouping
- compute the aggregate

Restructuring

... as usual via views:

```
create view
BtoA::salInfo(category, dept, salary) as
     select T.category, D, T.D
     from univ-B::salInfo→D, univ-B::salInfo T
     where D <> 'category'
```

creates a virtual database BtoA with a virtual relation salInfo in the same format as A::salInfo.

Restructuring

A to B: number of attributes of the result table depends on the number of departments.
⇒ Dynamic result schema

```
create view AtoB::salInfo(category,D) as
select A.category, A.salary
from univ-A::salInfo A, A.dept D
```

Result of the FROM-clause:

A.category	A.salary	A.dept D
Prof	65,000	CS
Assoc Prof	50,000	CS
Prof	60,000	Math
Assoc Prof	55,000	Math

Many-to-one-mapping into a schema of the form salInfo(category, $\text{dept}_1, \ldots, \text{dept}_n$).

AtoB::salInfo

category	CS	Math
Prof	65,000	60,000
Assoc Prof	50,000	55,000

2.3.4 Exercise

Create the following view that represents the information of all four databases in a uniform way:

```
create view
globalSchema::salInfo(univ, dept, category, salary) as
[TO BE COMPLETED]
```


SOLUTION

```
create view
globalSchema::salInfo(univ, dept, category, salary) as
   select "univ-A", T.dept, T.category, T.salary
   from univ-A::salInfo T
union
   select "univ-B", D, T.category, T.D
   from univ-B::salInfo T, univ-B::salInfo→D
   where D<>"category"
union
   select "univ-C", T, T.category, T.salary
   from univ-C→D, univ-C::D T
union
   select "univ-D", T.dept, C, T.D
   from univ-D::salInfo T, univ-D::salInfo→C
   where C<>"dept"
```


2.3.5 Query Evaluation

Federation System Table (FST): meta-information about the component databases, i.e. names of the databases, relations, attributes, or other statistical information that is useful for query evaluation (similar to the Data Dictionary in SQL).

Variable Instantiation Tables (VIT): contain the possible variable bindings during the evaluation (meta level).

Input: a *SchemaSQL* query

Output: bindings of the variables of the SELECT-clause of the query

Evaluation: two phases:

1. generation of the VITs according to the variables in the FROM-clause. For this, SQL queries are stated against the local databases and against the FST.
2. rewriting of the *SchemaSQL* query into an equivalent query using the VITs (Dynamic SQL). This query is then evaluated by the *resident SQL server*.

EVALUATION: EXAMPLE

```
select RelC
from univ-C→ RelC, univ-C::RelC C, univ-D::salInfo D
where RelC = D.dept and C.category = "Prof" and C.salary > D.Prof
```

Bindings for meta-variables (query against an FST):

VIT_{RelC}
RelC
CS
Math

Bindings for tuple variables (queries against component-DBS):

VIT_C (depends on Rel_C)		
RelC	category	salary
CS	Prof	60,000
CS	Assoc Prof	55,000
Math	Prof	70,000
Math	Assoc Prof	60,000

VIT_D		
Dept	Prof	AssocProf
CS	75,000	60,000
Math	60,000	45,000

Evaluation: Example

... again the query:

```
select RelC
from univ-C→ RelC, univ-C::RelC C,
   univ-D::salInfo D
where RelC = D.dept and
      C.category = "Prof" and
      C.salary > D.Prof
```

Query evaluation via standard SQL over the $VIT's$.

```
select VIT_RelC.RelC
from VIT_RelC, VIT_C, VIT_D
where VIT_C.RelC = VIT_RelC.RelC    % Correlation RelC, C
   and VIT_RelC.RelC = VIT_D.dept
   and VIT_C.category = "Prof"
   and VIT_C.salary > VIT_D.Prof
```


Exercise: Schema-SQL

Describe the evaluation of the query given on Slide 76 with its FST and VITs.

Solution

VIT_{dbname}
dbname
univ-A
univ-B
univ-C
univ-D

$VIT_{relname}$	
dname	relname
univ-A	salInfo
univ-B	salInfo
univ-C	CS
univ-C	math
univ-D	salInfo

SELECT VIT_{dbname}.dbname, $VIT_{relname}$.relname
FROM VIT_{dbname}, $VIT_{relname}$
WHERE VIT_{dbname}.dbname = $VIT_{relname}$.relname

2.3.6 Example: Integration of Stock Exchange Data

Frankfurt::Quota		
Date	Name	Price
3.3.93	sun	150
3.3.93	dc	151
3.3.93	b.u.	160
4.3.93	sun	153
4.3.93	dc	154
4.3.93	b.u.	163

Tokyo::Quota			
Date	sun	dc	fuji
3.3.93	150	151	140
4.3.93	153	154	140

Sydney::3.3.	
Name	Price
sun	150
dc	151
kiwi	130

Sydney::4.3.	
Name	Price
sun	153
dc	154
kiwi	135

New York::sun	
Date	Price
3.3.93	150
4.3.93	153

New York::dc	
Date	Price
3.3.93	151
4.3.93	154

New York::msoft	
Date	Price
3.3.93	148
4.3.93	74

Possible extension:
Euro vs. Dollar vs. Yen

Exercise: Schema-SQL

- Formulate the “On which days had which stocks the price of 150 $?” for the schemata given on Slide 91.
- In commercial database systems, the schema information is stored in the *Data Dictionary* (cf. the following excerpts of table definitions of the data dictionary):

```
SQL> desc sys.user_tables;
 Name                     Null?    Type
 ------------------------ -------- ----
 TABLE_NAME               NOT NULL VARCHAR2(30)
 SQL> desc sys.user_tab_columns;
 Name                     Null?    Type
 ------------------------ -------- ----
 TABLE_NAME               NOT NULL VARCHAR2(30)
 COLUMN_NAME              NOT NULL VARCHAR2(30)
 DATA_TYPE                VARCHAR2(30)
```

Describe how the above queries can be formulated in an environment where SQL is embedded into a procedural programming language (e.g. embedded-SQL or PL/SQL) (Pseudocode).

Solution: Schema-SQL

- SELECT Date, Name
  FROM Frankfurt::Quota
  WHERE Price=150;

  SELECT Date, AttrName
  FROM Tokyo::Quota.Date, Tokyo::Quota → AttrName
  WHERE AttrName ≠ 'Date' AND Price=150;

  SELECT NewYork::TabName.Date, TabName
  FROM NewYork → TabName
  WHERE Price=150;

  SELECT TabName, Sydney::TabName.Name
  FROM Sydney → TabName
  WHERE Price=150;

- Information from the Data Dictionary is only needed for Tokyo, New York and Sydney.

Solution: SQL

Algorithm for SQL in a procedural environment (database Tokyo):

- Store the result of

  SELECT ColumnName
  FROM Tokyo.user_tab_columns
  WHERE ColumnName ≠ 'Date';

  (result: the names of the companies) and for each result <cn> execute the query

  SELECT Date, <cn>
  FROM Tokyo.Quota
  WHERE <cn>= 150;

  and collect all results.

Solution: SQL

- database "New York": store the result of

  SELECT TableName
  FROM user_tables
  WHERE
  (SELECT ColumnName
  FROM user_tab_columns UTC
  WHERE UTC.TableName=TableName = {Date,Price});

  (the comparison of sets must be formulated in SQL) and for each result <tn> evaluate the query

  SELECT Date, <tn>
  FROM <tn>
  WHERE Price = 150;

  and collect all results.

Problem: SQL statements must be generated *dynamically*: the results of the first query are used in the second statement.

Solution: Dynamic SQL

This is e.g. possible in Oracle by using the DBMS_SQL-Package (to be used with PL/SQL), which allows to generate SQL statements at runtime:

```
create procedure findnumber as
declare
 cursor col_cursor is
   select column_name, data_type
   from sys.user_tab_columns
   where table_name = upper('&&table_name')
   order by column_id;
 lv_column_name   sys.user_tab_columns.column_name%TYPE;
 lv_column_typ    sys.user_tab_columns.data_type%TYPE;
 lv_rowid         varchar2(20);
 rows_processed   number;
 loop_count       number;
 stmnt            varchar2(2000);
 doublecur        BINARY_INTEGER;
 execute_feedback INTEGER;
 type colname_typ is table of lv_column_name%TYPE
      index by binary_integer;
 type rowid_typ is table of lv_rowid%TYPE
```

```
     index by binary_integer;
 colname_table      colname_typ;
 empty_colname      colname_typ;
 rowid_table        rowid_typ;
 empty_rowid_table  rowid_typ;

begin
   DBMS_OUTPUT.ENABLE(10000);
   rows_processed := 0;
   -- Search for attributes with datatype "Number"
   open col_cursor;
   loop
      fetch col_cursor into
         lv_column_name, lv_column_typ;
      exit when col_cursor%notfound;
      IF lv_column_typ='NUMBER' THEN
         rows_processed := rows_processed+1;
         colname_table (rows_processed)
              := lv_column_name;
      END IF;
   end loop;
   close col_cursor;

   -- Initialize query statement
   stmnt := 'select rowid from '
       || '&&table_name '
```



```
       || 'where ';

   -- generate the query iteratively
   loop_count := 1;
   WHILE loop_count <= rows_processed
   loop
      stmnt := stmnt
          || colname_table(loop_count)
          || ' = &&Price';
      if loop_count < rows_processed
      then
        stmnt := stmnt || ' or ';
      end if;
      loop_count := loop_count + 1;
   end loop;
   DBMS_OUTPUT.PUT_LINE
        ('Computed Query: ' || stmnt);

   -- execute the generated statement
   doublecur := DBMS_SQL.OPEN_CURSOR;
   DBMS_SQL.PARSE (doublecur
            ,stmnt
            ,DBMS_SQL.V7);
   DBMS_SQL.DEFINE_COLUMN
        (doublecur, 1, lv_rowid, 20);
   execute_feedback := DBMS_SQL.EXECUTE (doublecur);
```



```
   -- generate list of all resulting data records and
   -- RowIDs
   loop
     if DBMS_SQL.FETCH_ROWS (doublecur) = 0
     then
       exit;
     else
       DBMS_SQL.COLUMN_VALUE (doublecur,1, lv_rowid);
       DBMS_OUTPUT.PUT_LINE('RowID: ' ||lv_rowid);
     end if;
   end loop;

   -- cleaning ...
   DBMS_SQL.CLOSE_CURSOR (doublecur);
   colname_table := empty_colname;
end;
/
```


Solution: Dynamic SQL

```
SQL> execute find-number;
Give value for table_name: Tokyo
Give a value for price: 150
Generated Query:
  select rowid from Tokyo
  where SUN = 150 or DC = 150 or FUJI = 150

RowID: AAAA2MAADAAAD7nAAA

SQL> select * from Tokyo
     where rowid='AAAA2MAADAAAD7nAAA';
03.03.93        150        151       140
```

which must still be postprocessed for obtaining the answer `'sun', 3.3.93`.

- Conclusion: SchemaSQL helps to express such queries much shorter and more concise, and it is easier to learn than PL/SQL and DBMS_SQL.

2.3.7 Exercise: Horizontal and blockwise Grouping

- Consider the schemata `univ-B`, `univ-C` and `univ-D`. Give SchemaSQL queries that return for each kind of professors the average salary over all departments.

Solution: Horizontal and blockwise Grouping

- univ-A: same as in standard SQL: vertical aggregation:

  select T.category, avg(T.salary)
  from univ-A::salInfo T — tuple variable
  group by T.category

- univ-B: horizontal aggregation
  see Slide 81.

- univ-C: aggregation over different tables
  see Slide 82.

- univ-D: aggregation over different columns:

```
select T.category, avg(T.C)
from univ-B::salInfo T,   univ-B::salInfo → C
where C <> "dept"
group by C
```


Conclusion

- integration of *relational* databases with different schemas
- queries against metadata
- combination of metadata and data
- data-dependent *generation* of schema

New Features

Generalization of the use of variables:

- SQL: variables only ranging over tuples of a fixed relation,
- SchemaSQL: variables ranging over "everything": data: tuples, column values metadata: names of columns, names of relations, even names of databases,
- intuitively simple extension of SQL,
- powerful feature for data integration,

– But: classical query optimization/evaluation not applicable.

Such variables are more (F-Logic) or less extremely (XML: XPath/XQuery) used in Semistructured Data and XML.

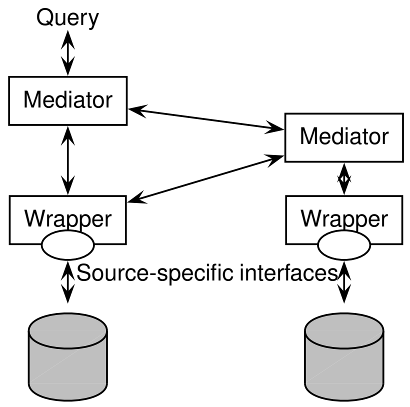
Chapter 3
Semistructured Data: Early Approaches

- Data integration
  - different, autonomous data sources
  - *different data models and schemata*
  - more advanced than the approach of SchemaSQL
- Knowledge representation, data exchange
  - schema- and meta-information inside the data
  - examples: KIF (Knowledge Interchange Format), F-Logic
  - up to ontology management ("Semantic Web")
- Management of data for presentation on the Web
- Extraction of data from the Web

SSD for Data Integration/Data Exchange:

Wrapper/Mediator-Based Architectures

- *Mediator* (Vermittler): between users and data sources (Middleware),
- *Wrapper (Translator)*: provides homogeneous access to heterogeneous sources
  (especially for information extraction from the Web:
  programming of wrappers for Web pages and then collect the data)



Wrapper/Mediator-Based Architectures

- sources: databases, interfaces to databases via forms (e.g. library search), search engines, simple Web pages
- each relevant Web source is associated to a wrapper
- mediator contains knowledge about the accessible sources
- mediators can be composed hierarchically

Virtual Approach

The users state queries against the upper level mediator (“external view”) which translates the queries against lower mediators and wrappers. Wrappers answer the queries from the sources. Mediators combine the answers and return them.

Materialized Approach

An integrated view of all data is completely materialized (and maintained). Users state their queries against the materialized database that directly answers them.

Requirements for Data Integration

- upper mediator level: a target data format
- interfaces between wrappers/mediators
  - a common data exchange format
  - a common query language/mechanism
- wrapper level: mapping from sources into the common format

Target Data Model and Languages

- flexible and extensible
  - “copy all properties of object X from data source A”
  - extensible to additional sources
  - different source data models and schemata
- handling metadata and content in combination
- self-describing data !?

3.1 TSIMMIS

(The Stanford-IBM Manager of Multiple Information Sources, 1995-2000)

Persons: J. Ullman, H. Garcia-Molina, J. Widom, Y. Papakonstantinou, etc.

Goal (several subprojects): construction of means for a consistent and efficient integrated access to information sources:

- Heterogeneous information sources
  - databases
  - Web pages
- ⇒ often no explicit schema known/present
- ⇒ mapping to a *common data model*:
  Object Exchange Model (OEM)

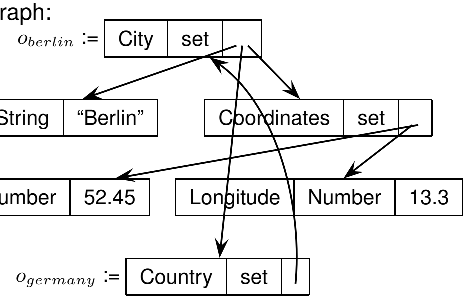
TSIMMIS: Concept

“Virtual” approach:

- users state queries against a mediator
- mediator forwards the subqueries to lower mediators or wrappers
- wrappers are programs that (logically) transform the objects of the data source into OEM and then answer the basic queries
- results of the wrappers are returned in OEM format to the mediator
- mediator integrates the results of the sources
- mediators can be composed hierarchically

3.1.1 OEM: Object Exchange Model

- very simple, “*self-describing*” object model
- knows only object identity and nesting as concepts:
- each object has an (optional) object-ID, a label ($\sim$ class), a (data)type and a value,
- values of complex types are sets of references to sub-objects
- labels: “self-describing data”
- top-level objects with semantic object identifiers as entry points (cf. OQL)
- can be represented as a graph:



OEM: EXAMPLE

Source 1: CIA World Factbook

Wrapper cia exports OEM objects as follows:

<&cont1, continent, set, {&a1, &n1, &c1, &c2, &c3, ...}>
<&n1, name, string, ‘Europe’>
<&a1, area, number, 9562488>
<&c1, country, set, {&cn1, &cc1, &ca1, &cp1, &cap1}>
<&c2, country, set, {&cn2, &cc2, &ca2, &cp2, &cap2}>

<&cn1, name, string, ‘Germany’>
<&cc1, code, string, ‘D’>
<&ca1, area, number, 356910>
<&cp1, population, number, 83536115>
<&cap1, name, string, ‘Berlin’>

<&cn2, name, string, ‘Sweden’>
<&cc2, code, string, ‘S’>
<&ca2, area, number, 449964>
<&cp2, population, number, 8900954>
<&cap2, name, string, ‘Stockholm’>

OEM: Example

Source 2: Global Statistics

Wrapper gs exports OEM objects as follows – also nesting is allowed:

<&cont1, continent, set, {&a1, &n1, &c1, &c2, &c3, ...}>
<&n1, name, string, ’Europe’>
<&c1, country, set, {&cn1, &ct11, &ct12, &ct13, ..., &prov11, &prov21,...}>
<&c2, country, set, {&cn2, &ct21, &ct22, &ct23, ..., &prov12, &prov22,...}>

<&cn1, name, string, ’Germany’>
<&ct11, city, set, { <name, string, ’Stuttgart’> <population, number, 588482>, &prov11 }>
<&ct12, city, set, { <name, string, ’Freiburg’> <population, number, 209628>, &prov11 }>
<&ct13, city, et, { <name, string, ’Berlin’> <population, number, 3292365>, &prov12 }>
<&prov11, province, set, { <name, string, ’Baden-Württemberg’>, <area, number, 35742>,
<population, number, 10272069>, &ct11, &ct12 ...}>
<&prov12, province, set, { <name, string, ’Berlin’>, <area, number, 891>,
<population, number, 3574830>, &ct13 ...}>

OEM

- another version of OEM has been presented that additionally allows for labeled edges; e.g. for *capital*-edges from a country to a city.

Exercise 3.1
Visualize an excerpt of the Mondial database with some countries, cities, continents and organizations as an OEM graph. □

... a very simple data model.

- how to query it?
- generally, the network model language could be used for navigating ...
- ... but in the meantime, *declarative* languages had been invented:
  – clause-based: SQL-style
  – logic-based: Datalog-style

3.1.2 TSIMMIS: Languages

Mediators are programmed in MSL (Mediator Specification Language; a rule-based query language for OEM):

MSL rules (cf. Prolog, Datalog):

head(Vars) :- body(Vars,databases)

- head and body consist of expressions over *patterns* of the form

  <*oid label type value*>
  or
  <*oid label value*>
  or
  <*label value*>

- *value* can be set-valued; in this case, the set consists itself of expressions of the form <. . . > .
- objects of different sources are identified by <object>@source.
- *body*: pattern that must be satisfied by suitable variable bindings,
- *head* describes the structure of the OEM object that is generated.

MSL

The country whose code is “D”:

```
?- <C country {<code "D">}>@cia
```

- the query generates a set-valued result object, whose sub-objects are the individual answers:

result: <answer {&c1}>

The names of all south-american countries in which there is a city with name “Santiago”:

```
<countryname N>:-
      <continent {<name "South America"> <country
           {<name N> <city {<name "Santiago">}>}>}>@gs
```

<&obj42, answer, set, {<countryname “Chile”>,
<countryname “Paraguay”>,
<countryname “Argentina”>}>

Example

Mediator med accesses wrappers cia and gs.

Query: all cities that are stored in gs whose names are mentioned in cia as names of capitals.

Mediator rule:

<capital {<name Cap> <country CN> R }>@med :-
<country {<name CN> <capital Cap>}>@cia
AND <country {<name CN> <city {<name Cap> | R}>}>@gs

- R is bound to the remaining sub-objects of the resulting city-objects.

⇒ object creation in the rule head $obj@med$ (cf. Views)

exported object e.g. <&cap, capital, set, {&ctn12, &c1, &ctp12, &ctprov12}>@med

- additionally: external predicates (string concatenation, substring, comparisons etc).
- variables can be bound to values, objects, sets and labels
- *syntactically* 2nd order

⇒ queries against the logical schema are possible.

3.1.3 TSIMMIS: User Queries Against OEM

Users can state queries in MSL or LOREL:

- MSL: see above – rule- and pattern-based language
- LOREL (Lightweight Object Repository Language):
  (LORE: DBMS for OEM data model, Stanford)
  clause- and path-based language (based on OQL)

Lorel

- Entry points are named constants (e.g. europe) or extents (e.g. countries)
- result of each query is a collection of OEM objects.

Semantics of 1:N-relationships

multi-valued instead of *set-valued* semantics:

- germany.city yields *multiple unary* answers instead of (as in ODMG) *one set-valued* answer.
- germany.city.name yields the names of all these cities.

Lorel

clause-based SQL/OQL-style language: SELECT - FROM - WHERE

```
% all europ.  capitals:
select europe.country.capital % note:  multivalued semantics

% the country with the code "S":
select c
from europe.country c
where c.code = "S"

% South-american countries such that ...
select c
from southamerica.country c
where c.city.name = "Santiago"
```

implicit existential semantics: ... if there is any city whose name is Santiago.

3.2 Frame-Based Models

- objects are represented by *object frames*,
- Frame contains *slots* for storing properties
  (“signature” of the slots *can* be given by a schema, but not necessarily (⇒ semistructured data))
- Slots can be literal-valued or object-valued (*internal* storage by references) for describing attributes and relationships.

A Self-Describing OO-Data Model: F-Logic

(M. Kifer, G. Lausen, 1989/1995; SIGMOD Test of Time Award 1999)

- full object-orientation (class hierarchy, inheritance)
- objects have properties
- metadata (class names, method names) as first-class-members of the data model
- “frame-based” model
- deductive language (Prolog-style)

F-Logic: Data Model and Syntax

- is-a relationship:
  <object> : <class>
- subclass-relationship:
  <class> :: <class>
- properties:
  <object>[<property>→<object/value>] (scalar)
  <object>[<property>↠{<set-of-objects/values>}] (set-valued/multi-valued)
  Analogously with parameters:
  <object>[<property>@(<list-of-objects/values>)→<object/value>]
  <object>[<property>@(<list-of-objects/values>)↠<{set-of-objects/values}>]
- inheritable properties:
  <class>[<property>•→<object/value>] (scalar)
  <class>[<property>•↠{<set-of-objects/values>}] (set-valued)
  nonmonotonic inheritance semantics with overriding.

F-Logic: Example

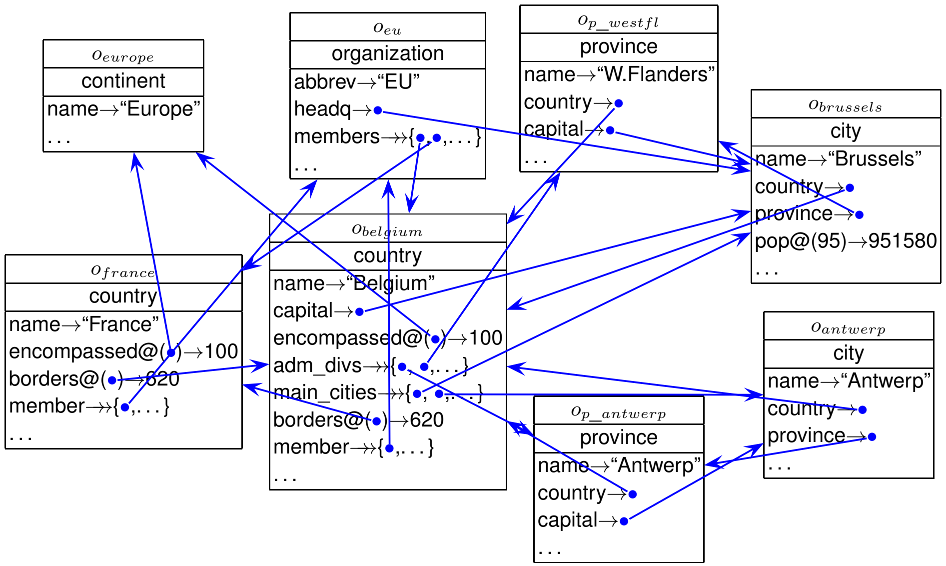
$o_{belgium}$: *country*[name→"Belgium"; car_code→"B";
capital→$o_{brussels}$; independence→"04 10 1830";
total_area→30510; population→10170241;
encompassed@(o_{europe})→100; pop_growth→0.33;
adm_divs→→{$o_{p\_antwerp}$, o_{p_westfl}, ...};
main_cities→→{$o_{brussels}$, $o_{antwerp}$, ...};
borders@(o_{france})→620; borders@($o_{germany}$)→167;
borders@($o_{luxembourg}$)→148; borders@($o_{netherlands}$)→450].

$o_{brussels}$: city[name→"Brussels"; country→$o_{belgium}$;
province→o_{p_westfl}; population@(95)→951580].

o_{p_westfl} : prov[name→"West Flanders"; country→$o_{belgium}$;
capital→$o_{brussels}$; area→3358; population→2253794].

o_{eu} : org[abbrev→"EU"; name→"European Union";
established→"07 02 1992"; headq→$o_{brussels}$;
members@("member")→→{$o_{belgium}$, o_{france}, ...};
members@("applicant")→→{$o_{hungary}$, $o_{slovakia}$, ...}].

Representation as a Graph



Properties

- arbitrary properties of an objects can be stored – simply generate and fill a slot. *self-describing* data model
- null values need not to be stored explicitly, slots are simply not filled.
- *navigation* with *path expressions* in combination with *patterns*:
  - population of the province where the capital of Belgium is located:
    ?- C:country[name→"Belgium"].capital.province[population →P].
  - all names of cities in Belgium:
    ?- C:country[name→"Belgium"]..main_cities[name→N].
  - sum of population of all provinces of Belgium:
    ?- Z = sum{X | C:country[name→"Belgium"]..province[population→X]}.
  - can be nested in complex conditions:

?- C:country[encompassed@(_Cont[name→"Europe"])→_X; member→→_O[name→"EU"];
population→CP; area→CA]..city[population→CitP; name→N],
CP > 1000000, CA > 100000, CitP > 0.25 * CP.

F-Logic: Signature

The signature can also be formalized in F-Logic:

- properties:
  <type>[<property>⇒<type>] (scalar)
  <type>[<property>⇒⇒<type>] (set-valued)
  analogously with parameters:
  <type>[<property>@(<list-of-types>)⇒<type>]
- country[name⇒string; capital⇒city;
  total_area⇒number; population⇒number;
  encompassed@(continent)⇒number;
  adm_divs⇒⇒province; main_cities⇒⇒city;
  borders@(country)⇒number].

  city[name⇒string; country⇒country;
  province⇒province; population@(number)→number].
- queries against metadata: ?- X:country[M→(_V:C)] or ?- country[M⇒C].

F-Logic as a Programming Language

- object-oriented logic
- *deductive* database language (i.e. Prolog-style) with fixpoint semantics

```
<head>:- <body>.
?- <query>.
```

- e.g., transitive closure can be expressed:

  R[transitive_flows$\twoheadrightarrow$S] :- R:river[to@(sea)$\rightarrow$S].

  R1[transitive_flows$\twoheadrightarrow$S] :- R1:river[to@(river)$\rightarrow$R2], R2:river[transitive_flows$\twoheadrightarrow$S].

Implementations

- The FLORID system (F-Logic Reasoning in Databases; C++):
  `http://www.informatik.uni-freiburg.de/~dbis/florid`
- FLORA/FLORA II; XSB-Prolog: `http://flora.sourceforge.net/`

... current use: reasoning in the Semantic Web.

Data Integration from the Web with Florid

- warehouse approach
- direct mapping from HTML trees to F-Logic
- queries against Web pages possible
- wrapper + mediator functionality programmed by F-Logic rules

1998: Generation of the MONDIAL database from the Web

- F-Logic wrappers for several Web pages
  - CIA World Factbook Country Pages
  - CIA World Factbook Organizations Pages
  - “Global Statistics”
  - some smaller sources + the original Karlsruhe TERRA database

  $\Rightarrow$ materialized F-Logic representation of each source
- F-Logic integration program that *stepwise* materializes an integrated database
- advantages of the warehouse approach for complex integration tasks (basic rules + exceptional and refining rules)

3.3 Summary: Database Aspects (1995)

- Integration of data from different, heterogeneous sources:
  - relational distributed/federated databases: metadata queries and -integration (SchemaSQL)
  - arbitrary sources (Tsimmis): metadata queries, general, flexible data model
  - sources can also be Web pages (HTML) (Tsimmis, Florid)
- semistructured nature of data
  - no fixed schema
  - implicit null values
  - easily extensible (adding new properties)
  - object as a collection of properties
  - self-describing data + metadata
- languages for semistructured data
  - metadata queries
  - generation of structure

$\Rightarrow$ flexible

Query Languages

- declarative
- functional, closed languages, usually clause-based: iterator variables and SELECT-FROM-WHERE-clause
  (SQL, OQL, Lorel)
- logical/deductive; binding of variables by patterns: constructive semantics of rule heads
  - patterns as terms
    (WSL/MSL)
  - patterns as extended path expressions
    (F-Logic)
  - as programming languages: fixpoint semantics
- multivalued vs. set-valued semantics
- rule-based (more or less explicit):
  - binding of variables in the “body” (SQL/OQL: FROM-WHERE clause)
  - result generation in the head

F-Logic [1989] as a predecessor of XML, RDF, OWL

- semistructured (⇒ XML, RDF)
- self-describing (⇒ XML, RDF)
- data model
  - database model: complex objects, properties, relationships (⇒ XML, RDF)
  - but also knowledge representation model with *built-in reasoning* (⇒ OWL)
  - optional schema information (⇒ XSD, RDFS, OWL)
- query language
  - navigation, path expressions with predicates and multivalued semantics (⇒ XPath)
- derivation rules (⇒ OWL + Rules (SWRL))

RDF: Resource Description Format, 1997, see Lecture "Semantic Web"
OWL: Web Ontology Language, 2002 [OIL: 2000], see Lecture "Semantic Web"

3.4 Situation 1996

- Experiences with SQL (and ODMG/OQL) as database languages
  - standardization vs. products
- document management with SGML (Structured Generic Markup Language), CSS (Cascading Stylesheets) and DSSSL
- data exchange/access via internet/Web:
  - homogeneous solution necessary
  - availability of documents and data in HTML:
    - very simple variant of SGML
    - "native" HTML data (handwritten)
    - mapping of SGML (document management) to HTML (publication) by CSS
    - HTML-Web-Servers over relational databases

⇒ "Global" approach coordinated by the W3C (World Wide Web Consortium): development of a data model (+ language), that can handle (legacy-)databases, documents and Web (=HTML)

The W3C (World Wide Web Consortium)

- `http://www.w3.org`.
- founded in 1994 for developing common protocols and languages for the World Wide Web and to ensure interoperability of applications in the Web.
  (Tim Berners-Lee, MIT, CERN)
- following the principles of OMG/ODMG who developed the CORBA and ODL/OIF/OQL standards
- members: companies and research institutes
- definition of working groups
- notes → working drafts → recommendations
- not only XML, but also many other Web-related issues

3.5 Documents: SGML and HTML

- Structuring (und presentation) are called *(logical and optical) "markup"*.
  (document = content + markup)
- SGML (Standard Generalized Markup Language),
  development (IBM) since 1979, standard 1986.
  structuring and markup of documents, widely used in publishing.
- for publishing in the Web:
  HTML (Hypertext Markup Language), development since 1989 (CERN), standard 1991.

⇒ HTML is *an SGML application* with a fixed syntax
(tags, attributes, later: DTD).
goal: optical markup, as a side effect also some structuring of the documents (cf. <P>-Tag).

- SGML much more flexible than HTML → more complex → not suitable for browsers
  (HTML allows for efficient and fault-tolerant parsing)
- SGML sources can be transformed to HTML by stylesheets (CSS: Cascading Style Sheets).

Chapter 4
XML (Extensible Markup Language)

Introduction

- SGML *very* expressive and flexible
  HTML very specialized.
- Summer 1996: John Bosak (Sun Microsystems) initiates the XML Working Group (SGML experts), cooperation with the W3C.
  Development of a subset of SGML that is simpler to implement and to understand
  `http://www.w3.org/XML/`: the homepage for XML at the W3C

⇒ XML is a “stripped-down version of SGML”.

- for understanding XML, it is not necessary to understand everything about SGML ...

HTML

let’s start the other way round: HTML ... well known, isn’t it?

- tags: pairwise opening and closing: <TABLE> ... </TABLE>
- “empty” tags: without closing tag <BR>, <HR>
- <P> is *in fact* not an empty tag (it should be closed at the end of the paragraph)!
- attributes: <TD colspan = “2”> ... </TD>
- empty tags with attributes:
  <IMG SRC=“http://www.informatik.uni-goettingen.de/photo.jpg” ALIGN=“LEFT”>
- content of tag structures: <TD>123456</TD>
- nested tag structures: <TH><B>Name</B></TH>
  <A href=“http:www.ifi.informatik.uni-goettingen.de”>
  <B>Homepage of the IFI</B></A>

⇒ hierarchical structure

- Entities: ä = ä ß= ß

HTML

- browser must be able to interpret tags
  → semantics of each tag is fixed for all (?) browsers.
- fixed specifications how tags can be nested
  (described by a DTD (Document Type Definition))

```
<body><H1>...</H1><H2>...</H2>
                  <P> ... </P>
                  <H2>...</H2>
                  <P> ... </P>
      <H1>...</H1><H2>...</H2>
                  <P> ... </P>
</body>
```

- analogously for tables and lists ...
- reality: people do in general not adhere to this structure
  - closing tags are omitted
  - structuring levels are omitted

  → parser has to be fault-tolerant and auto-completing

Knowledge of HTML for XML?

- intuitive idea – but only of the *textual/unicode representation*
- this is *not a data model*
- no query language
- only a very restricted viewpoint:
  HTML is a markup language for browsers
  (note: we don’t “see” HTML in the browser, but only what the browser makes out of the HTML).

  Not any more.

Goals of the Development of XML

- XML must be directly usable and transmitted in the internet (unicode-files/streams),
- XML must support a wide range of applications,
- XML must be compatible with SGML,
- XML documents must be human-readable and understandable,
- XML documents must be easy to create,
- it must be easy to write programs that evaluate/process/parse XML documents.

Differences between XML and HTML?

- Goal: *not browsing*, but representation/storage of (semistructured) data (cf. SGML)
- SGML allows the definition of new tags according to the application semantics; each SGML application uses its own *semantic tags*.
  These are defined in a DTD (Document Type Definition).
- HTML is *an SGML application* (cf. `<HTML> at the beginning of each document </HTML>`), that uses the DTD "HTML.dtd".
- In XML, (nearly) arbitrary tags can be defined and used:

```
<country> ... </country>
<city> ... </city>
<province> ... </province>
<name> ... </name>
```

- These *elements* represent objects of the application.

XML as a Meta-Language for Specialized Languages

- For each application, it can be chosen which "notions" are used as element names etc.:
  ⇒ document type definition (DTD)
- the set of allowed element names and their allowed nesting and attributes are defined in the DTD of the document (type).
- the DTD describes the *schema*
- XML is a *meta-language*, each DTD defines an own language
- for an application, either a new DTD can be defined, or an existing DTD can be used
  → standard-DTDs
- HTML has (as an SGML application) a DTD
- Remark: XML is case-sensitive.

Example: Mondial

```
<mondial>
  :
 <country car_code="D" capital="city-D-Berlin" memberships="org-EU org-NATO org-UN ...">
   <name>Germany</name>
   <encompassed continent="europe">100</encompassed>
   <population year="1997">82501000</population>
   <population year="2011">80219695</population>
   <ethnicgroup name="German">95.1</ethnicgroup>
   <ethnicgroup name="Italian">0.7</ethnicgroup>
   <religion name="Roman Catholic">37</religion>
   <religion name="Protestant">45</religion>
   <language name="German">100</language>
   <border country="F" length="451"/>
   <border country="A" length="784"/>
   <border country="CZ" length="646"/>
     :
```

Example: Mondial (Cont'd)

```
      :
    <province id="prov-D-berlin" capital="city-D-berlin">
      <name>Berlin</name>
      <population year="1995">3472009</population>
      <city id="city-D-berlin">
        <name>Berlin</name> <population year="1995">3472009</population>
      </city>
    </province>
    <province id="prov-D-baden-wuerttemberg" capital="city-D-stuttgart">
      <population year="1995">10272069</population>
      <name>Baden Wuerttemberg</name>
      <city id="city-D-stuttgart">
        <name>Stuttgart</name> <population year="95">588482</population>
      </city>
      <city id="cty-D-mannheim"> ... </city>
      :
    </province>
    :
  </country>
  :
</mondial>
```


CHARACTERISTICS:

- hierarchical "data model"
- subelements, attributes
- references
- ordering? documents – yes, databases – no

Examples can be found at

`http://dbis.informatik.uni-goettingen.de/Mondial/#XML`

XML AS A DATA MODEL

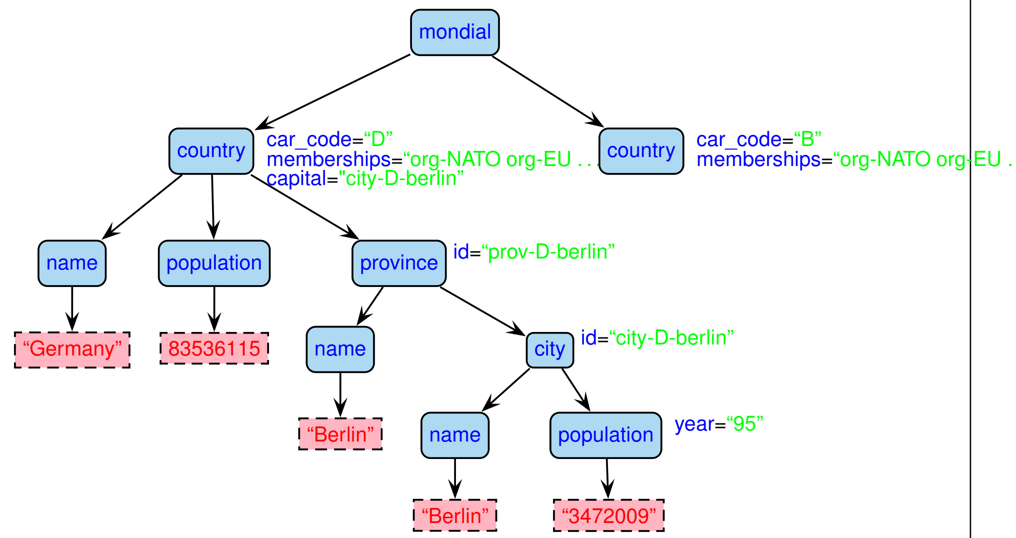
XML is much more than only the character/unicode representation shown above as known from HTML
(see also introductory talk)

- abstract data model (comparable to the relational DM)
- abstract datatype: DOM (Document Object Model) – see later
- many concepts around XML
  (XML is *not* a programming language!)
  - higher-level declarative query/manipulation language(s) XPath and XQuery
  - transformation language: XSLT
  - notions of "schema": DTD and XML Schema

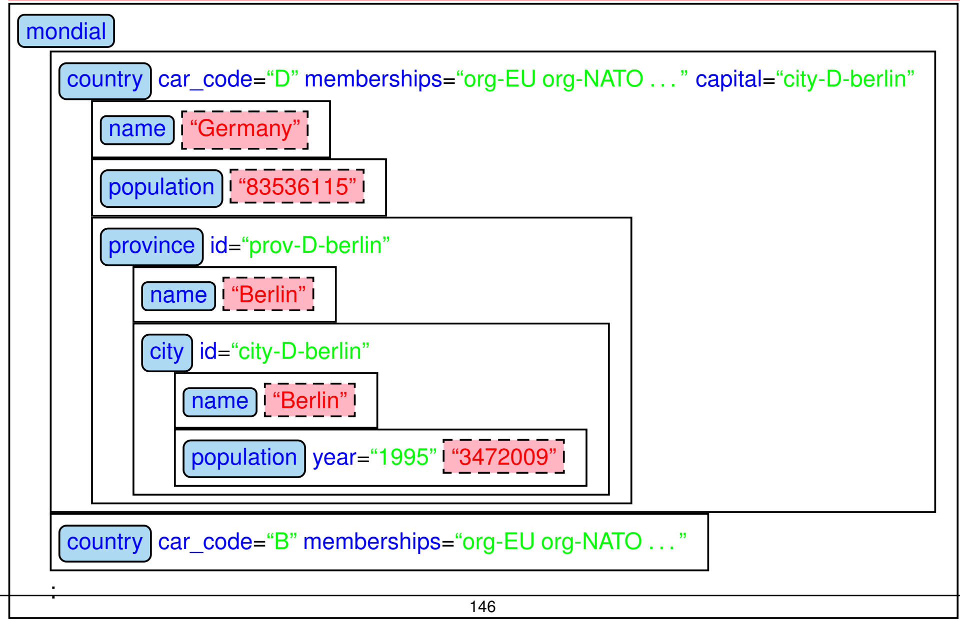
4.1 Structure of the Abstract XML Data Model (Overview)

- for each document there is a document node which "is" the document, and which contains information about the document (reference to DTD, doctype, encoding etc).
- the document itself consists of nested *elements* (tree structure),
- among these, exactly one *root element* that contains all other elements and which is the only child of the document node.
- elements have an element type (e.g. Mondial, Country, City)
- element content (if not empty) consists of text and/or *subelements*.
  These *child nodes* are ordered.
- elements may have *attributes*.
  Each attribute node has a name and a value (e.g. (car_code, "D")).
  The attribute nodes are unordered.
- *empty elements* have no content, but can have attributes.
- a *node* in an XML document is a logical unit, i.e., an element, an attribute, or a text node.
- the allowed structure can be restricted by a schema definition.

Example: Mondial as a Tree



Example: Mondial as a Nested Structure



Observations

- there is a global order (preorder-depth-first-traversing) of all element- and text nodes, called *document order*.
- actual text is only present in the text-nodes
  Documents: if all text is concatenated in document order, a pure text version is obtained.
  Exercise: consider an HTML document.
- element nodes serve for structuring (but do not have a "value" for themselves)
- attribute nodes contain values whose semantics will be described in more detail later
  - attributes that describe the elements in more detail
    (e.g. td/@colspan or population/@year)
  - IDs and references to IDs
  - can be used for application-specific needs

4.2 XML Character Representation

- Tree model and nested model serve as abstract datatypes (see later: DOM)

data exchange? how can an XML document be represented?

- a relational DB can be output as a finite set of tuples (cf. relational calculus)
  positional (cf. Datalog): country("Germany", "D", 83536115, 356910, "Berlin", "Berlin")
  or slotted (cf. formal definition of tuples in the relational algebra):
  country(Name: "Germany", Code: "D", Population: 83536115, Area: 356910,
  Capital: "Berlin", CapitalProvince: "Berlin")
- object-oriented databases: OIF (Object Interchange Format)
- OEM-tripels, F-Logic-frames
- XML:
  Exporting the tree in a *preorder-depth-first-traversing*.
  The node types are represented in a specified syntax:
  ⇒ XML as a representation language
- JSON (spec as RFC in 2006, standardized in 2013)

XML as a Representation Language

- elements are limited by
  - opening <country> and
  - closing *tags* </country>,
  - in-between, the *element content* is output recursively.
- Element content consists of text
  <name>United Nations</name>
- and *subelements*: <country> <city> ... </city>
  <city> ... </city>
  </country>
- *attributes* are given in the opening tag:
  <country car_code="D"> . . . </country>
  where attribute values are always given as strings, they do not have further structure. The difference between value- and reference attributes is not visible, but is only given by the DTD.
- *empty elements* have only attributes: <border country="F" length="451"/>

XML as a Representation Language: Grammar

The language "XML" defined as above can be given as an BNF grammar:

Document	::=	Element
Element	::=	"<" ElementName Attributes ">" Content "</" ElementName ">"
		| "<" ElementName Attributes "/>"
Content	::=	ε | Element Content | Chars Content
Chars	::=	*characters including whitespace*
Attributes	::=	ε | AttributeName "='"Chars"'" Attributes
ElementName, AttributeName	::=	*character string with some restrictions*

- note that this grammar does not guarantee that the opening and closing tags match!
- instead of ′, also the usual " are allowed
- strict adherence to these rules (closing and empty elements) is required.
- an XML instance (as a sequence of Unicode/UTF-8/UTF-16 characters) is *well-formed*, if it satisfies these rules.
- "XML parsers" process this input.

XML Parser

- an *XML parser* is a program that processes an XML document given in Unicode representation according to the XML grammar, and generates a result:
  - correctness: check for well-formedness (and adherence to a given DTD)
  - DOM-parser: transformation of the XML instance into a DOM model (implementation of the abstract datatype; see later).
  - SAX-parser: traversing the XML tree and generation of a sequence of "events" that *serialize* the document (see later).
- XML parsers are required to accept only well-formed instances.
  - simple grammar, simple (non-fault-tolerant) parser
  - HTML: fault-tolerant parsers are much more complex
    (fault tolerance wrt. omitted tags is only possible when the DTD is known)
- each XML application must contain a parser for processing XML instances in Unicode representation as input.

XML Parsing in the General Case

- ElementName is a separate production and
  Element ::= "<" ElementName Attributes ">" Content "</" ElementName ">"
  | "<" ElementName Attributes "/>"
  does not guarantee matching tags

$\Rightarrow$ not context-free!

- Nevertheless, context-free-style parsing with push-down-automaton *without fixed stack alphabet* possible:
  - for every opening tag, put ElementName on the stack
  - for every closing tag, compare with top of stack, pop stack.

$\Rightarrow$ linear-time parsing

- Exercise: give an automaton for parsing XML and describe the handling of the stack (solution see Slide 179).

Viewing XML Documents?

- as a file in the editor
  - emacs with xml-mode
  - Linux/KDE: kxmleditor
- browser cannot "interpret" XML
  (in contrast to HTML)
- with "show source" in a browser:
  current versions of most browsers show XML in its Unicode representation with indentation and allow to open/close elements/subtrees.
- but, in general, XML is not intended for viewing:
  → transformation to HTML by XSLT stylesheets
  (see later)

4.3 Datatypes and Description of Structure for XML

- relational model: atomic data types and tuple types
- object-oriented model: literal types and object types, reference types

Data Types in XML

- data types for text content
- data types for attribute values
- element types (as "complex objects")
- somewhat different approaches in DTD (document-oriented, coarse) and XML Schema (database-oriented, fine)

Document Type Definition – DTD

- the set of allowed tags and their nestings and attributes are specified in the DTD of the document (type).
- the idea of the DTD comes from the SGML area
  - meets the requirements for describing document structure
  - does not completely meet the requirements of the database area
    → XML Schema (later)
  - simple, and easy to understand.
- the DTD for a document type $doctype$ is given by a grammar (context-free; regular expression style) that characterizes a class of documents:
  - what types of elements are allowed in a document of the type $doctype$,
  - what subelements they have (element types, order, cardinality)
  - what attributes they have (attribute name, type and cardinality)
  - additionally, "entities" can be defined (they serve as constants or macros)
- Remark: the DTD language is case-sensitive.
  ALL DTD KEYWORDS AND DATATYPES ETC MUST BE WRITTEN WITH CAPITAL LETTERS!

Data types of XML and DTDs

- text content of elements: PCDATA – "parsed character data"; (nearly) arbitrary strings;
  it is up to the application to distinguish between string data and numerical data;
  for having "<" in element contents, see Slide 181
- data types for attribute values:
  - CDATA: (Character data) arbitrary strings
  - NMTOKEN: string without blanks; some special chars not allowed
  - NMTOKENS: a list of NMTOKENs, separated by blanks
  - ID: restriction of NMTOKEN, start with [a-zA-Z:_],
    each value must be unique in the document,
  - IDREF: like ID, each value must occur in the same document as an ID value
  - IDREFS: the same, multivalued
  - for the ugly details which characters are (dis)allowed, see
    `https://www.w3.org/TR/2008/REC-xml-20081126/#sec-attribute-types`
- element types: definition of structure in the style of regular expressions.

DTD: Element Type Definition – Structure of the Element Contents

<!ELEMENT *elem_name struct_spec*>

- EMPTY: empty element type,
- (#PCDATA): text-only content
- (*expression*): expression over element names and combinators (same as for regular expressions). Note that the expression must be deterministic.
  - ",": sequence,
  - "|": (exclusive-)or (choice),
  - "*": arbitrarily often,
  - "+": at least once,
  - "?": optional
- (#PCDATA|*elem_name*$_1$|...|*elem_name*$_n$)*
  mixed content, here, only the types of the subelements that are allowed to occur together with #PCDATA can be specified; no statement about order or cardinality.
- ANY: arbitrary content

Element Type Definition: Examples

- from HTML: images have only attributes and no content

```
<!ELEMENT img EMPTY >
```

- from Mondial:

```
<!ELEMENT country (name, encompassed+, population*,
                   ethnicgroup*, religion*, border*,
                   (province+ | city+))>
<!ELEMENT name (#PCDATA)>
```

- for text documents:

```
<!ELEMENT Section (Header,
                   (Paragraph|Image|Figure|Subsection)+,
                   Bibliography?)>
```

- Element type definitions by regular expressions
  ⇒ can be checked by finite state automata

DTD: Attribute Definitions

- General: an element contains at most one attribute of every attribute name.
- details about allowed attribute names and their types are specified in the DTD.

<!ATTLIST *elem_name*
attr_name$_1$ *attr_type*$_1$ *attr_constr*$_1$
: : :
attr_name$_n$ *attr_type*$_n$ *attr_constr*$_n$>

- *attr_type*$_i$: value/reference attribute and scalar/multi-valued
  - CDATA, NMTOKEN, NMTOKENS, ID, IDREF, IDREFS: see Slide 156.
  - (*const*$_1$|…|*const*$_k$): scalar, from a given domain.
- *attr_constr*$_i$: minimal cardinality
  - #REQUIRED: attribute must be present for each element of this type.
  - #IMPLIED: attribute is optional.
  - *default*: Default-value (non-monotonic value inheritance).
  - #FIXED *value*: attribute has the same (given) value for each element of this type (monotonic value inheritance).

DTD: Attribute-Definitions (Examples)

```
<!ATTLIST country
    car_code      ID          #REQUIRED
    capital       IDREF       #IMPLIED
    memberships   IDREFS      #IMPLIED
    products      NMTOKENS    #IMPLIED >
<!ATTLIST desert
    id            ID          #REQUIRED
    type          (sand|rocks|ice) 'sand'
    climate       NMTOKEN     #FIXED 'dry' >
```

- when an XML parser reads an XML instance and its DTD, it fills in default and fixed values.

DTD and XML Instances

- Each DTD defines an own markup language (i.e., an XML application – HTML is one, Mondial is another).
- an XML instance has a *document node* (which is not the root node, but even "superior") that contains among other things information about the DTD.
  (see next slides ...)
- the root element of the document must be of an element type that is defined in the DTD.
- an XML instance is *valid* wrt. a DTD if it satisfies the structural constraints specified in the DTD.
  Validity can be checked by an extended finite state automaton in linear time.
- XML-instances can exist without a DTD (but then, it is not explicitly specified what their tags "mean").

XML Document Structure: The Prolog

The *prolog* of an XML document in Unicode representation contains additional information about the document (associated with the document node):

- XML declaration (with optional attributes):
  <?xml version="1.0" encoding="utf-8"?> allows all characters (chinese, cyrillic, arabian, . . .)
  e.g. encoding="ISO-8859-1" allows German "Umlauts".
- document type *declaration*: indication of the document type, and where the document type *definition (DTD)* can be found.
  - <!DOCTYPE *name* {SYSTEM *own-url* | PUBLIC *public-id public-url*}>
    name: one of the element names given in the DTD
    SYSTEM *own-url*: own document type,
    <!DOCTYPE mondial SYSTEM "mondial.dtd">
    PUBLIC *public-id public-url*: standard document type (e.g. XHTML), or
  - <!DOCTYPE *name* [*dtd*]>
    with DTD directly included *in* the document.
- then follows the document content (i.e., the root node with the document body as its content).

Note: Document Type Declaration with PUBLIC id, PUBLIC url

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

- id is a globally agreed string
- url looks like a URL for being accessed through the Web
  ... maybe this was intended at the beginning.
  – any software that processes a document accesses the DTD at the URL.
  ⇒ turned out to be a bad idea: billions of accesses to this URL (`http://www.w3.org/blog/systeam/2008/02/08/w3c_s_excessive_dtd_traffic`)
  ⇒ W3C blocked access to this URL!
  ⇒ problem for the users who now get unintelligible error messages when using any tools (e.g., creating the DBIS Web pages with XSLT).
- W3C: this URL is to be understood as a URI (Uniform Resource Identifier; in a sense that rather belongs to the Semantic Web area) that only tells the tool that the document "is" XHTML 1.0; *not* that the XHTML DTD should/can be accessed there.
- technically to be solved by using "XML Catalogs", cf. Slide 235

Example: Mondial

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial>
 <country car_code="AL" area="28750" capital="cty-cid-cia-Albania-Tirane"
         memberships="org-BSEC org-CE org-CCC ...">
  <name>Albania</name> <population>3249136</population>
  <encompassed continent="europe" percentage="100"/>
  <ethnicgroup percentage="3">Greeks</ethnicgroup>
  <ethnicgroup percentage="95">Albanian</ethnicgroup>
  <border country="GR" length="282"/> <border country="MK" length="151"/>
  <border country="YU" length="287"/>
  <city id="cty-cid-cia-Albania-Tirane" country="AL">
   <name>Tirane</name>
   <latitude>46.2</latitude> <longitude>10.7</longitude>
   <population year="87">192000</population>
  </city>
  :
 </country>
 :
</mondial>
```

Tool: XMLLINT

xmllint is a simple tool that allows (among other things – see later) to validate a document (belongs to libxml2):

- `man xmllint`: lists all available commands
- currently, we are mainly interested in the following:
  `xmllint -loaddtd --noblanks -valid -noout mondial-europe.xml`
  validates an XML document wrt. the DTD given in the prolog
  (`--noblanks` ignores (indentation) whitespaces that would otherwise be seen as mixed content)

XMLLINT: Further Functionality (see later)

XMLLINT can be used to “visit” the document, and to walk through it:

- call `xmllint -loaddtd -shell mondial-europe.xml`.

Then, one gets a “navigating shell” “inside” the XML document tree
(very similar to navigating in a UNIX directory tree):

- `validate`: validates the document
- `cd xpath-expression`: navigates into a node
  (the XPath expression must uniquely select a single node)
  relativ: `cd country[1]`
  absolut: `cd //country[@car_code="D"]`
- `pwd`: gives the path from the root to the current position
- `cat`: prints the current node
- `cat xpath-expression`
  `cat .//city/name`
- `du xpath-expression` lists the content of the node that is selected by `xpath-expression` (starting from the current node)
- `dir xpath-expression` prints the node type and attributes of the selected node

Example: “Books” from W3C XML Use Cases

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE bib SYSTEM "books.dtd">
<!-- from W3C XML Query Use Cases -->
<bib>
  <book year="1994">
    <title>TCP/IP Illustrated</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="1992">
    <title>Advanced Programming in the Unix environment</title>
    <author><last>Stevens</last><first>W.</first></author>
    <publisher>Addison-Wesley</publisher>
    <price>65.95</price>
  </book>
  <book year="2000">
    <title>Data on the Web</title>
    <author><last>Abiteboul</last><first>Serge</first></author>
    <author><last>Buneman</last><first>Peter</first></author>
    <author><last>Suciu</last><first>Dan</first></author>
    <publisher>Morgan Kaufmann Publishers</publisher>
    <price>39.95</price>
  </book>
  <book year="1999">
    <title>Economics of ... for Digital TV</title>
    <editor>
      <last>Gerbarg</last><first>Darcy</first>
      <affiliation>CITI</affiliation>
    </editor>
    <publisher>Kluwer Academic Publishers</publisher>
    <price>129.95</price>
  </book>
</bib>
```

[see XML-DTD/books.xml]

Exercise: Generate a DTD for the above XML

... do it step-by-step, using a validator:

- for all element types:
  <!ELEMENT *name* ANY>
- declare <!ATTLIST *name* ...> where needed
- validate
- stepwise refinement of content models ...
- ... blackboard demonstration ...
- solution see Slide 175

Data-Centered vs. Document-Centered XML Documents

Data-Centered XML Documents

- very regular structure with "data fields"
- only some text
- no naturally induced tree structure

Document-Centered XML Documents

- tree structure with much text (text content is the text of the document)
- non-regular structure of elements
- logical markup of the documents
- annotations of the text by additional elements/attributes

Semistructured XML Documents

- combine both (e.g. medical information systems)

Subelements vs. attributes

When designing an XML structure, often the choice of representing something as subelement or as attribute is up to the designer.

Document-Centered XML

- the concatenation of the whole text content should be the "text" of the document
- element structures for logical markup and annotations
- attributes contain additional information *about* the structuring elements.

Data-Centered XML

- more freedom
- attributes are unstructured and cannot have further attributes
- elements allow for structure and refinement with subelements and attributes
- using DTDs as schema language allows the following functionality only for attributes:
  – usage as identifiers (ID) and references (IDREF/IDREFS)
  – restrictions of the domain
  – default values
  (XML Schema allows many more things)

Examples and Exercises

- The MONDIAL database is used as an example for practical experiments. See `http://dbis.informatik.uni-goettingen.de/Mondial#XML`.
- many W3C documents base on examples about a literature database (book, title, authors, etc.).
- each participant (possibly in groups) should choose an *own* application area to set up an own example and to experiment with it.
  - from the chosen branch of study?
  - database of music CDs
  - lectures and persons at the university
  - exams (better than FlexNever?)
  - calendar and diary
  - other ideas ...

Exercise: Define a DTD and generate a small XML document for your chosen application.

Exercises

- Validate your example document with a suitable prolog and internal DTD.
- put your DTD publicly in your public-directory and validate a document that references this DTD as an external DTD.
- take a DTD+url from a colleague and write a small instance for the DTD and validate it.
- note: if you do this with an XHTML document and W3Cs XHTML DTD, care for the XML Catalog issue, cf. Slides 163 and 235.

Data Exchange with XML

For *Electronic Data Interchange (EDI)*, a commonly known+used DTD is required

- producers and suppliers in the automobile industry
- health system, medical area
- finance/banking

Proceeding

Usually, XML data is exchanged in its Unicode representation.

- XML-Server make documents in the Unicode representation accessible (i.e., as a stream or as a textfile)
- applications *parse* this input (linear) and store it internally (DOM or anything else).

4.3.1 Aside: XML Parsing

... side objective of this lecture: show applications and connections of basic concepts of CS:

- XML/DTD: content models are regular expressions
  ⇒ can be checked by finite state automata
  - design one automaton for each <!ELEMENT ...> declaration
  - design a combined automaton for validating documents against a given DTD (recursion requires usage of a return-stack, still linear time)
  - extension to attributes: straightforward (when processing opening tags, dictionary-based)
  - checking for well-formedness and validity in linear time
    - with a DOM parser: during generation of the DOM
    - with a SAX parser: streaming, on the fly
    - using a DOM instance: depth-first traversal
- without a DTD: requires a push-down automaton (remembering opening tags); still linear time
  - checking well-formedness
  - generating a DOM instance, or on-the-fly (SAX)

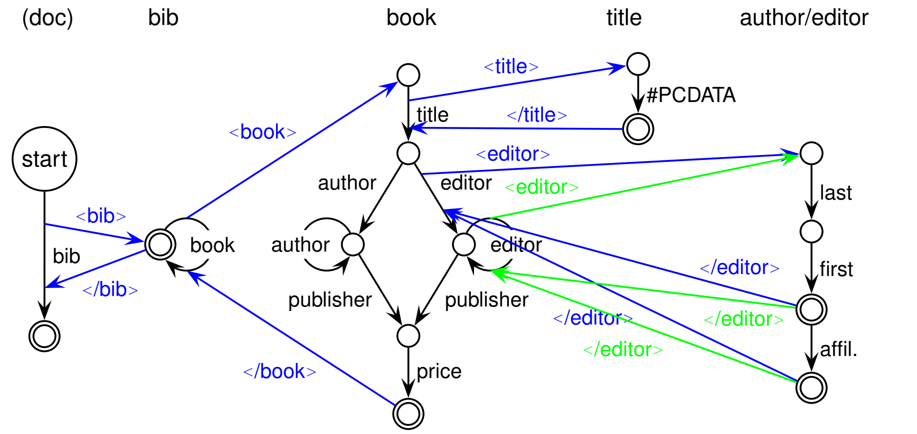
Finite State Automata for Validation
Example: books.dtd

Consider the "books" example:

```
<!ELEMENT bib (book*)>
<!ELEMENT book (title, (author+ | editor+), publisher, price)>
<!ATTLIST book year CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (last, first, affiliation?)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT editor (last, first, affiliation?)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT affiliation (#PCDATA)>
```


Finite State Automata

- individual automata for element content models
  (recall that the content model must be deterministic)
- combined by nesting (jumping and returning on opening/closing tags)



- author edges use the same author/editor subautomaton → use return-stack

XML Grammar in presence of a DTD

Consider the grammar from Slide 150:

- Element names known from a DTD: context-free grammar (nonterminals in BLUE)
  (translate regexps in BNF as in the CS I course)

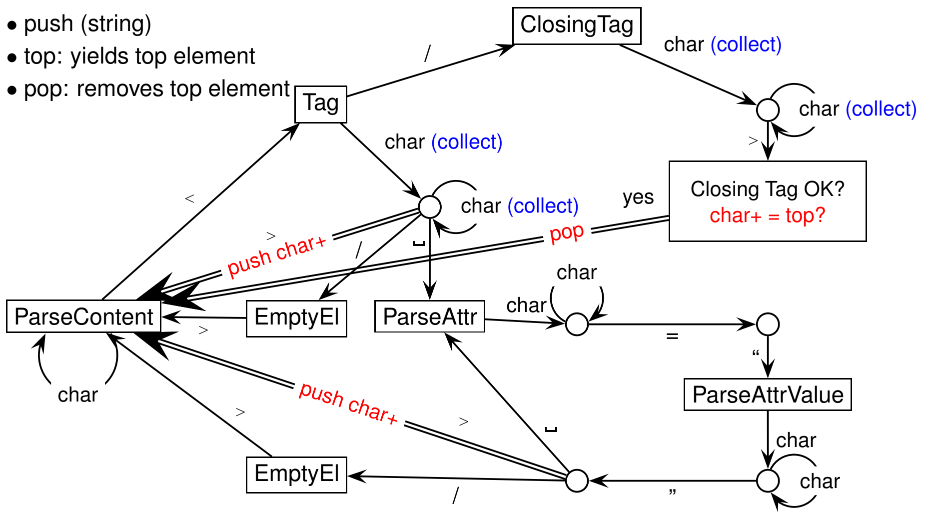
DOCUMENT	::=	BIB	
BIB	::=	"<bib>" BOOKS "</bib>"	
BOOKS	::=	ε | "<book year='"CHARS"'>" TITLE AUTHORS PUBLISHER PRICE "</book>" BOOKS	regexp: book*
		| "<book year='"CHARS"'>" TITLE EDITORS PUBLISHER PRICE "</book>" BOOKS	regexp: ...(auth+|edi+)...
TITLE	::=	"<title>" CHARS "</title>"	
AUTHORS	::=	AUTHOR | AUTHOR AUTHORS	regexp: author+
AUTHOR	::=	"<author>" LAST FIRST AFFILIATION "</author>"	
AFFILIATION	::=	ε | "<affiliation>" CHARS "</affiliation>"	regexp: affiliation?
⋮	⋮	⋮	
CHARS	::=	*characters*	

XML Grammar in General

- no DTD present/element names not known:
  Consider the grammar from Slide 150:
- ElementName is a separate production and
  Element ::= "<" ElementName Attributes ">" Content "</" ElementName ">"
  | "<" ElementName Attributes "/>"
  does not guarantee matching tags.
- Nevertheless, context-free-style parsing with push-down-automaton *without fixed stack alphabet* possible:
  - for every opening tag, put ElementName on the stack
  - for every closing tag, compare with top of stack, pop stack.
- Automaton: see next slide.

XML Grammar in General

Stack Commands:
- push (string)
- top: yields top element
- pop: removes top element



4.4 Example: XHTML

- XML documents that adhere to a strict version of the HTML DTD
- Goal: browsing, publishing
- DTD at `http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd`
  (note that the DTD requires also some entity files)
- Validator at `http://validator.w3.org/`
- Example at ... DBIS Web Pages
- only the text content is shown in the browser, all other content describes *how* the text is presented.
- no logical markup of the documents (sectioning etc), but
- only optical markup ("how is it presented").

Exercise

Design (and validate) a simple homepage in XHTML, and put it as `index.html` in your public-directory.

4.5 Miscellaneous about XML

4.5.1 Remarks

- all letters are allowed in element names and attribute names
- text (attribute values and element content) can contain nearly all characters.
  Western european umlauts are allowed if the XML identification contains encoding="UTF-8" or encoding="ISO-8859-1" etc.
- comments are enclosed in <!-- ... -->
- inside XML content,

  <![CDATA[...]]>

  (*character data sequences*) can be included that are not parsed by XML parsers, but which are copied character-by-character.

  E.g. in HTML:

  coloring: <![CDATA[XXX]]>
  prints XXX

  yields

  • coloring: XXX prints XXX

4.5.2 Entities

Entities serve as macros or as constants and are defined in the DTD. They are then accessible as "&*entityname*;" in the XML instance and in the DTD:

<!ENTITY *entity_name replacement_text*>

- additional special characters, e.g. ç:

  DTD: <!ENTITY ccedilla "&#231">
  XML: president="Fran&ccedilla;ois Mitterand"
- reserved characters can be included as *references to predefined entities*:
  < = < (less than), > = > (greater than)
  & = & (ampersand), *space* = , *apostroph* = ', *quote* = "
  ä = ä, ..., Ü = Ü

  <name>Düsseldorf </name>
- characters can also be given directly as character references, e.g. &#x20 (space), &#xD (CR).

Entities (cont'd)

- global definitions that may change can be defined as constants:

  DTD: <!ENTITY server "http://dbis.informatik.uni-goettingen.de">
  XML(HTML): Teaching
- macros that are needed frequently:

  DTD: <!ENTITY european
  "<encompassed continent='europe'>100</encompassed>">
  XML: <country car_code="D">
  <name>Germany</name>
  &european; ...
  </country>
- note: single and double quotes can be nested.

Parameter Entities

Entities that should be usable only in the DTD are defined as *parameter entities*:

- macros that are needed frequently:

  <!ENTITY % namedecl "name CDATA #REQUIRED">
  <!ATTLIST city
  %namedecl;
  zipcode ID #REQUIRED>
- define enumeration types:

  <!ENTITY % waters "(river|lake|sea)">
  <!ATTLIST city_located_at
  type %waters; #REQUIRED
  at IDREF #REQUIRED>

Entities from External Sources

External entities: the SYSTEM keyword allows to import external, reusable parts of a DTD (elements, attributes, often entity definitions):

<!ENTITY *entity_name* SYSTEM "*url or filename*">

e.g. a set of technical symbols:

<!ENTITY % isotech SYSTEM
"http://www.schema.net/public-text/ISOtech.pen">
%isotech;

the reference %isotech; makes then some technical symbols accessible that are defined in the external resource.

For another example, "importing" the XHTML DTD into Mondial, see Slide 240.

This can be iterated for defining "style files" that collect a set of external resources that are used by an author.

4.5.3 Processing Instructions

<?*pi-target something-without-"?>"* ?>

- Embedded PHP calls are an example for processing instructions

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head><title>PHP test</title></head>
  <body>
    <p>This should print hello world (if PHP is activated):
       <?php echo 'Hello World '; echo date('D, d M Y H:i:s'); ?></p>
  </body>
</html>
```

[Filename: XML-DTD/html-php.xml]

- the document validates: `xmllint -noout -valid html-php.xml`
- Browsing: PHP must be activated on the server (cf. Slide 617), files must be named *filename.php*
- Querying: see Slide 315.

4.5.4 Integration of Multimedia

- for (external) non-text resources, it must be declared which program should be called for showing/processing them. This is done by NOTATION declarations:

  <!NOTATION *notation_name* SYSTEM "*program_url*">
  <!NOTATION postscript SYSTEM "file:/usr/bin/ghostview">

- the entity definition is then extended by a declaration which notation should be applied on the entity:

  <!ENTITY *entity_name* SYSTEM "*url*"
  NDATA *notation_name*>
  <!ENTITY manual SYSTEM "file:/.../name.ps"
  NDATA postscript>

- the *application program* is then responsible for evaluating the entity and the NDATA definition.
- [XLink provides another mechanism for referencing resources – rarely used].

4.6 Summary and Outlook

XML: "basic version" consists of DTD and XML documents

- tree with additional cross references
- hierarchy of nested elements
- order of the subelements
  – documents: 1st, 2nd, ... section etc.
  – databases: order in general not relevant
- attributes
- references via IDREF/IDREFS
  - documents: mainly cross references
  - databases: part of the data (relationships)
- XML model similar to the network data model:
  relationships are mapped into the structure of the data model
  - the basic explicit, stepwise navigation commands of the network data model have an equivalent for XML in the DOM-API (see later), but
  - XML also provides a declarative, high-level, set-oriented language.

Requirements

- Documents: logical markup (Sectioning etc.)
  presentation on Web pages in (X)HTML? – transformation languages
- databases: structuring of data;
  several equivalent alternatives
  query languages?
  presentation on Web pages in (X)HTML? – transformation languages
- application-specific formats:
  DTDs are induced by the application-programs
  XHTML: browsing
  ant: configuration of automated software build process
  Web-Services: WSDL, UDDI; CAD; ontology languages; . . .
  transformation between different XML languages
  application-programs must "understand" XML internally

Further Concepts of the XML World

Extensions:

- namespaces: use of different DTDs in a database
  (see Slide 226)
- APIs: DOM, SAX
- theoretical foundations
- query languages: XPath, XQL, XML-QL, Quilt, XQuery
- stylesheets/transformation languages: CSS, DSSSL, XSL
- better schema language: XML Schema
- [XML with inter-document handling: XPointer, XLink – rarely used]

4.7 Recall

- XML as an abstract *data model*
  - cf. relational DM
  - XML now has become less abstract: creation of instances in the editor, validating, viewing ...
- a data model needs ... implementation? theory?
- ... first, something else: *abstract datatype, interface(s)*
  - constructors, modifiers, selectors, predicates (cf. Info I)
- here: "two-level model"
  - as an ADT (programming interface): Document Object Model (DOM):
    detailed operations as usual in programming languages (Java, C++).
  - as a database model (end user interface; declarative):
    import (deserialize, parser), *queries*, updates, export (serialize)
- theory: formal specification of the semantics of the languages, other issues are the same as in classical DB theory (transactions etc.).

Chapter 5
Query Languages: XPath

- Network Data Model: no query language; only some specific commands extending the host language
- SQL – only for a flat data model, but a "nice" language
  (easy to learn, descriptive, relational algebra as foundation, clean theory, optimizations)
- OQL: SQL with object-orientation and path expressions
- Lorel (OEM): extension of OQL
- F-Logic: navigation in a graph by path expressions with additional conditions
  descriptive, complex.

Requirements on an XML Query Language

- suitable both for databases and for documents
- declarative: binding variables and using them
  - rule-based, or
  - SQL-style clause-based (which is in fact only syntactic sugar)
- binding variables in the rule body/selection clause:
  suitable for complex objects
  - navigation by path expressions, or
  - patterns
- generation of structure in the rule head/generating clause

Evolution of XPath

- when defining a query language, constructs are needed for addressing and accessing individual elements/attributes or sets of elements/attributes.
- based on this *addressing mechanism*, a clause-based language is defined.

Early times of XML (1998)
different navigation formalisms of that kind:

- XSL Patterns (inside the stylesheet language)
- XQL (XML Query Language)
- XPointer (referencing of nodes/areas in an XML document)

used all the same basic idea with slight differences in the details:

- paths in UNIX notation
- conditions on the path

  /mondial/country[@car_code="D"]/city[population > 100000]/name

5.1 XPath – the Basics

1999: specification of the navigation formalism as *W3C XPath*.

- Base: UNIX directory notation

  in a UNIX directory tree: /home/dbis/Mondial/mondial.xml
  in an XML tree: /mondial/country/city/name

  Straightforward extension of the URL specification:
  http://.../dbis/Mondial/mondial.xml#mondial/country/city/name [XPointer until 2002]
  http://.../dbis/Mondial/mondial.xml#xpointer(mondial/country/city/name) [XPointer now]
- W3C: XML Path Language (XPath), Version 1.0 (W3C Recommendation 16. 11. 1999)
  `http://www.w3.org/TR/xpath`
- W3C: XPath 2.0 and XQuery 1.0 (W3C Recommendation 23. 1. 2007)
  `http://www.w3.org/TR/xquery`
- Tools: see Web page
  - XML (XQuery) database system "eXist"
  - lightweight tool "saxonXQ" (XQuery)

XPath: Navigation, Simple Examples

XPath is based on the UNIX directory notation:

- /mondial/country
  addresses all country elements in Mondial,
  the result is a set of elements of the form
  <country code="..."> ... </country>
- /mondial/country/city
  addresses all city elements, that are direct subelements of country elements.
- /mondial/country//city
  adresses all city elements that are subelements (in any depth) of country elements.
- //city
  addresses all city elements in the current document.
- wildcards for element names:
  /mondial/*/name
  addresses all name elements that are grandchildren of the mondial elements
  (different from /mondial//name which goes to arbitrary depth!)

... and now systematically:

XPath: Access Paths in XML Documents

- Navigation paths

  /step/step/.../step

  are composed by individual navigation steps,
- the result of each step is a sequence of nodes, that serve as input for the next step.
- each step consists of

  axis::*nodetest*[*condition*]*

  – an axis (optional),
  – a test on the type and the name of the nodes,
  – (optional) predicates that are evaluated for the current node.
- paths are combined by the "/"-operator
- additionally, there are function applications
- the result of each XPath expression is a *sequence* of nodes or literals.

XPath: Axes

Starting with a *current node* it is possible to navigate in an XML tree to several "directions" (cf. xmllint's "cd"-command).

In each navigation step

path/*axis*::*nodetest*[*condition*]/*path*

the *axis* specifies in which direction the navigation takes place. Given the sequence of nodes that is addressed by *path*, for *each* node, the step is evaluated.

- Default: child axis: child::country ≡ country.
- Descendant axis: all sub-, subsub-, ... elements:
  country/descendant::city
  selects all city elements, that are contained (in arbitrary depth) in a country element.
  Note: *path*//city actually also addresses all these city elements, but "//" is *not* the exact abbreviation for "/descendant::" (see later).

XPath: Axes

... another important axis:

- attribute axis:
  attribute::car_code ≡ @car_code
  wildcard for attributes: attribute::* selects all attributes of the current context node.
- and a less important:
  self axis: self::city ≡ ./city
  selects the current element, *if* it is of the element type city.

for the above-mentioned axes there are the presented abbreviations. This is important for *XSL patterns* (see Slide 351):

XSL (match) patterns are those XPath expressions, that are built *without* the use of "axis::" (the abbreviations are allowed).

XPath: Axes

Additionally, there are axes that do not have an abbreviation:

- parent axis: //city[name="Berlin"]/parent::country
  selects the parent element of the city element that represents Berlin, *if* this is of the element type country.
  (*only* the parent element, not all ancestors!)
- ancestor: all ancestors:
  //city[name="Berlin"]/ancestor::country selects all country elements that are ancestors of the city element that represents Berlin (which results in the Germany element).
- siblings: following-sibling::..., preceding-sibling::...
  for selecting nodes on the same level (especially in ordered documents).
- straightforward: "descendant-or-self" and "ancestor-or-self".
  Note: The popular short form country//city is defined as
  country/descendant-or-self::node()/city.
  This makes a difference only in case of *context functions* (see Slide 221).

XPath: Axes for use in document-oriented XML

- following: all nodes after the context node in document order, excluding any descendants and excluding attribute nodes
- preceding: all nodes that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes

Note: For each element node x, the ancestor, descendant, following, preceding and self axes *partition* a document (ignoring attribute nodes): they do not overlap and together they contain all the nodes in the document.

Example:

Hamlet: what is the next speech of Lord Polonius after Hamlet said "To be, or not to be"?
(note: this can be in a subsequent scene or even act)

Exercise:

Provide equivalent characterizations of "following" and "preceding"

i) in terms of "preorder" and "postorder",

ii) in terms of other axes.

XPath: Nodetest

- The *nodetest* constrains the node type and/or the names of the selected nodes
- test if something is a node: //city[name="Berlin"]/descendant::node()
  returns all descendant nodes.
- test if something is an element node: //city[name="Berlin"]/descendant::element()
  returns all descendant *elements* (i.e., not the text nodes).
- test if something is a text node: //city[name="Berlin"]/descendant::text()
  returns all descendant *text nodes*.
  //city[name="Berlin"]/population/text()
  returns the text contents of all population child elements (as a sequence of text nodes).
- test for a given element name:
  //country[name="Germany"]/descendant::element(population)
  or short form:
  //country[name="Germany"]/descendant::population
  returns all descendant *population elements*.
- "*" as wildcard: //city[name="Berlin"]/child::*
  returns all child *elements* of any element name (analogously for attribute::* and @*).

XPath: Tests

In each step

path/*axis*::*nodetest*[*condition*]/*path*

condition is a predicate over XPath expressions.

- The expression selects only those nodes from the result of *path*/*axis*::*nodetest* that satisfy *condition*. *condition* contains XPath expressions that are evaluated relative to the current *context node* of the respective step.
  //country[@car_code="D"]
  returns the country element whose car_code attribute
  has the value "D"
- When comparing an element with something, the string() method is applied implicitly:
  //country[name = "Germany"] is equivalent to
  //country[name/string() = "Germany"]
- If the right hand side of the comparison is a number, the comparison is automatically evaluated on numbers:
  //country[population > 1000000]

XPath: Tests (cont'd)

- boolean connectives "and" and "or" in *condition*:
  //country[population > 100000000 and @area > 5000000]
  //country[population > 100000000 or @area > 5000000]
- boolean "not" is a *function*:
  //country[not (population > 100000000)]
- XPath expressions in *condition* have existential semantics:
  The *truth value* associated with an XPath expression is *true*, if its result set is non-empty:
  //country[inflation]
  selects those countries that have a subelement of type inflation.

  ⇒ formal semantics: a path expression has
  - a semantics as a result set, and
  - a truth value!

XPath: Tests (cont'd)

- XPath expressions in *condition* are not only "simple properties of an object", but are path expressions that are evaluated wrt. the current context node:

  //city[population/@year='1995']/name

- Such comparisons also have existential semantics, when one comparand is a node sequence:

  //country[.//city/name='Cordoba']/name

  returns the names of all countries, in which *some* city with name Cordoba is located.

  //country[not (.//city/name='Cordoba')]/name

  returns the names of those countries where no city with name Cordoba is located.

XPath: Evaluation Strategy

- Input for each navigation step: A *sequence* of nodes (*context*)
- each of these nodes is considered separately for evaluation of the current step
- and returns zero or more nodes as (intermediate) result.
  This intermediate result serves as context for the next step.
- finally, all partial results are collected and returned.

Example

- conditions can be applied to multiple steps

  //country[population > 10000000]
  //city[located_on and population > 1000000]
  /name/text()

  returns the names of all cities that have more than 1,000,000 inhabitants and are located (at least partially) on an island and in a country that has more than 10,000,000 inhabitants.

Absolute and Relative Paths

So far, conditions were always evaluated only "local" to the current element on the main navigation path.

- Paths that start with a name are *relative* paths that are evaluated against the current context node (used in conditions):

  //city[name = "Berlin"]

- Semijoins: comparison with results of independent "subqueries":
  Paths that start with "/" or "//" are absolute paths:

  //country[number(@area) > //country[@car_code='B']/@area]/name

  returns the names of all countries are bigger than Belgium.

  – automatically, the string values of the attributes are taken,
  – casting to number must be applied on (at least) one side.

- conflict between "//" for absolute paths and for descendant axis:

  //country[.//city/name="Berlin"]
  (equivalent: //country[descendant::city/name="Berlin"])

  can be used for starting a relative path.

XPath: Functions

Input: a node/value or a set of nodes/values.
Result: in most cases a value; sometimes one or more nodes.

- dereferencing (see Slide 210)
- access to text value and node name (see Slide 213)
- aggregate functions count(*node_set*), sum (*node_set*)

  count(/mondial/country)

  returns the number of countries.

- context functions (see Slide 220)
- access to documents on the Web:

  doc("*file or url*")/*path*
  doc('http://www.dbis.informatik.uni-goettingen.de/index.html')//text()

  (for querying external HTML documents, consider use of namespaces as described on Slide 239 - nodetests work only with namespace!)

- see W3C document *XPath/XQuery Functions and Operators*

IDREF ATTRIBUTES

- ID/IDREF attributes serve for expressing cross-references
- SQL-style: (single-IDREF) references can be resolved by semi-joins: (similar to foreign keys in SQL)

  //city[@id = //organization[abbrev=“EU”]/@headq]

  SQL equivalent (uncorrelated subquery):

```
SELECT *
FROM city
WHERE (name, country, province) IN
    (SELECT city, country, province
     FROM organization
     WHERE abbrev = 'EU')
```

... not a really elegant way in a graph-based data model ...
and would not work for IDREFS (white-space-separated tokens)

XPATH: DEREFERENCING

Access via “keys”/identifiers

The function id(*string**) returns all elements (of the current document) whose id’s are enumerated in *string**:

- id(“D”) selects the element that represents Germany
  (country/@car_code is declared as ID)
- id(//country[car_code=“D”]/@capital)
  yields the element node of type city that represents Berlin.

This notation is hard to read if multiple dereferencing is applied, e.g.

id(id(id(//organization[abbrev=’IOC’]/@headq)/@country)/@capital)/name

Alternative syntaxes:

//organization[abbrev=’IOC’]/id(@headq)/id(@country)/id(@capital)/name
//organization[abbrev=’IOC’]/@headq/id(.)/@country/id(.)/@capital/id(.)/name

XPath: Dereferencing (Cont’d)

Analogously for multi-valued reference attributes (IDREFS):

- //country[@car_code=“D”]/@memberships
  returns “org-EU org-NATO ...”
- id(//country[@car_code=“D”]/@memberships)
  //country[@car_code=“D”]/id(@memberships)
  returns the set of all elements that represent an organisation where Germany is a member.
- id(//organization[abbrev=“EU”]/members/@country)
  //organization[abbrev=“EU”]/members/id(@country)
  returns all countries that are members (of some kind) in the EU.

Aside: Dereferencing by Navigation [Currently not supported]

Syntax:

attribute::*nodetest*⇒*elementtype*

Examples:

- //country[car_code=“D”]/@capital⇒city/name
  yields the element node of type city that represents Berlin.
- //country[car_code=“D”]/@memberships⇒organization
  yields elements of type organization.
- Remark: this syntax is not supported by all XPath Working Drafts:
  - XPath 1.0: no
  - has originally been introduced by Quilt (2000; predecessor of XQuery)
  - XPath 2.0: early drafts yes, later no
  - announced to be re-introduced later ...

XPath: string() Function

The *function* string() returns the string value of a node:

- straightforward for elements with text-only contents:
  string(//country[name='Germany']/population[1])
  Note: for these (and only for these!) nodes, text() and string() have the same semantics.
- for attributes: //country[name='Germany']/string(@area)
  Note: an attribute node is a name-value pair, not only a string (will be illustrated when constructing elements later in XQuery)!
  free-standing attribute nodes as result cannot be printed!
- the string() function can also be appended to a path; then the argument is each of the context nodes: //country[name='Germany']//name/string()
- the string value of a subtree is the concatenation of all its text nodes:
  //country[@name='Germany']/string()
  Note: compare with //country[@name='Germany']//text() which lists all text nodes.
- string() *cannot* be applied to node sequences: string(//country[name='Germany']//name) results in an error message.
  (see W3C XPath and XQuery Functions and Operators).

XPath: Some more Details on Comparisons

- in the above examples, all predicate expressions like [name="Berlin"] or [@car_code="D"] always *implicitly* compare the string value of nodes, e.g., here the string values of <name>Berlin</name> or attribute: (car_code, "D").

Usage of Numbers

- comparisons using $>$ and $<$ and a number literal given in the query implicitly cast the string values as *numeric* values.

  //city[population > 200000]
  returns the all cities with a population higher than 200,000.

  //city[population > '200000']
  returns the all cities with a population *alphabetically* "bigger" than 200,000, e.g., 3500, but not 1,000,000!

  //city[population > //city[name="Munich"]/population]
  does *not* recognize that numerical values are meant:
  All cities with population alphanumerically bigger than "1244676" are returned.

  //city[population > //city[name="Munich"]/population/number()]
  It is sufficient to apply the number() casting function (see later) to one of the operands.

XPath: Comparison between Nodes

Usage of Node Identity

- as seen above, the "=" predicate uses the string values of nodes.
  In most cases, this is implicitly correct:
  Consider the following query: "Give all countries whose capital is the headquarter of an organization":

  //country[id(@capital)=//organization/id(@headq)]/name

  Compares the overall string values of city elements, e.g., "Brussels 4.35 50.8 951580".
- but for empty nodes, the result is not as intended ...

Comparison of Nodes

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mondial-simple SYSTEM "mondial-simple.dtd">
<mondial-simple>
 <country car_code="D" capital="Berlin"/>    <city name="Berlin"/>
 <country capital="Brussels" car_code="B"/>  <city name="Brussels"/>
 <organization name="EU" headq="Brussels"/>
</mondial-simple>
```
[Filename: XPath/node-comparison.xml]

- the query //country[id(@capital)=//organization/id(@headq)]/string(@car_code) yields both "D" and "B" (city@name is the id attribute).
- Test for node identity see Slide 223 (since XPath 2.0).
- "deep equality" of nodes can be tested with the predicate deep-equal(x, y).
  (by this, two subtrees are checked to have the same structure+contents (including (unordered) attribute sets))
- the query
  //country[deep-equal(id(@capital), //organization/id(@headq))]/string(@car_code)
  yields only "B".

XPath: Predicates and Operations on Strings

- concat(string, string, string*)
  also the SQL-like infix operator || is allowed (since XQuery 3.0)
- startswith(string, string)
  //city[starts-with(name,'St.')]/name
- contains(string, string)
  //city[contains(name,'bla')]/name
- substring-before(string, string, int?)
- substring-after(string, string, int?)
- substring(string, int, int): the substring consisting of i_2 characters starting with the i_1th position.

XPath: Name Function

- the function name() returns the element name of the current node:
  - name(//country[@car_code='D']) or
    //country[@car_code='D']/name()
  - //*[name='Monaco' and not (name()='country')] yields only the city element for Monaco.

XPath: IDREF Function

- the function idref(string*) returns all nodes that have an IDREF value that refers to one of the given strings (note that the results are attribute nodes):
  idref('D')/parent::*/name yields the name elements of all "things" that reference Germany.

Functions on Nodesets

- Aggregation: count(*nodeset*), sum(*nodeset*), analogously min, max, sum, avg
  sum(//country[encompassed/id(@continent)/name="Africa"]/@area)
  count(//country)
  all numeric functions implicitly cast to numeric values (double).
- removal of duplicates:
  - recall that the XPath strategy works on *sets of nodes* in each step - duplicate *nodes* are automatically removed:
    //country/encompassed/id(@continent)
    Starting with 244 countries, yielding a *set* of five continent nodes
  - function distinct-values(*nodeset*):
    takes the *string values* of the nodes and removes duplicates:
    doc('hamlet.xml')//SPEAKER
    returns lots of <SPEAKER>...</SPEAKER> *nodes*.
    distinct-values(doc('hamlet.xml')//SPEAKER)
    returns only the different (text) *values*.
- and many more (see W3C XPath/XQuery Functions and Operators).

XPath: Context Functions

- All functions retain the order of elements from the XML document (document order).
- the position() function yields the position of the current node in the *current result set*.
  /mondial/country[position()=6]
  Abbreviation: [x] instead of [position()=x]; [last()] yields the last node:
  /mondial/country[population > 1000000][6]
  selects the 6th country that has more than 1,000,000 inhabitants (in document order, not the one with the 6th highest population!)
  /mondial/country[6][population > 1000000]
  selects the 6th country, if it has more than 1,000,000 inhabitants.
- the last() function returns the position of the last elements of the current sub-results, i.e., the size of the result.
  //country[@car_code='D']/population[position()=last()] or
  //country[@car_code='D']/population[last()]
  for the most recent (last) population count.

XPath: Context Functions (Cont'd)

- consider again the "//" abbreviation (cf. Slide 200):
  - /mondial/descendant::city[18] selects the 18th city in the document,
  - /mondial/descendant-or-self::node()/city[18] selects each city which is the 18th child of its parent (country or province).
    (note that some implementations are buggy in this point ...)
- Note: above, position() is used for *filtering*.
  Selecting and displaying the position is more complex:

  //country[@car_code='D']/position()

  yields "1" (for every country), since after evaluating //country[@car_code='D'], the context contains only one element – and its position in the context is 1.

  See Slides 304 (XQuery) and Slides 393 (XSLT).
- Example queries against mondial.xml and hamlet.xml.

XPath: Forward- and Backward Axes

- the result of each query is a *sequence of nodes*
- document order (and final results): forward
- context functions: forward or backward
- all axes enumerate results starting from the current node.
  - forward axes: child, descendant, following, following-sibling
  - backward axes: ancestor, preceding, preceding-sibling

    //SPEECH[contains(.,'To be, or not to be')]/preceding-sibling::SPEECH

    selects all preceding speeches.

    The result is -as always- output in document order.

    //SPEECH[contains(.,'To be, or not to be')]/preceding-sibling::SPEECH[1]

    selects the last preceding speech (context function on backward axis)
  - undirected: self, parent, attribute.
- only relevant for queries against document-oriented XML.

Extensions with XPath 2.0

- first draft already in 2001 after first XQuery drafts; W3C Recommendation since 2007
- more complex path constructs (alternatives, parentheses)
  (//city|//country)[name='Monaco']
  /mondial/country/(city|(province/city))/name
- constructor "," for sequences, e.g., to be used in (item-wise!) comparisons:
  - /mondial/country[@car_code = ('D', 'B', 'F')]
  - /mondial/country[position() = (1, 5 to 9, 64)]
    yields the first, the 5th to 9th, and the 64th country
- Comparison wrt. *node identity* is done by "is"
  - recall from Slide 216: node comparison only by string value comparison or deep-equality in XPath 1.0
  - "is" requires both comparands to be single nodes; not node sequences (cf. Slide 224)
  - //country[id(@capital) is //organization[abbrev='EU']/id(@headq)]/name
- alignment of the whole XML world (XPath, XQuery) with datatypes (data model and XML Schema)

Extensions with XPath 2.0: every and some – logical quantifiers

- logical $\forall$ and $\exists$ semantics for conditions:
  countries where all/at least one city has more than 1000000 inhabitants:
  //country[every $p in .//city/population[last()] satisfies $p > 1000000]
  //country[some $p in .//city/population[last()] satisfies $p > 1000000]

Quantifiers extend the language to more than navigation

- the usage and syntax of variables is inherited from XQuery 1.0 (2001),
- quantifiers motivated by the relational calculus
  (recall also EXISTS from SQL),
- break with the simplicity of XPath,
- "some"? – the XPath 1.0 comparisons have existential semantics
  ... when sequences are allowed in the comparison; otherwise the explicit "some" has to be used:
  //country[some $org in //organization satisfies $org/id(@headq) is id(@capital)]/name
- "every" is obviously useful
  (remember the usage of relational division in SQL)

XPath with XPath 2.0's logical quantifiers

Compare with relational algebra, relational calculus:

- inside of "[...]", variables and (even nested) quantifiers are allowed:
  - selection: filters
  - projection: not supported (but inside conditions everything where a projection is used can be replaced by variables and "and")
  - join: some $\$x_1$ in $expr_1$ satisfies (...(some $\$x_n$ in $expr_n$ satisfies *subexpr*($\$x_1 ... \$x_n$))...)
  - union: "|", "or"
  - non-atomic negation/set difference: not
  - universal quantification: "every" or like in SQL via "not some ... not"

⇒ wrt. boolean queries (yes/no) and unary (i.e. result has a single column) queries, relational completeness is obtained.

- missing: recombination of results (joins, generation of XML structures)
- complex queries are hard to write (and to test)

Exercise

- Give the names of all organizations that have at least one member on each continent.

5.2 Aside: Namespaces

The names in an XML instance (i.e., tag names and the attribute names) actually consist of two parts:

- localpart + namespace (which can be empty, as in the previous examples)

Use of Namespaces

- a namespace is similar to a language: defining a set of names and sometimes having a DTD (if intended as an XML vocabulary).
- e.g. "mondial:city", "bib:book", "xhtml:tr" "dc:author", "xsl:template" etc.
- used for distinguishing coinciding element names in different application areas.
- each namespace is associated with a URI (which can be a "real" URL), and abbreviated by a *namespace prefix* in the document.
- e.g., associate the namespace prefix `xhtml` with url `http://www.w3.org/1999/xhtml`. these things will become clearer when investigating the RDF, RDFS, and Semantic Web Data Models.

Usage of Namespaces in XML Documents

- each element can have (or can be in the scope of) multiple *namespace declarations* (represented by a node in the data model, similar to an attribute node).
- namespace declarations are inherited to subelements
- the element/tag name and the attribute names can then use one of the declared namespaces.
  By that, every element can have one *primary namespace* and "knows" several others.

Alternatives:

1. the elements have no namespace (e.g. mondial),
2. the document declares a default namespace (for all elements (not the attributes!) that do not get an explicit one (often in XHTML pages)),
3. elements have an explicit namespace (multiple namespaces allowed in a document; e.g. an XSL document that operates with XHTML markup and "mondial:" nodes).

- (2) and (3) are semantically equivalent.

... see next slides.

Explicit Namespace in an XML Document

```
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml">
 <xh:body>
  <xh:h3>Header</xh:h3>
  <xh:a href="http://www.informatik.uni-goettingen.de">IFI</xh:a>
 </xh:body>
</xh:html>
```

[Filename: XML-DTD/xhtml-expl-namespace.xml]

- Note: attribute is not in the HTML namespace!

This is actually already not XPath, but a simple XQuery query:

```
declare namespace ht = "http://www.w3.org/1999/xhtml";
/ht:html//ht:a/string(@href)
```

[Filename: XPath/xhtml-query.xq]

- Note: the namespace *must* be used in the query, i.e., "ht:html" is different from just "html"
- more accurate, it means something like `<{http://www.w3.org/1999/xhtml}html>...</...>` since not the chosen namespace prefix matters, but only the URI assigned to it.

Two Explicit Namespaces in an XML Document

- “Dublin Core” defines a vocabulary for metadata description of resources (here: of XML documents); cf. `http://dublincore.org/documents/dces/`

```
<xh:html xmlns:xh="http://www.w3.org/1999/xhtml"
         xmlns:dc="http://purl.org/dc/elements/1.1/">
 <xh:head> <dc:creator>John Doe</dc:creator>
           <dc:date>1.1.2000</dc:date> </xh:head>
 <xh:body> ... </xh:body> </xh:html>
```

[Filename: XML-DTD/xhtml-expl-namespaces.xml]

```
declare namespace ht = "http://www.w3.org/1999/xhtml";
declare namespace dc = "http://purl.org/dc/elements/1.1/";
/ht:html//dc:creator/text()
```

[Filename: XPath/xhtml-dc-query.xq]

- the document is *not* valid wrt. the XHTML DTD since it contains additional “alien” elements.
  (combination of languages is a problem in XML – this is better solved in RDF/RDFS)
- in RDF, dc:creator from above expands to the URI `http://purl.org/dc/elements/1.1/creator`.

(Different) Default Namespaces in an XML Document

- a Default Namespace can be assigned to an element (and inherited to all its subelements where it is not overwritten):

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:dc="http://purl.org/dc/elements/1.1/">
 <head> <dc:creator>John Doe</dc:creator>
        <date xmlns="http://purl.org/dc/elements/1.1/">1.1.2000</date> </head>
 <body> ... </body> </html>
```

[Filename: XML-DTD/xhtml-def-namespaces.xml]

```
declare namespace ht = "http://www.w3.org/1999/xhtml";
declare namespace dc = "http://purl.org/dc/elements/1.1/";
/ht:html/ht:head/dc:date/text()
```

[Filename: XPath/xhtml-dc-def-query.xq]

Namespaces and Attributes

- Namespaces are *not* inherited to attributes in any case. If an attribute should be associated with a namespace, this *must* be done explicitly:

```
<ht:html xmlns:ht="http://www.w3.org/1999/xhtml">
 <ht:body>
  <ht:a href="1+" ht:href="2-">IFI</ht:a>
  <x:a xmlns:x="http://www.w3.org/1999/xhtml" href="3+" x:href="4-">IFI</x:a>
  <a xmlns="http://www.w3.org/1999/xhtml" href="5+" ht:href="6-">IFI</a>
 </ht:body> </ht:html>
```

[Filename: XML-DTD/namespaces-attr.xml]

```
declare namespace ht = "http://www.w3.org/1999/xhtml";
/ht:html//ht:a/@href/string()
```

[Filename: XPath/namespaces-attr-query.xq]

- the “HTML-correct” attributes “1+”, “3+”, and “5+” are returned,
- the query /ht:html//ht:a/@ht:href/string() returns the “wrong” attributes “2-”, “4-”, and “6-”.

Declaring Namespaces in the DTD Document

- introduce default namespace in the DTD as attribute of the root element (e.g. in the W3C XHTML DTD) described at
  `https://www.w3.org/TR/xhtml1/dtds.html#a_dtd_XHTML-1.0-Strict`

```
<!ELEMENT html (head, body)>
<!ATTLIST html
   xmlns  %URI;  #FIXED 'http://www.w3.org/1999/xhtml' >
```

- an XHTML instance (or subtree, cf. Slide 240) loading this DTD automatically extends to:
  <html *xmlns=“http://www.w3.org/1999/xhtml”*> <body> ... </body></html>
- introduce explicit namespaces as attribute of the root element:
  <!ELEMENT html (head, body)>
  <!ATTLIST html xmlns:xh %URI; #FIXED 'http://www.w3.org/1999/xhtml' >

  This is extensively used with RDF/XML in the Semantic Web.

Declaring a Default Namespace in XQuery

XQuery allows to declare default namespaces for elements and for functions:

- are then added to each element and function step, respectively;
- not for attributes (recall that namespaces from elements are not inherited to attributes). (cf. Slide 231)

```
declare default element namespace "http://www.w3.org/1999/xhtml";
/html//a/@href/string()
```

[Filename: XPath/namespaces-default-query.xq]

- the "HTML-correct" attributes "1+", "3+", and "5+" are returned,
- the equivalent query is `/h:html//h:a/@href/string()`.

Exclusive Canonical XML

- Required for some applications (e.g., usage of XMLLiteral values in the "Jena" Semantic Web Framework)
- XML fragments/subtrees must be processable without their context – thus, namespaces must be present at appropriate levels in the tree.
- Details: `http://www.w3.org/TR/xml-exc-c14n/`
- in case you ever need it: can be obtained with `xmllint -exc-c14n x.xml > y.xml` (and analogously by other tools)

5.3 Aside: XML Catalogs

(cf. introductory note at Slide 163)

Accessing an XHTML document that contains a reference to W3Cs XHTML DTD at `http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd` via software (other than a browser) fails since the DTD is not accessible.

- an XML catalog is a dictionary *uri*→*accessible_url*:
- whenever the resource identified by *uri* is referenced, take the resource that is actually *accessible* at *accessible_url* (usually a local copy of the item).
  - DTDs
  - entity references (cf. Slide 185),
  - a graphics for an HTML `<img src="uri"/>`, e.g. a company's logo
  - anything for an XML Inclusion (XInclude; cf. Slide 491)
- Software then uses a *Resolver* instance.

XML Catalog

- XML catalogs are XML documents themselves
- a catalog contains different subelements
- default catalog at /etc/xml/catalog (only root can change it),
- usage from several tools: put it in a central place (e.g., `~/teaching/ssd/XMLCatalog`),
- if a tool or a servlet uses an own catalog (e.g., the XQuery Web interface) it can have an own, local one.
- put the DTDs (etc.) that should be made accessible somewhere, e.g., next to the catalog in a "DTD" subdirectory.

```
<?xml version="1.0"?>
<!DOCTYPE catalog PUBLIC "-//OASIS//DTD XML Catalogs V1.0//EN"
  "file:///usr/share/xml/schema/xml-core/catalog.dtd">
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
  <system systemId="http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd"
          uri="DTD/xhtml1-strict.dtd"/>
  <system systemId="http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd"
          uri="DTD/xhtml1-transitional.dtd"/>
</catalog>
```

[Filename: ~dbis/XMLTools/XMLCatalog/catalog]

Required files for XHTML

- xhtml1-strict.dtd, xhtml1-transitional.dtd,
- xhtml-lat1.ent, xhtml-symbol.ent, xhtml-special.ent

Using the XML Catalog

- software comes with a resolver, or
- get the XML Commons Resolver (resolver.jar) from Apache, put it somewhere (e.g. also below the XMLCatalog directory).
- since version 9.4 (Dec. 2011), saxon uses local copies of the W3C DTDs automatically.
- when (non-XHTML) XML documents with public DTD references are used frequently, copying them and using a catalog entry saves time and Web traffic.
- Technical description for using catalogs in saxon can be found at `http://sourceforge.net/apps/mediawiki/saxon/index.php?title=XML_Catalogs` and `http://saxonica.com/documentation/sourcedocs/xml-catalogs.xml`.

Saxon Call with Catalog until 9.3

- Java -D: set environment variable for java
- saxon -r,-x allows to refer to appropriate classes explicitly

```
java -cp $DBIS/XML-Tools/saxon/saxon9.jar:$DBIS/XML-Tools/XMLCatalog/resolver.jar \
 -Dxml.catalog.files=$DBIS/XML-Tools/XMLCatalog/catalog \
 net.sf.saxon.Query \
 -r:org.apache.xml.resolver.tools.CatalogResolver \
 -x:org.apache.xml.resolver.tools.ResolvingXMLReader \
 catalogtest.xq
```

[Filename: XMLCatalog/saxon.call.old]

- (for saxonXSL: -r, -x, -y)

Shorter with -catalog (Saxon 9.4)

```
java -cp $DBIS/XML-Tools/saxon/saxon9.jar:$DBIS/XML-Tools/XMLCatalog/resolver.jar \
 net.sf.saxon.Query \
 -catalog:$DBIS/XML-Tools/XMLCatalog/catalog \
 catalogtest.xq
```

[Filename: XMLCatalog/saxon.call]

```
doc('http://www.dbis.informatik.uni-goettingen.de/')
```

[Filename: XMLCatalog/catalogtest.xq]

Example: Querying XHTML in Presence of Namespaces

XHTML DTD at `http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd` contains:

```
<!ELEMENT html (head, body)>
<!ATTLIST html id    ID    #IMPLIED
               xmlns %URI; #FIXED 'http://www.w3.org/1999/xhtml'>
```

Sample XHTML files:

- DBIS Web pages:

```
declare namespace h = "http://www.w3.org/1999/xhtml";
doc('http://www.dbis.informatik.uni-goettingen.de/')//h:li/h:a/@href/string()
```

[Filename: XPath/web-queries.xq]

5.4 Use Case: Mondial with Embedded HTML Fragments

- define a default namespace for Mondial
  - is then inherited to all subelements (except they overwrite it, as the html element will do ...)
  - could also be declared as FIXED in the DTD (cf. the XHTML DTD)
- just use (XHTML-valid) html subelements (their default namespace URL will be FIXED in the original W3C XHTML DTD), inside the city elements,
- non-default namespace prefixes can be used, but then prefix:localname acts in the DTD like a name that contains a “:” – the prefix cannot be changed.

Use Case: Mondial with Embedded HTML and Dublin Core Fragments

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE mondial SYSTEM "monhtml.dtd">
<mondial xmlns="https://www.semwebtech.org/mondial/10/">
  <country car_code="D" capital="berlin">
    <name>Germany</name>
    <html><head><title>GERMANY</title></head><body><p>...</p></body></html>
    <city id="berlin"><name>Berlin</name>
      <dc:author>John Doe</dc:author>
      <html><head><title>BERLIN</title></head><body><p>...</p></body></html>
    </city>
    <city id="hambg"><name>Hamburg</name>
      <dc:author>Jane Doe</dc:author>
      <html><head><title>HAMBURG</title></head><body><p>...</p></body></html>
    </city>
  </country>
</mondial>
```
[Filename: XQuery/monhtml.xml]

- Next slide: DTD that defines the XHTML default namespace by a FIXED attribute for the <html> element

The Mondial+HTML DTD

- extend the Mondial DTD:
  - allow for an xmlns attribute (could be FIXED),
  - allow nested html elements where needed,
  - add the city's dc:author subelement declaration stuff (with hardcoded prefix),
  - reference the XHTML DTD (by its URL) using a SYSTEM external entity.

```
<!ELEMENT mondial (country*)>
  <!ATTLIST mondial xmlns CDATA #IMPLIED>
<!ELEMENT country (name+, html?, city+)>
<!ATTLIST country car_code ID #REQUIRED
                  capital IDREF #IMPLIED>
<!ELEMENT name (#PCDATA)>
<!ELEMENT city (name+, dc:author?, html?)>
  <!ATTLIST city id ID #REQUIRED>

<!ELEMENT dc:author (#PCDATA)>
  <!ATTLIST dc:author xmlns:dc CDATA #FIXED 'http://purl.org/dc/elements/1.1/'>

<!ENTITY % htmldtd SYSTEM 'http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd'>
%htmldtd;
```
[Filename: XQuery/monhtml.dtd]

Validate Mondial+HTML wrt. both DTDs

- use saxon to validate – it will automatically use its internal XHTML DTD
- xmllint tries (unsuccessfully) to access the W3C DTD. For it, a dirty workaround is

```
<!ENTITY % htmldtd SYSTEM 'myxhtml.dtd'>
%htmldtd;
```

  and copy the W3C DTD to this local file (and download or remove references at its beginning).

Querying Mondial+HTML

- The XHTML DTD defines (fixed)

```
<!ELEMENT html (head, body)>
<!ATTLIST html xmlns %URI; #FIXED 'http://www.w3.org/1999/xhtml'>
```

- in this example mondial also defines a namespace.
- note that the "dc" prefix cannot be changed since it is hardcoded in the monhtml.dtd.

```
declare namespace mon="https://www.semwebtech.org/mondial/10/";
declare namespace h="http://www.w3.org/1999/xhtml";
declare namespace dc="http://purl.org/dc/elements/1.1/";
/mon:mondial//mon:city[dc:author]//h:title
```
[Filename: XQuery/monhtml.xq]

- recall that the element namespace is *not* applied to the attributes:

```
declare namespace mon="https://www.semwebtech.org/mondial/10/";
declare namespace h="http://www.w3.org/1999/xhtml";
/mon:mondial//mon:country/id(@capital)/h:html/h:head/h:title
```
[Filename: XQuery/monhtml2.xq]

5.5 XPath: Conclusion

XPath without variables

- simple (and cheap) navigation language
- only following a "main path" for addressing sets of nodes (including semijoins)
- not "give all pairs of ..."
- selection/filtering: yes
- projection/reduction: no. Only complete nodes can be selected
- join/combination: no. Only semi-joins can be expressed in the conditions
- subqueries: inside the conditions as semijoins
- restructuring of the results: no

⇒ only a fragment of a query language for addressing nodes.
  - compared with SQL, XPath allows only for "SELECT *" and a unary "FROM" clause
  - XQL (Software AG, 1998/1999) for some time followed (as one of the predecessors of XPath) an approach to add join variables and constructs for projection and restructuring/grouping to the path language (cf. Slides 251 ff).

XPath (3.0) with "some"/"every" and Variables (cf. Slide 224)

- relational completeness (cf. SQL, relational calculus) *inside* filters
- still no generation of structures/joins on the result level.

Exercise

Consider the following query that yields the highest mountain in Africa:
(without variables, using semijoins)

```
doc('mondial.xml')//mountain[
  id(id(located/@country)/encompassed/@continent)/name='Africa'
  and
  not (number(elevation) <
   //mountain[
    id(id(located/@country)/encompassed/@continent)/name='Africa']/elevation)]
 /name
```

[Filename: XPath/highestmountain.xq]

Give the names of all mountains that are the highest ones on the continent where they are located.
(two properties of the same object (elevation, continent) must be compared independently → requires variable binding)

Importance of XPath in the XML-World

- adressing mechanism for nodes in XML documents
- navigation in the tree structure
- serves as base for different concepts:
  - XQuery
  - XSL/XSLT: stylesheets, transformation language
  - other query languages
  - XML Schema
  - [XPointer/XLink – rarely used]

Chapter 6
XML Query Languages

- XPath is not a query language:
  selects only sets of nodes
- additional functionality of query languages:
  - composition of tuples/structures from several nodes of a path
  - joins
  - dereferencing
    - via joins
    - via direct resolving of IDs (seen as values)
    - via dereferencing of ID attributes
  - aggregations
  - formatting and restructuring of results
  - operations on the order of nodes!

XML Query Languages

Collected experiences from SQL, OQL, OEM/WSL/MSL/Lorel, F-Logic and some more ...

- predecessors of XPath: XSL Patterns/XPointer/XQL (1998)
- XQL extended the early "basic form" to a query language
  - adding several constructs to the path expressions
  - increasingly complicated
  - still not sufficiently expressive
  - showed the limits and requirements
- XML-QL (1998): pattern-matching-based "extraction language"
  - not path-based, but XML-pattern/template-based binding of variables
  - semantics by a clause-construct
  - generation and structuring of the result by an XML pattern with variables

XML Query Languages (cont'd)

- Quilt (2000): SQL-style extension of XPath
  - binding of variables by XPath expressions
  - nested loops by "for"-clauses
  - additional conditions in a "where"-clause
  - structuring of the result by a "return"-clause
- XQuery (2001): "official" version of Quilt
  - W3C Working Draft XQuery first version from 15 February 2001
  - XQuery 1.0: W3C Recommendation since 23.1.2007
  - `http://www.w3.org/TR/xquery/`

6.1 XQL

XQL (XML Query Language; 1998) is a simple query language based on early constructs of XPath:

- all XPath expressions that can be expressed without the use of "*axis*::" (cf. Slide 199 - axes have only been added later).
- text() was a function,
- function applications have been expressed by "!" at the end of the path expression: //country/name!text()

Further querying functionality was integrated syntactically into the path expressions.

XQL: Boolean Operations and Set Operations

- q_1 or q_2
- q_1 and q_2
- q_1 union q_2, $q_1 \mid q_2$
- q_1 intersect q_2
- $q_1 \sim q_2$ (union, in case that both are non-empty)

XQL: Return Operators (Projections on the path)

Operators that output the node that is addressed at the given position:

??: the complete node is added to the output structure (including attributes and subelements)

?: only the element "hull" is added to the output

- country/city[@isCountryCap]/name

```
<name>Berlin</name>
<name>Rome</name>
```

- country?/city[@isCountryCap]/name

```
<country> <name>Berlin</name> </country>
<country> <name>Rome</name> </country>
```

- country?[@car_code?]/city[@isCountryCap]/name

```
<country car_code="D"> <name>Berlin</name> </country>
<country car_code="I"> <name>Rome</name> </country>
```

- country?[@car_code?]/city?[@isCountryCap]/name!text()

```
<country car_code="D"> <city>Berlin</city> </country>
<country car_code="I"> <city>Rome</city> </country>
```


XQL: Grouping

- copy a part of the original document structure:
  $path_1$ { $path_2$ }
- without grouping:
  country?[@car_code?]/city?/name!text()

```
<country car_code="D"> <city>Berlin</city> </country>
<country car_code="D"> <city>Hamburg</city> </country>
<country car_code="D"> <city>Munich</city> </country>
```

- with grouping:
  country?[@car_code?] {/city?/name!text()}

```
<country car_code="D">
  <city>Berlin</city>
  <city>Hamburg</city>
  <city>Munich</city>
</country>
```


XPath: Semijoins are Possible

- Semi-joins via subqueries in the condition:

$$\pi[A](r \ltimes s)\ , \quad A \subset \text{attr}(r)$$

Query: name of the continent where Germany is located:

```
/mondial/continent[@id =
                  /mondial/country[@car_code="D"]
                                /encompassed/@continent]
                  /name!text()
```

Problems

- full joins with join conditions not possible
- no restructuring/generation of answer structure

XQL: Joins

Asymmetric full joins expressed by *correlating variables* and "alternative"-construct: Filters may contain variable assignments of the form

[$\$var$:= $expr$]

that are then used in another condition

[$expr'$ = $\$var$]

//organization?[$s := @headq] {name?? | abbrev?? | member?? | //city[@id=$s]?? }

```
<organization>
 <name>European Union</name>
 <abbrev>EU</abbrev>
 <member type="member" country="GR F E A D I B L NL DK SF S IRL P GB"/>
 <member type="membership applicant"
    country="AL CZ H SK LV LT PL BG RO EW M CY"/>
 <city> <name>Brussels</name> ... </city>
</organization>
```

Equivalent:

//organization?[$s := @headq and name?? | abbrev??] {member?? | //city[@id=$s]?? }

XQL: Conclusion

- Ad-hoc-constructs (in different versions)
- insufficient restructuring functionality
  - tree structure of the input is in principle retained
- insufficient join functionality
- no clear semantics for the result format
- queries cannot be nested (cf. SQL, OQL: results are again relations); here is even no notion of a subquery
- one of the reasons: no clean variable concept (the variable concept of XPath 2.0 motivated by logical quantifiers is clean, but also leads to complex expressions)
- implemented and used up to 2002 in the "Tamino" system of Software AG.

6.2 Query Languages: Requirements

Requirements on XML Query Languages [David Maier and W3C XML Query Requirements]

- closedness: output must be XML
- orthogonality/composability: everywhere where a set of XML elements is required, also a query is allowed.
- clean definition and nesting of operations: selection, extraction/projection, restructuring, combination/join, fusion of elements,
- applicable without presence of schema, but can use a schema,
- retaining the order of nodes,
- [queries should have an XML representation, especially, XML documents should be able to contain embedded queries]
- resolving of XPointer and XLink [never completed since XLink is rarely used]
- formal semantics: deriving structure of the result, equivalence and query containment

6.3 XML-QL

- `http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/`
- simple, pattern-based XML query language:

  WHERE *xml-pattern* IN *url* CONSTRUCT *result*
- usage of variable bindings:
  xml-pattern contains variables that can be used in *result*,
- declarative,
- "relationally complete", i.e., joins can be expressed.

Example:

```
WHERE <country car_code=$id>
        <name>$name</name>
      </country>
IN "http://www.../mondial.xml"
CONSTRUCT <country car_code=$id name=$name/>
```


XML-QL: Joins

Joins are expressed as a list of
WHERE ($expr_1$ IN doc_1, ... , $expr_n$ IN doc_n)-clauses:

- equijoin inside a document:

```
WHERE
  <country car_code=$c></country>
  IN mondial.xml,
  <organization abbrev=$org>
    <members type=$type country=$c/>
  </organization>
  IN mondial.xml,
CONSTRUCT ...
```

XML-QL: Joins

- Joins that combine multiple documents:

```
WHERE
  <city name=$name1>
  IN http://www.../europe.xml,
  <city name=$name2>
  IN http://www.../america.xml,
  <connection from=$name1 to=$name2>
  IN http://www.../lufthansa.xml
CONSTRUCT
 <connection>
   <from continent="europe" city=$name1/>
   <to continent="america" city=$name2/>
 </connection>
```


XML-QL: Nested Queries

WHERE *xml-pattern* IN *url* CONSTRUCT *result*

- *result* can contain nested WHERE ... IN ... CONSTRUCT statements.

Further Functionality

- tag-variables: WHERE <$tag> ... </>
- regular path expressions: instead of XPath's "//", `<*> ... </>` is used.

Data Model

- Graph-based: XML-tree with IDREF edges:

```
WHERE
  <country car_code=$cc>
    <capital><name>$name</name><population>$pop</population></capital>
  </country>
IN ... CONSTRUCT ...
```


XML-QL: Conclusion

- clause-based high-level language
- selection and construction pattern-based (by binding variables in the patterns; similar to Logic Programming)
- join conditions: not in a WHERE clause, but implicitly expressed by *join variables* (like in Logic Programming)
- graph data model; no difference between tree edges and reference edges
- has been implemented
- used in different projects (e.g. MIX – Mediation in XML; UC San Diego 1999/2000)
  - allows for access and combination of different HTML/XML-sources in a query.

6.4 Recall basic concepts from SQL, OQL etc.

- set-oriented (sets of tuples or objects) language
- implicit iteration over sets:

  SELECT ... FROM *relation-or-extent* c
- variable c ranges over *data items*
- join: use several such variables and correlate them
- WHERE and SELECT part: use these variables

Similar constructs for XML?

- variables range over sets of nodes
- ... sets of nodes can be addressed by XPath
- straightforward and intuitive:

  for $c in //country
  where $c/population > 1000000
  return $c/name/text()

6.5 XQuery

- First proposed as "Quilt" by IBM, Software AG, INRIA at WebDB2000-Workshop
- a Quilt is a "Flickenteppich" ...
- Structure similar to SQL/OQL: clause-based, functional language (arbitrary nesting of FLWR expressions allowed),
- Use of variables similar to SQL/OQL,
- based upon XPath (previously XQL/XSL Patterns) in the selection part and upon XML-QL (XML patterns) in the construction part:
- For Let Where Return-clauses

  for *variable* in *xpath-expr* // from XQL/XPath and XML-QL
  let *additional_variable* := *xpath-expr*
  where *condition*
  return *xml-expr* // from XML-QL
- has been moved into W3C's "XML Query" in 2001 with only small changes.
- Remark: XQuery is case-sensitive.
  ALL KEYWORDS MUST BE WRITTEN WITH non-capital LETTERS!

XQuery: Example

- for-clause: binding of variables (cf. SQL: FROM)
- where-clause: evaluation of conditions
- return-clause: generation of the result (cf. SQL: SELECT)

```
<result>
 { for $c in /mondial/country
   where $c/@area > 500000
   return <bigcountry>
            { $c/name }
            <area> { string($c/@area) } </area>
          </bigcountry>
 } </result>
```

[Filename: XQuery/first-example.xq]

generates

<result><bigcountry><name>France</name><area>547030</area></bigcountry>
<bigcountry><name>Spain</name><area>504750</area></bigcountry>
:
</result>

Some notes on the Mondial Database

- Cities can have more than one name ("München","Munich",. . .).
  (this is important for data integration when combining Mondial with external sources)
  Access the first one with //city[name='Munich']/name[1].
- Countries, provinces and cities usually have several
  <population year="...">*value*</population>
  subelements.
  These subelements are in temporal order, the last one ist the newest.
  Access the last one with //country[name='Germany']/population[last()].
  Aside: note that this is much more longwinded with SQL:

```
select country, year, population
from countrypops p1
where year = ( select max(year)
               from countrypops p2
               where p2.country=p1.country )
```


Aside: Tools – XQuery as a Database and Web Query Language

XML Databases

- local repository of XML documents
- adding documents to the Database
- access only against locally stored documents
- presence of access paths like indexes etc
- manipulation of documents

Queries against the Web

- querying the whole Web
- documents not locally stored; only on-the-fly-indexing possible
- access to remote documents by their url

path expressions: doc('*filename or url*')//country/name

6.5.1 XQuery: the basis

- for-clause: defines nested loops where each of the variables runs over the set of selected values
- variables in XPath expressions: bound in for/let (or by surrounding statements), they are used as starting points for paths and in conditions
- joins:
  - multiple variables in a for-clause:
    for $\$var_1$ in $doc_1/path_1, \ldots, \$var_n$ in $doc_n/path_n$
  - correlated definition of the variables in the for-clause:
    for $\$var_1$ in $doc_1/path_1$, $\$var_2$ in $\$var_1/path_2, \ldots$
- let-clause for definition of "constants":
  let *$var := expr*
  binds *$var* to the *whole result* of *expr* (in general, a sequence of nodes).
- nested/iterated for-let-for-let-clauses allowed
- where clause: conditions on the variables defined so far
- generation of nested structures:
  the return-clause may contain further FLWR-clauses (which can use variables from the outer clause).

Simplest XQuery Queries: XPath

- Each XPath query is also an XQuery query
  result: a sequence of nodes or literal values

```
doc('mondial.xml')//country/name
```

Note: different behavior when returning attribute nodes!

```
doc('mondial.xml')//country/@area
```

XQuery: For-clause

for $*var* = *xpath-expr*

- iterates over the result of *xpath-expr*

```
for $x in /mondial//country/name
return $x
```

[Filename: XQuery/for-example.xq]

XQuery: Return-clause

Output of all statements must be XML.

- simple case: content of a variable

```
for $x in /mondial//country/name
return $x
```

- and generation of structured results (cf. OQL)

Generation of Structures

- literal XML
- computed element- and attribute constructors (later)

Use of Computed Values/Structures

- enclosed between "{" . . . "}"
- evaluation of variables and XPath expressions
- nested FLWR-clauses

Return-Clause: Construction of Result Elements

- literal XML, values of variables and results of XPath expressions

```
<html><table>
<tr><th>Name</th><th>Area</th><th>Population</th></tr>
{ for $c in /mondial/country
  return
    <tr><td>{$c/name/text()}</td>
        <td>{string($c/@area)}</td>
        <td>{$c/population[last()]/text()}</td>
    </tr>
}
</table></html>
```

[Filename: XQuery/table-example.xq]

returns one table row for each country.

XQuery: For-Clause

Multiple Variables in a For-Clause

- cartesian product
  (cf. FROM-clause in SQL)

```
for $c in /mondial//country,
    $o in /mondial//organization
where $c/@capital = $o/@headq
return
  <answer>
     <country>{$c/name[1]/text()}</country>
     <organization>{$o/name/text()}</organization>
  </answer>
```

[Filename: XQuery/cartesian-example.xq]

- compare where clause with equivalent
  where $c/id(@capital) is $o/id(@headq)
  on node level ("=" would also be correct here, taking the string value of the nodes).

XQuery: For-Clause

Multiple Variables in a For-Clause

- "correlated" Join
  (cf. FROM-clause in Schema-SQL and OQL)
- subset of the cartesian product

```
for $c in /mondial/country,
    $p in $c/province
return
  <answer>
     <country>{$c/name/text()}</country>
     <prov>{$p/name/text()}</prov>
  </answer>
```

[Filename: XQuery/correlated-join-example.xq]

Return-Clause with Nested FLWR-Clause

- inner query used in the outer return-clause (cf. OQL)

```
for $c in /mondial/country
where $c/province
return
  <answer>
     {$c/name}
     { for $p in $c/province
       return
       <prov>{$p/name/text()}</prov>
     }
  </answer>
```

[Filename: XQuery/nested-flwr-example.xq]

generates for each country that has provinces an <answer> element that contains a <name> element and a sequence of <prov> elements.

Let-Clause

let $*var* := *xpath-expr*

- does not iterate over the result of *xpath-expr*
- but binds the complete result of *xpath-expr* as sequence of nodes to the variable:

```
for $c in /mondial/country
let $cities := $c//city/name[1]      (: first name of each city :)
return
  <country>
     {$c/name}
     {$cities}
  </country>
```

[Filename: XQuery/let-example.xq]

- useful for keeping intermediate results for reuse (often missed in SQL)
  - Note: if a fragment with idref-attributes is created, these cannot be dereferenced during later use (the created fragment is not part of the "main" tree)

WHERE-CLAUSE: CONDITIONS

Similar to XPath's conditions (same predicates etc):

- logical "and" and "or"
- "not(...)" as a boolean function
- Comparisons: "is" for node identity, "<<" and ">>" for document order, "follows" and "precedes"
- Quantifiers: where some|every $*var* in *expr* satisfies *condition*

```
for $c in /mondial/country
where some $city in $c//city satisfies $city/population[last()] > 1000000
return $c/name
```

```
for $c in /mondial/country
where every $city in $c//city satisfies $city/population[last()] > 1000000
return $c/name
```

[Filenames: XQuery/some-example.xq and every-example.xq]

USE CASE: JOIN BETWEEN DIFFERENT DOCUMENTS

- doc(...) function to access files (local or from the Web)
- here: join by a subquery

```
for $c in doc(concat('http://www.dbis.informatik.uni-goettingen.de',
                     '/Mondial/mondial-europe.xml'))/mondial/country
where some $l in doc('hamlet.xml')//LINE
      satisfies contains($l, $c/name[1])
return
  <country>
    {$c/name}
  </country>
```

[Filename: XQuery/join-web-documents.xq]

- "contains" requires (node,node) as arguments, does not accept (node, node+). Exercise: compare not only with name[1] but with all of them.

ATTRIBUTES IN THE RETURN-CLAUSE

- note that expressions the form "@bla" return *attribute nodes* - these are (AttrName, value)-pairs:

```
<result>
  {//country[name='Germany']/@car_code}
</result>
```

generates <result car_code="D"/>.

- attribute nodes are always added to the surrounding element.
- if only their value is needed, apply string().

```
for $c in /mondial/country
return
  <country>
    {$c/@area}
    {string($c/@car_code)}
  </country>
```

Result:

<country area="28750">AL</country>

<country area="131940">GR</country>

:

[Filename: XQuery/attribute-example.xq]

ORDER OF RESULT SET

XPath: the result is *always* returned in *document order*:

- purely navigational access:

  `//country/city/name`

- even when a backward axis is used during navigation, the nodes are enumerated in document order:

  `//country[name='Germany']/province[last()]/preceding-sibling::*/name`

  (backward axis is only relevant for context functions in immediate conditions)

- or when id-referencing is used:

  `id(//organization/@headq)/name`

  (note: cities are *not* ordered according to the order of the organizations!)

XQuery: result set is ordered according to for-clause:

```
for $c in //organization
return id($c/@headq)/name
```

let-clause: binds the result set according to the respective order.

Ordering

- order by: *expr* order by *expr* [ascending|descending]

```
for $c in //country
order by $c/name[1]
return $c/name[1]
```

[Filename: XQuery/orderby-example.xq]

- note that the interpreter must be told whether the values should be regarded as numbers or as strings (default: alphanumerical) (see below).
- Note: ordering cannot only be used for finally presenting the output (like in SQL), but also to create an ordered sequence to bind it with “let” and do something with it:

```
let $ordered := (for $c in //country
                 order by number($c/@area) descending
                 return $c/name[1] )
return $ordered[position() = (1 to 4)]
```

[Filename: XQuery/bigcountries.xq]

Grouping (by “let”) and Aggregation

- aggregate functions over result sets (avg, sum, min, max, count).
- bind group-by variable(s) with “for”-clause,
- assign group with “let” (dependent on the current value in the for-clause) to a variable,
- apply aggregate functions to the nodesets bound by the let.

```
for $c in /mondial/country
let $cities := $c//city
where sum($cities/population[last()]) > 10000000
return
  <answer>
     {$c/name}
     {sum($cities/population[last()])}
  </answer>
```

[Filename: XQuery/aggr-1-example.xq]

Aggregation

- aggregation over result of a FLWR subquery
- bind (single) intermediate result by “let”

```
for $c in /mondial/country
let $maxpop := max( for $citypop in $c//city/population[last()]/text()
                    return $citypop )
return
  <answer>
     {$c/name}
     {$maxpop}
  </answer>
```

[Filename: XQuery/aggr-2-example.xq]

Conditional Evaluation and Alternatives

- if-then: alternative choice of subelements
  if (*expr*) then *expr* else *expr*

```
for $c in /mondial/country
return
  <country>
    {$c/name}
    {if ($c/province) then $c/province/city else $c/city}
  </country>
```

[Filename: XQuery/if-else-example.xq]

- same as SQL’s CASE ... WHEN ...
- since XQuery 3.0: “switch/case+/default” expression.

6.5.2 XQuery: Further Functionality

Computed Element- and Attribute Names

- explicit constructors
  - element *expr attrs-and-content*
    the evaluation of *expr* yields the name of the element, the result of *attrs-and-content* is then inserted as attributes and content
    Note: content is a node sequence, separated by ","
  - attribute *expr expr-value*
    the evaluation of *expr* yields the name of the attribute, *expr-value* yields its value.

```
for $c in doc('mondial.xml')//country
return
element { $c/@car_code }
        { attribute {$c/encompassed[1]/@continent} {"yes"},
          $c/name
        }
```

```
<B europe="yes">
  <name>Belgium</name>
</B>
```

[Filename: XQuery/computed-constructors-example.xq]

... for *one* continent for each country

Computed Element- and Attribute Names: Another Example

- the same, iterating over *all* "encompassed" elements of a country:

```
for $c in doc('mondial.xml')//country
order by count($c/encompassed) descending
return
element { $c/@car_code }
        { for $e in $c/encompassed
          return attribute {string($e/@continent)} {$e/@percentage},
          $c/name
        }
```

[Filename: XQuery/computed-constructors-example2.xq]

- returns e.g.

```
<R europe="25" asia="75"> <name>Russia</name> </R>
```

- note: `text { $car_code }` can be used to create text nodes, e.g. from a string bound to a variable (when operating on sequences, a sequence of text nodes is different from a sequence of strings. E.g., union and except are only applicable on sequences of nodes).

Handling Duplicates

- recall from XPath: results (and intermediate results) of XPath expressions are *node sets* in document order
  ⇒ for $x in *xpath-expr*, let $y := *xpath-expr*
  always results in a set (i.e., duplicates removed)
- recall Slide 219 for removal of duplicate *values*: distinct-values(...)

  ```
  distinct-values(doc('...')//SPEAKER)
  ```

  How many speeches has each of the speakers in "Hamlet"?

```
for $a in distinct-values(doc('/db/xmlcourse/hamlet.xml')//SPEAKER)
let $n := count(//SPEECH[SPEAKER = $a])
order by $n descending
return
  <answer>
    {$a}
    {$n}
  </answer>
```

[Filename: distinct-values.xq]

- takes only the string values (⇒ no further navigation applicable)

Handling Duplicates in XQuery(cont'd)

- FLWR expressions (e.g., for $c in ... return $c) do *not* eliminate duplicates automatically
- for $o in //organization return $o/id(@headq)
  returns duplicates
- distinct-values(for $o in //organization return $o/id(@headq))
  returns only the string values
- so it must be done programmatically (often, specific for the given problem: iterate over the target set and do the test in a subquery) – cf. SQL:

  ```
  select * from <table-of-entity-tuples> where <condition>
  ```

- or by a generic function – see Slide 300

OPERATING WITH SEQUENCES

Comparisons are existentially quantified and instance-based: if one operand is a sequence, each value is compared, and if one value satisfies the condition, the whole filter is satisfied:

- ... as we have seen for XPath: country[.//city/name = "Cordoba"]/name
  country[.//city/population > 1000000]/name
- the same holds when comparing with a sequence bound to a variable by a "let"-view:

```
let $europnames := //country[encompassed/@continent="europe"]/name
for $country in //country
where not ($country/name = $europnames)
return $country/name
```

  [Filename: XQuery/seq-comparison-example.xq]

  outputs all names of non-european countries.
- selection from let-sequences is also instance-based:

```
let $europcountries := //country[encompassed/@continent="europe"]
return $europcountries[@area>300000]/name
```

  [Filename: XQuery/seq-selection-example.xq]

Operations on Nodes and Node Sequences

- "=" compares the string-values of nodes, not "correct" if node identity has to be checked
- "is" compares node identity:

```
for $c in //country,
    $o in //organization
where $c/id(@capital) is $o/id(@headq)
return <pair country='{$c/@car_code}' org='{$o/abbrev}'/>
```

[Filename: XQuery/node-comparison.xq]

- "is" is not allowed for node sequences:

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where $hq is $caps       (: not allowed :)
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-1.xq]

⇒ Error: "A sequence of more than one item is not allowed as the second operand of "is" "

Operations on Nodes and Node Sequences

- explicit iteration via some:

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where some $cap in $caps satisfies $hq is $cap
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-2.xq]

- index-of(*sequence(atomic type)*,*item*) → integer

```
let $caps := //country/id(@capital)
for $hq in //organization/id(@headq)
where index-of($caps,$hq)      (: checks if $caps contains $hq :)
return $hq/name
```

[Filename: XQuery/nodes-comparison-example-3.xq]

- ... or "where $caps intersect $hq", or even shorter:

```
let $caps := //country/id(@capital)
let $hqs := //organization/id(@headq)
return ($hqs intersect $caps)/name
```

[File: XQuery/nodes-comparison-example-4.xq]

- "intersect", "union", and "except" are only allowed on sequences of *nodes*, not on sequences of literals (like e.g. strings).

GROUP BY (SINCE XQUERY 3.0)

- At first sight like SQL ...
- recall SQL: only group-by variables and aggregate function applications over the group can be selected.
- group by variable: uses/binds the *string value* of the path expression!
- XQuery: all non-group-variables are bound to the sequence of all values of that variable (can be used later for aggregate functions).

```
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergroup country="{$cc}">
  { $c/name[1] }
  <sumpop> {sum($c/population[last()])} </sumpop>
</answergroup>
```

[Filename: XQuery/group-by-simple.xq]

Group By (cont'd) 3.0

consider again

```
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergroup country="{$cc}">
  { $c/name[1] }
  <sumpop> {sum($c/population[last()])} </sumpop>
</answergroup>
```

[Filename: XQuery/group-by-simple.xq]

- one <answergroup> per country (in arbitrary order if the countries),
- note: $c is *implicitly* reassigned after the group-by, holding the sequence of all city elements in this country;
- $c/name[1] results in the sequence of all their (first) name subelements,
- sumpop holds then the sum of their newest population counts.
- similar (and preferable?): for $c in //country let $cities := $c//city ... (cf. Slide 282)

Group By – Danger!

- Recall: all non-group-variables are bound to the sequence of all values of that variable (can be used later for aggregate functions).

```
let $x := 3, $y := ("a","b")
for $c in //city
group by $cc := $c/ancestor::country/name[1]
return
<answergrp country="{$cc}"> {$x} {$y}</answergrp>
```

[Filename: XQuery/group-by-ugly.xq]

- after the group-by, $x and $y are a *sequences* that consist of as many 3's and so many a-b-a-b-... as the country has cities!
- if a non-group-by-variable is not used later, the optimizer will (hopefully) never instantiate its sequence, but
- if such a non-group-by-variable $x is used later, one should consider to access it by $x[1],
- if such a non-group-by-variable $y contains a sequence before, there is no simple way to "keep it" (the duplicated sequence is flattened)!

⇒ in general, prefer to use
for *groupingvar* in ... let *groupvar* := *correlatedpath* ... (cf. Slide 282)

Group By – a statistical use case

- Sometimes,
  for *groupingvar* in ... let *groupvar* := *correlatedpath* ... is not more intuitive:

```
for $c in //city[population]
group by $pp := round($c/population[last()] div 10000)
where count($c) > 0     (: another where, not "having" :)
order by $pp descending
return
<answergrp pp="{$pp * 10000} to {$pp * 10000 + 10000}" count="{count($c)}">
  {$c/name[1]} </answergrp>
```

[Filename: XQuery/group-by-popgrps.xq]

```
let $allpps := distinct-values(
     for $p in //city/population[last()] return round(/$p div 10000))
for $pp in $allpps
let $c := //city[round(population[last()] div 10000) = $pp]
where count($c) > 0
order by $pp descending
return
<answergrp pp="{$pp * 10000} to {$pp * 10000 + 10000}" count="{count($c)}">
  {$c/name[1]} </answergrp>
```

[Filename: XQuery/group-by-popgrps2.xq]

FUNCTIONS AND OPERATORS

XPath and XQuery Standard Operators

- Recall Slide 213 for string() and name(), and Slide 210 for id().
- Mathematical functions have been added as builtins in XQuery 3.0:
  - `declare namespace math="http://www.w3.org/2005/xpath-functions/math";`
  - use math:sqrt, math:sin, ...
- See "W3C XML Query Functions and Operators 3.0" for predefined functions:
  `https://www.w3.org/TR/xpath-functions-30/`
  - functions/operators using strings (cf. Slide 217);
    for splitting and re-constructing IDREFS attributes, especially,
    `tokenize`(*string-to-be-tokenized*, *separator*) → xs:string* and
    `string-join`(xs:string*, *separator*) → xs:string are useful.
  - functions/operators using numbers,
  - functions/operators using date and time (cf. Slide 309),
  - functions/operators on sequences of nodes.

6.5.3 User-Defined Functions/Functional Programming

- definition and reuse of functions simplifies daily work with XML in general,
- up to a full-fledged functional programming language (like Lisp or Haskell)

Syntax

- User defined functions are declared in the prolog:
  declare function *func_name* ([*$var*$_1$, ..., *$var*$_n$]) [as *returnType*]
  {
  expr that uses $var*$_1$, ..., *$var$_n$
  }
- Parameters: *$var*$_i$ [as *paramType*], default for parameter and return types is item()* (i.e. a sequence of nodes, literals etc.),
- Any sequence type may be used for *paramType* and *returnType* (cf. XML Schema),
- Any XQuery expression is allowed in the function body.

User-Defined Functions: Examples

- A function computing the population density for a given country:

```
declare function local:density ($name as xs:string) as element(density)?
{                                          (: returns zero or one element :)
  for $c in doc('mondial.xml')//country[name=$name]
  let $density :=
      if ($c/@area > 0) then $c/population[last()] div $c/@area else 0
  return <density>{$density}</density>
};
local:density('Germany')
```

[Filename: XQuery/function-density.xq]

- Example for a recursive function:

```
declare function local:depth($e as node()) as xs:integer
{
   if (fn:empty($e/*)) then 1
   else fn:max(for $c in $e/* return local:depth($c)) + 1
};
local:depth(/)
```

[Filename: XQuery/function-depth.xq]

User-Defined Functions: Example

Ignoring the FLWR, XQuery can even be used as a common functional language:

- every (arithmetic + if) expression is a valid XQuery expression
- example: the "factorial" function (german: "Fakultät"): $n! := \begin{cases} 1 & \text{if } n = 1 \\ n \cdot (n-1)! & \text{if } n > 1 \end{cases}$

```
(: command line: call  saxonXQ factorial.xq x=5
                   or  saxonXQ ?x=5 factorial.xq   :)
declare variable $x external;
declare function local:factorial($n as xs:integer) as xs:integer
{  if ($n=1) then 1
   else $n * local:factorial($n - 1)
};
local:factorial($x)
```

[Filename: XQuery/factorial.xq]

- saxon from command line: for strings as parameters use
  `saxonXQ ?foo='xs:string("blabla")' anyprogram.xq`

User-Defined Function: A Useful Example

Remove duplicates from a node set (taken from the example Section from W3C XPath/XQuery Functions and Operators):

```
declare function distinct-nodes-stable ($arg as node()*) as node()*
{
   for $a at $apos in $arg
   let $before_a := fn:subsequence($arg, 1, $apos - 1)
   where every $ba in $before_a satisfies not($ba is $a)
   return $a
}
```

XQuery Exception Handling (since XQuery 3.0/2014)

- like in Java: try-catch, embedded as functional programming syntax.
- example: catch any exception and report it (as result!)

```
declare function local:trycatchtest($x as xs:integer) {
  try { 1000 div $x }
  catch * {
    $err:code, $err:description, $err:value,
    if ($err:module) then (" module: " || $err:module) else (),
    "(line " || $err:line-number || ", col " || $err:column-number || ")"
  }};
local:trycatchtest(0)
```

[Filename: XQuery/trycatch.xq]

- example: catch div by zero (which is exception err:FOAR0001)

```
declare function local:catchdivby0($x as xs:integer) {
  try { 1000 div $x }
  catch err:FOAR0001 { 12345 }};
local:catchdivby0(0)
```

[Filename: XQuery/catchdivby0.xq]

- List of all error codes: <https://www.w3.org/2005/xqt-errors/>

Web Access requires try-catch

- many Web pages are not available or not valid XML

⇒ raises exceptions (different types)

```
declare function local:getwebpage($x as xs:string) {
  try { doc("http://www." || $x || ".de") }
  catch * { " not found or invalid as XML: " || $x}
};
for $n in //province[name='Niedersachsen']//city/name
return local:getwebpage($n)
```

[Filename: XQuery/catchwebaccess.xq]

Throwing an exception: fn:error

- call fn:error;
- this will not create a result, but throw an exception:

```
declare function local:throwexc($x as xs:integer) {
  try { if ($x = 0)
        then fn:error(fn:QName('http://www.w3.org/2005/xqt-errors',
                               'err:FOER0000'))
        else if ($x = 1)
        then fn:error(fn:QName('http://www.semwebtech.org/foo/', 'err:foo'),
                      "I don't like that!")
        else $x
      }
  catch * {
    $err:code, $err:description, $err:value,
    if ($err:module) then (" module: " || $err:module) else (),
    "(line " || $err:line-number || ", col " || $err:column-number || ")"
  }};
local:throwexc(0)
```

[Filename: XQuery/throw.xq]

Querying for Context Positions by position()

- position() is always evaluated *for a given node* wrt. *a given context (sequence)*
- in Mondial, languages, religions, and ethnic groups in each country are ordered by percentage.
  What is the ranking of the french language in each country?

```
let $lgs := //country[@car_code='CH']/language
return
<result allpositions = '{$lgs/position()}'
  everypos='{for $x in $lgs return $x/position()}'
  frenchpos1 = '{$lgs[text()='French']/position()}'
  frenchpos2 = '{for $x in $lgs return if ($x/text()="French")
                   then $x/position() else ()}'            />
```

[Filename: XQuery/queryposwrong.xq]

- allpositions is ’1 2 3 4’, everypos is ’1 1 1 1’, the others are both ’1’.

⇒ the “context” is always only the individual node bound to $x, not the sequence bound to $l

- see next slide for the workaround design in XQuery.

Querying for context positions by position() - “Positional variables”

- The XQuery language introduced the so-called “positional variables”:

  for $x at $i in *expr*

  $i is then bound to the position of $x in the context given by *expr*:

```
for $c in //country[language/text()='French']
let $lgs := $c/language
return <result country='{$c/@car_code}'
               rank= '{ for $x at $pos in $lgs
                        return if ($x/text()="French") then $pos else ()}'/>
```

[Filename: XQuery/frenchpos.xq]

- As shown on Slide 393, this is solved better in XSLT.
- an XQuery workaround is shown on the next slide.

Finding in another Sequence — the index-of() Function

- index-of(*sequence of atomic type*,*value*) → integer
- for querying the position of a *(atomic!) value* in a sequence of *(atomic!) values*,
- the sequence can be obtained directly representing the “context”, or can be computed in any other way.

```
for $c in //country[language/text()='French']
let $l := $c/language
return <result country='{$c/@car_code}'
               rank= '{ index-of($l/text(), "French") }'/>
```

[Filename: XQuery/frenchpos2.xq]

Functional Programming (incomplete list)

- “simple” map operator (“!”): syntactic sugar for
  sequence ! *expr* → for $x in *sequence* return *expr*($x)
  //country[name='Germany']//city ! concat(name[1], " has pop ", population[last()])
- handling sequences for rewriting iteration into recursion: fn:head, fn:tail
  e.g. sum(*sequence*) = head(sequence) + sum(tail(sequence)),

```
declare function local:sum($x as xs:integer*) as xs:integer {
    if (not(fn:empty($x))) then fn:head($x) + local:sum(fn:tail($x)) else 0 };
local:sum((1,2,3,4,5))
```

[Filename: XQuery/sum.xq]

- Note: function references and inline functions require the saxon EE version
  - Lisp-style: fold-left, fold-right
  - Inline functions, Dynamic function calls

```
let $f := function ($x) { 2 * $x },
for $c in (1 to 10)
return $f($c)
```

[Filename: XQuery/inlinefct.xq]

```
declare namespace math="http://www.w3.org/2005/xpath-functions/math";
let $f := math:sqrt#1      (: $f unary function math:sqrt :)
for $c in (1 to 10) return $f($c)
```

[Filename: XQuery/dynfctcall.xq]

6.5.4 Miscellaneous

Exercises

... see Web.

Exercise 6.1
Determine the lowest mountain that is the highest mountain of the continent where it is located.

Solve the problem for the relational Mondial-DB in SQL, and for XML in XQuery. □

Specialized Datatypes for Time etc.

The datatypes specified by XML Schema are used in XPath/XQuery (and in XSLT)

- Syntax: constructors like xs:dateTime('*syntactical representation*')
- syntactical representations:
  - xs:dateTime: *yyyy-mm-dd* T*hh:mm:ss*[.*xx*][{+|-}*hh:mm*]
  - xs:date: *yyyy-mm-dd* and xs:time: *hh:mm:ss* [{+|-}*hh:mm*]
  - xs:duration: P [*n* Y][*n* M][*n* D][T[*n* H][*n* M][*n* [.*n*] S]], where *n* can be any natural number
  - xs:dayTimeDuration, xs:yearMonthDuration: restrictions of xs:duration.

```
let $x := xs:dateTime('2009-08-01T13:51:20.99'),
    $y := xs:date('2008-12-31'),
    $t1 := xs:time('12:50:00+01:00'),      (: timezone +1 = Frankfurt :)
    $t2 := xs:time('15:35:00.50-05:00')   (: timezone -5 = New York  :)
return <e const="{$x}" diff="{$t2 - $t1}" d="{$y + xs:yearMonthDuration("P1Y2M")}"
 sum1="{xs:time("11:12:00") + xs:dayTimeDuration("PT1H75M")}"
 sum2="{xs:dateTime("2009-01-10T11:12:00") + xs:dayTimeDuration("P3DT26H40M")}"/>
```

[Filename: XQuery/datetime-test.xq]

- resulting diff = "PT8H45M0.5S" (an xs:duration), sum1 = "13:27:00" (an xs:time), sum2= "2009-01-14T13:52:00" (an xs:dateTime), d= "2010-02-28" (an xs:date)

Actual Usage of xs:date etc.

... is often simple (recall that any text contents in an XML file is PCDATA or CDATA):

```
<country car_code="B">
   <indep_date>1830-10-04</indep_date>
</country>
```

```
for $c in //country[indep_date < '1900-01-01']
return concat($c/name, $c/indep_date)
```

[Filename: XQuery/simple-date-example.xq]

note: explicit `[indep_date < xs:date('1900-01-01')]` would be more explicit, but string comparison has the same result for the standard textual representation ('0' $<$ '1').

- functions current-date() and current-time(),
- extraction functions like days-from-duration, minutes-from-dateTime etc. (see W3C XQuery and XPath Functions and Operators)

```
for $c in //country[indep_date]
order by xs:date($c/indep_date) ascending
return
  concat($c/name, " is ", days-from-duration(current-date() - xs:date($c/indep_date)),
         " days old.")
```

[Filename: XQuery/current-date-example.xq]

Durations and Time Differences

- difference between two times is a duration
- add $x to a time: $x must be a duration (SQL: DATE + number means days)
  - Note: in XPath's built-in functions there is also nothing like SQL's ADD_MONTHS(date, number) function.
- duration serialization format: like "PnDTnHnMdecS", n positive integers, m positive decimal, any n, mX might be missing, dec might be a decimal with a decimal point. Negative durations: "-P..."
- functions adjust-dateTime-to-timezone($dt, $timezone) and adjust-time-to-timezone($dt, $timezone);
  $timezone must hold a dayTimeDuration!
  not allowed: adjust-time-to-timezone(xs:time("12:00:00"), 1)
  adjust-time-to-timezone(xs:time("12:00:00"), xs:dayTimeDuration("PT1H30M")) $\rightarrow$ 12:00:00+01:30
- Mondial has airport timezones (gmtOffset elements) as numbers, which might be negative, and even non-integer:

```
<airport iatacode="YYT" city="cty-Canada-Saint-Johns" country="CDN">
  <name>St Johns Intl</name>  <latitude>47.61861</latitude> <longitude>-52.751945</longitu
  <gmtOffset>-3.5</gmtOffset> <located_on island="island-Newfoundland"/>  </airport>
```


Durations and Time Differences: example

```
let $flights :=  (   (: all times in localtime :)
   <flight nr="LH123" from="FRA" to="LIS" departure="12:30:00" arrival="14:40:00"/>,
   <flight nr="LH124" from="FRA" to="JFK" departure="12:40:00" arrival="15:10:00"/>,
   <flight nr="LH125" from="FRA" to="YYT" departure="12:50:00" arrival="14:20:00"/> )
for $fl in $flights[position() = (1 to 2)]     (: not for YYT: offset 3.5h :)
let $dur1 := xs:time($fl/@arrival) - xs:time($fl/@departure),
    $deptoffset := //airport[@iatacode = $fl/@from]/gmtOffset,
    $arroffset := //airport[@iatacode = $fl/@to]/gmtOffset,
    $deptOffsetDur := if ($deptoffset >= 0) then
            xs:dayTimeDuration(concat('PT', $deptoffset, 'H'))
       else xs:dayTimeDuration(concat('-PT', abs($deptoffset), 'H')),
    $arrOffsetDur :=   if ($arroffset >= 0) then
            xs:dayTimeDuration(concat('P0DT', $arroffset, 'H'))
       else xs:dayTimeDuration(concat('-P0DT', abs($arroffset), 'H')),
    $depteff := adjust-time-to-timezone(xs:time($fl/@departure),$deptOffsetDur),
    $arreff := adjust-time-to-timezone(xs:time($fl/@arrival),$arrOffsetDur),
    $dureff := $arreff - $depteff (: XQuery knows that these are xs:time :)
return
<res dur1="{$dur1}" dDur="{$deptOffsetDur}" aDur="{$arrOffsetDur}"
     depteff="{$depteff}" arreff="{$arreff}" dureff="{$dureff}"/>
```

[Filename: XQuery/flights-duration-example1.xq]

Converting numbers to durations

- there seems to be no built-in function to turn a number into a duration.
  (number can mean hours/minutes/seconds)
  So, let's write one.

```
declare function local:numToDur($x as xs:decimal, $unit as xs:string)
   as xs:duration {
   xs:dayTimeDuration(local:ntd($x, $unit, false())) };

declare function local:ntd($x as xs:decimal, $unit as xs:string,
   $rec as xs:boolean?) as xs:string {
 let $sign := if($x<0) then "-" else "",
     $pre := if (not($rec)) then "P" else ""
 return if ($unit = "H") then
   concat($sign, $pre, "T", floor(abs($x)),
                       "H", local:ntd((abs($x)-floor(abs($x)))*60,"M", true()))
   else if ($unit = "M") then
     concat($sign, $pre,      floor(abs($x)),
                       "M", local:ntd((abs($x)-floor(abs($x)))*60,"S", true()))
     else if ($unit = "S") then concat(abs($x), "S") else ()
};
```



```
let $flights := (
   <flight nr="LH123" from="FRA" to="LIS" departure="12:30:00" arrival="14:40:00"/>,
   <flight nr="LH124" from="FRA" to="JFK" departure="12:40:00" arrival="15:10:00"/>,
   <flight nr="LH125" from="FRA" to="YYT" departure="12:50:00" arrival="14:20:00"/> )
   (: all times in localtime :)
for $fl in $flights
let $deptOffsetDur := local:numToDur(//airport[@iatacode = $fl/@from]/gmtOffset,"H"),
    $arrOffsetDur := local:numToDur(//airport[@iatacode = $fl/@to]/gmtOffset,"H"),
    $depteff := adjust-time-to-timezone(xs:time($fl/@departure),$deptOffsetDur),
    $arreff := adjust-time-to-timezone(xs:time($fl/@arrival),$arrOffsetDur),
    $dureff := $arreff - $depteff (: XQuery knows that these are xs:time :)
return
<res dOffsetDur="{$deptOffsetDur}" aOffsetDur="{$arrOffsetDur}"
     depteff="{$depteff}" arreff="{$arreff}" dureff="{$dureff}"/>
```

[Filename: XQuery/flights-duration-example2.xq]

```
<res dOffsetDur="PT1H" aOffsetDur="PT0S" depteff="12:30:00+01:00"
                       arreff="14:40:00Z" dureff="PT3H10M"/>
<res dOffsetDur="PT1H" aOffsetDur="-PT5H" depteff="12:40:00+01:00"
                       arreff="15:10:00-05:00" dureff="PT8H30M"/>
<res dOffsetDur="PT1H" aOffsetDur="-PT3H30M" depteff="12:50:00+01:00"
                       arreff="14:20:00-03:30" dureff="PT6H"/>
```

The third flight takes 6hrs from +1 to the -3.5h-timezone

Handling XML processing instructions in XQuery

- PIs are also children of their parent element in the XML tree (cf. Slide 186):
- can be detected with nodetest “processing-instruction()”,
- access: use name() and string()

```
let $d :=
 doc("https://www.dbis.informatik.uni-goettingen.de/Teaching/SSD/XML-DTD/html-php.xml")
for $n in $d//processing-instruction()
return ($n, " --- ",  name($n), " --- ", string($n))
```

[Filename: XQuery/pis.xq]

Practical Hints: Output

When creating output, most XQuery engines generate the XML declaration, and output “<” and “>” in text nodes as “<” and “>”, respectively.

Output Options

- W3C XSLT and XQuery Serialization 3.1:
  declare namespace output = "http://www.w3.org/2010/xslt-xquery-serialization";
  declare option output:method "xml";
  declare option output:encoding "UTF-8";   (or e.g. "ISO-8859-1")
  declare option output:indent "yes";
- for saxon, look at its own extensions
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "saxon:indent-spaces=1";
- In case it is intended to generate e.g. LaTeX, SQL input statements, RDF/Turtle/N3 or whatever plain text output, the XML declaration and the “<”/“>”-conversion must be avoided.
  W3C: declare option output:method "text";
  saxon: declare option saxon:output "method=text";

Add Doctype Declaration to the Output (saxon)

- XQuery engines output only the XML structure itself
- how to add the <!DOCTYPE mondial SYSTEM "mondial.dtd"> preamble?
  With saxon, use
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "doctype-system=mondial.dtd";

Generation of Multiple Instances (and for Debugging)

- fn:put(*node,uri*) (belongs to XQuery Update Facility; requires saxonEE)
- side-effect during executing the program,
- also possible with dynamically computed filenames:
  Generate a file for each country:

```
(: saxonXQEE -update:on \!indent=yes redirected-output-countries.xq  :)
  declare namespace saxon="http://saxon.sf.net/";
  declare option saxon:output "doctype-system=mondial.dtd";
  declare option saxon:output "saxon:indent-spaces=1";
 for $c in doc("file:mondial.xml")//country
 return put($c, concat("tmp/",$c/@car_code, ".xml"))
```

[Filename: XQuery/redirected-output-countries.xq]

Flexibility

For each task, there is a multitude of possible solutions ...

Example: Uncorrelated Subqueries

Names of all countries that are larger than Germany:

- XPath:

```
//country[@area > number(//country[@car_code='D']/@area)]/name
```

- XQuery and SQL: uncorrelated subquery/semijoin

```
for $c in //country
where $c/@area >
 number(//country[@car_code='D']/@area)
return $c/name
```

```
SELECT c.name
FROM country c
WHERE c.area > (SELECT c2.area
                FROM country c2
                WHERE c2.code = 'D')
```

- binding the uncorrelated subquery to a variable:

```
let $germanyarea := number(//country[@car_code='D']/@area)
for $c in //country
where $c/@area > $germanyarea
return $c/name
```


6.5.5 XQuery: Conclusion

Design and Functionality

- combines the positive experiences of previous approaches
- avoids their drawbacks
- intuitively clear syntax and semantics
- declarative, orthogonal, functional style: every expression is a function on nodes/sequences of nodes that also returns a sequence of nodes
  - explicit, variable-based iteration: "for *var* in *expression*"
  - implicit iteration: "*collection*[*condition*]" or "*collection*/*path*"
- Theoretical background (see W3C XML Query Formal Semantics; datatypes of the XML Schema and XML Query Data Model)
  - for each expression (and thus also for its result), the formal type (according to the XML Schema datatypes) can be determined.
  - the type of each variable is determined in the same way.
  - formal, denotational semantics of queries:
    "what is the answer set of a given expression?"

XQuery: Conclusion (Cont'd)

W3C XML Query Formal Semantics:

- XPath/XQuery is a *functional* language (like SQL, LISP, Haskell, ...).
- is built from expressions, rather than statements. Every construct in the language (except for the XQuery query prolog) is an expression and expressions can be composed arbitrarily.
- The result of one expression can be used as the input to any other expression, as long as the type of the result of the former expression is compatible with the input type of the latter expression with which it is composed.
- Another characteristic of a functional language is that variables are always passed by value, and a variable's value cannot be modified through side effects.

XQuery: Conclusion (Cont'd)

- Note: XQueryX provides a syntax that is formulated in XML

Restrictions

- up to now no resolving of XLink/XPointer (see later)
- only a *query language*:
  decision of the W3C: first complete XQuery 1.0 as a query language and make it consistent with XML Schema and XML Query Data Model as a "Recommendation", and then consider updates in XQuery 2.0.
- started as an "XML Query Language" . . .
- . . . XQuery 3.0 became a full-fledged functional programming language.

General Design Patterns for Database Query Languages

SQL, OQL, XML-QL, XQuery (and many others) use the same underlying principle:

- binding variables
- evaluating a condition
- generating a result (which is a set of data items of the underlying data model)

Note: XQL did not follow this idea ⇒ restricted expressiveness and clarity

... let's now have a look on one more XML query language

- the underlying principle is the same

⇒ everything else is "just syntax"!

6.6 Further (Academic) Query Languages

XPathLog

- Prolog-/Datalog-style (May, DBPL and VLDB 2001; TPLP 2004)
- based on F-Logic
  - path syntax changed from *step.step.step* to *step/step/step*
  - same syntax for conditions as for F-Logic: "[...]" could be reused
  - F-Logic semantics (1989) closely related with XPath semantics
  - new: distinction between attributes/subelements
- Binding of variables at *arbitrary* positions of an expression
- joins as conjunction (as in Prolog/Datalog)

XPathLog

- implicit resolving of multi-valued attributes
- implicit resolving of reference attributes

```
?- //country->C[name->N and @membership->O/name->A].
```

- access to signature/metadata

```
?- //country[name="Germany"]/M.
?- //country[name="Germany"]/@A.
```

- class membership and -hierarchy

```
?- C isa country[name->N]/M.
?- _C isa country/@A->_O, _O isa X.
?- country[@M=>C].      % from DTD
```

XPathLog

- declarative language
- (equi-)join variables

```
?- //country->_C[name->N and @capital->_X[name->XN],
   //organization->_O[@abbrev->A and @headq->_X].
N/"Belgium", A/"EU" X/"Brussels"
N/"Austria", A/"OSCE" X/"Vienna"
   :           :          :
```

- XPath-style semantics in rule heads for *generation* and *manipulation* of XML data
- first implementation of an update language for XML (Demo VLDB 2001)
  generation of XML in rule heads:
  C[density -> D] :- C isa country[population -> P; @area -> A], D is P div A.
- fixpoint semantics for Datalog-style rules
  ⇒ possible to compute transitive closure etc.

```
R[tr_flows_into -> S]   :-   R isa river, R/to[@water -> S], S isa water.
R[tr_flows_into ->S]    :-   L isa lake, L/to[@water -> R[tr_flows_into -> S]].
R[tr_flows_into ->S]    :-   R isa river, R/to[@water -> W[tr_flows_into -> S]].
```


General Design Principles for Database Query Languages

SQL, OQL, XML-QL, XQuery (and many others) use the same underlying principle:

- binding variables
- evaluating a condition
- generating a result (which is a set of data items of the underlying data model)

	SQL/OQL	XML-QL	XQuery	XPathLog
variables:	1-step-navig.	XML patterns	XPath navig.	XPath navig.+
	SQL: flat data model			XPath patterns
	OQL: + path navig.			
conditions:	WHERE clause	Patterns	XPath fragment	XPath filters
		(equality join conds)	(only tree-style join-conds)	(join conds)
		WHERE clause	WHERE clause	separate conjuncts
		(comparisons+joins)	(all)	(comparisons+joins)

- the underlying Logic Programming fixpoint semantics enables XPathLog to compute the transitive closure
- ... but it does not allow for syntactically nested statements

Further (Academic) Query Languages

- XML-GL (Comai, Politecnico Milano, 1999): graphical “language”
- Lorel-XML (Stanford Univ., 1999): OQL-style language, migration of Lorel
- YATL-XML (Cluet, INRIA, 2000): term-based language, migration of YATL
- Lixto/Elog (Gottlob, TU Wien, 2001): graphical tool for data extraction from the Web, Datalog-based internals
- Xcerpt, XChange (Bry et al, LMU München, 2002): term- and unification-based language

... many different approaches to the same goal (mainly in Europe).
Overview in (May, TPLP 2004).

Chapter 7
Manipulating XML Data

- XML data in files:
  - usually no changes (except manually or by scripts)
  - transformations XML → HTML etc: XSLT
- XML data in application systems
  - inside the application programming language; mostly by the DOM-API
  - no special data manipulation language necessary (cf. OQL)?
- different proposals
  - pre-XQuery commercial area:
    - XMLDB: XUpdate (1999)
    - eXcelon (2000; XUL as extension of XSLT)
  - academic area:
    - “Updating XML” (Halevy et al, SIGMOD 2001) as an extension to XQuery
    - XPathLog (May, VLDB 2001): Prolog-style query- and manipulation language

Extending XQuery with Updates – Concepts

- always wrt. a context node
- base operations:
  - delete *node*
  - rename *node* as *name*
  - insert *node/nodes* before|after|into *node*
- combined operations:
  - replace *node* with *node*
  - move *node* before|after|into *node*

7.1 XML:DB Initiative's XUpdate

- XML:DB Initiative founded in late 1999
  Goal: interface for storing XML in databases
- Low-level API (Java etc., using DOM + XPath ...)
- an update concept: XUpdate
- Implementation:
  dbXML Core XML Database released as Open Source software in Sept. 2000
  transferred to the Apache Software Foundation ("Xindice"), abandoned in 2011.
- The XML:DB database API is implemented in several systems:
  eXist (`http://exist-db.org`; open-source), Tamino (Software AG)

... but here we are mainly interested in XUpdate ...
(note that XUpdate (1999) is not related with XQuery (2001))

XML:DB XUpdate

Situation in 1999: XML, XPath, XSLT [see later], low-level APIs

- Requirement: "The XML Update specification MUST be an XML element"
  i.e., the language is itself in XML syntax (like XSLT and XML Schema)
- XUpdate: a very basic description of update operations:
  - which node (elements, attributes)
  - which operation (delete, update value, append/insert to contents)
  - new value (in case of update/append/insert)

Basic structure:

```
<xu:modifications xmlns:xu= "http://www.xmldb.org/xupdate">
  <xu:operation select= "xpath-expression">
    contents (e.g. new value)
  </xu:operation>
</xu:modifications>
```

... submit such an element as a message (e.g., HTTP POST) to the DB and get the update.

XUpdate: Example

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:append select="/mondial/country[name='Germany']">
    <xu:element name='localname'>Deutschland</xu:element>
  </xu:append>
</xu:modifications>
```

[Filename: XUpdate/append.xu]

Calling eXist with (see `client.sh -h`)

`/bin/gen_client.sh -u user -P password -c /db/may -f mondial.xml -X append.xu`

executes the update.

- `select= "xpath"` is the same as in XSLT (see later), XML Schema etc. – a widely used concept in the XML world.
  (if multiple nodes are addressed, each one is modified)
- `<xu:element>` constructor is the same as in XSLT (1998) and later in XQuery's RETURN clause
- analogously insert-before and insert-after.

XUpdate: Examples (Cont'd)

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:remove select="/mondial/country[name='Germany']/localname"/>
</xu:modifications>
```

[Filename: XUpdate/remove.xu]

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:update select="/mondial/country[name='Germany']/population">
     80000000
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update.xu]

XUpdate: Examples (Cont'd)

- get the new value from the database:

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:update select="/mondial/country[name='Germany']/population/text()">
     <xu:value-of select="/mondial/country[name='Germany']/@area"/>
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update-select.xu]

note: the inner `select` cannot depend on the current node.

```
<?xml version="1.0"?>
<xu:modifications version="1.0" xmlns:xu="http://www.xmldb.org/xupdate">
  <xu:variable name="bla"
      select="/mondial/country[name='Germany']/gdp_total/text()"/>
  <xu:update select="/mondial/country[name='Germany']/population/text()">
     <xu:value-of select="$bla"/>
  </xu:update>
</xu:modifications>
```

[Filename: XUpdate/update-variable.xu]

XUpdate: Conclusion and Comments

- XML syntax of the language strongly influenced by XSLT (1998)
  - elements as commands
  - `select="..."` selects nodes to which the command is applied
  - use of variables `select="$`*variable*`"` as in XSLT, and later also in XQuery
  - element/command contents specifies what is to be done
  - element generation by literal XML (also in XSLT and later XQuery)
- only very simple functionality
  - no way to compute the inner value,
  - no iteration etc.
- same time: combination with XSLT and XUpdate to XUL (XML Update Language/Updategrams [Excelon, 2002 bought by Progress Software]):
  XSLT program structures + XUpdate operations, applied to "current node" of XSLT.

7.2 XQuery with Updates

- extend a *declarative query language* with updates
- based on *variable bindings*
- SQL: FROM-WHERE for selecting nodes ...
  ... that are then modified.
- XQuery: FOR-LET-WHERE for selecting nodes ...
  ... that are then modified.
- execute update instead of the return-clause (cf. SQL: UPDATE/DELETE vs. SELECT)?

Aside: XQuery with Updates – an early proposal

- QuiP: the 2001/02 XQuery prototype of Software AG, later integrated into the Tamino system (before: XQL).
- calling `quip filename.xq > bla.xml` wrote the modified XML to a file.

```
update
for $c in document("twocountries.xml")//country
let $area := string($c/@area)
delete $c/@area
insert <area>{$area}</area> after $c/name
rename $c//city[@id=$c/@capital] as capital
replace $c/@car_code with
    attribute code {concat($c/name/text(), ":", string($c/@car_code))}
replace $c/population/text() with
   $c/population/text() * (1 + $c/population_growth div 100)
insert "biggest city" into
   $c//city[population = max(for $citypop in $c//city/population/text()
                              return int($citypop))]
```

[Filename: XQuery/update.quip]

XQuery with Updates

- XQuery reached recommendation state in 2007 ... as a query language still without updates.
- "XQuery Update Facility", first W3C Working Draft has been published 27 January 2006; `http://www.w3.org/TR/xqupdate`
- XQuery Update Facility 1.0, W3C Recommendation 17 March 2011; `https://www.w3.org/TR/xquery-update-10/`

Update Expressions

delete (node|nodes) *TargetExpr*
insert node|nodes *SourceExpr* (([as (first | last)] into) | after | before) *TargetExpr*
rename node *TargetExpr* as *Expr*   *// Expr must result in a qname*
replace [value of] node *TargetExpr* with *Expr*

- Syntax: "node"/"nodes" does not matter.
- for insert, rename, and replace, *TargetExpr* must evaluate to a single node.

Simple Updates in XQuery

- not implemented in saxonHE, only in (commercial) saxonEE (download 30 days eval)
- changes the XML file,
- in an XML database system, the internal database would be changed.
- call `saxonXQEE -update:on update.xq`
  changes the document (if given as local file; not via http:...)
  (use `-update:discard` just for testing syntax without changing the document)
- call `saxonXQEE -update:on -tree:linked` *filename* for (required for transformation commands, cf. Slide 341)

```
(: saxonXQEE -update:on \!indent=yes simpleinsert.xq  :)
insert node <bla/>
after fn:doc("mondial.xml")//country[name='Albania']
```

[Filename: XQuery/simpleinsert.xq]

```
delete nodes fn:doc("mondial.xml")//bla
```

[Filename: XQuery/simpledelete.xq]

```
 replace value of node fn:doc("mondial.xml")//country[name='Germany']/name
         with 'Deutschland'
```

[Filename: XQuery/simpleupdate.xq]

XQuery with Updates: Embedding into FLWR Clauses

- can be embedded into the return part of FLWR clauses when variables should be used (note: the return must still be there!)
- update statements return nothing
- sequence for executing multiple updates
- no queries allowed in the same sequence

```
for $c in  fn:doc("mondial.xml")//country
return (
  insert node <bla/> after $c/name,
  delete node $c/@area
  (: , $c/@car_code         not allowed :)
)
```

[Filename: XQuery/updateseq.xq]   (changes the mondial.xml file)

XQuery with Updates – Copy-Modify Expressions

1. copy: assign variable(s) to fragments if the tree,
2. modify: update things bound to the copy-variable(s),
3. return: return something generated from the copied+updated variables.

- not an update! – (often called “hypothetical update”)
- rather belongs to the querying/constructing functionality; sometimes shorter than reconstructing as a new fragment

Syntax:

copy $*VarName* := *Expr* (, $*VarName* := *Expr*)*
modify *UpdateExprSeq*
return *Expr*

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked copymodify.xq :)
for $c in fn:doc("mondial.xml")//country[@area > 1000000]
return
  copy $cc := $c
  modify delete nodes $cc/(province|city)
  return $cc
```

[Filename: XQuery/copymodify.xq] / / (outputs the result doc.)

XQuery with Updates – how to perform complex tasks

(see example next slide)

- embed into a copy-modify-return-statement
  - ⇒ cannot write file directly
  - ⇒ pipe output
- works differently than expected when knowing SQL updates:
  - collect *“pending updates”* as a sequence in the modify statement
  - use “for ... return *updates*” to collect them

XQuery with Updates – code example next slide

- for *each* country: update most recent population (+1 year, using population growth rate),
- turn Bavaria into an independent country (element), and delete it as province (semantically absolutely incomplete update)

XQuery with Updates – complex task example

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked updmondial.xq > bla.xml :)
let $mm := fn:doc("mondial.xml")
return
  copy $m := $mm
  modify
  ( for $c in $m//country[population_growth and population]
    return ( replace value of node $c/population[last()]
               with $c/population[last()] * (1 + $c/population_growth div 100),
             replace value of node $c/population[last()]/@year
               with $c/population[last()]/@year + 1 ),
    insert node
      ( copy $bav := $m//country[name='Germany']/province[name='Bayern']
        modify ( rename node $bav as 'country',
                 insert node attribute area{string($bav/area)} into $bav,
                 delete node $bav/area )
        return $bav )
    after $m//country[name='Germany'],
    delete node $m//country[name='Germany']/province[name='Bayern']
  )
  return $m
```

[Filename: XQuery/updmondial.xq]

Updating Functions

- declare updating function ... { returns *update statements* };

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked updfct.xq :)
declare updating function local:growpop($tree, $n as xs:string) {
  let $c := $tree//country[name=$n]
  return
  ( replace value of node $c/population[last()]
      with round($c/population[last()] * (1 + $c/population_growth div 100)),
    replace value of node $c/population[last()]/@year
      with $c/population[last()]/@year + 1 ) };

let $m := doc("mondial.xml")
return (local:growpop($m, "Germany"),
        local:growpop($m, "Austria") )
```

[Filename: XQuery/updfct.xq]   updates mondial.xml

XQuery with Updates 3.1 – Transform Expressions NOT YET SUPPORTED

- apply a transformation to a single node
- shorthand for copy-modify-return

```
(:  saxonXQEE -update:on \!indent=yes -tree:linked transform.xq :)
let $x := fn:doc("mondial.xml")
return
  $x transform with
  { modify delete nodes $x//country }
```

[Filename: XQuery/transform.xq]

Chapter 8
The Transformation Language XSL

8.1 XSL: Extensible Stylesheet Language

- developed from
  - CSS (Cascading Stylesheets) scripting language for transformation of data sources to HTML or any other optical markup, and
  - DSSSL (Document Style Semantics and Specification Language), stylesheet language for SGML. Functional programming language.
- idea: rule-based specification how elements are transformed and formatted *recursively*:
  - Input: XML
  - Output: XML (special case: HTML)
    special case: a single text node (=string), e.g. SQL input statements, LaTeX code, ...
- declarative/functional: XSLT (XSL Transformations)

APPLICATIONS

- XML → XML
  - Transformation of an XML instance into a new instance according to another DTD,
  - Integration of several XML instances into one,
  - Extraction of data from an XML instance,
  - Splitting an XML instance into several ones.
- XML → HTML
  - Transformation of an XML instance to HTML for presentation in a browser
- XML → string
  - since no data structures, but only Unicode is generated, LaTeX, postscript, pdf can also be generated
  - ... or transform to XSL-FO (Formatting objects).

THE LANGUAGE(S) XSL

Partitioned into two sublanguages:

- functional programming language: XSLT
  "understood" by *XSLT-Processors* (e.g. xt, xalan, saxon, xsltproc ...)
- generic language for document-markup: XSL-FO
  "understood" by XSL-FO-enabled *browsers* that transform the XSL-FO-markup according to an internal specification into a direct (screen/printable) presentation.
  (similar to LaTeX)
- programming paradigm: self-organizing tree-walking
- XSL itself is written in XML-Syntax.
  It uses the namespace prefixes "xsl:" and "fo:",
  bound to `http://www.w3.org/1999/XSL/Transform` and
  `http://www.w3.org/1999/XSL/Format`.
- XSL programs can be seen as XML data.
- it can be combined with other languages that also have an XML-Syntax (and an own namespace).

APPLICATION: XSLT FOR XML → HTML

- the prolog of the XML document contains an XML processing instruction that specifies the stylesheet to be used:

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="mondial-simple.xsl"?>
<!DOCTYPE mondial SYSTEM "mondial.dtd">
<mondial> ...  </mondial>
```

- if an (XSL-enabled) browser finds an XML document with a stylesheet instruction, then the XML document is processed according to the stylesheet (by the browser's own XSLT processor), and the result is shown in the browser.
  `http://dbis.informatik.uni-goettingen.de/Teaching/SSD/XSLT/mondial-with-stylesheet.xml`
  ⇒ click "show source" in the browser
- Remark: not all browsers support the full functionality (id()-function)
  (recall that this requires accessing the DTD)
- in general, for every main "object type" of the underlying application, there is a suitable stylesheet how to present such documents.

8.2 XSLT: Syntax and Semantics

- Each XSL-stylesheet is itself a valid XML document,

```
<?xml version="1.0">
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    ...
</xsl:stylesheet>
```

- contains elements of the namespace xsl: that specify the transformation/formatting,
- contains literal XML for generating elements and attributes of the resulting document,
- uses XPath expressions for accessing nodes in an XML document. XPath expressions (mostly) occur as attribute values of `<xsl:...>` elements,
  (e.g., `<xsl:copy-of select='`*xpath*`'/>`)
- XSL stylesheets/programs recursively generate a result tree from an XML input tree.

8.2.1 XSLT: Flow Control by Templates

The stylesheet consists mainly of *templates* that specify the instructions *how* elements should be processed:

- xsl:template:

```
<xsl:template match="xsl-pattern">
    content
</xsl:template>
```

- *xsl-pattern* is an XPath expression without use of "*axis*::" (cf. Slide 199). It indicates for which nodes (e.g. elements or attributes) the template is applicable:
  a node x satisfies *xsl-pattern* if there is some k in the document, such that x is in the result set of evaluating *xsl-pattern* with k as context node.
  (often, k is an ancestor node of x, but not always)
  (another selection takes place at runtime when the nodes are processed for actually deciding to apply a template to a node).
- *content* contains the XSL statements for generation of a fragment of the result tree.

TEMPLATES

- ```
  <xsl:template match="city">
    <xsl:copy-of select="current()"/>
  </xsl:template>
  ```
  is a template that can be applied to cities and copies them unchanged into the result tree.
- `<xsl:template match="lake|river|sea"> ... </xsl:template>`
  can be applied to waters.
- `<xsl:template match="country/province/city"> ... </xsl:template>`
  can be applied to city elements that are subelements of province elements that in course are subelements of country elements.
- `<xsl:template match="id('D')"> ... </xsl:template>`
  can be applied to the element whose ID is "D".
- `<xsl:template match="city[population[last()] &gt; 1000000]"> ... </xsl:template>`
  can be applied to city elements that have more than 1000000 inhabitants.
  ```

Execution of Templates: "Tree Walking"

- xsl:apply-templates:

  <xsl:apply-templates select="*xpath-expr*"/>

- *xpath-expr* is an XPath expression that indicates for which elements (starting from the node where the current template is applied as context node) "their" template should be applied.
  Note that elements are processed in order of the final axis of the select expression.
- By <xsl:apply-templates> elements inside the content of <xsl:template> elements, the hierarchical structure of XML documents is processed
  - simplest case (often in XML → HTML): depth-first-search
  - can also be influenced by the "select" attribute: "tree jumping"
- if all subelements should be processed, the "select" attribute can be omitted.

  <xsl:apply-templates/>

Templates

- <xsl:apply-templates select="country"/>
  processes all country subelements of the current context element.
- <xsl:apply-templates select="country/city"/>
  processes all city subelements of country subelements of the current context element,
- <xsl:apply-templates select="/mondial//city[population[last()] > 1000000]"/>
  processes all city elements that are contained in Mondial and whose population is more than 1000000,
- <xsl:apply-templates select="id(@capital)"/>
  processes the element whose ID equals the value of the capital-(reference) attribute of the current context element.
- <xsl:apply-templates select="located/@province"/>
  processes all @province attributes of located elements. This can e.g. be used to change them in a transformation.

Templates

- One template must be applicable to the *document node* or to the *root element* for initiating the processing (both of them can be used):
  - <xsl:template match="*name_of_the_root_element*">
  - <xsl:template match="/">
    matches the *document node*; its (only) child is then the root element (e.g., mondial)
  - <xsl:template match="/*">
    matches the unique root element (e.g., mondial)
  - <xsl:template match="*">
    matches any element (see conflict resolution policies later)

Rule-Based "Programming"

- local semantics: templates as "rules"
- global semantics: built-in implicit tree-walking combines rules

Templates: Example

Presentation of the country information as a table (→ HTML)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">

 <xsl:template match="mondial">
   <html> <body> <table>
            <xsl:apply-templates select="country"/>
       </table>  </body> </html>
 </xsl:template>

 <xsl:template match="country">
   <tr><td> <xsl:value-of select="name"/> </td>
       <td> <xsl:value-of select="@car_code"/> </td>
       <td align="right"> <xsl:value-of select="population[last()]"/> </td>
       <td align="right"> <xsl:value-of select="@area"/> </td>
   </tr>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-simple.xsl]

Templates: Example

Presentation of the country and city information as a table:

```
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="mondial">
 <html><body><table>
     <xsl:apply-templates select="country"/>
 </table></body></html>
</xsl:template>

<xsl:template match="country">
<tr valign="top">
  <td><xsl:value-of select="name"/></td>
  <td><xsl:value-of select="@car_code"/></td>
  <td align="right"><xsl:value-of select="population[last()]"/></td>
  <td align="right"><xsl:value-of select="@area"/></td>
  <td valign="top">
    <table><xsl:apply-templates select=".//city"/></table>
  </td>
</tr>
</xsl:template>

<xsl:template match="city">
<tr> <td width="100"><xsl:value-of select="name[1]"/></td>
     <td align="right" width="100">
        <xsl:value-of select="population[last()]"/>
     </td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-nested.xsl]

Templates: Example

The following (transformation: XML $\rightarrow$ XML) stylesheet copies all country and city elements from Mondial and outputs first all country elements, and then all city elements as top-level elements:

```
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
   <!-- Template that copies elements -->
   <xsl:template match="city|country">
      <xsl:copy-of select="current()"/>
   </xsl:template>
   <xsl:template match="mondial">
      <!-- apply templates: first countries -->
      <xsl:apply-templates select="/mondial/country">
      <!-- apply templates: then cities -->
      <xsl:apply-templates select="//country/city | //country/province/city"/>
   </xsl:template>
</xsl:stylesheet>
```


Templates

Difference between:

1. <xsl:template match="*xsl-pattern*">
   content
   </xsl:template>
2. <xsl:apply-templates select="*xpath-expr*"/>

- select="..." is evaluated wrt. the current context node (selects which elements are addressed by the given XPath expression),
- match="..." is evaluated wrt. the document structure starting from "below" (checks if the document structure matches with the pattern),
- xsl:apply-templates selects nodes for application by its *xpath-expr*, and then the suitable templates are applied,
- the order of templates has no effect on the order of application (document order of the selected nodes).

Templates

Exercise 8.1
Describe the difference between the following stylesheet fragments:

1.
```
<xsl:template match="city">
     <xsl:copy-of select="current()"
</xsl:template>
<xsl:apply-templates select="//country/city"/>
<xsl:apply-templates select="//country/province/city"/>
```

2.
```
<xsl:template match="country/city">
     <xsl:copy-of select="current()"
</xsl:template>
<xsl:template match="country/province/city">
     <xsl:copy-of select="current()"
</xsl:template>
<xsl:apply-templates select="//country/city|//country/province/city">
```

□

Conflicts between Templates

When using non-disjoint match-specifications of templates (e.g. *, city, country/city, city[population[last()]>1000000]) (including possibly templates from imported stylesheets), several templates are probably applicable.

- in case that during processing of an <xsl:apply-templates>-command several templates are applicable, the one with the most specific match-specification is chosen.
- defined by *priority rules* in the XSLT spec.
- <xsl:template match="..." priority="n"> for manually resolving conflicts between incomparable patterns.

Overriding (since XSLT 2.0)

The above effect is similar to *overriding* of methods in object-oriented concepts: always take the most specific implementation

- <xsl:next-match>: apply the next-lower-specific rule (among those defined in the same stylesheet)
- <xsl:apply-imports>: apply the next-lower-specific rule (among those defined in imported stylesheets (see later))

Resolving Template Conflicts Manually

Process a node with different templates depending on situation:

- associating "modes" with templates and using them in apply-templates

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="mondial">
  <xsl:apply-templates select="country[@area>1000000]"/>
  ... and now the second part ...
  <xsl:apply-templates select="country[@area>1000000]" mode="bla"/>
</xsl:template>
<xsl:template match="country">
  <firsttime> <xsl:value-of select="name"/> </firsttime>
</xsl:template>
<xsl:template match="country" mode="bla">
  <secondtime> <xsl:value-of select="name"/> </secondtime>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-modes.xsl]

Named Templates

Named templates serve as macros and can be called by their name.

- xsl:template with "name" attribute:

  <xsl:template name="*name*">
  content
  </xsl:template>

  - *name* is an arbitrary name
  - *content* contains xsl-statements, e.g. xsl:value-of, which are evaluated against the current context node.
- xsl:call-template

  <xsl:call-template name="*name*"/>
- Example: Web pages – templates for upper and left menus etc.

Default Templates

- there is a built-in set of *default templates* that apply if the program does not specify a template:
- altogether they return *the string value of a node*

```
<xsl:template match="*|/">
  <xsl:apply-templates/>  <!-- recursively over children, NOT attributes! -->
</xsl:template>
<xsl:template match="text()|@*">
  <xsl:value-of select="."/>
</xsl:template>
```

- they have lowest priority ("imported before any stylesheet")
- one can e.g. use their recursive behavior and override only some of them.
- they also apply when called in some mode m and there is no user-defined template.

Default Templates: Example

- recurse by default through all children, not through the attributes.
- do not do that for cities, but output only their country attribute
  (when the default template *is* invoked for an attribute, then it outputs its value)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
<xsl:template match="city">
  <xsl:apply-templates select="@country"/>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/default-example.xsl]

8.2.2 XQuery and XSLT

- both are declarative, functional languages ...
- ... with completely different strategies:
  - XQuery: nesting of the return-statement directly corresponds to the structure of the result
  - XSLT: the nested processing of templates yields the structure of the result.

XSLT

- modular structure of the stylesheets
- extensibility and reuse of templates
- flexible, data-driven evaluation

XQuery

- better functionality for joins (for $a in ..., $b in ...)
- XSLT: joins must be programmed explicitly as nested loops (xsl:for-each)

TRANSLATION XSLT → XQUERY

- each template is transformed into an FLWR statement,
- inner template-calls result in nested FLWR statements inside the return-clause
- genericity of e.g. <apply-templates/> cannot be expressed in XQuery since it is not known which template is activated

⇒ the more flexible the schema (documents), the more advantages show up for XSLT.

Exercise 8.2

- Give XQuery queries that do the same as mondial-simple.xsl and mondial-nested.xsl.
- Give an XQuery query that does the same as the stylesheet on Slide 358. □

8.2.3 XSLT: Generation of the Result Tree

Nodes can be inserted into the result tree by different ways:

- literal XML values and attributes,
- copying of nodes and values from the input tree,
- generation of elements and attributes by constructors.

Configuring Output Mode

- recommended, top level element (see xsl doc. for details):
  <xsl:output method="xml|html|xhtml|text" indent="yes|no"/>
  (not yet supported by all XSLT tools; saxon has it)

Generation of Structure and Contents by Literal XML

- All tags, elements and attributes in the content of a template that do not belong to the xsl-namespace (or to the local namespace of an xsl-tool), are literally inserted into the result tree.
- with <xsl:text> *some_text*</xsl:text>, text can be inserted explicitly (whitespace, e.g. when generating IDREFS attributes).

Generation of the Result Tree

Copying from the Input Tree

- <xsl:copy>*contents*</xsl:copy>
  copies the current context node (i.e., its "hull"): all its namespace nodes, but *not* its attributes and subelements (note that contents can then be generated separately).
- <xsl:copy-of select="*xpath-expr*"/>
  copies the result of *xpath-expr* (applied to the current context) unchanged into the result tree.
  (Note: if the result is a sequence of complex subtrees, it is completely copied, no need for explicit recursion.)
- <xsl:value-of select="*xpath-expr*" [separator="*char*"]/>
  generates a text node with the string value of the result of *xpath-expr*.
  (Note: if the result is a sequence of complex subtrees, the string value is computed *recursively* as the concatenation of all text contents.)
  If the result is a sequence, the individual results are separated by *char* (default: space).
  [note: the latter changed from XSLT 1.0 (apply only to 1st node) to 2.0]

Generation of the Result Tree

Example:

```
<xsl:template match="city">
  <mycity>
    <xsl:value-of select="name[1]"/>
    <xsl:copy-of select="latitude|longitude"/>
  </mycity>
</xsl:template>
```

- generates a mycity element for each city element,
- the name is inserted as text content,
- the subelements latitude and longitude are copied:

```
<mycity>Berlin
  <latitude>52.45</latitude>
  <longitude>13.3</longitude>
</mycity>
```


Generation of the Result Tree: Inserting attribute values

For inserting attribute values,

<xsl:value-of select="*xpath-expr*"/>

cannot be used *directly*. Instead, XPath expressions have to be enclosed in {...}:

```
<xsl:template match="city">
  <mycity key="{@id}">
    <xsl:value-of select="name[1]"/>
    <xsl:copy-of select="latitude|longitude"/>
  </mycity>
</xsl:template>
```


Generation of the Result Tree

Example:

```
<xsl:template match="city">
  <mycity source="mondial"
          country="{ancestor::country/name[1]}">
    <xsl:apply-templates/>
  </mycity>
</xsl:template>
```

- generates a "mycity" element for each "city" element,
- constant attribute "source",
- attribute "country", that indicates the country where the city is located,
- all other attributes are omitted,
- for all subelements, suitable templates are applied.

XSLT: Generation of the Result Tree

Generation of Elements and Attributes

- <xsl:element name="*xpath-expr*">
  content
  </xsl:element>
  generates an element of element type *xpath-expr* in the result tree, the content of the new element is *content*. This allows for computing element names.
- <xsl:attribute name="*xpath-expr*">
  content
  </xsl:attribute>
  generates an attribute with name *xpath-expr* and value *content* which is added to the surrounding element under construction.
- With <xsl:attribute-set name="*name*"> xsl:attribute* </xsl:attribute-set>
  attribute sets can be predefined. They are used in xsl:element by
  use-attribute-sets="attr-set$_1$ ... attr-set$_n$"

Generation of IDREFS attributes

- XML source: "border" subelements of "country" with an IDREF attribute "country":
  <border country="*car_code*" length="...">
- result tree: IDREFS attribute country/@neighbors that contains all neighboring countries
- two ways how to do this (both require XSLT 2.0)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
<xsl:template match="*"><xsl:apply-templates select="country"/></xsl:template>
<xsl:template match="country">
  <country neighbors1="{border/@country}"> <!-- note: adds whitespace as separator -->
    <xsl:attribute name="neighbors2">
      <xsl:value-of select="border/@country"/> <!-- default separator: whitespace -->
    </xsl:attribute>
   </country>
 </xsl:template></xsl:stylesheet>
```

[Filename: XSLT/mondial-neighbors.xsl]

8.2.4 XSLT: Control Structures

... so far the "rule-based", clean XSLT paradigm with implicit recursive semantics:

- templates: recursive control of the processing

... further control structures inside the content of templates:

- iterations/loops
- branching

Design of XSLT Command Elements

- semantics of these commands as in classical programming languages (Java, C, Pascal, Basic, Cobol, Algol)
- Typical XML/XSLT design: element as a command, further information as attributes or in the content (i.e., iteration specification, test condition, iteration/conditional body).

Iterations

For processing a list of subelements or a multi-valued attribute, local iterations can be used:

<xsl:for-each select="*xpath-expr*">
  content
</xsl:for-each>

- inside an iteration the "iteration subject" is not bound to a variable (like in XQuery as `for $x in` *xpath-expression*), but
- the current node is that from the xsl:for-each, not the one from the surrounding xsl:template
- an xsl:for-each iteration can also be used for implementing behavior that is different from the templates "matching" the elements (instead of using modes).

For-Each: Example

Presentation of the country and city information as a table:

```
<xsl:stylesheet version="2.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="mondial">
 <html><body><table>
     <xsl:apply-templates select="country"/>
 </table></body></html>
</xsl:template>

<xsl:template match="country">
<tr valign="top">
  <td><xsl:value-of select="name"/></td>
  <td><xsl:value-of select="@car_code"/></td>
  <td align="right"><xsl:value-of select="population[last()]"/></td>
  <td align="right"><xsl:value-of select="@area"/></td>
  <td valign="top">
    <table>
      <xsl:for-each select=".//city">
       <tr> <td width="100"><xsl:value-of select="name[1]"/></td>
            <td align="right" width="100">
              <xsl:value-of select="population[last()]"/>
            </td>
       </tr>
      </xsl:for-each>
    </table>
  </td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-nested-for-each.xsl]

XSLT: Conditional Processing

- Simple Test:
  <xsl:if test="*predicate*"> *content* </xsl:if>

Example:

<xsl:template match="country">
  <table>
    <tr> <th colspan="2"> <xsl:value-of select="name"> </th>
    </tr>
    <xsl:if test="@area">
      <tr>
        <td> Area: </td>
        <td> <xsl:value-of select="@area"> </td>
      </tr>
    </xsl:if>
    ⋮
  </table>
</xsl:template>

XSLT: Conditional Processing

- Multiple alternatives:

<xsl:choose>
  <xsl:when test="*predicate*$_1$">
    content$_1$
  </xsl:when>
  <xsl:when test="*predicate*$_2$">
    content$_2$
  </xsl:when>
  ...
  <xsl:otherwise>
    content$_{n+1}$
  </xsl:otherwise>
</xsl:choose>

8.2.5 XSLT: Variables and Parameters

Variables and parameters serve for binding values to names.

Variables

- variables can be assigned only once (in their definition). A later re-assignment (like in C or Java) is not possible.
- variables can be defined as top-level elements which makes them visible in the whole document (as a constant).
- a variable definition can take place at an arbitrary position inside a template - such a variable is visible in all its following siblings, e.g.,
  - a variable before a <xsl:for-each> is visible inside the <xsl:for-each>;
  - a variable inside a <xsl:for-each> gets a new value for each iteration to store an intermediate value.

Binding and Using Variables

- value assignment either by a “select” attribute (value is a string, a node, or a set of nodes)

  <xsl:variable name=“*var-name*” select=“*xpath-expr*”/>

- or as element content (then, the value can be a tree which is generated dynamically by XSLT)

  <xsl:variable name=“*var-name*”>
  content
  </xsl:variable>

- Usage: by select=“$*var-name*”

Example: Variables

A simple, frequent use is to “keep” the outer current element when iterating by an xsl:for-each:

- Consider the previous “country”-example
- now: generate a table of all *pairs* of neighbors

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
<xsl:template match="*">
  <table><xsl:apply-templates select="country"/></table>
</xsl:template>
<xsl:template match="country">
  <xsl:variable name="mycode" select="@car_code"/>
  <xsl:for-each select="border">
   <tr>
    <td><xsl:value-of select="$mycode"/></td>
    <td><xsl:value-of select="@country"/></td>
   </tr>
  </xsl:for-each>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/mondial-neighbors-table.xsl]

Parameters

- ... similar to variables
- values are communicated to called templates by parameters,
- the definition of parameters is allowed *only at the beginning* of xsl:template elements. The defined parameter is then visible everywhere in the template body.
- the assignment of a value takes place in the calling <xsl:apply-templates> or <xsl:call-template> element.
- pure call-by-value, no call-by-reference possible.

Remark: since a parameter can be an element with substructures, theoretically, a single parameter is always sufficient.

Communication of Parameters to Templates

- Parameters are declared at the beginning of a template:

  <xsl:template match=“...”>
  <xsl:param name=“*param-name*”
  select=“*xpath-expr*”/> <!-- with a default value -->
  :
  </xsl:template>

- the parameter values are then given with the template call:

  <xsl:apply-templates select=“*xpath-expr*$_1$”>
  <xsl:with-param name=“*param-name*” select=“*xpath-expr*$_2$”/>
  </xsl:apply-templates>

- Often, parameters are propagated downwards through several template applications/calls. This can be automatized (since XSLT 2.0) by

  <xsl:param name=“*param-name*” [select=“*xpath-expr*”] tunnel=“yes”/>

  where it should be used,

  <xsl:with-param name=“*param-name*” select=“*xpath-expr*” tunnel=“yes”/>

  where it should be passed downwards.

Example: Parameters

Generate a table that lists all organizations with all their members. The abbreviation of the organisation is communicated by a parameter to the country template which then generates an entry:

→ next slide

[Filename: orgs-and-members.xsl]

Exercise 8.3

- Extend the template such that it also outputs the type of the membership.
- Write an equivalent stylesheet that does not call a template but works explicitly with <xsl:for-each>.
- Give an equivalent XQuery query (same for the following examples). □

Example (cont'd)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:template match="mondial">
    <html><body> <h2>Membership Table</h2>
      <table> <xsl:apply-templates select="organization"/>
      </table></body></html>
  </xsl:template>
  <xsl:template match="organization">
    <tr><td colspan="2"><xsl:value-of select="name"/></td></tr>
    <xsl:apply-templates select="id(members/@country)">
       <xsl:with-param name="the_org" select="name/text()"/>
    </xsl:apply-templates>
  </xsl:template>
  <xsl:template match="country">
    <xsl:param name="the_org"/>
    <tr><td><xsl:value-of select="$the_org"/></td>
        <td><xsl:value-of select="name/text()"/></td></tr>
  </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/orgs-and-members.xsl]

Example/Comparison of Mechanisms

Example: This example illustrates the implicit and explicit iterations, and the use of variables/parameters

[use file:XSLT/members1.xsl and develop the other variants]

- Generate a list of the form

  ```
  <organization> EU <member>Germany</member>
                    <member>France</member> ... </organization>
  ```

  – using template-hopping [Filename: XSLT/members1.xsl]

  – using xsl:for-each [Filename: XSLT/members2.xsl]

- Generate a list of the form

  ```
  <membership organization="EU" country="Germany"/>
  ```

  based on each of the above stylesheets.

  – template hopping: requires a parameter [Filename: XSLT/members3.xsl]

  – iteration: requires a variable [Filename: XSLT/members4.xsl]

A Powerful Combination: Variables and Control

<xsl:variable name="*var-name*">

contents

</xsl:variable>

Everything inside the *contents* is bound to the variable – this allows even to generate complex structures by template applications (similar to XQuery's "let"):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:template match="mondial">
    <xsl:variable name="bigcities">
      <xsl:apply-templates select="//city"/>
    </xsl:variable>
    <xsl:copy-of select="$bigcities//name[1]"/>
    <xsl:copy-of select="sum($bigcities//population[last()])"/>
  </xsl:template>
  <xsl:template match="city">
    <xsl:if test='number(population[last()])>1000000'>
      <xsl:copy-of select="current()"/>
    </xsl:if>
  </xsl:template> </xsl:stylesheet>
```

[Filename: XSLT/letvar.xsl]

XQuery's "let" vs. XSLT variables

- XQuery's "let" is a sequence of nodes:

```
let $x := for $c in //country[@area > 1000000] return $c/name
return $x[5]
```
[Filename: XQuery/let-sequence.xq]

returns the single, 5th name element <name>Kazakhstan</name>.

- XSLT variables have children instead:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
<xsl:template match="mondial">
  <xsl:variable name="x">
    <xsl:copy-of select="//country[@area > 1000000]/name"/>
  </xsl:variable>
  <xsl:copy-of select="$x/name[5]"/>
</xsl:template>
</xsl:stylesheet>
```
[Filename: XSLT/var-children.xsl]

also returns the 5th name element.

- there is a crucial difference if the sub-results are not elements, but text nodes (or numbers!):
  - in XQuery, the variable is then bound to a list of these text nodes;
  - in XSLT, all text "children" of the variable are concatenated to a single text node.

External Parameters

Stylesheets can be called with external parameters (e.g., from the shell, or from a Java environment):

- define formal parameters for the stylesheet:

  <xsl:stylesheet ...>
  <xsl:parameter name="*name*$_1$"/>
  <xsl:parameter name="*name*$_2$"/>
  stylesheet contents
  (parameters used as $name$_i$*)*
  </xsl:stylesheet ...>

- call e.g. (with saxon)
  saxonXSL bla.xml bla.xsl *name*$_1$=*value*$_1$ *name*$_2$=*value*$_2$

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:param name="country"/>
 <xsl:template match="mondial">
  <xsl:copy-of select="//country[name=$country]"/>
 </xsl:template>
</xsl:stylesheet>
```
[Filename: XSLT/external-param.xsl]

8.2.6 XSLT: Miscellaneous

Sorting

For the set-based XSLT elements

- xsl:apply-templates and
- xsl:for-each

it can be specified whether the elements should be processed in the order of some key:

<xsl:sort select="*xpath-expr*"
data-type = {"text"|"number"}
order = {"descending"|"ascending"}/>

- "select" specifies the values according to which the nodes should be ordered (evaluated wrt. the node as context node),
- "data-type" specifies whether the ordering should be alphanumeric or numeric,
- "order" specifies whether the ordering should be ascending or descending,
- if an "xsl:apply-templates"- or "xsl:for-each" element has multiple "xsl:sort" subelements, these are applied in a nested way (as in SQL).

Sorting (Example)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="mondial">
  <html><body><table>
   <xsl:apply-templates select=".//city|river|lake|mountain|island">
    <xsl:sort select="descendant::elevation[1]" data-type="number" order="descending"/>
         <!-- river: use source/elevation -->
   </xsl:apply-templates>
  </table></body></html>
 </xsl:template>
 <xsl:template match="*">
   <tr><td><xsl:value-of select="name"/></td>
       <td><xsl:value-of select="name()"/></td>
       <td><xsl:value-of select="descendant::elevation[1]"/></td></tr>
 </xsl:template>
</xsl:stylesheet>
```
[Filename: XSLT/sorting.xsl]

- ".//elevation[1]" results in two values (source/elevation, estuary/elevation) for rivers (cf. Slide 221).

  `XTTE1020: A sequence of more than one item is not allowed as the @select attribute of xsl:sort (<elevation>, <elevation>)`

Querying for context positions by position()

- position() is always evaluated *for a given node* wrt. *a given context (sequence)*
- in XSLT, the context is the surrounding iteration (apply-templates, for-each)
- in Mondial, languages, religions, and ethnic groups in each country are ordered by percentage.
  What is the ranking of the german language in each country?

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
  <xsl:template match="mondial">
    <html><body><table>
      <xsl:apply-templates select="//country[language[text()='German']]"/>
    </table></body></html>
  </xsl:template>
  <xsl:template match="country">
    <tr><td> pos in <xsl:value-of select="@car_code"/>:
      <xsl:apply-templates select="language"/>
    </td></tr>
  </xsl:template>
  <xsl:template match="language">
    <xsl:if test="text()='German'">
      <xsl:value-of select="position()"/>
    </xsl:if>
  </xsl:template> </xsl:stylesheet>
```

[Filename: XSLT/germanpos.xsl]

... position() within xsl:for-each

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
<xsl:template match="mondial">
  <html><body><table>
  <xsl:for-each select="//country[language[text()='German']]">
    <tr><td> rank in <xsl:value-of select="@car_code"/>:
      <xsl:for-each select="language">
        <xsl:if test="text()='German'">
          <xsl:value-of select="position()"/>
        </xsl:if>
      </xsl:for-each>
    </td></tr>
  </xsl:for-each>
</table></body></html>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/germanpos2.xsl]

- in both cases, any kind of filter can be used in the select=“language[...]” to the context.

GROUPING (SINCE XSLT 2.0)

Extends the <xsl:for-each> concept to groups:

<xsl:for-each-group select=“*xpath-expr*” group-by=“*local-key*”>
    content
</xsl:for-each-group>

Inside the content part:

- current element is the *first* element of the current group
  ⇒ for accessing/returning the whole group, something else must be used:
- current-group() returns the sequence of all elements of the current group (e.g., current-group()/name for all their names); can e.g. be used for aggregation
- current-grouping-key() returns the current value of the grouping key
- position() returns the number of the current group

Grouping (Example)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:template match="mondial">
  <xsl:for-each-group select="country" group-by="encompassed/@continent">
   <xsl:sort select="id(current-grouping-key())/area" data-type="number" order="ascending"/>
   <continent nr="{position()}">
    <xsl:copy-of select="id(current-grouping-key())/name"/>
    <xsl:copy-of select="current-group()/name"/>
   </continent>
  </xsl:for-each-group>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/for-each-group.xsl]

- note: <xsl:sort .../> is also allowed, using the current-grouping-key()

Exercise 8.4

- output the country names of each continent ordered by the @area of the country
- the same, but ordered by the portion of the area located on the respective continent
- Do the same in XQuery (note: use “let” or “group-by”).

□

Handling non-XSLT Namespaces in XSLT

- namespaces used in the queried document (e.g., xhtml)
- namespaces to be used in the generated document
- namespaces used in the XSLT stylesheet (xsd, fn, ...)

Declare the namespaces in the surrounding <xsl:stylesheet> element:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ht="http://www.w3.org/1999/xhtml"
                version="2.0">
```

tells the XSL processor that the namespace bound to 'http://www.w3.org/1999/xhtml' is denoted by "ht:" in this document.
(and <ht:body> is different from <body>)

Querying XHTML documents with namespace

```
<!--
 call: saxonXSL http://www.dbis.informatik.uni-goettingen.de/index.html
                xsl-html.xsl
 note: takes some time ...
-->
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ht="http://www.w3.org/1999/xhtml"
                version="2.0">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/">
 <result>
  <xsl:copy-of select="//ht:li"/>
 </result>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/xsl-html.xsl]

Functions

- the functions and operators from "XQuery Functions and Operators" (e.g., aggregations) and math operators are also available in XSLT.

User-Defined Functions (since XSLT 2.0)

<xsl:function name="*local-ns*:*fname*">
<xsl:param name="*param*$_1$"/>
:
<xsl:param name="*param*$_n$"/>
contents
</xsl:function>

- the *local-ns* must be declared by xmlns:*local-ns*='*uri*' in the xsl:stylesheet element;
- function can then be used with n parameters in "select='...'" XPath expressions; e.g., <xsl:value-of select="*local-ns*:*fname*(*value*$_1$,...,*value*$_n$)"/>

XSLT Exception Handling

- again, like in Java: try-catch, XSLT-ized
- recall Slide 301 (XQuery error handling) and the XPath error() function
- <xsl:try ...> ... </xsl:try>
  inside anything applies *either* to its select="..." attribute or to its contents (each of them generates the "result" when successfully evaluating);
- contains one or more
  <xsl:catch errors="*the errors to be caught; default **" ...> ... </xsl:catch>
  elements;
  each of them with a @select attribute or contents that generate output,
- see documentation: boolean xsl:try/@rollback-output in case of real result streaming!

XSLT Debugging

- if something is not working correctly, it is hard to debug an XSLT stylesheet ...
- <xsl:message ... > *the message ...* </xsl:message>
- outputs *the message ...* to stdout – which is important in case that the XSL output is written silently to another file)
- the content of the message can be any XSLT – e.g., it might use xsl:value-of statements reporting the value of some variables;
- @terminate="yes" additionally immediately terminates the processing.

Debugging Example

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:template match="mondial">
   <xsl:apply-templates select="country"/>
 </xsl:template>
 <xsl:template match="country">
   <xsl:value-of select=".//city/name"/>
   <xsl:if test="name='Namibia'">
     <xsl:message terminate="yes">
       <xsl:value-of select="population"/>
       Stop here.
     </xsl:message>
   </xsl:if>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/message.xsl]

- note: in presence of parallelization (in normal processing, the output is later serialized correctly), the processing is immediately interrupted in case of message/@terminate="yes". Then, previous output before the message might be interrupted/missing.

xsl:output

- (cf. Slide 368) top level element
  <xsl:output *attributes*/>
- method="xml|html|xhtml|text"
  indent="yes|no"   control some output formatting (mainly if humans will read it),
- Note: the XML declaration `<?xml version="1.0">` and a (optional) DTD reference `<!DOCTYPE mondial SYSTEM "mondial.dtd">` are *not* part of the XML tree, but belong to the document node. They can also be controlled via xsl:output:
- omit-xml-declaration = "yes" | "no"
- doctype-public = *string*
  doctype-system = *string*

... and what about associating an XSL stylesheet with the output?

Generating Processing Instructions

- Things in <? ... ?> are *Processing Instructions*, and they are intended for some processing tool.
- They are generated by the constructor <xsl:processing-instruction .../>.
- e.g., associate an XSL- or CSS-stylesheet with the generated document (here: mondial-simple.xsl from Slide 349 and 356):

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:template match="mondial">
 <xsl:processing-instruction name="xml-stylesheet"
      select="('href=&quot;mondial-simple.xsl&quot;', 'type=&quot;text/xsl&quot;')"/>
  <mondial>   <!-- copy the small countries to the result -->
    <xsl:copy-of select="country[100>@area]"/>  </mondial>
   </xsl:template> </xsl:stylesheet>
```

[Filename: XSLT/pi.xsl]

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="mondial-simple.xsl" type="text/xsl"?>
<mondial><country car_code="MC" ...></>...  </mondial>
```

(the output is (maybe) located at XSLT/pi-created.xml)

Access to Data from Multiple Source Documents

- using the doc()-function from XPath:
  (for historical compatibility, XSLT also allows to call it "document()")
- recall that an XML Catalog can be used for providing DTDs etc. (cf. Slides 235 ff.)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:ht="http://www.w3.org/1999/xhtml"
                version="2.0">
<xsl:output method="xml" indent="yes"/>
<xsl:template match="/"> <!-- "/", so one can call it for any xml document -->
 <result>
  <xsl:copy-of
   select="doc('http://www.dbis.informatik.uni-goettingen.de/index.html')//ht:li"/>
 </result>
</xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/web-queries.xsl]

Generation of Multiple Instances

- controlling output to different files (since XSLT 2.0):
  <xsl:result-document href="*output-file-url*"> *generate output* </xsl:result-document>
- note: generates directories if required.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="*">
  <xsl:result-document href="tmp/countries.xml">
    <countries><xsl:apply-templates select="country"/></countries>
  </xsl:result-document>
  <xsl:result-document href="tmp/organizations.xml">
    <organizations><xsl:apply-templates select="organization"/></organizations>
  </xsl:result-document>
 </xsl:template>
 <xsl:template match="country"><xsl:copy-of select="name"/></xsl:template>
 <xsl:template match="organization"><xsl:copy-of select="name"/></xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/redirected-output.xsl]

Generation of Multiple Instances

- also possible with dynamically computed filenames:
  Generate a file for each country:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0">
 <xsl:output method="xml" indent="yes"/>
 <xsl:template match="*">
    <xsl:apply-templates select="country"/>
 </xsl:template>
 <xsl:template match="country">
  <xsl:variable name="filename" select="concat('tmp/',@car_code)"/>
  <xsl:result-document href="{$filename}">
   <xsl:copy-of select="name"/>
  </xsl:result-document>
 </xsl:template>
</xsl:stylesheet>
```

[Filename: XSLT/redirected-output-countries.xsl]

Import Mechanisms

XSL-stylesheets can import other stylesheets (i.e., they import their rules):

- <xsl:include href="*url*"/>
  for conflict-free stylesheets,
- <xsl:import href="*url*"/>
  definitions of the importing document have higher priority than definitions from the imported documents,
  the xsl:import subelements must precede all other subelements.

Example: DBIS Web pages

- general macros, frames etc. as templates in separate files
- individual page content in XML
- stylesheets generate Web pages from XML content file

8.3 XSL-FO

XSL-FO specifies *formatting objects*, that are added to a result tree and describe the later formatting

- page layout, areas, frames, indentation,
- colors, fonts, sizes,
- structuring, e.g. lists, tables ...

XSL-FO provides similar concepts as known from LaTeX.

FO-objects are e.g. (Namespace fo:) fo:block, fo:character, display-graphic, float, footnote, inline-graphic, list-block, list-item, list-item-body, list-item-label, multi-case, page-number, page-number-citation, region-before/after, region-body, simple-link, table, table-and-caption, table-body, table-caption, table-cell, table-column, table-footer, table-header, table-row.

- Each of these objects has appropriate attributes.

XSL-FO

- result tree contains *formatting objects* elements
- the result tree is then input to a formatter that generates HTML/LaTeX/RTF/PDF etc.
- currently only understood by
  - FOP (originally by James Tauber, now by Apache), a Java program that translates XML documents that include XSL-FO-markup to PDF:
    `http://xml.apache.org/fop/`
  - Adobe Document Server (XML → PDF)
- the same idea is followed by the XML-based DocBook markup language, which is (mainly) used for technical documentation; transformed by XSLT stylesheets into XHTML, man pages, PDF etc.

8.4 XSLT: Language Design in the XML-World

- XSLT is itself in XML-Syntax
- there is a DTD for XSLT: `http://www.w3.org/TR/xslt#dtd`

⇒ Analogously, there is an XML syntax for XQuery: XQueryX,
`http://www.w3.org/TR/xqueryx-3`.

- XSLT uses an own *namespace*, xsl:....
- there are several further languages of this kind (programming languages, markup languages, representation languages ...):
  XLink, XML Schema,
  SOAP (Simple Object Access Protocol)
  WSDL (Web Services Description Language)
  OWL (Web Ontology Language)
  DocBook
  ... lots of application-specific languages.

8.5 Concepts

(cf. Slide 10)

- XML as an abstract *data model* (Infoset) with an *abstract datatype* (DOM) and several implementations (*physical level*),
- *declarative*, *set-oriented* query language on the *logical level* of the data model: XPath/XQuery
- new: XSLT: transformational language
- two possibilities to define *views*:
  - XQuery: views as queries,
  - XSLT: views by transformations, especially XHTML views to the user.

Chapter 9
XML Schema

9.1 Motivation

- Database area: schema description
  - cf. SQL datatypes, table schemata
  - constraints
- Programming languages: typing – *real* typing – means: theory
  - every expression (query, variable etc) can be assigned with a type
  - structural induction
  - static typechecking for queries/programs/updates
  - validation of resulting structures wrt. target DTD/Schema

XML Query Formal Semantics: Overview

Every (query) expression is assigned with a semantics

Static Semantics

Given a static environment, an expression is of a certain type:
(static env.: namespace decl, typedefs, type decls. of variables)

- $statEnv \vdash Expr : Type$

Dynamic Semantics

Given a dynamic environment, an expression yields a certain result:
(dynamic env.: context node, size+position in context, variable bindings)

- $dynEnv \vdash Expr \Rightarrow Value$
  (equivalent to "classical" notation: $[[Expr]]_{dynEnv} = Value$)

... both defined by structural induction.
(for short example: show 2.1.5 and "if" in 4.10 of W3C XQFS document)

XML Query Data Types

... by examples:

```
define type coordinates {
    element latitude  of type xs:float &     -- in any order
    element longitude of type xs:float}

define type city {
    attribute country of type xs:string,
    attribute province of type xs:string?,    -- optional
    element name of type xs:string,
    element population of type {             -- anonymous, local type
           attribute year of type xs:integer
           xs:nonNegativeInteger } *,        -- arbitrarily often
    element coordinates of type coordinates, -- defined above
   }
```

... similar to DTD expressions extended with primitive datatypes

XML Query Data Types (Cont'd)

XML type theory:

- operations on types (e.g. "union"): result type of a query that yields either a result of type a or type b
- derivation of new types by
  - additional constraints
  - additional content
- constraints: does the derived result type for some expression guarantee that some conditions hold?
- containment of types: is the derived result type for some expression covered by a certain target type?
  (static type checking of programs)
- can e.g. be applied for query and storage optimization, indexing etc.

Requirements on an XML Schema Language

- requirement: a schema description language for the user that is *based* on these types: (usage is optional – XML is self-describing)
- DTD: heritage of SGML; database-typical aspects are not completely supported (datatypes [everything is CDATA/PCDATA], cardinalities); but: order, iteration.
- DTD: syntax is not in XML.

⇒ better formalism for representing schema information

- XML syntax → easy to process
- more detailed information as in the DTD
- database world: datatypes with derived types, constraints etc.

XML Schema: Ideas

- no actually new concepts (in contrast to the definition of the object-oriented model), but ...
- combination of the power of previous schema languages:
  - datatype concepts from the database area (SQL)
  - idea of complex object types/classes from the OO area
  - structured types from the area of tree grammars (e.g. DTD)
- new in contrast to DTDs: distinguishes between (element content) types and elements. E.g., a *type* `percentageProperty` that is a simple element with string contents and a `percentage` attribute which is a decimal between 0 and 100. This *type* is then used for defining elements `language`, `ethnicgroup`, and `religion`.

⇒ more complex and modular than DTDs.

9.2 XML Schema: Design

Using XML syntax and a verbose formalism, XML Schema uses a very detailed and systematic approach to type definitions and -derivations.

- only a few primitive, atomic datatypes
- other *simple types* are derived from these by *restriction*,
- *complex types* with text-only contents are derived by *extension* from simple types,
- other (*complex types*) are derived by *restriction* from a general *anyType* (cf. class *object* in OO),
- these types are then used for declaring elements.

XML Schema: the Standard

The XML Schema Recommendation (since May 2001; version 1.1 Recommendation since 2012) consists of 2 parts:

- Part 2: "Datatypes"
  - Definition of *simple types*: have no attributes and no element content; are used only for text content and as attribute values.
- Part 1 "Structures":
  - Definition of structured datatypes (*complex types*): with subelements and attributes; are used as element types.
    - names/types of the subelements and attributes
    - order of the subelements
  - Definition of elements using the complex types.
- many syntax definitions
- Part 0: "Primer" (`http://www.w3.org/TR/xmlschema-0/`) explains and motivates the concepts.

Usage of XML Schema

- understand concepts and ideas (XML Schema Primer, lecture)
- apply them in practice
- lookup for syntax details in the W3C documents
- make experiences
  - Validation with xmllint:
    `xmllint -noout -schema` *schemafilename file.xml*
  - Validation with saxonEE
  - JAXB (cf. Slide 545)

XML Schema Documents

An XML-Schema document consists of

- a preamble and
- a set of definitions and declarations

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
content
</xs:schema>

content uses the following kinds of *schema components*:

- datatype definitions (simple types and complex types)
- attribute declarations
- element declarations
- miscellaneous ...

9.3 Datatypes

Datatypes are seen as triples:

- range (possible values, cardinality, equality and ordering),
- lexical representation by *literals* (e.g. 100 also as 1.0E2 and 1.0e+2)
- set of properties

The set of possible values can be specified in different ways:

- extensional (enumeration)
- intensional (by axioms)
- restriction of another domain
- construction from other domains (lists, sets ...)

Datatypes

Datatypes are characterized by their domain and *properties (called facets)* in multiple independent "dimensions". The facets describe the differences between datatypes.

The basic facets (present for each datatype) are

- equality,
- order relation,
- upper and lower bound,
- cardinality (finite vs. countable infinite),
- numerical vs. non-numerical.

Datatypes

- primitive datatypes (predefined)
- generated datatypes (derived from other datatypes). This can happen by aggregation (lists) or restriction of another datatype.
- Primitive predefined types in XML Schema:
  - string (with many subtypes: token, NMTOKEN),
  - boolean (lexical repr.: true, false),
  - float, double,
  - decimal (with several subtypes: integer etc.),
  - duration, time, dateTime, ...
  - base64Binary, hexBinary
  - anyURI (Universal Resource Identifier).
- generated predefined types:
  - integer, [non]PositiveInteger, [non]NegativeInteger, [unsigned](long|short|byte)

XML-specific Datatypes

There are some XML-specific datatypes (subtypes of string) that are defined based on the basic XML recommendation. They are only used for attribute types (atomic and list types):

- NMTOKEN (restriction of string according to the definition of XML tokens),
- NMTOKENS derived from NMTOKEN by list construction,
- IDREF/IDREFS analogously,
- Name: XML Names,
- NCName: non-colonized names,
- language: language codes according to RFC 1766.

Constraining Facets

By specifying constraining facets, further datatypes can be derived:

- for sequences of characters: length, minlength, maxlength, pattern (by regular expressions);
- for numerical datatypes: maxInclusive, minInclusive, maxExclusive, minExclusive,
- for lists: length, minLength, maxLength
- for decimal datatypes: totalDigits (number of digits), fractionDigits (number of positions after decimal point);
- enumeration (definition of the possible values by enumeration),

... for a description of all details, see the W3C XMLSchema Documents.

Generation of Simple Datatypes

Simple datatypes can be derived as <simpleType> from others:

Derivation by Restriction

Restriction of a base type (i.e., specification of further restricting facets):

```
<xs:simpleType name="name">
   <xs:restriction base="simple-type">
      facets
   </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="carcodeType">
   <xs:restriction base="xs:string">
      <xs:minLength value="1"/>
      <xs:maxLength value="4"/>
      <xs:pattern value="[A-Z]+"/>
   </xs:restriction>
</xs:simpleType>
```

Derivation by Restriction

Example:

```
<xs:simpleType name="latitudeType"
   <xs:restriction base="xs:decimal">
      <xs:minInclusive value="-90"/>
      <xs:maxInclusive value="90"/>
   </xs:restriction>
</xs:simpleType>
```

```
<xs:simpleType name="longitudeType"
   <xs:restriction base="xs:decimal">
      <xs:minExclusive value="-180"/>
      <xs:maxInclusive value="180"/>
   </xs:restriction>
</xs:simpleType>
```

defines two derived simple datatypes.

Remark: Usage of SimpleTypes

- as attributes (details later):

  ```
  <xs:attribute name="car_code" type="carcodeType"/>
  ```

  can e.g. be used in

  ```
  <country car_code="D"> ... </country>
  ```

- as elements (details later):

  ```
  <xs:element name="longitude" type="longitudeType"/>
  ```

  can e.g. be used in

  ```
  <longitude>48</longitude>
  ```

  Only if also attributes are required, a <complexType> must be defined.

Derivation by Restriction: Enumeration

- Enumeration of the allowed values.

Example (from XML Schema Primer)

```
<xs:simpleType name="USState">
   <xs:restriction base="xs:string">
      <xs:enumeration value="AK"/>
      <xs:enumeration value="AL"/>
      <xs:enumeration value="AR"/>
      <!– and so on ... –>
   </xs:restriction>
</xs:simpleType>
```

- so far, this functionality is similar to what could be done in SQL by attribute types and integrity constraints.
- additionally:
  - "multi-valued" list types (but still simple types)
  - complex types

Derivation as List Types

```
<xs:simpleType name="name">
   <xs:list itemType="simple-type">
      facets          <!-- optional -->
   </xs:list>
</xs:simpleType>
```

- *simpleType* must be a non-list type,
- facets of the list (e.g., maxLength, minLength, pattern) can be defined by subelements

Example

Datatype for a list of country codes:

```
<xs:simpleType name="countrylist">
   <xs:list itemType="carcodeType"/>
</xs:simpleType>
```

```
<xs:attribute name="neighbors" type="countrylist"/>
```

for <country neighbors="NL L F CH ..."> . . . </country>

Derivation as Union Types

- Analogously union of sets with xs:union and @xs:memberTypes.

Component of a data type for postal addresses for US suppliers: send e.g., to D 37075 Göttingen (car code), or CA 94065 Redwood (US State Code)

```
<xs:simpleType name="stateOrCountry">
   <xs:union memberTypes="carcodeType USState"/>
</xs:simpleType>
```


9.4 Attribute and Element Declarations and (Structural) Type Declarations

Modular design with named types and element (and attribute) declarations:

- Attributes are always based on simple types:

```
<xs:attribute name="attrname" type="simpletypename"/>
<xs:attribute name="car_code" type="carcodeType"/>
```

- Elements based on simple types (DTD: #PCDATA without attributes), without attributes:

```
<xs:element name="elemname" type="simpletypename">
<xs:element name="name" type="xs:string"/>
```

- Elements using structural content types:

```
<xs:element name="elemname" type="complextypename"/>
```

- Complex types can be defined and named by

```
<complexType name="complextypename"> contentModelDerivationSpec
</complexType >
```

as described next. One goal is to make *contentModelDerivationSpecs* reusable.

(Structural) Content Model/Type Declarations

Complex element content *types* can be derived from others by

<complexType [name="*complextypename*"]> *contentModelDerivationSpec* </complexType>.

- *contentModelDerivationSpec*: describes how the structural content type can derived by "extending" or "restricting" another simpleContentType or a complexContentType.

(1) Define a complex type "from scratch"
(this is an abbreviation of case (4) below):

```
<xs:complexType [name="complextypename"]>
   content model
   attribute usage declarations
</xs:complexType>
```

where *content model* is of a content group definition of the following forms like regular expressions (details see later):

- <xs:sequence> . . . </xs:sequence>
- <xs:choice> . . . </xs:choice>
- <xs:all> . . . </xs:all>
- (or <xs:group ref="*groupname*"> . . . </xs:group>)
- which in turn can be nested and finally contain <xs:element . . . ...> elements.

(Structural) Content Model/Type Declarations

contentModelDerivationSpec variants based on a simpleContentType:

(2) Simple type content "extended" with attributes:

```
<xs:simpleContent>
  <extension base="simpletypename" >
    attribute usage declarations
  </extension>
</xs:simpleContent>
```

used for element types xxx of the form <xxx *attrs-as-defined*> *simple-type-value* </xxx>

(3) (rarely used) restricting the content of another complex content type with simple content by facets

```
<xs:simpleContent>
  <restriction base="complextypename (!)" >
    facets
    attribute usage declarations
  </restriction>
</xs:simpleContent>
```

(Structural) Content Model/Type Declarations

contentModelDerivationSpec variants restricting a complexContentType:

(4) (rarely used) complex type by restriction of another complex type (overwrites is mainly)

<xs:complexContent [mixed="true"]>
<restriction base="*complextypename*" >
content model
attribute usage declarations
</restriction>
</xs:complexContent>

(4b) The complex type "from scratch" case in (1) corresponds to an abbreviation of (4) formally "by restricting xs:anyType":

<xs:complexContent>
<restriction base="xs:anyType" >
content model
attribute usage declarations
</restriction>
</xs:complexContent>

(the optional "mixed="true"" attribute is then added to the <xs:complexContent> element)

(Structural) Content Model/Type Declarations

contentModelDerivationSpec variants extending a complexContentType:

(5) Complex type by extension (basically appends its *content model* to the content model of the type it extends (if both are xs:all, then the result is xs:all, otherwise xs:sequence (seems to be strange if both are xs:choice!)) ,

<xs:complexContent [mixed="true"]>
<restriction base="*complextypename*" >
content model
attribute usage declarations
</restriction>
</xs:complexContent>

Using these structural types, the XML element types can be defined.

(Structural) Type Declarations and Element (Type) Declarations

- Modular design with named types and element declarations: declare types and elements referring to them separately:
  - Declare a named type globally (may be the same name as the later name of some element)

    <xs:complexType name="*complextypename*">
    content (incl attributes) model specification contains:
    <xs:element ref="*elemname*" *subelem-cardinality-spec etc.* />
    <xs:attribute ref="*attrname*" *attr-cardinality-spec etc.* />
    </xs:complexType>
- Alternative: unnamed, local declarations for subelements and attributes locally inside the *content model specification*.

Complex Element Content Types: Simple Type with Attributes

Population: text content and an attribute:

<population year="1997">130000</population>

- take the simpleType for the text content and extend it with an attribute:

```
<xs:complexType name="population">
  <xs:simpleContent>
    <xs:extension base="xs:nonNegativeInteger">
      <xs:attribute name="year" type="xs:nonNegativeInteger"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

- the complexType can have (but does not necessary have) the same name, as the element to be defined later.

Complex Element Content Types: Empty Element Types

Border: only two attributes:

<border country="F" length="500"/>

follows the case (1), defining a compleyType from scratch:

```
<xs:complexType name="border">
   <xs:attribute name="country" type="xs:IDREF"/>
   <xs:attribute name="length" type="xs:decimal"/>
</xs:complexType>
```


Complex Element Content Types Element Content Types: Arbitrary Element Types

- element types with complex content use (nested) structure-defining elements (called *Model Groups*):
  - <xs:sequence> ... </xs:sequence>
  - <xs:choice> ... </xs:choice>
  - <xs:all> ... </xs:all>
    ("all" with some restrictions - only top-level, no substructures allowed)
- inside, the allowed element types are specified:

  <xs:element name="*name*" type="*typename*"/>
- note: even if only one type of subelements is contained, one of the above must be used around it.
- note: the XML Schema definition requires to list the content model specification before the attributes.

Complex Element Content Types: Example

```
<xs:element name="country">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" maxOccurs="unbounded"/>    <!-- defined elsewhere -->
      <xs:element minOccurs="0" maxOccurs="1" ref="population"/>
      <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element minOccurs="0" maxOccurs="1" ref="indep_date"/>
        <xs:element minOccurs="0" maxOccurs="1" ref="dependent"/>
      </xs:choice>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="encompassed"/>
      <xs:choice>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="province"/>
        <xs:element minOccurs="1" maxOccurs="unbounded" ref="city"/>
      <xs:choice>
    </xs:sequence>
    <xs:attribute name="car_code" type="xs:ID"/>
    <xs:attribute name="area" type="xs:positiveDecimal"/>
    <xs:attribute ref="capital"/>                      <!-- defined elsewhere -->
    <xs:attribute name="memberships" type="xs:IDREFS"/>
  </xs:complexType>
</xs:element>
```


9.5 Composing Declarations

Further Attributes of Attribute Definitions

- use (optional, required, prohibited)
  default is "optional"
- default (same as in DTD: attribute is added if not given in the document)
- fixed (same as in DTD)

Further Attributes of Subelement Definitions

- minOccurs, maxOccurs: default 1.
- <default value="*value*"/> (bit different from attribute default): if the element is given in a document with empty content, then the default contents *value* is inserted.
  In case that an element is not given at all, no default is used.
- <fixed value="*value*"/>: analogous.

Examples: later.

Global Attribute- and Element Definitions

... up to now, arbitrary element *types* have been defined.

At least, for the root element, a separate element declaration is needed.

- <xs:attribute> and <xs:element> elements can not only occur inside of <xs:complexType> elements, but can also be global.
- as global declarations, they must not contain specifications of @use, @minOccurs, or @maxOccurs.
- global declarations can then be used in type definitions by @ref.
  Then, they have @use, @minOccurs and @maxOccurs.
- especially useful if the same element type is used several times.

Example: reusing a type definition: percentage-property

```
<xs:complexType name="percentage-property">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute ref="percentage"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
<xs:element name="ethnicgroup" type="percentage-property"/>
<xs:element name="religion" type="percentage-property"/>
<xs:element name="language" type="percentage-property"/>
<xs:element name="country">
  <xs:complexType>
    <xs:sequence>
      <xs:element minOccurs="1" maxOccurs="unbounded" ref="name"/>
        :
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="ethnicgroup"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="religion"/>
      <xs:element minOccurs="0" maxOccurs="unbounded" ref="language"/>
        :
    </xs:sequence>
    <xs:attribute name="car_code" type="xs:ID"/>
      :
  </xs:complexType>
</xs:element>
```


AUFRAEUMEN

- Instead of
  <xs:element name="*name*" type="*typename*"/>
  <xs:attribute name="*name*" type="*typename*"/>
  anonymous, local type definitions in the content of such elements are allowed:

```
<xs:complexType name="city">
  <xs:sequence>
    <xs:element name="name" type="xs:string" maxOccurs="unbounded"/>
    <xs:element name="population" minOccurs="0" maxOccurs="unbounded">
      <xs:simpleContent>
        <xs:extension base="xs:nonNegativeInteger">
          <xs:attribute name="year" type="xs:nonNegativeInteger">
        </xs:extension>
      </xs:simpleContent>
    </xs:element>
  </xs:sequence>
  <xs:attribute name="country" type="xs:IDREF"/>
</xs:complexType>
```


Local Declarations

- <complexType> declarations define local *symbol spaces*, i.e., the same attribute/element names can be used in different complex datatypes with different specifications of result-datatypes (this is not possible in DTDs; cf. country/population and city/population elements)

Using global types:

<xs:complexType name="countrypop"> ... without @year ... </xs:complexType>

<xs:complexType name="citypop"> ... with @year ... </xs:complexType>

```
<xs:complexType name="countryType">
  :
  <xs:element name="population" type="countrypop"/>
</xs:complexType>
```

```
<xs:complexType name="cityType">
  :
  <xs:element name="population" type="citypop"/>
</xs:complexType>
```

Local Declarations (Cont'd)

Using local "population" types:

```
<xs:complexType name="countryType">
   <xs:complexType name="pop"> ... without @year ... </xs:complexType>
     :
   <xs:element name="population" type="pop"/>
</xs:complexType>
```

```
<xs:complexType name="cityType">
   <xs:complexType name="pop"> ... with @year ... </xs:complexType>
     :
   <xs:element name="population" type="pop"/>
</xs:complexType>
```


Attribute Groups

Groups of attributes that are used several times can be defined, named and then reused:

```
<xs:attributeGroup name="groupname">
   attributedefs
</xs:attributeGroup>
```

```
<xs:complexType name="name" ...>
   :
   <xs:attributeGroup ref="groupname"/>
</xs:complexType>
```

- group definitions can also be nested ...

Content Model Groups

In the same way, parts of the content model can be predefined:

```
<xs:group name="groupname">
   modelgroupdef
</xs:group>
```

```
<xs:complexType name="name" ...>
   :
   <xs:group ref="groupname"/>
</xs:complexType>
```

Exercise 9.1
Use the following group definitions in your MONDIAL schema:

- an attribute group for (country, province) in city, lake, mountain etc.
- a content model group for (latitude, longitude)

□

Practical Issues: xsi:schemaLocation

In addition to use separate separate .xsd and .xml files (call e.g. saxonXSD bla.xml bla.xsd), the XML Schema can be identified in the XML instance:

- simple things without namespace: the xsi:noNamespaceSchemaLocation attribute gives the URI or local file path of the XML Schema file:

```
<mondial xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="mondial.xsd"> ... </mondial> <!-- local -->
<mondial xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://.../mondial.xsd"> ... </mondial>
```

- when a namespace is used: declare the namespace, and the xsi:schemaLocation attribute is of the form xsi:schemaLocation="*namespace uri-of-xsd-file*":

```
<mon:mondial xmlns:mon="http://www.semwebtech.org/Mondial"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.semwebtech.org/Mondial
                      http://www.semwebtech.org/Mondial/mondial.xsd">
  ... </mon:mondial>
```

- if a document uses several namespaces, several xsi:schemaLocations gan be given; also inside of inner elements.

9.6 Integrity Constraints

XML Schema supports three further kinds of integrity constraints (*identity constraints*):

- unique, key, keyref

that have *very* strong similarities with the corresponding SQL-concepts:

- a name,
- a *selector*: an XPath expression, e.g. //city, that describes the set of elements for which the condition is specified (stronger than SQL: relative to the instance of the element type where the spec is a child of),
- a list of fields (relative to the result of the selector), that are subject to the condition,
- for keyref: the name of a key definition that describes the corresponding referenced key.

More expressive than ID/IDREF:

- not only document-wide keys, but can be restricted to a set of nodes (by type, and by subtree),
- multiple fields; can not only contain attributes, but also (textual) element content,
- but not applicable to IDREFS (then, e.g., “D NL B ...” would be seen as a single value).

Integrity Constraints

- are subelements of an element type. The scope of them is then each instance of that element type (e.g., allows for having a key amongst all cities of a given country, and keyrefs in that country only referring to such cities)
- document-wide: define them for the root element type.

```
<xs:unique name=“...” >
   <xs:selector xpath=“xpath-expr”/>
   <xs:field xpath=“xpath-expr”/> ··· <xs:field xpath=“xpath-expr”/>
</xs:unique>

<xs:key name=“ name ” >
   <xs:selector xpath=“xpath-expr”/>
   <xs:field xpath=“xpath-expr”/> ··· <xs:field xpath=“xpath-expr”/>
</xs:key>

<xs:keyref name=“...”  refer=“ name ” >
     <xs:selector xpath=“xpath-expr”/>
     <xs:field xpath=“xpath-expr”/> ··· <xs:field xpath=“xpath-expr”/>
</xs:keyref>
```


Integrity Constraints: Example

```
<xs:element name="mondial">
  <xs:complexType>
    <xs:element ref="country" maxOccurs="*"/>
     :      <!-- with <border country="..."/> subelements -->
  </xs:complexType>

  <xs:key name="countrykey"> <-- key amongst all countries -->
    <xs:selector xpath="country"/>  <!-- range: unique amongst all countries -->
    <xs:field xpath="@car_code"/>   <!-- is the field @car_code -->
  </xs:key>

  <xs:keyref name="bordertocountry" refer="countrykey">
    <xs:selector xpath=".//border"/> <!-- for all border elements, -->
    <xs:field xpath="@country"/>     <!-- the @country attr refs to a country -->
  </xs:keyref>
</xs:element>
```


Integrity Constraints: Local Keys Example

```
<?xml version="1.0" encoding="UTF-8"?>
<countries-and-cities
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="keys.xsd">
  <country code="D">
   <neighbor code="A"/>
   <neighbor code="CH"/>
   <county code="FR" name="Freiburg">
    <neighbor code="VS"/>
    <city>Freiburg</city>
   </county>
   <county code="VS" name="Villingen-Schwenningen">
    <neighbor code="FR"/>
    <city>Villingen</city>
   </county>
   <county code="D" name="Duesseldorf"/>
  </country>
  <country code="CH">
   <neighbor code="D"/>
   <neighbor code="A"/>
   <county code="FR" name="Fribourg">
    <neighbor code="VS"/>
    <city>Fribourg</city>
   </county>
   <county code="VS" name="Valais/Wallis">
    <neighbor code="FR"/>
    <city>Sion</city>
   </county>
   <county code="VD" name="Vaud/Waadt">
    <neighbor code="FR"/>
    <neighbor code="VS"/>
   </county>
  </country>
  <country code="A"/>
</countries-and-cities>
```

[Filename: XMLSchema/keys.xml]

Integrity Constraints: Local Keys Example (Cont'd)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="countries-and-cities">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="country" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
   <xs:key name="countrykey"> <!-- all countries -->
    <xs:selector xpath=".//country"/>
    <xs:field xpath="@code"/>
   </xs:key>
   <xs:keyref name="neighbortocountry" refer="countrykey">
    <xs:selector xpath=".//country/neighbor"/>
    <xs:field xpath="@code"/>
   </xs:keyref>
  </xs:element>
  <xs:element name="country">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="neighbor" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element ref="county" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="code" type="xs:string"/>
   </xs:complexType>
   <xs:key name="countykey">             <!-- key is local to the country -->
    <xs:selector xpath=".//county"/>
    <xs:field xpath="@code"/>
   </xs:key>
   <xs:keyref name="neighbortocounty" refer="countykey"> <!-- local in the country -->
    <xs:selector xpath=".//county/neighbor"/>
    <xs:field xpath="@code"/>
   </xs:keyref>
  </xs:element>
  <xs:element name="county">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="neighbor" minOccurs="0" maxOccurs="unbounded"/>
     <xs:element ref="city" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="code" type="xs:string"/>
    <xs:attribute name="name" type="xs:string"/>
   </xs:complexType>
  </xs:element>
  <xs:element name="city" type="xs:string"/>
  <xs:element name="neighbor">
   <xs:complexType>
     <xs:attribute name="code" type="xs:string"/>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

[Filename: XMLSchema/keys.xsd]

Use of Key/Keyref

- allows for local keys and multi-field keys.
- queries can only be stated by joins (as in SQL); then local keys are only of limited use.
- no multivalued keyrefs as in IDREFS. Each reference must be in a separate subelement.

General Assertions

Assertions between attributes and/or subelements on instances of an element type:

- children of a complexType declaration,
- <xs:assert test="*xpath-based test*"/>

9.7 Applications and Usage of XML Schema

- (simple) datatype arithmetics and reasoning (numbers, dates)
  The simpleTypes and restrictions are also used in the Semantic Web languages RDF/RDFS/OWL.
- specification of allowed structure: validation of documents

The information about a class of documents is also often used:

- derive an efficient mapping to relational storage (cf. Slide 688)
- definition of indexes (over keys and other properties)
- the type definitions can be used to derive corresponding Java Interfaces (JAXB; cf. Slide 545)
  - get/set methods for properties,
  - automatical mapping between Java and XML serialization,
  - classes that add behavior can then be programmed by extending the interfaces.

Chapter 10
XPointer and XLink

- Considered up to now: XML as a data model, data representation, queries
- only single, isolated documents

World Wide Web: References between documents

- links in HTML: point to a document, sometimes to an *anchor* in a document:
  `http://user.informatik.uni-goettingen.de/~may/Mondial/mondial.html#XML`
- browser: when clicking on a link, something happens.

World Wide XML Web? - finally, not. [⇒ This chapter can be skipped]

- basic definitions 1997-2001
- never widely used, no real implementations (not supported by XQuery)
- XLink – who cares? `http://www.xml.com/pub/a/2002/03/13/xlink.html`
- For *Linked (open) Data*, RDF (1999) is used instead (since ∼ 2005)

- a language for expressing references: XPointer
- a language for specifying the semantics of references: XLink

10.1 XPointer

- links in HTML: point to a document, sometimes to an *anchor* in a document:
  `http://user.informatik.uni-goettingen.de/~may/Mondial/mondial.html#XML`
  (the target must be prepared in the remote document with `<A name="XML"/>`),
- Goal in XML: express a pointer to *something* in another XML document.
- possibilities to address individual elements, attributes, or also characters in an XML document:
  - element-, attribute-, comment-, processing instruction nodes,
  - all "information" that can be selected on the monitor by "mousing" can also be addressed by an XPointer.
    (independent from borders of elements – can start in the middle of an element and end in the middle of another element).
  - each point directly before or after an element can be addressed.

XPointer

- XPointer is a *semantical*, not a syntactical (wrt. the target document) concept. XPointers must be transparent against mechanical changes in the target document (i.e., not "point to the 3rd character in the 6th line in the browser").
- extends the URL concept with the use of XPath:

  XPointer = *url#xpointer-expr*

  http://.../Mondial/mondial.xml#xpointer(descendant::country[@car_code="D"])
- "shorthand pointer": *url#id*
  – as in HTML: <a name="bla"> and <a href="http://filename#bla">
  addresses the element that has *id* as its ID-value
  (DTD: value of an attribute declared as ID)
- full form – "xpointer scheme" (there are also other schemes):

  url#xpointer(*xpointer-expr*)
- For this, XPath is extended with some constructs.
- alternative: element() scheme, e.g. element(D), element(/1/4/3), element(D/8/3)
  (last: third child of the eight child of the element identified by "D")

XPointer

- every XPath epression is also an XPointer expression
- *xpath-expr*$_1$/range-to(*xpath-expr*$_2$) is a pointer, that selects an area in an XML document:

  mondial.xml#xpointer(//country[name="Germany"]/city[1]/range-to(../city[6]))

  selects the area from the 1st to the 6th city of Germany in mondial.xml.
  (*not* as set of nodes, but as an area. This can e.g. include changing from one province element to another).
- string-range(*xpath-expr*, *string*, m, n) selects sequences of characters in documents: for each result of *xpath-expr*, the first occurrence of *string* is searched, and the characters from positions m-n are "referenced".
  Markup is ignored in this sequence (including attribute values!)

Remark: since we speak about pointers, the result is not a fragment of an XML document, but simply two positions in a document!

XPointer: Examples

- Addressing via the id-function:
  mondial.xml#xpointer(id("D"))
  shorthand: mondial.xml#D
  - robust against changes in the XML document structure,
  - requires knowledge about the schema definition (ID-declaration)
- "object-oriented" addressing via semantic "keys":
  mondial.xml#xpointer(//country[name="Germany"])
  mondial.xml#xpointer(//country[name="Germany"]//city[name="Berlin"])

10.2 XLink: World Wide Linking

- extended possibility for specifying hyperlinks.
- relationships between resources (documents, elements, ...)
  resources can also be programs, images, movies, etc.
- – Language: "XLink"
  – Namespace xlink:
- uses (naturally) XPointer

Requirement Analysis

- What "kinds" of references are needed?
  Is the functionality of HTML's <a>-tag sufficient?
- semantics of references?
  click? and then?
- ... up to now, XLink is officially only investigated for browsing applications.

Semantics of Existing Reference Types: HTML

HTML: <A HREF="*url#anchor*">

- specified in the source document, unidirectional, only one target,
- either the whole page, or to a predefined anchor.
- behavior?
  - standard: when clicked, the target page is shown in the current window.
    user-activated, "replace"
  - alternative: when clicked, the target page is shown in a new window.
    user-activated, "new"
  - alternative: instead of building up a page, another page is shown in the current window (forwarding)
    automatically activated, "replace"
  - alternative: when building up a page in the browser, other pages are shown in small, separate windows
    automatically activated, "new"

... sufficient for clicking/browsing, but not for a data model.

HTML: <IMG SRC="*url*"/>

... is also a "link"!

- specified in the source document, unidirectional, only one non-HTML/XML target,
- behavior?
  - standard: when the page is loaded, the image is embedded at the given position.
    automatically activated, "embed"
  - alternative: when building up a page in the browser, show pictures in small, separate windows
    automatically activated, "new"

Semantics of existing Reference Types: ID/IDREF

ID/IDREF/IDREFS is already a reference mechanism in XML: Simplest kind of references *inside an XML document*:

- unidirectional, internal to the document, one or more targets
- "Activation"?
  ... when a query is executed (dereferencing; "user-activated")

... insufficient for a data model, useless for clicking ...

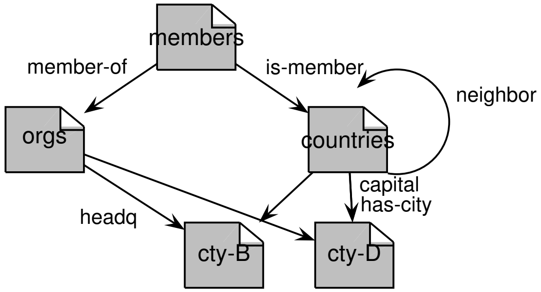
Example-Scenarios

World-Wide-Web

- Web pages
- Hyperlinks
- other kinds of relationships between Web pages

Storage of XML Data in XML (Mondial)

- Distribution over multiple documents
  - countries.xml
  - cities-*car-code*.xml (cities and provinces of each country)
  - organizations.xml
  - memberships.xml



XLink: Basic Notions

Resources: XML documents, parts of XML documents, HTML pages, images, movies, Web services ...

- local resource: a resource that belongs as a structure to the content of the XLink element itself (or that is the link itself)
- remote resource: a resource that is given by a URI

Examples

- `<a href='http://www.goettingen.de'>Göttingen</a>` is a simple link: connects the (local) resource “Göttingen” (string to be clicked) with a (remote) resource located at the URL `www.goettingen.de` (Web page).
- `<img src='...'/>` is an even simpler link: has no local resource, but points only to a remote one

XLink: Basic Notions (Cont'd)

Arcs: directed connections between resources (starting point $\rightarrow$ endpoint)

- outbound: the starting point is a local resource, the end is a remote resource.
  - `<a href="..."> ...</a>`,
  - country-capital-relationship: a country element is the local resource, and city element is the other, remote, resource.
- inbound: the starting point is a remote resource, the endpoint is a local one. Inbound-arcs cannot be represented in the same document as their starting point.
- third-party: starting point and endpoint are remote resources.
  - e.g. own linkbase over the Web: each link connects two remote resources (an area of an HTML document with another URL).
  - e.g. memberships of countries in organizations:
    - each link connects two remote resources, a country and an organisation
    - n:m-relationship ... see later

XLink: Kinds of Links and their Semantics

XLink offers a meta-semantics for describing references that is then used by applications.

- different kinds of references
  - simple: like `<a href='...'>...</a>` or `<img src='filename'/>`
  - links to multiple targets/resources/documents
    activate several resources at the same time
    DB: a country has several cities
  - the links described above are *inline*-links, i.e., contained in the document itself (outbound arcs).
  - *out-of-line*-links: a user can define connections between (sets of) documents that are owned by somebody else (third-party arcs).
    “overlay” own hyperlinks for clicking over the Web
    DB: connections between countries and organizations
- timepoint of activation (onLoad, onRequest)
- action (new, replace, embed)

XLINK ELEMENTS

- Element- and attribute names from the *xlink:* namespace
- *Each* element can become a link ...
- ... by adding an *xlink:type* attribute having one of the values defined by XLink, the element is assigned XLink functionality.
- Properties and substructures (chosen from a predefined set of XLink behavior) can then be specified.

```
<!ELEMENT linkelement (contentmodel)>
<!ATTLIST linkelement
      xmlns:xlink   CDATA   #FIXED   "http://www.w3.org/1999/xlink"
      xlink:type    (simple|extended|locator|resource|arc)   #FIXED   "..."
      ... >
```

- different possibilities for the *content model* (depending on xlink:type, some things are required).
- additional attributes (depending on xlink:type)

XLINK ELEMENT TYPES

- Definition of arbitrary *link element* types,
- XLink defines 5 basic types of element types:
  - simple: similar to what is known from HTML's <A href="...">,
  - extended: multiple targets,
  - locator: is used in extended links for specifying individual remote resources,
  - resource: is used in extended links for specifying individual local resources,
  - arc: is used in extended links for specifying connections between locator- and resource elements.

XLINK -ELEMENTS AND THEIR ATTRIBUTES

Structure

- xlink:type: chooses a link type,
- xlink:href: specification of target(s) (for simple or locator elements)
  (note that an XPointer can specify multiple targets)
- xlink:label: give names for locators and resources.
- xlink:from, xlink:to: for arcs – references to an xlink:label.

Behavior

- xlink:actuate: specifies, when the link is "activated":
  - onLoad: when loading/parsing the XML document,
  - onRequest: only when the user does something (e.g. clicking).
- xlink:show: specifies what happens when the link is activated:
  - new: target appears in a new window,
  - replace: the current document is replaced by the target,
  - embed: the target is embedded into the current document.

XLINK ELEMENTS AND THEIR ATTRIBUTES (CONT'D)

Further Attributes (application-specific)

- xlink:title: the user can give a comment about the target.
- xlink:role and xlink:arcrole: allows to group links to roles, e.g., xlink:role="Annotation" for giving additional information to the application.
- xlink:arcrole is mainly used for annotations for mappings to RDF (Resource Description Framework):

  object has property *rolename value*

  (RDF is e.g. used in the Semantic Web)

Simple Links

- By setting xlink:type to “simple”, a simple link analogous to the <A>-tag in HTML is defined.
- the xlink:href attribute specifies the target
- xlink:actuate and xlink:show specify the behavior.

Example: <A>-Element in HTML (known behavior)

```
<!ELEMENT A ANY>
<!ATTLIST A xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink"
    xlink:type (simple|extended|locator|resource|arc) #FIXED "simple"
    xlink:href CDATA #REQUIRED
    xlink:actuate (onLoad|onRequest) "onRequest"
    xlink:show (new|embed|replace) "new">
```

A sample element that embeds an HTML fragment when clicking on it:

```
For getting the whole list, please click
<A xlink:href="liste-fragment.html" xlink:show="embed">here</A>.
```


Simple Link: Example

- country/capital as simple link

```
<!ELEMENT country (name,population*, ..., capital,cities, ...)>
<!ELEMENT capital EMPTY>
<!ATTLIST capital xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink"
                  xlink:type (simple|extended|locator|resource|arc)
                        #FIXED "simple"
                  xlink:href CDATA #REQUIRED>
```

In the XML document:

```
<country car_code="D" area="356910">
  <name>Germany</name>
    :
  <capital xlink:type="simple"
           xlink:href="cities-D.xml#xpointer(//city[name='Berlin'])"/>
</country>
```

- the (empty) “capital” element is the local resource, the link is specified between country/capital and Berlin.

Exercise: Define the HTML <IMG> element with XLink.

XLink: Inline Extended Links

- Extended links can contain multiple XPointers,
- they have no own href attribute,
- the specification of the targets is done by locator subelements and resource subelements.

<!ELEMENT *linkelement* (... *locatorelement** ... *resourceelement** ...)>
<!ATTLIST *linkelement*
    xmlns:xlink ... ...
    xlink:type (simple|extended|locator|resource|arc) #FIXED “extended” >

<!ELEMENT *locatorelement* (*contentmodel*)>
<!ATTLIST *locatorelement*
    xmlns:xlink ... ...
    xlink:type (simple|extended|locator|resource|arc) #FIXED “locator”
    xlink:href CDATA #REQUIRED >

resourceelement: any element can be made a local resource by xlink:type=“resource”.

Example: Extended Link in the WWW

Pictures in the Web:

```
<!ELEMENT a-loc EMPTY>
<!ATTLIST a-loc xlink:type (...) FIXED "locator"
                xlink:href CDATA>
<ul> <li> <ext-a xlink:type="extended">Goettingen
             <a-loc xlink:type="locator" xlink:href="goe1.jpg">
             <a-loc xlink:type="locator" xlink:href="goe2.jpg">
             <a-loc xlink:type="locator" xlink:href="goe3.jpg">
          </ext-a>
     <li> <ext-a xlink:type="extended">Freiburg
             <bla-loc xlink:type="locator" xlink:href="fr1.jpg">
             <bla-loc xlink:type="locator" xlink:href="fr2.jpg">
          </ext-a>
</ul>
```

when the user (in an XLink-enabled browser) clicks on such an <ext-a> element, all corresponding pictures are shown in separate new windows.

Example: Extended Link in Mondial

The country elements can also be regarded as extended links:

- each country element as a whole is a local resource
- with own resources: name etc.
- with references (locators) to other resources.

```
<!ELEMENT country (name,...,capital,encompassed*,border*,cities)>
<!ATTLIST country car_code ID #IMPLIED
                :
                xlink:type CDATA #FIXED 'extended'>
<!ELEMENT capital EMPTY>
<!ATTLIST capital xlink:type CDATA #FIXED 'locator'
                xlink:href CDATA #REQUIRED >
```

Analogously for encompassed, border, provinces, cities.

Exercise 10.1
Give a DTD for these elements (note that encompassed and border have further attributes). □

Example (cont'd)

XML instance:

```
<country xlink:type="extended"
         car_code="D" area="356910" ...>
  <capital xlink:type="locator"
           xlink:href="cities-D.xml#xpointer(//city[name='Berlin'])"/>
  <cities xlink:type="locator"
           xlink:href="cities-D.xml#xpointer(//city)"/>
    :
</country>
```

Exercise 10.2
Complete the XML instance. □

Example (cont'd)

- the attributes xlink:show and xlink:actuate are not relevant here
- application: not browsing, but queries, transformations etc.
- not considered neither in XPath/XQuery nor in XLink.
- These aspects are investigated in the LinXIS project:
  `http://www.dbis.informatik.uni-goettingen/LinXIS`
  (see later)

- up to now: only outbound links
  – country/capital is an implicit outbound-arc from the local resource (country) to a city.
  – country/cities references multiple targets (city elements), defines multiple outbound arcs.

XLink: Out-of-Line-Links

Link elements that are not in the document, but in separate documents (i.e., possible to "add" links to other people's documents):

- expressed by extended link elements with locators and resources;
  these are equipped with an xlink:label attribute.
- in addition to the locator elements (that address the (remote) resources), additional information must be stored:
  – which resources are connected by an arc,
  – and the direction of the connection.

⇒ additional *arc* elements
connect resources/locators by xlink:from and xlink:to attributes.

XLink: Out-of-Line-Links

- element content allows for subelements of locator element types (as above) and subelements of arc element types that describe relationships between locator elements:

```
<!ELEMENT linkelement  ((locatorelement*|resourceelement*|arcelement)*)>
<!ATTLIST linkelement as above >
<!ELEMENT locatorelement as above >
<!ATTLIST locatorelement as above
     type, href, label: NMTOKEN , title, role >
<!ELEMENT arcelement (contentmodel)>
<!ATTLIST arcelement
     xmlns:xlink  CDATA   #FIXED   "http://www.w3.org/1999/xlink"
     xlink:type   (simple|extended|locator|resource|arc)  #FIXED "arc"
     xlink:from   NMTOKEN  #IMPLIED
     xlink:to     NMTOKEN  #IMPLIED
     xlink:arcrole  CDATA  #IMPLIED  >
```

- Note: xlink:label is not an ID; there can be several elements with the same label.

Example Out-of-Line-Links: Web

Adding an own link base in addition to the `<a href="..">`-entries in HTML Web documents:

- two kinds of locator elements:
  1. XPointers on keywords and sentences where the link should start from:
     ```
     <univis xlink:type="locator" xlink:label="univis-ssdxml"
       xlink:href="http://www.univis.goe/somepath#xpointer(//h1[text()='...'])"/>
     ```
  2. URLs of target pages:
     ```
     <teaching xlink:type="locator" xlink:label="ssdxml-homepage"
       xlink:href="http://www.cs.uni-goettingen.de/teaching/ssdxml.html"/>
     ```
- Arc elements:
  ```
  <arc xlink:type="arc"
       xlink:from="univis-ssdxml" xlink:to="ssdxml-homepage"/>
  ```
- when using an XLink-enabled browser that has access to the link base, there is a link on the Univis page from the headline to the XML lecture.

Example Out-of-Line-Links: Mondial

Memberships of countries in organizations:

```
<!ELEMENT memberships (country*,organization*,membership*)>
<!ATTLIST memberships xlink:type (...)  #FIXED "extended"  >

<!ELEMENT country EMPTY>
<!ATTLIST country xlink:type     (...)  #FIXED "locator"
                  xlink:href     CDATA  #REQUIRED
                  xlink:label    NMTOKEN #REQUIRED  >

<!ELEMENT organization EMPTY>
<!ATTLIST organization xlink:type    (...)  #FIXED "locator"
                       xlink:href    CDATA  #REQUIRED
                       xlink:label   NMTOKEN #REQUIRED  >

<!ELEMENT membership EMPTY>
<!ATTLIST membership xlink:type   (...)  #FIXED "arc"
                     xlink:from         NMTOKEN  #IMPLIED
                     xlink:to           NMTOKEN  #IMPLIED
                     membership_type CDATA  #REQUIRED  >
```


Example Out-of-Line-Links: Mondial

```
<memberships>
  <country xlink:label="B"  xlink:type="locator"
     xlink:href="../countries.xml#xpointer(//country[@car_code='B'])"/>
  <country xlink:label="D"  xlink:type="locator"
     xlink:href="…/countries.xml#xpointer(//country[@car_code='D'])"/>
  <organization xlink:label="org-EU"  xlink:type="locator"
     xlink:href="…/organizations.xml#xpointer(//organization[@abbrev='EU'])"/>
  <organization xlink:label="org-UN"  xlink:type="locator"
     xlink:href="…/organizations.xml#xpointer(//organization[@abbrev='UN'])"/>
  <membership xlink:from="B"  xlink:to="org-EU"  xlink:type="arc"
              membership_type="member"/>
  <membership xlink:from="B"  xlink:to="org-UN"  xlink:type="arc"
              membership_type="member"/>
</memberships>
```

Semantics of Arcs

- In case that all xlink:label in an extended link element are unique, each arc element stands for the unique relationship given by the xlink:from and xlink:to attributes.
- In case that the labels are not unique, every arc stands for all relationships between pairs of locators that have the corresponding from- and to-labels.
- an arc that has no xlink:to attribute, stands for a connection to each locator (analogously for from).
- an arc that has neither from nor to stands for all possible relationships.

XLink: Usage

Browsing: obvious. xlink:show and xlink:actuate

- W3C Amaya (http://www.w3.org/Amaya): partially understands XLink and is open-source
  - use XLink for annotations to Web pages (→ RDF).
- queries against XML data sources:
  - The *W3C XML Query Requirements* state that the query language must support queries over references. The XLink/XQuery combination does not (yet) satisfy this.
  - behavior of XPath and XLink has not yet been considered in the W3C documents:
  - there is even no *data model* for XLinks
  - currently: requires real programming for resolving XLink elements and evaluating the references dynamically.

10.3 XInclude: Database-Style Use of XPointer

Include-elements are *replaced* by the corresponding included items:

<xi:include parse= "xml|text" href= "*url*" xpointer= "*xpointer*"/>

- no browsing semantics (XHTML: include must be resolved when loading)
- query/database semantics: obvious

```
<country xlink:type="extended" car_code="D" area="356910" ...>
  <xi:include parse="xml" href="mondial-D-cities.xml" xpointer="//city"/>
</country>
```

becomes

```
<country xlink:type="extended" car_code="D" area="356910" ...>
  <city><name>Berlin</name> ... </city>
  <city><name>Stuttgart</name> ... </city>
  :
</country>
```

- resolve inclusion when loading
- resolve inclusion on-demand when querying

10.4 The LinXIS Project – Linked XML Information Sources

Research project (2001–2006) in the DBIS group at Göttingen:

... combine XLink with XInclude-style functionality

- extend XLink with data model semantics: insert referenced targets
- dbxlink: namespace for specifying the data model

```
<country car_code="D" area="356910" ...>
 <capital xlink:type="simple" dbxlink:transparent="keep-element insert-contents"
      xlink:href="cities-D.xml#xpointer(//city[name='Berlin'])"/>
</country>
```

```
<country car_code="D" area="356910" ...>
 <capital><name>Berlin</name><population>3472009</population></capital>
</country>
```

- query: `document(countries.xml)//country[name="Germany"]/capital/population` resolves the xlink and "jumps" automatically to the target of the reference to `mondial/cities-D.xml#xpointer(//city[name="Berlin"])`
- `http://www.dbis.informatik.uni-goettingen.de/LinXIS`

Chapter 11
Algorithms and APIs

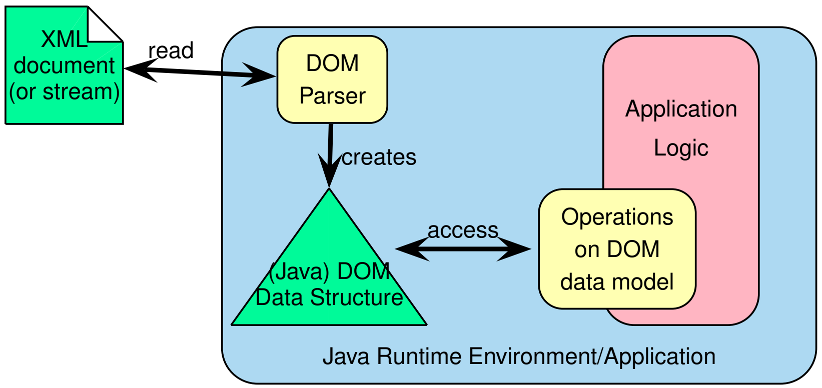
- XML as a data structure:
  - *abstract datatype* with API: DOM
  - (mainly main-memory) implementations; used e.g. in Java applications
  - low-level API with variable-based access
- Databases?
  - high-level API: XPath, XQuery
  - mapping to relational model (Oracle, IBM DB2) or ObjectTypes (Oracle, DB2)
  - “Native” storage: Software AG-Tamino
  - classical database functionality: multiuser, transactions, recovery
  - integration of XML and SQL: SQLX-standard

Algorithms and APIs (Cont’d)

- Stream Processing:
  - XML data transfer as sequence of events
  - SAX (Simple Application Interface for XML), StAX (Streaming API for XML)
- XML as Data Exchange Format in Web Services
  - serialize application objects as XML
  - SOAP: generic [not discussed in this course]
  - JAXB: "model-aware" infrastructure
- an intermediate rule-based concept:
  - apache.commons.digester

11.1 DOM

- DOM (Document Object Model) defines a platform- and language-independent object-oriented *interface* (i.e., an *abstract datatype*) for generating, processing and manipulating XML data.



DOM

- DOM is a specification of an interface/abstract datatype for the XML data model, *not a* data model and *not a* programming language!
- implementations in Java, C++, etc; usually main-memory-based;
  specialized Java interface definitions:
  - recommended for this course: JDOM2: org.jdom2.*, jdom2.jar,
  - original jdom (=jdom1) deprecated (mainly XPath handling changed; 2013),
  - another alternative: dom4j,
  - not recommended: org.w3c.dom.* (the plain dom is an implementation that exists in nearly all programming languages and does not make use of Java’s advantages);
- language base of the DOM specification: OMG-IDL
- Main-memory-based:
  - handling *small* XML fragments for data exchange

DOM: PRINCIPLES

- only one document in a single DOM instance
- step-by-step-access to the data:
  based on variable assignments in the surrounding imperative/object-oriented programming language and on iterators (cf. proceeding in the network data model):
  - class "Document": represents the complete document,
    - doctype declaration, getRootElement()
  - class "Node": getNodeType(), getChildren(), getFirstChild(), getNextSibling(), getParentNode(), ...
  - class "Element": getName(), getAttributes(), getContent(), ...
  - class "Attribute": getName(), getValue(), ...
  - corresponding methods for generating and changing nodes.
- additionally, XPath and XSLT can be applied to instances of Document and Element;
- based on DOM, XPath and XQuery can be implemented (cf. Apache Xerces (XML/DOM)/Xalan (C++/Java; XPath 1.0/XSLT 1.0 [in 2016])
- XPath/XSLT often inefficient (no indexes, query optimization), restricted functionality

JDOM – sample code fragment

```
// apt-get install libjdom2-java; add jdom2.jar to the classpath
import java.io.File;
import java.util.List;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.input.SAXBuilder;

public class MondialJDOMSimple {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("../mondial.xml"));
      Element mondial = doc.getRootElement();
      List<Element> countries = mondial.getChildren("country");
      String firstcode = countries.get(0).getAttributeValue("car_code");
      System.out.println(firstcode);
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/JDOM/MondialJDOMSimple.java]

- SAXBuilder can be set on any input stream.
- XMLOutputter([Format])/SAXOutputter

JDOM2: XPath

- similar to JDBC/SQLJ statement concept for SQL in Java:
- Basic: compile Strings into XPath expressions, evaluate to Object or List<Object>,
- Optional: add (type) Filter for result node type,
- Advanced: XPath expression contains variables
  - must be declared as a map of variables (optionally with preset values)
  - call then requires also namespace blabla, see doc,
- XPath handling changed severely from jdom to jdom2 (2013).
- id(), number(), and thus also max(), sum() etc. not supported.

JDOM2: XPath example

```
import java.io.File;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.Attribute;
import org.jdom2.Namespace;
import org.jdom2.input.SAXBuilder;
import org.jdom2.xpath.XPathFactory;
import org.jdom2.xpath.XPathExpression;
import org.jdom2.filter.Filters;

public class MondialJDOMXPath {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("../mondial.xml"));
      Element mondial = doc.getRootElement();
      XPathFactory xpf = XPathFactory.instance();
```

```
    XPathExpression xpath = xpf.compile("//country[@car_code='R']/@area");
    Attribute a = (Attribute) xpath.evaluateFirst(doc);     // -> casting
    int area = Integer.valueOf(a.getValue());
    System.out.println(area);

    xpath = xpf.compile("//country[@area > 7000000]/name");
    List<Object> names = xpath.evaluate(doc);   //  <Object> -> casting
    for (Object n : names) System.out.println(((Element) n).getTextTrim());

    Map<String,Object> vars = new HashMap<String,Object>();
    vars.put("code","D");
    XPathExpression<Element>   // due to filter: result type guaranteed
      xp2 = xpf.compile("//country[@car_code=$code]/population[last()]",
                    Filters.element(), vars, (Namespace[]) null);
    Element res = xp2.evaluateFirst(doc);  // no casting necessary
    int pop = Integer.valueOf(((Element)res).getTextTrim());
    System.out.println(pop);
    xp2.setVariable("code", "F");
    res = (Element) xp2.evaluateFirst(doc);
    pop = Integer.valueOf(((Element)res).getTextTrim());
    System.out.println(pop);
  } catch (Exception e) { e.printStackTrace(); }
}}
```
[Filename: java/JDOM/MondialJDOMXPath.java]

Deprecated: JDOM1 – sample code fragment: XPath

```
public Element getCity(Element country, String provname, String cityname)
{
  Element city = null;
  XPath xpath;
  try
  {  if (provname != null) {
        xpath = XPath.newInstance("./province[name=$pn]/city[name=$cn]");
        xpath.setVariable("pn", provname); }
     else
        xpath = XPath.newInstance(".//city[name=$cn]");
     xpath.setVariable("cn", cityname);
     // xpath.addNamespace(... an instance of Namespace ...);
     city = (Element) xpath.selectSingleNode(country);
  }
  catch (Exception e) {...}
  return city;
}
```


JDOM and XSLT

XSLT: $xsl(xml) \times xml \to xml$

- Simple: class org.jdom2.transform.XSLTransformer (XSLT 1.0 only?)
- More complex: JAXP TrAX classes
  - JDOMSource/JDOMResult as generic interfaces to XML representations
- saxon can/could also be combined with JDOM2 (via a JDOM wrapper class provided by Saxon)

JDOM Output etc

- XMLOutputter
- predefined formats available
- for adding a DTD reference:
  document.setDocType(new DocType("*elementname*", "*dtdfilename*"));


```
import java.io.File;
import org.jdom2.input.SAXBuilder;
import org.jdom2.Document;
import org.jdom2.transform.XSLTransformer;
import org.jdom2.output.XMLOutputter;
import org.jdom2.output.Format;

public class MondialJDOMXSL {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("mondial.xml"));
      XSLTransformer xsltr = new XSLTransformer("../../XSLT/mondial-simple.xsl");
      Document result = xsltr.transform(doc);
      Format fmt = Format.getPrettyFormat();
      fmt.setLineSeparator(System.getProperty("line.separator"));
        // default would be DOS/windows style \r\n
      XMLOutputter outputter = new XMLOutputter(fmt);
      outputter.output(result, System.out);
  } catch (Exception e) { e.printStackTrace(); }
}}
```
[Filename: java/JDOM/MondialJDOMXSL.java]

```
import java.io.File;
import org.jdom2.input.SAXBuilder;
import org.jdom2.Document;
import org.jdom2.transform.JDOMSource;
import javax.xml.transform.*;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class MondialJDOMXSL2 {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("mondial.xml"));
      TransformerFactory factory = TransformerFactory.newInstance();
      Source xsltSource = new StreamSource("../../XSLT/mondial-simple.xsl");
      Transformer transformer = factory.newTransformer(xsltSource);
      StreamResult result = new StreamResult("bla.xml");
      JDOMSource src = new JDOMSource(doc);
      transformer.transform(src, result);
      } catch (Exception e) { e.printStackTrace(); }
} }
```
[Filename: java/JDOM/MondialJDOMXSL2.java]

11.2 HTML vs XHTML in Java

- HTML/XHTML has a DTD that restricts the element types, nesting, and attributes,
- HTML applications know+use a fixed DTD,
- sloppyness/tolerance can exploit that the DTD is known, allow to omit “obvious” things.
- HTML *usage* is less strict than XHTML (cf. parser exercise to the lecture)
  - mainly concerned: HTML parsers in the browsers that should accept handwritten dirty/sloppy HTML.
  - data model tree: intermediate levels (e.g. table: thead, tbody) may be missing browsers “know” how to render it (best-effort)
  - parsing the input character stream:
    - closing tags may be missing, e.g., <tr><td>bla <td>blubb </tr>
    - empty elements like <br> instead of
 (HTML tools know that br-elements are empty)
- XML tools must not accept any sloppyness

HTML to XHTML

- tidy: `https://www.html-tidy.org/`
  console application (→ can be used in shell scripts) C library libtidy
- JSoup: Java HTML DOM Parser and API: `https://jsoup.org/`
  - uses an own (HTML) DOM implementation
    ```
    Document doc = Jsoup.parse(string s; [baseUri for hrefs]);
    Document doc = Jsoup.connect("http://en.wikipedia.org/").get();
    Document doc = Jsoup.parse(File in, String charsetName, String baseUri);
    ```
  - stepwise navigation in the DOM
  - allows simple XPath style selects
    `doc.select("selector pattern");`
  - set/modify content, attributes and text contents
  - serialization: toString() (obviously!)
  - document properties (DTD, output mode, ...) can be set:
    charset, syntax (html/xml), indentAmount, escapeMode (how to handle HTML entities), e.g.
    document.outputSettings().syntax(Document.OutputSettings.Syntax.xml);
  - note: JSoup materializes the whole document tree

JSoup example

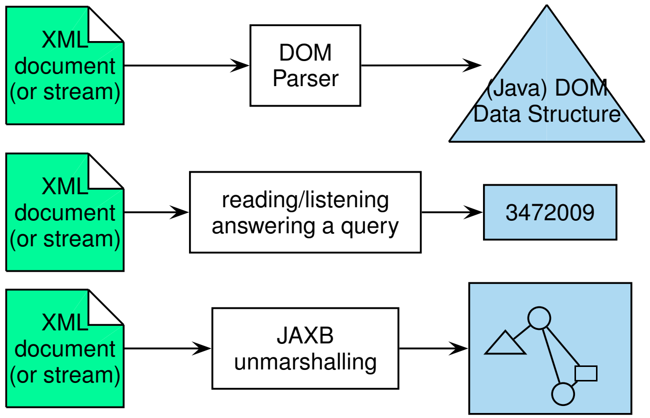
```
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import java.io.*;
public class JSoupTest {
  public static void main(String[] args) {
    try {
      Document doc = Jsoup.connect("http://en.wikipedia.org/wiki/Ball's_Pyramid").get();
      // File input = new File("bla.html");
      // Document doc = Jsoup.parse(input, "iso-8859-1", "");
      doc.outputSettings().syntax(Document.OutputSettings.Syntax.xml);
      String xml = doc.toString();
      BufferedWriter writer = new BufferedWriter(new FileWriter("jsoupoutput.xml"));
      writer.write(xml);
      writer.close();
      } catch (Exception e) { e.printStackTrace(); }
} }
```
[Filename: java/JSoupTest.java]

11.3 XQJ: XQuery API for Java

- similar to JDBC for “SQL in Java”
- access to a remote XML DB: open a connection, send an XQuery query, receive an iterable set of results.
- Tutorial: `https://www.progress.com/tutorials/xquery/api-for-java-xqj`
- saxonEE (commercial version) implements XQJ (although it’s not a remote XML DB server then)

11.4 XML Stream Processing

- reading from a file or from an HTTP connection both is actually reading char by char from a stream
- the stream is parsed, resulting in something that can be used by application:



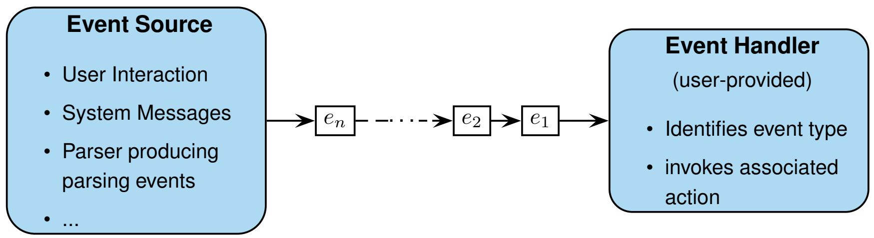
PARSING: GENERAL CONCEPTS

- Compiler Construction in general:
  1. input: a sequence of characters
  2. lexical analysis ("lexer") extracts the keywords (e.g. “begin”, “end”, “for”) and outputs a sequence of *tokens*
  3. syntactical (grammar) analysis: check grammatical structure and generate the *parse tree* (e.g. via automaton)
  4. tools: lex & yacc/bison
  5. interpreter, optimizer, compiler, visualizer etc. process the parse tree
- XML:
  1. lexical: split unicode input sequence into opening tags, closing tags, attributes, PCDATA, processing instructions, etc.
  2. syntactical and structural: is it well-formed XML? (if not: “debug” it?)
  3. processing: build DOM tree, build JAXB structure, visualize, ...
- the above DOM and JAXB are actually parser+specific processing

⇒ XML Stream Processing: works on the tokens sequence!

EVENT-BASED PROCESSING AS A *General Design Pattern*

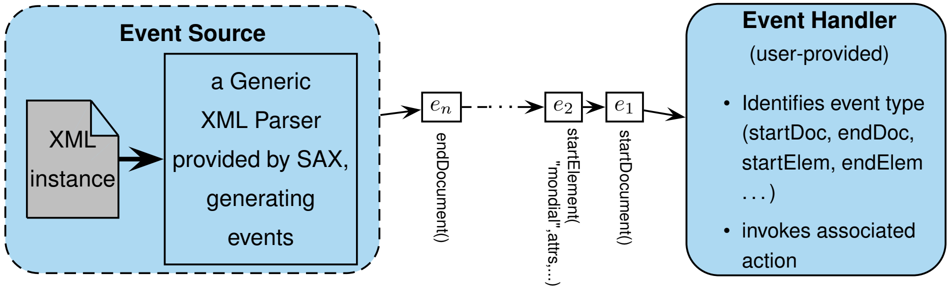
- A stream of (high-level) items that carry some inherent semantics can be seen as a stream of “events”
  (in contrast to a simple 0-1-stream, a byte stream or similar low-level streams)



- The application programmer provides the Event Handler implementation, containing actions for each type of event;
- kind of *rule-based*;
- programmer is *not* in charge of the control flow

11.5 Event-based XML Parsing with SAX

- SAX (“The Simple API for XML”) is an *event-based interface/model*



Represents/processes an XML document as a sequence of events (depth-first traversal), e.g.

- startDocument(), endDocument()
- startElement(Name, attributesList) – attributes not split
- endElement(Name)
- characters(string)

XML Parsing with SAX

SAX: parse XML tokens from a character stream (e.g., a file)

- import classes: javax.xml.parsers.*, org.xml.sax.*
- SAXParserFactory (configurable with setValidating(.), setSchema(.)) creates a generic SAX XML Parser which is then called to parse and to feed a *Content Handler* (plus *Error Handler*, *DTD Handler*, and *EntityResolver*) implementation.
- The most trivial Content Handler is the *DefaultHandler* that does nothing: the document is parsed, events are detected, but no action is performed
- Content (Event) handler programmed wrt. a “*push* API”.
- Normally, the user-provided Content Handler extends the DefaultHandler, overwriting (some of) its event methods.
- With the content handler implementation, the user provides “actions” in form of Java code, associated with specific events (and even dependent on context information).
- If during parsing of the XML document, a specific event occurs, the code of the associated action from the content handler is invoked (*“callback”*).

SAX (and StAX): Applications

Only events are signaled: linear processing based on incoming sequence of events (=tokens).

- ... among many other things, one can generate a DOM tree structure,
- validation according to a DTD (using an automaton as given on Slide 176) in linear time,
  - SAX (and StAX) are only on an intermediate level between character streams and XML
    - ⋆ have lower requirements on input syntax and structure
    - ⋆ input must be allowed XML tokens incl. text
    - ⋆ closing tags may be missing, entities may be undeclared
    - ⇒ allows e.g. for fault-tolerant parsing of sloppy HTML.
- stream-processing of XML input
  - start processing already when input document is not yet complete,
  - filtering for elements that are relevant for a given application,
  - linear search for something, e.g., names of countries,
  - stop evaluation when finished before reading the whole document.
- if necessary: application needs to maintain context.

SAX Example Code

Consider a very simple application that

- detects all elements with attributes,
- for each element, output the element’s name,
- for each element, output the name-value pairs of its attributes,
- end the evaluation when “Göttingen” is found.

```
element: country
 - attribute: 'car_code' value: 'AL' type: 'ID'
 - attribute: 'area' value: '28750' type: 'CDATA'
 - attribute: 'capital' value: 'cty-cid-cia-Albania-Tirane' type: 'IDREF'
 - attribute: 'memberships' value: 'org-BSEC org-CE org-CCC org-ECE org-EU ...' type: 'IDREFS'
element: encompassed
 - attribute: 'continent' value: 'europe' type: 'IDREF'
 - attribute: 'percentage' value: '100' type: 'CDATA'
element: ethnicgroups
 - attribute: 'percentage' value: '3' type: 'CDATA'
element: ethnicgroups
 - attribute: 'percentage' value: '95' type: 'CDATA'

... Göttingen found - ready.
```

```
import java.io.IOException;
import javax.xml.parsers.*;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class PrintAttributes {

  public static void main(String[] args) {
    DefaultHandler handler = new ContentHandlerPrintAttributes();
    SAXParserFactory factory = SAXParserFactory.newInstance();
    try {
          SAXParser parser = factory.newSAXParser();
          parser.parse("file:../mondial.xml", handler);  // <<<<<<<<< and now it runs
    } catch (ContentHandlerPrintAttributes.MySAXTerminatorException e) {
        // exception for smoothly stopping the processing:   <<<<<<<<<<<<<<<
        System.out.println(" ... Göttingen found - ready."); }
      catch (IOException e1) { e1.printStackTrace(); }
      catch (ParserConfigurationException e) { e.printStackTrace(); }
      catch (SAXException e) { e.printStackTrace(); }
}   }
```
[Filename: java/SAX/PrintAttributes.java]


```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class ContentHandlerPrintAttributes extends DefaultHandler {
    // react on opening elements:
    public void startElement(String url, String localName, String qName,
            Attributes attrs) throws SAXException {
        if (attrs.getLength() > 0) {
            String elementName;
            if (qName == null || qName.equals("")) elementName = localName;
            else elementName = qName;
            System.out.println("element: " + elementName);
            for (int i = 0; i < attrs.getLength(); i++) {
                System.out.println(" - attribute: '" + attrs.getQName(i)
                        + "' value: '" + attrs.getValue(i) + "' type: '"
                        + attrs.getType(i)+"'");
            }
            System.out.println();
    }   }
    // methods for endElement(), startDocument(), endDocument() omitted

    // continue next page
```



```
    // continue next page


    public void characters(char[] text, int from, int length)
       throws SAXException  {
       // stop evaluation by throwing an exception:
       String textString = (new String(text)).substring(from, from + length);
       if (textString.contains("Göttingen"))
          throw new MySAXTerminatorException();
    }
    // the exception stub:
    public class MySAXTerminatorException extends SAXException { ; }
}
```
[Filename: java/SAX/ContentHandlerPrintAttributes.java]

- evaluation can only be stopped by raising an exception;
- the events are on the level of structural XML parsing,
  an XML element/subtree consists of several (often: many) events.
- all PCDATA/CDATA values are strings
  → numeric computations require conversion to Java literals or class instances.

SAX: Applications to XPath Query Answering

Forward queries

XPath-queries like //country[@car_code='D']/population[last()] can be answered very (time- and memory-)efficient,

- use the sequence of events (linear)
- maintain some context (often LOGSPACE/additional LOGTIME sufficient)

... works only for queries, that contain only forward steps,

General queries

which XPath expressions can be *transformed* in equivalent forward-expressions (and with what efforts)?

- “XPath: Looking forward”; F. Bry et al ; 2002; LMU München
- theory: complexity, connections to linear temporal logic
  For every linear temporal logic formula that uses past and future operators, there is an equivalent formula that uses only future operators
  ... but in general of exponential size.

11.6 XML Streams/StAX - The Streaming API for XML

Higher abstraction level (than character-based XML) for XML data exchange:
javax.xml.stream (rt.jar)

Reconsider SAX

- on-the fly processing, no in-memory representation for good performance
- idea of "XML Event Stream": a char stream (File, HTTP) can be converted into an XML Event Stream by an XML parser; see example's main() method.
- SAX does not make the XML Event Stream accessible, but only via calls of methods of the Event Handler.

Generalization and Abstraction: XML Streams

- XMLEvent types: StartDocument, DTD, StartElement, Characters, EndElement, EndDocument, (Attribute), (Namespace) ...
- XMLStreamReader, XMLStreamWriter,
  XMLEventReader, XMLEventWriter,
- XML Streams also can be connected *directly* as an *abstract* means to exchange XML

XML Streams: Application Scenarios

- READ: usage analogous to SAX: process an XML file input as an XML Event Input Stream:
  control flow is not passed to the parser (**unlike** *SAX*), but XML events are accessed using an *iterator*, controlled by the Java program using the StAX API (*Pull-API*).
  [Note: iterators are a common design pattern, not only applied to collections, but as we see here also to streams: init(), next(), ...]
  ⇒ application code: same as for SAX, only operational embedding done differently.
- WRITE and READ: streamed data exchange between processed on the XML level.
- Two variants exist:
  - XMLStreamReader, XMLStreamWriter ("Cursor")
  - XMLEventReader, XMLEventWriter ("Iterator")
- XML-S/E-Readers/Writers can be put on any input/output stream (FileInput/OutputStream, BufferedInput/OutputStream, System.out, HTTP stuff (see Web Services) or directly connected to each other:
  XMLS/EWriter->PipedOutputStream->PipedInputStream->XMLS/EReader of the next application)

Interfaces XMLStreamReader, XMLStreamWriter

XMLStreamReader

- int eventtype = r.next() and then switch based on eventtype
  javax.xml.stream.XMLStreamConstants.XX:
  START_DOCUMENT, START_ELEMENT, CHARACTERS, END_ELEMENT, ...
- access methods when on START_DOCUMENT: getEncoding() etc.
- goal-driven access methods on the reader when on START_ELEMENT:
  r.getLocalName(), r.getAttributeValue(*name*),
  r.getAttributeCount(), getAttributeLocalName(*n*), getAttributeValue(*n*) for iteration,
  r.getElementText() (reads also the next EndElement from the stream!),
  getName() (as qname),
  namespace handling: getPrefix(), getNamespaceURI() (default NS),
  getNamespaceURI(prefix),
- goal-driven access method when on CHARACTERS: r.getText(), r.isWhiteSpace();
- goal-driven access methods on the reader when on END_ELEMENT:
  r.getLocalName() + namespace handling
- note again: all PCDATA/CDATA values are strings.

XMLStreamWriter

- w.writeStartDocument()
- w.writeStartElement(*name*),
- w.writeEmptyElement(*name*),
  Note: there is writeEmptyElement(*name*), although for the Reader, there is no event type EMPTY_ELEMENT; instead also for empty Elements, START_ELEMENT and END_ELEMENT are separately read
  ⇒ copying straightly to output will create an non-empty element with ""-content!
- w.writeAttribute(*name*, *value*), (and all three also with namespace handling)
- w.writeCharacters(*text*);
- w.writeEndElement(): closes the innermost open element;
- w.writeEndDocument(): closes all open elements.
- w.flush(): force write any data to the underlying output mechanism.

StAX StreamReader Example

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.FileInputStream;

public class StAXPrintAttributes {
  public static void main(String[] args) {

  try {
    FileInputStream inputStream = new FileInputStream("../mondial.xml");
    XMLInputFactory inputFactory = XMLInputFactory.newInstance();
    XMLStreamReader parser = inputFactory.createXMLStreamReader(inputStream);
    boolean goOn = true;

    while (goOn) {
      int eventtype = parser.next();
      switch(eventtype) {
        case XMLStreamConstants.END_DOCUMENT:
          goOn = false;
          break;   // << break after each case!

        // continue next page
```



```
        // continue next page

        case XMLStreamConstants.START_ELEMENT:
          if (parser.getAttributeCount() > 0) {
            System.out.println(parser.getLocalName());
            for (int i = 0; i < parser.getAttributeCount(); i++) {
           System.out.println(" - attribute: '" + parser.getAttributeLocalName(i)
                                    + "' value: '" + parser.getAttributeValue(i)
                             + "' type: '"  + parser.getAttributeType(i));
            }
          }
          break;
        // cases for endElement(), startDocument(), endDocument() omitted
        case XMLStreamConstants.CHARACTERS:
          String textString = parser.getText();
          if (textString.contains("Göttingen"))
               goOn = false;
      }
    }
    System.out.println(" ... Göttingen found - ready.");
    parser.close();
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/StAX/StAXPrintAttributes.java]

XMLEventReader/XMLEventWriter

- above: XML*Stream*Reader/Writer:
  - XML-parsing level "events" like in SAX
  - the reader is the central object (`r.next()`→`int`, `r.getLocalName(), ...`)
- alternative: XML*Event*Reader/Writer:
  - consider (empty or CDATA) XML Elements as *events* on the application level,
  - XMLEventReader as an *Iterator* over a sequence of events
    (actually, `XMLEventReader extends Iterator { ...}`),
    applicable to pure XML files, but also to incoming HTTP-XML streams (→ Web Services)
    - hasNext() → boolean: check if there are more events.
    - nextEvent() → XMLEvent: get the next XMLEvent
    - getElementText() → String: reads the content of a text-only element.
    - nextTag() → XMLEvent: skips any insignificant space events until a START_ELEMENT or END_ELEMENT is reached. [what about CHARACTERS?]
    - peek()→ XMLEvent: check the next XMLEvent without reading it from the stream.

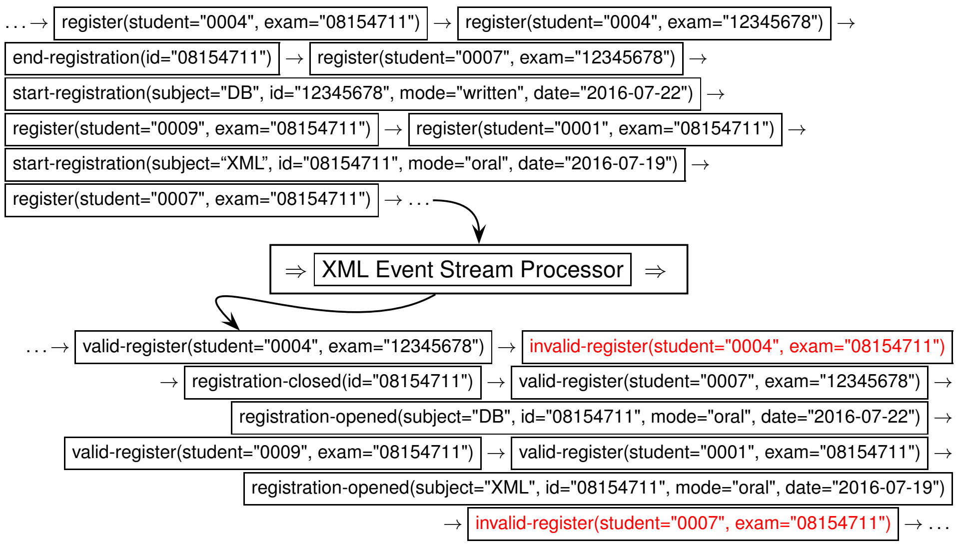
StAX Event Example: Exam Registration

Assume the administration of exams in a student's office ("Prüfungsamt"):

- The *subject* (e.g., "Semi-Structured Data and XML") and ID of lectures/exams,
- whether the exam is *written* or *oral*,
- for written exams, the date of the exam,
- for oral exams, a number of dates is given when the single exams are held.
- the registration period *starts* when receiving an incoming XML message `start-registration`
- the registration period *ends* when receiving an incoming XML message `end-registration`
- for all students that did (`register`) correctly, the student's relevant details are extracted and written to an XMLOutputStream stream (`valid-register`; in the example, we pipe it to `stdout`.)
- students that register before beginning or after the end of registration, are not accounted for the exam; an error message/event `invalid-register` goes to the XMLOutputStream.

StAX Example: Exam Registration

- the program should allow the management of registrations for multiple exams at one time (all incoming over the same continuous input stream).



StAX Example Cont'd:

Consider the following XML sequence as input stream:

```
<?xml version="1.0" encoding="UTF-8"?>
<stream>
  <register student="0007" exam="08154711"/>
  <start-registration id="08154711" subject="Semistructured Data and XML"
     date="2016-07-19" mode="oral"/>
  <register student="0001" exam="08154711"/>
  <register student="0009" exam="08154711"/>
  <start-registration id="12345678" subject="Databases"
     date="2016-07-22" mode="written"/>
  <register student="0007" exam="12345678"/>
  <end-registration id="08154711"/>
  <register student="0004" exam="08154711"/>
  <register student="0004" exam="12345678"/>
</stream>
```

[Filename: java/StAX/exams.xml]

StAX Example Cont'd:

Code for the `Exam` bean, containing the exam's properties and some constants):

```
import java.util.Date;
import java.text.SimpleDateFormat;
import javax.xml.namespace.QName;
import javax.xml.stream.events.StartElement;

public class Exam {
  private String id;     private String subject;
  private String date;   private boolean oral;
  private boolean registeringClosed = false;
  private String startOfReg;   private String endOfReg;

  public Exam(StartElement ev) {  // the <start-registration> element "event"
    this.id = ev.getAttributeByName(new QName("id")).getValue();
    this.subject = ev.getAttributeByName(new QName("subject")).getValue();
    this.oral = "oral".equals(ev.getAttributeByName(new QName("mode")).getValue());
    this.date = ev.getAttributeByName(new QName("date")).getValue();
    this.setStartOfReg(getTodayDate());
  }

  // continue next page
```



```
  public String getId() { return id; }
  public String getDate() { return date; }
  public String getSubject() { return subject; }
  public boolean isOral() { return oral; }
  public boolean isWritten() { return (!oral); }
  public String getMode() {  if (oral) return "oral"; else return "written"; }
  public boolean isRegisteringClosed() { return registeringClosed; }
  public void setRegisteringClosed(boolean registeringClosed) {
    this.registeringClosed = registeringClosed;  }
  public String getEndOfReg() { return endOfReg; }
  public String getStartOfReg() {  return startOfReg; }
  public void setStartOfReg(String startOfReg) { this.startOfReg = startOfReg;  }
  public void setEndOfReg(String endOfReg) { this.endOfReg = endOfReg; }

  public static String getTodayDate() {
    return new SimpleDateFormat().format(new Date());   }
/*  private String getTodayDate() {
    DateFormat format = new SimpleDateFormat().;
    return format.format(new Date());   }
*/
}
```

[Filename: java/StAX/Exam.java]

StAX Example Cont'd:

Code for the main parser class, containing the `main` method:

```
import java.io.File;          import java.io.FileInputStream;
import java.io.OutputStream;
import java.text.DateFormat;   import java.text.SimpleDateFormat;
import java.util.HashMap;      import java.util.Map;
import java.util.Iterator;

import javax.xml.namespace.QName;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.events.XMLEvent;
import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.Attribute;
import javax.xml.stream.events.Characters;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;

// continue next page
```



```
import javax.xml.stream.XMLEventWriter;

// continue next page

public class ExamStreamParser {
  public static void main(String[] args) {
  try{
    FileInputStream in = new FileInputStream("exams.xml");
    OutputStream out = System.out;
    XMLInputFactory inputFactory = XMLInputFactory.newInstance();
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventReader parser = inputFactory.createXMLEventReader(in);
    XMLEventWriter writer = outputFactory.createXMLEventWriter(out);
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
    Map<String,Exam> exams = new HashMap<String,Exam>();
    boolean goOn = true;

    while (goOn) {
      XMLEvent event = parser.nextEvent();
      int eventtype = event.getEventType();
      switch(eventtype) {




      // continue next page
```



```
      // continue next page

        case XMLStreamConstants.START_ELEMENT:
          StartElement ev = (StartElement)(event.asStartElement());
          if("start-registration".equals(ev.getName().getLocalPart())) {
            Exam exam = new Exam(ev);
            exams.put(exam.getId(), exam);
            Iterator<Attribute> attrs = ev.getAttributes();
            StartElement se = eventFactory.createStartElement("", null, "registration-opened",
            writer.add(se);
            writer.add(eventFactory.createEndElement("", null, "registration-opened"));
            writer.add(eventFactory.createCharacters("\n"));
          }
          else if("end-registration".equals(ev.getName().getLocalPart())) {
              String examId = ev.getAttributeByName(new QName("id")).getValue();
            Exam exam = exams.get(examId);
            if(exam == null) {
              System.err.println("no such exam with id '"+examId+"' open for registration!");
              break;
              }
            exam.setEndOfReg(Exam.getTodayDate());
            exam.setRegisteringClosed(true);
            }
          // continue next page
```



```
          // continue next page

          else if("register".equals(ev.getName().getLocalPart())) {
            String studentId = ev.getAttributeByName(new QName("student")).getValue();
            String examId = ev.getAttributeByName(new QName("exam")).getValue();
            if(exams.containsKey(examId)) {
              Exam exam = exams.get(examId);
              if(! exam.isRegisteringClosed()) {
               StartElement se = eventFactory.createStartElement(
                    "", null, "valid-register", ev.getAttributes(), null);
                writer.add(se);
                writer.add(eventFactory.createEndElement("", null, "valid-register"));
                writer.add(eventFactory.createCharacters("\n"));
              }
              else {  //  exam.isRegisteringClosed()
                StartElement se = eventFactory.createStartElement(
                    "", null, "invalid-register", ev.getAttributes(), null);
                writer.add(se);
                writer.add(eventFactory.createCharacters("(reg. ended on " + exam.getEndOfReg(
                writer.add(eventFactory.createEndElement(
                   "", null, "invalid-register"));
                writer.add(eventFactory.createCharacters("\n"));
              }
            }
            // continue next page

            else {  // not (exams.containsKey(examId))
```

```
// continue next page

else {  // not (exams.containsKey(examId))
  StartElement se = eventFactory.createStartElement(
       "", null, "invalid-register", ev.getAttributes(), null);
  writer.add(se);
  writer.add(eventFactory.createCharacters("reg. for exam '"+examId+"' not open"));
  writer.add(eventFactory.createEndElement("", null, "invalid-register"));
  writer.add(eventFactory.createCharacters("\n"));
}
}
break;
case XMLStreamConstants.END_DOCUMENT:
  parser.close();
  writer.flush();
  writer.close();
  goOn = false;
  break;
}
}} catch (Exception e) { e.printStackTrace(); }
}
}
```

[Filename: java/StAX/ExamStreamParser.java]

Some comments on XMLEventReader/Writer

- XMLEventReader/Writer
  - no simple getLocalName()/getAttributeByName(), but only via qnames or getAttributes() as Iterator<Attribute>.
    Note: real event-based applications usually use namespaces.
  - no getAttributeValue(. . .), but only via getAttribute(...).getValue().
  - generates instances of Event class
    - memory-intensive, garbage-collector-intensive
    - instances can be given away to threads for processing
  - For output, also event instances have to be created (use EventFactory).
  - No EmptyElement class - neither for Reader nor Writer.
  - EndElement explicitly needs element name again.

Some notes for both XMLStreamReader and XMLEventReader

- Only XMLStreamWriter has a notion of empty elements:
  - XMLStream/EventReader: empty elements also have an EndElement event;
  - XMLEventWriter: empty elements require to write an explicit EndElement!
- The accessors to attributes differ between XMLStreamReader on START_ELEMENT and XMLEventReader→StartElement.

StAX Comparison with SAX

SAX: • "Push" API

- feeds a single event handler that is given "away" to the parser
- Common pattern: methods for each event type, where startElement() and endElement() contain large `ifs`.

StAX: ... next page ...

StAX Comparison with SAX (Cont'd)

StAX: • "Pull" API

- Common pattern: huge `switch` command whose cases again contain large `ifs`.
- the design as a "pull-interface" where the user has control allows to use Reader.next()/Reader.nextEvent() whenever the programmer wants it:
  - in the "case"-code for StartElement, one can call next() to read the text content immediately for further processing. This saves some booleans.
- Modularization:
  the while(...){... parser.next() ...} loop is parameterized with the parser. For "subtasks", other procedures can be called that have specific implementation of the cases:

```
case XMLStreamConstants.START_ELEMENT:
  if parser.getLocalName.equals("country") {
    Country country = processCountry(parser); // run another while-loop
    // (that first must process the attributes of the START_ELEMENT)
    mondial.add(country);  }
```

Note that such methods can be wrapped as constructor calls like

```
prov = new Province(parser, country)
```

to parse a province into an object of class Province that belongs to an object of class Country. (e.g. to implement JAXB; see Slide 545 ff.)

StAX Comparison with SAX (Cont'd)

- Performance: no difference.
  The underlying XMLStream is the same.
- both can easily produce XML output via `XMLStreamWriter/XMLEventWriter` (e.g. to another SAX/StAX appl.)
- The actual code to be written is not much different in both cases.
- SAX maps a unicode input stream directly to the EventHandler calls.
- StAX makes the intermediate abstraction level of XML event streams accessible.

SAX and StAX: Applications

Stream-based processing can be applied to XML data on multiple levels:

- low-level applications:
  SAX is often used for building a DOM from Unicode XML input: "opening tag with attributes", "text", "closing tag" can immediately be translated into the DOM constructors.
- low-level streaming of an XML instance:
  answering XPath (forward-axes only) queries; optionally maintaining some context (e.g., stack).
- higher level "application-level events":
  the XML stream is not seen as the traversal of a large instance, but as a sequence of (independent) XML fragments that are seen as application-level events
  [RFID applications, time series of stock quotes, RSS feeds]

Example: XML Stream Communication

```
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.io.OutputStream;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class XMLStreamTestWriter implements Runnable
{
  OutputStream outputStream;

  public XMLStreamTestWriter(OutputStream out) {
    this.outputStream = out;
  }

  public void run() {
    try {
      XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
      XMLStreamWriter writer = outputFactory.createXMLStreamWriter(outputStream);
      writer.writeStartElement("foo");
      int i=1;
      while (i<100) {
        writer.writeStartElement("bla");
        writer.writeCharacters(" " + i);
        writer.writeEndElement();
        System.out.print("Write  <bla>" + i + "</bla> ");
        //writer.flush(); // if not uncommented: strictly alternating
        // comment out flush: sleep < 700 causes alternating after blocks of 2...5 elements
        try{ java.lang.Thread.sleep(50); }
          catch (Exception e) { e.printStackTrace(); }
        i++;
      }
      // writer.writeEndElement();  // close </foo> is done by the next line:
      writer.writeEndDocument(); // docu: closes all tags, but does not send anything else
      writer.flush();
      writer.close();
    } catch (Exception e) { e.printStackTrace(); }
    System.out.println("Writer finished");
  }

  public static void main(String[] args) throws Exception{
    PipedOutputStream pos = new PipedOutputStream();
    PipedInputStream pis = new PipedInputStream();
    pis.connect(pos);
    new Thread (new XMLStreamTestWriter(pos)).start();
    new Thread (new XMLStreamTestReader(pis)).start();
  }
}
```

[Filename: java/StAX/XMLStreamTestWriter.java]

- underlying: connected PipedOutput/InputStream

Example: XML Stream Communication (Cont'd)

```
import java.io.PipedInputStream;
import java.io.InputStream;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;

public class XMLStreamTestReader implements Runnable {

  InputStream inputStream;

  public XMLStreamTestReader(PipedInputStream in) {
    this.inputStream = in;
  }

  public void run() {
    try {
      XMLInputFactory inputFactory = XMLInputFactory.newInstance();
      XMLStreamReader parser = inputFactory.createXMLStreamReader(inputStream);
      boolean goOn = true;
      while (goOn) {
        int event = 0;
        try {
          event = parser.next();
          switch(event) {
            case XMLStreamConstants.START_ELEMENT:
              System.out.println("Read start element " + parser.getLocalName());
              break;
            case XMLStreamConstants.CHARACTERS:
              System.out.println("Read " + parser.getText());
              break;
            case XMLStreamConstants.END_ELEMENT:
              System.out.println("Read end element " + parser.getLocalName());
              break;
              case XMLStreamConstants.END_DOCUMENT: // never happens!
              System.out.println("Read end document");
              goOn = false;
            default: System.out.println("Read something else. event: " + event);
          }}
          catch(Exception e) { parser.close(); goOn = false; }
      }
      parser.close();
      System.out.println("Reader finished");
    }
    catch (Exception e) { e.printStackTrace(); }
  }
}
```

[Filename: java/StAX/XMLStreamTestReader.java]

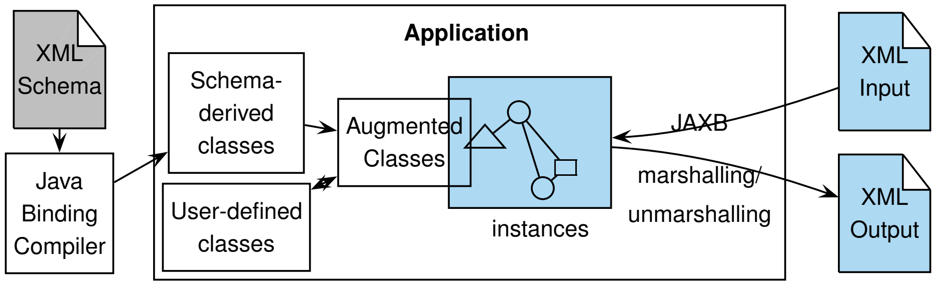
11.7 JAXB - The Java/Jakarta API for XML Binding

- API changes its position within Java from time to time
  - was published first in 2003 as part of the Java Web Services Developer Pack (which was replaced in 2006 by GlassFish)
  - From Java SE 6 until Java SE 10 (Sept. 2018), it was part of Java SE.
  - Part of Jakarta EE (formerly Java Platform, Enterprise Edition (Java EE)) current implementations e.g. `https://javaee.github.io/jaxb-v2/` (2020)
- XML elements describe objects with properties,
- correspond to classes of an application,
- derive interface with setX/getX methods (= Java Beans) as skeletons for these classes (automatically generated from an XML Schema description),
- user derives classes from these interfaces by adding behavior,
- application logics implemented by using these classes,
- import/export of XML instances of these classes via generic mappings (derived from the XSD).

JAXB ARCHITECTURE

- map XML Schemas to Java classes (get/set methods),
- methods for *unmarshalling* XML data (file, DOM instance, javax XMLEvents stream, etc.) into Java objects,
- methods for *marshalling* Java objects back into XML data (DOM instance, SAX, javax XML Events stream, etc.),
- for input/output XML data formats see online documentation.

Architecture



JAXB - EXAMPLE

```
<?xml version="1.0"?>
<BookCollection>
  <books>
    <book isbn="111-1234">
      <name>Learning JAXB</name>
      <price>34</price>
      <authors>
        <authorName>Jane Doe</authorName>
      </authors>
      <language>English</language>
      <language>French</language>
      <promotion>
        <Discount>10% until March 2003</Discount>
      </promotion>
      <publicationDate>2003-01-01</publicationDate>
    </book>
    <book isbn="112-0815">
      <name>Java Web Services Today and Beyond</name>
      <price>29</price>
      <authors>
        <authorName>John Brown</authorName>
        <authorName>Peter T.</authorName>
      </authors>
      <language>English</language>
      <promotion> <None/> </promotion>
      <publicationDate>2002-11-01</publicationDate>
    </book>  </books>  </BookCollection>
```

Values for xs:date and xs:time must conform to the syntax required for these XML types (cf. Slide 309)

[Filename: java/JAXB/books.xml]

JAXB - Example: XSD

[Filename: java/JAXB/books.xsd]

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0">

<xs:element name="BookCollection">
 <xs:complexType>
  <xs:sequence>
   <xs:element name ="books">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="book" type="bookType"
                  minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:sequence>
 </xs:complexType>
</xs:element>

  <!-- continue next page -->
```

```
<xs:complexType name="bookType">
 <xs:sequence>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="price" type="xs:long"/>
  <xs:element name="authors" >
   <xs:complexType>
    <xs:sequence>
     <xs:element name="authorName" type="xs:string" minOccurs="1" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="language" type="xs:string" minOccurs="1"  maxOccurs="unbounded"/>
  <xs:element name="promotion">
   <xs:complexType>
    <xs:choice>
     <xs:element name="Discount" type="xs:string" />
     <xs:element name="None" type="xs:string"/>
    </xs:choice>
   </xs:complexType>
  </xs:element>
  <xs:element name="publicationDate" type="xs:date"/>
 </xs:sequence>
  <xs:attribute name="isbn" type="xs:string" />
</xs:complexType>
</xs:schema>
```


JAXB HowTo

- README file in java/JAXB/JAXB-README.txt:

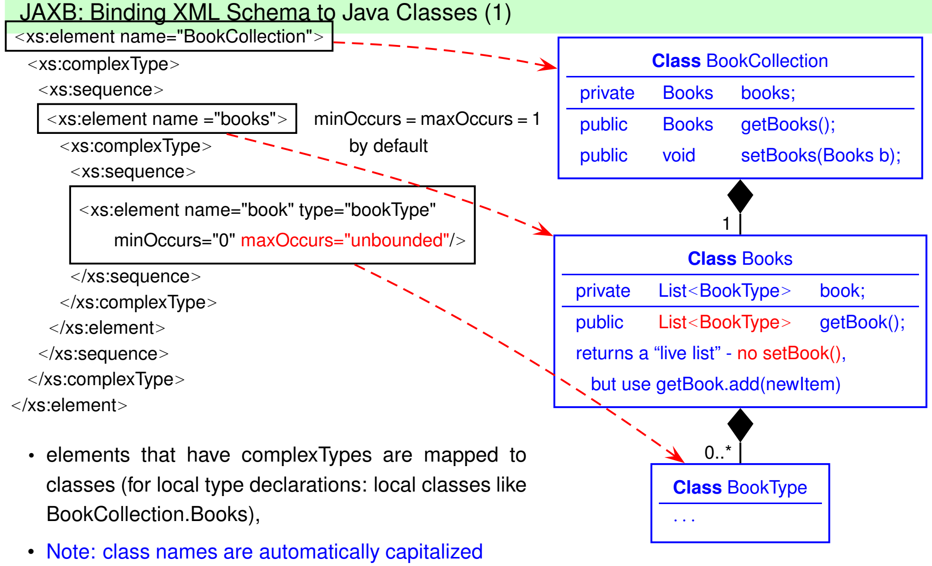
```
mkdir gensrc
xjc -p JAXBbooks books.xsd -d gensrc
# created classes can then be found in gensrc/JAXBbooks:
#  BookType.java, BookCollection.java, and ObjectFactory.java
# compile: generated classes can then be found in ./JAXBbooks
javac -d . $(find gensrc -name '*.java')
javac JAXBbooks.java
java JAXBbooks books.xml

xjc -p JAXBmondial mondial-jaxb.xsd -d gensrc
javac -d . $(find gensrc -name '*.java')
javac JAXBmondial/JAXBmondial.java
java JAXBmondial/JAXBmondial mondial-jaxb.xml
javac JAXBmondial/MyCountry.java
javac JAXBmondial/MondialObjectFactory.java
javac JAXBmondial/JAXBmondialExt.java
java JAXBmondial/JAXBmondialExt mondial-jaxb.xml
```


JAXB - Steps

1. `xjc -p` *`pckgname xsdfile`* `-d gensrc`:
   - generates class files (with local classes) from the XML Schema specification (put them into `gensrc/`*`pckgname`*),
   - all classes *classname* have only the standard constructor *classname()* and getXXX()/setXXX(...) methods.
   - generates an *ObjectFactory.java* with methods of the form
     `public` *`classname`* `create`*`classname`*`() {`
     `        return new` *`classname`*`(); }`
2. `javac -d . $(find gensrc -name '*.java')`
   compiles all files found in the `gensrc` subtree
   (generates package directory *pckgname* in ".").
3. write a a java application which
   (a) creates an Unmarshaller (which uses the ObjectFactory),
   (b) unmarshalls an XML file into objects (using the ObjectFactory),
   (c) optionally uses a Marshaller to transform the object graph back to XML.

JAXB: Binding XML Schema to Java Classes (1)



- elements that have complexTypes are mapped to classes (for local type declarations: local classes like BookCollection.Books),
- Note: class names are automatically capitalized
- elements of simpleTypes and attributes are mapped to instance properties,
- multivalued properties are handled by lists; updates not via setXXX(), but via list modifications.

JAXB: Binding XML Schema to Java Classes (2)

```
<xs:complexType name="bookType">
  <xs:sequence>
    <xs:element name="name" type="xs:string"/>
    <xs:element name="price" type="xs:string"/>
    <xs:element name="authors" >
      <xs:complexType>
        <xs:sequence>
          <xs:element name="authorName" type="xs:string"
            minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="language" type="xs:string"
    minOccurs="1" maxOccurs="unbounded"/>
    <xs:element name="promotion">
      <xs:complexType>
        <xs:choice>
          <xs:element name="Discount" type="xs:string" />
          <xs:element name="None" type="xs:string"/>
        </xs:choice>
      </xs:complexType>
    </xs:element>
    <xs:element name="publicationDate" type="xs:date"/>
  </xs:sequence>
  <xs:attribute name="isbn" type="xs:string"/>
</xs:complexType>
```

Class BookType

```
private String      name;
private String      isbn;
private long        price;
private XMLGregorianCalendar publicationDate;
private Authors     authors;
private List        language
private Promotion   promotion;
---
public  String      getName();    void setName(String s);
public  String      getIsbn();    void setIsbn(String s);
public  long        getPrice();   void setPrice(long x);
public  XML-G-C.    getPublicationDate();   void set(...)
public  Authors     getAuthors();   void setAuthors(Authors as);
public  List        getLanguage();   no setLanguage()
public  Promotion   getPromotion();   void setPromotion();
```

1   1

Class Authors

```
private List<String> authorName;
---
public  List<String> getAuthorName();
live list - no setAuthorName()
```

Class Promotion

```
private String discount;
private String none;
---
public get/set methods
```


JAXB - Example Usage

```
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import javax.xml.bind.Marshaller;
import java.io.File;
import javax.xml.datatype.XMLGregorianCalendar;
import org.w3c.dom.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import JAXBbooks.*; // import classes generated by binding compiler

public class JAXBbooks {

  public static void main (String args[]) { try
    { // generate the context using the JAXBBooks directory
      // where the generated classes and the ObjectFactory are
      JAXBContext jc = JAXBContext.newInstance(ObjectFactory.class);
      Unmarshaller unmarshaller = jc.createUnmarshaller();

      //   continue next page
```

[Filename: java/JAXB/JAXBbooks.java]


```
      //   continue next page

      BookCollection collection =
           (BookCollection) unmarshaller.unmarshal(new File( "books.xml"));
      BookCollection.Books books = collection.getBooks();

      for (BookType book : books.getBook()) {
        System.out.println("Book details "   );
        System.out.println("Book Name: " +  book.getName().trim());
        System.out.println("Book ISBN: " +  book.getIsbn().trim());
        System.out.println("Book Price: " +  book.getPrice());
        if (book.getPromotion().getDiscount() != null)
        System.out.println("Book promotion: " +  book.getPromotion().getDiscount().trim());
        System.out.println("No of Authors " + book.getAuthors().getAuthorName().size());

        BookType.Authors authors = book.getAuthors();
        for (String authorName : authors.getAuthorName())
            System.out.println("Author Name " + authorName.trim());
        XMLGregorianCalendar date = book.getPublicationDate();
        System.out.println("Date " + date);
        for (String language: book.getLanguage())
            System.out.println("Language " + language.trim());
        // add an element to a live list:
        book.getLanguage().add("Kisuaheli");
      }
```



```
      BookCollection collection =
        for (String language: book.getLanguage())
            System.out.println("Language " + language.trim());
        // add an element to a live list:
        book.getLanguage().add("Kisuaheli");
      }

      // transform the result into a DOM and write to an XML file:
      Marshaller m = jc.createMarshaller();
      DOMResult domResult = new DOMResult();
      m.marshal(collection, domResult);
      Document doc = (Document) domResult.getNode();
      // transformer stuff is only for writing DOM tree to file/stdout
      TransformerFactory factory = TransformerFactory.newInstance();
      Source docSource = new DOMSource(doc);
      StreamResult result = new StreamResult("foo.xml");
      System.out.println("look into foo.xml for the result");
      Transformer transformer = factory.newTransformer();
      transformer.transform(docSource, result);
    } catch (Exception e) { e.printStackTrace(); }
 }}
```

Marshaller

- generate an instance m of javax.xml.bind.Marshaller,
- The call of m.marshal(collection, destination) supports multiple kinds of destination (see javadoc for javax.xml.bind.Marshaller):
  (File)OutputStream (including System.out), SAX ContentHandler, DOM(Result) (as above), XMLStreamWriter, XMLEventWriter etc.
- Some methods and properties for controlling output:
  - `m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true)`
    `m.setProperty("com.sun.xml.bind.indentString", "  ")`
    formatted output with indentation __ (default: ____),
  - `m.setProperty("com.sun.xml.bind.xmlHeaders",`
    `"\n<!DOCTYPE mondial SYSTEM 'mondial.dtd'>")`
    outputs doctype declaration,
  - `m.setProperty(Marshaller.JAXB_FRAGMENT, true)`
    if result should be output as a *fragment* into something bigger (depends on destination, does not output xml declaration or START/END_DOCUMENT).

JAXB - Example: Mondial XSD with an annotation

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="2.0">
 <xs:element name="mondial">
  <xs:complexType> <xs:sequence>
       <xs:element name="country" type="country" minOccurs="0" maxOccurs="unbounded"/>
     </xs:sequence> </xs:complexType>
 </xs:element>
 <xs:complexType name="country">
  <xs:sequence>
   <xs:element name="population" type="populationtype" minOccurs="0" maxOccurs="1" />
   <xs:element name="province" type="province" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:attribute name="name" type="xs:string" use="required"/>
  <xs:attribute name="area" type="xs:decimal" use="optional"/>
  <xs:attribute name="car_code" type="xs:ID" use="optional"/>
  <xs:attribute name="indep_date" type="xs:date" use="optional"/>
  <xs:attribute name="capital" type="xs:IDREF" use="optional">
   <xs:annotation>  <!-- annotation of the target type  <<<<<<<< -->
    <xs:appinfo>
     <jaxb:property>
      <jaxb:baseType name="City"/>  <!-- Note: capitalization - otherwise: error !!!!!!!!!!! -->
     </jaxb:property>
    </xs:appinfo>
   </xs:annotation>
  </xs:attribute>
 </xs:complexType>   <!-- continue next page -->
```

[Filename: java/JAXB/mondial-jaxb.xsd]


```
<xs:complexType name="province">
 <xs:sequence>
  <xs:element name="population" type="populationtype" minOccurs="0" maxOccurs="1" />
  <xs:element name="city" type="city" minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="name" type="xs:string" use="required"/>
 <xs:attribute name="area" type="xs:decimal" use="optional"/>
</xs:complexType>

<xs:complexType name="city">
 <xs:sequence>
  <xs:element name="population" type="populationtype"  minOccurs="0" maxOccurs="unbounded"/>
 </xs:sequence>
 <xs:attribute name="name" type="xs:string" use="required"/>
 <xs:attribute name="id" type="xs:ID"/>
</xs:complexType>

<xs:complexType name="populationtype">
 <xs:simpleContent>
  <xs:extension base="xs:nonNegativeInteger">
   <xs:attribute name="year" type="xs:nonNegativeInteger" use="optional"/>
  </xs:extension>
 </xs:simpleContent>
</xs:complexType>
</xs:schema>
```


JAXB example: Mondial – Datatypes

(see `java/JAXB/gensrc/JAXBmondial/XXX.java`)

- annotation of the country/@capital IDREFS attribute:
  ⇒ Country: `public City getCapital()`,
- without annotation (like standard mondial):
  ⇒ Country: `public Object getCapital()`, requires casting.
- countries have only one, and cities may have several (complex) population subelements (of type “PopulationType” → class PopulationType.java):
  - Country: `public PopulationType getPopulation()`,
  - City: `public List<PopulationType> getPopulation()`,
- Populationtype: content is xs:nonNegativeInteger – `public BigInteger getValue()`,
- Country/@Area: value is xs:decimal – `public BigDecimal getArea()`,
- Date values:
  Class IndepDate (<indep_date from=“GB”>1776-07-04</indep_date>):
  `public XMLGregorianCalendar getValue()`,
  Class Organization (<organization ...> <established>1992-02-06</>...</>):
  `public XMLGregorianCalendar getEstablished()`.

JAXB – Datatypes

- Complex Types → application-specific auto-generated (bean) classes with setter/getter methods.
- Text content types and attribute value types:
  - "High-level" datatypes like xs:decimal, xs:nonNegativeInteger, xs:integer are mapped to *Java literal classes* java.math.BigDecimal, java.math.BigInteger, etc.
  - xs:int, xs:long (see Books.xsd: BookType.price) are mapped to *Java literal types* int, log, etc.
- Usage
  - XML + XML Schema for data exchange:
    use implementation-level types xs:int, xs:long etc. in XSD
  - XML + XML Schema as data model (+ ontology): comes with semantic datatypes like xs:nonNegativeInteger,
    ⇒ JAXB programs must do conversions.

JAXB - Mondial

```
<?xml version="1.0"?>
<mondial>
 <country name="Austria" area="83850" indep_date="1918-11-12" capital="cty-Austria-Vienna">
  <population>8023244</population>
  <province name="Vienna" area="415">
   <population>1583000</population>
    <city name="Vienna" id="cty-Austria-Vienna">
     <population year="1994">1583000</population>
     <population year="2013" measured="admin.">1761738</population>
    </city>
  </province>
  <province name="Upper Austria" area="11979">
   <population>1424910</population>
   <city name="Linz">
     <population year="1994">203000</population>
     <population year="2013">193511</population>
   </city>
   <city name="Wels">
     <population year="2013">59239</population>
   </city>
  </province>
 </country>
</mondial>
```

[Filename: java/JAXB/mondial-jaxb.xml]

JAXB - Example Usage

```
package JAXBmondial;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import java.io.File;
import java.math.BigInteger;
import java.math.BigDecimal;
import java.util.List;

public class JAXBmondial {
public static void main (String args[]) {
 try {
  JAXBContext jc = JAXBContext.newInstance("JAXBmondial");
  Unmarshaller unmarshaller = jc.createUnmarshaller();
  Mondial mondial = (Mondial) unmarshaller.unmarshal(new File("mondial-jaxb.xml"));

  List<Country> countryList  = mondial.getCountry();
  for (Country country: countryList)
  {
    System.out.println("Country: " + country.getName()  );
    BigDecimal area = country.getArea();
    BigInteger countrypop = country.getPopulation().getValue();
      // datatype class handling is ugly (pop is BigInteger, area is BigDecimal):
    System.out.println("   pop: " +  countrypop + ", area: " + area + ", density: "
            + new BigDecimal(countrypop).divide(area, java.math.BigDecimal.ROUND_HALF_UP));
```



```
    System.out.println("   pop: " +  countrypop + ", area: " + area + ", density: "
            + new BigDecimal(countrypop).divide(area, java.math.BigDecimal.ROUND_HALF_UP));


    // Java knows from the annotation of the IDREF attribute that this is a city:
    City cap = country.getCapital();
    System.out.println("   cap: " +  cap.getName());

    List<Province> provList = country.getProvince();
    for (Province prov: provList) {
      System.out.println("   Province name: " + prov.getName().trim());
      System.out.println("            area: " + prov.getArea().toString());
      System.out.println("            pop : " + prov.getPopulation().getValue());
      List<City> cityList = prov.getCity();
      for (City city : cityList) {
        System.out.println("            City name: "+city.getName().trim());
        List<Populationtype> poplist = city.getPopulation();
        for (Populationtype p : poplist)
          System.out.println("                  pop "
                        + p.getYear() + ": " + p.getValue());
  } } }
 } catch (Exception e ) { e.printStackTrace(); }
}}
```

[Filename: java/JAXB/JAXBmondial/JAXBmondial.java]

JAXB Integration with Java Application?

Classes in an application program

- application-specific methods
- properties that are local to the Java existence of the object

JAXB-generated classes vs. user-defined classes

- user-defined class my_xxx where xxx is a subclass of:
  - useful from the java point of view: extend application class with bean functionality and marshalling.
  - cannot be communicated declaratively to the JAXB generation of the classes (annotation with xjc:superClass c in the XML Schema does only allow to make all classes subclasses of c)
- define a subclass: my_xxx extends xxx
  - after unmarshalling, the objects are only instances of xxx

  ⇒ methods of my_xxx not applicable

  ⇒ Different alternatives.

User-Defined Extension of JAXB-Created Classes

Manual editing of generated classes themselves

- edit the generated xxx.java files
- if instance attributes are added, they must also be added either to propOrder, or get an anntotation as @XmlAttribute – and then they will be exported when marshalling them.

⇒ must be manually redone/adapted after schema changes.

User-Defined Subclasses (I)

- (manually) write application subclasses my_xxx that extend the JAXB-generated classes,
- after unmarshalling, traverse the tree and re-create the objects as instances of the my_xxx subclasses.

User-Defined Subclasses (II) – Overwrite Generated Object Factory

- create the instances of the my_xxx subclasses during unmarshalling: JAXB allows to create the unmarshaller over a user-defined Object Factory.

JAXB - Example Usage with extended class definition

```
package JAXBmondial;

public class MyCountry extends Country {
  // a method for more comfortable manipulation:
  public void addProvince(Province p) {
      getProvince().add(p);
  }
  // a "useful" method:
  public void printCityNames() {
    for (Province prov : getProvince()) {
        for (City city : prov.getCity()) {
            System.out.println(city.getName().trim());
        }
    }
}}
```

[Filename: java/JAXB/JAXBmondial/MyCountry.java]

JAXB - Example Extended Object Factory

- original auto-generated ObjectFactory can be found in `java/JAXB/gensrc/JAXBmondial/ObjectFactory.java`:

  `public Country createCountry() { return new Country(); }`

```
package JAXBmondial;
import JAXBmondial.ObjectFactory;
public class MondialObjectFactory extends ObjectFactory {
  @Override
  public Country createCountry() {
    System.out.println("create MyCountry");
    return new MyCountry();
  }}
```

[Filename: java/JAXB/JAXBmondial/MondialObjectFactory.java]

JAXB - Example Usage with extended class definition

- tell the unmarshaller to use the modified MondialObjectFactory
- *unm*.setProperty(*jaxb-unm-propertyId*, *myClass*);
- supported Values for *jaxb-unm-propertyId*: in general none, only provider-dependent ones. Download most recent JAXB2.

JAXB - Example Usage with extended class definition

```
package JAXBmondial;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import java.io.File;

public class JAXBmondialExt {
 public static void main (String args[]) { try {
  JAXBContext jc = JAXBContext.newInstance("JAXBmondial");
  Unmarshaller unm = jc.createUnmarshaller();
  unm.setProperty("com.sun.xml.bind.ObjectFactory", new MondialObjectFactory()); // <<<<
  Mondial mondial = (Mondial) unm.unmarshal(new File("mondial-jaxb.xml"));

  for ( Country x : mondial.getCountry() ) {  // <<<<<<<  only "Country" possible
    MyCountry country  = (MyCountry) x;   // <<<<<<<<<
    System.out.println("Country: " + country.getName()  );
    country.printCityNames();  // <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
  }} catch (Exception e ) { e.printStackTrace(); }
}}
```

[Filename: java/JAXB/JAXBmondial/JAXBmondialExt.java]

JAXB Mapping: Summary

- allows for easy and lightweight unmarshalling, bean-based manipulation and marshalling of XML data,
- higher level of abstraction from XML representation, compared with DOM and SAX,
- but still actually just a way to manipulate XML data without having to know the specific notions of the XML data model.

Minor Comments

- naming (getBook() for a list etc.) not always intuitive;
  can be customized by annotations to the XSD;
- intermediate elements (example: Books, Authors) lead to unnecessary classes;
  can often be omitted (example: Book/Language elements)

⇒ to get a better “modeling”, do not use structures like
  Country-hasProvince-Province-hasCity-City
  (as in Striped RDF/XML [Semantic Web lecture]; this generates intermediate classes), but
  Country-Province-City.

- Care for datatypes/classes (cf. Slide 561).

Aside: SOAP (Simple Object Access Protocol)

- Generic “protocol” (nevertheless, HTTP-based)
- Any object can be serialized in XML, sent, and deserialized
  (only having the Java class code, without having an XSD).
  So far similar to the OIF (Object Interchange Format) of ODMG (cf. Slide 53 ff.).
- The XML representation is not intended to be processed on the XML level, but only by soap-unpacking it.
- Bad experience: correct packing/unpacking only between same SOAP implementations.
- Note: Instances of Java XML (DOM) are not serialized as plain XML, but as SOAP serialization of an instance of the underlying DOM implementation class.
  (not intended *for* exchanging XML, but for exchanging objects *by* XML).

⇒ when messages are designed to be XML, SOAP is not the right way, but use simple, plain HTTP!

- One does not need to have any knowledge of XML to use soap (actually, knowledge of XML doesn’t help).

⇒ so it does not fit in this course.

11.8 XML Digester

Comparison

- SAX/StAX:
  - fine granularity,
  - extremely flexible,
  - hard to write and read
- JAXB:
  - whole document is transformed into objects
  - unflexible
  - self-explaining mapping

... in-between:

- http://commons.apache.org/digester/

XML Digester: Principle

- rule-based: on simple XPath patterns ... do something.
- internally based on SAX,
- rules hook on beginElement, PCDATA, and endElement,
- provides a stack with automatical and user-defined behavior,
- building an object graph/tree by traversing the XML tree:
  - predefined rules for user-defined selective XML-to-objects mapping supported by a stack and bean-style user-defined classes.
- implementing other XML-tree-traversal-based algorithms:
  - rules with user-defined program code on the predefined hooks
- mixed functionality:
  - can be used for SAX/StAX-style filtering from the stream (and building intermediate objects) and query answering.
- Comparison with JAXB:
  transformation XML→objects, but not the other direction.

XML Digester: Stack

Predefined rule types mirror XML element tree/hierarchy:

- top-of-stack-element is always the one that is currently accessed by methods,
- ancestors are down the stack.
- push(), pop(),
- peek(n) accesses the n-th element on the stack (top=0),
- generation of objects on the stack is controlled by rules:
- ObjectCreate(*pattern*, *class*.class)
  - if an element satisfying *pattern* is started, create a new instance of class *class* and push it on the stack.
  - endElement(): pops the topmost element from the stack.

⇒ On-the-fly-filtering: specify addObjectCreate() only for relevant element types (=object classes).

- During traversal, map the element hierarchy to the created objects:
  SetNext(*pattern*, *method*): on endElement() of x satisfying *pattern*, calls *method* of the next object on the stack, *method*'s argument type must be the class of x.
  (i.e. applies peek(1).method(peek(0)))

Built-In Rules for XML-to-Objects/Beans Mapping

Rule specifications consist of a match pattern (similar to XSL patterns) and specifications of the action:

- Patterns: only *elname/.../elname* and **/elname/.../elname* where * stand for an arbitrary number of child navigation steps,
- digester.addObjectCreate(*pattern*, *class*);   (see above)
- digester.addSetProperties(*pattern*, *attrname*, *property*);
  sets *property* of the top object to the value of *attrname*;
  also [...]- lists of attrnames/properties are allowed.
- digester.addBeanPropertySetter(*pattern*);
  given a node x matching the pattern, sets the property with x's name to the value of x.
- digester.addCallMethod(*pattern*, *method*, n);
  digester.addCallParam(*pattern*, i); ($i \leq n$)
  executes a *method* call to the top object with n parameters, which are set by the value(s) of the subsequent addCallParam rules.
- digester.addSetNext(*pattern*, *method*);
- see Javadoc at `http://commons.apache.org/digester/` for details.

XML Digester: Example

```
import java.io.File;
import java.util.TreeSet;
import org.apache.commons.digester3.Digester;

public class GetMillionCities {
  public static class CityCollection extends TreeSet<City>  {
      public void addCity(City c) { if (c.population > 1000000) this.add(c);}  }
  public static void listCities(CityCollection cities) {
      for (City c : cities) {
         System.out.println(c.name + " " + c.country + " " + c.population); }  }
  public static class City implements Comparable{
    String name = null;
    String country = null;
    int population = -1;
    public void setName(String n) { if (name == null) name = n; }
    public void setCountry(String code) { this.country = code; }
    // note: all PCDATA/CDATA values are strings!
    public void setPopulation(String pop) { this.population = new Integer(pop); }
    public int compareTo(Object o) {
         if (this.population < ((City)o).population) return 1; else return -1; }
  }
  // continue next page
```

[Filename: java/Digester/GetMillionCities.java]

```
  public static void main(String[] args) {
    File mondial = new File("mondial.xml");
    final Digester digester = new Digester();
    digester.push(new CityCollection());

    // note: reacts only on cities as direct children of countries
    // */city would take all cities,
    // country//city, country/*/city, mondial/*/city are not allowed;
    digester.addObjectCreate("mondial/country/city", City.class);
    digester.addSetProperties("mondial/country/city", "country", "country");
    digester.addBeanPropertySetter("mondial/country/city/name");
    digester.addCallMethod("mondial/country/city/population", "setPopulation", 1);
    digester.addCallParam("mondial/country/city/population", 0);
    // Digester (clearly) does not like population[last()]. //
    // note: at the end, the last=most recent population is the stored value
    // at </city> calls addCity() of the now-top-of-stack-object which is the CityCollection
    digester.addSetNext("mondial/country/city", "addCity");
    try { digester.setValidating(false);
          final CityCollection cities = digester.parse(mondial);
          System.out.println("####  now listing cities #### ");
          listCities(cities);
        } catch (Exception e) { e.printStackTrace(); }
} }
```


Digester: Rules

The above are shortcuts for typical rule patterns.
Every rule implementation class implements three methods:

- begin(): executed at startElement(),
- end(): executed at endElement(),
- body(): executed at PCDATA contents of an element.

Example: addObjectCreate(*pattern*, *class*)

- Class ObjectCreateRule(*class*.class)
- begin(): create object of given class, initialize from attributes in the opening tag (if required), push it on the stack;
- end(): pop object from stack.

Example: addSetNext(*pattern*, *method*)

- Class SetNextRule(*class*.class), subclass of AbstractMethodRule
- begin(): nothing,
- end(): pop object obj from stack and execute *method* of then-top with argument obj.

XML Digester: Example using Basic Rule Implementations

[Filename: java/Digester/GetMillionCities2.java]

```
import java.io.File;
import java.util.HashSet;
import org.apache.commons.digester3.Digester;
import org.apache.commons.digester3.Rule;
import org.apache.commons.digester3.ObjectCreateRule;
import org.xml.sax.Attributes;

public class GetMillionCities2 {

  public static class CityCollection extends HashSet<City>  {
      public void addCity(City c) {
        if (c.population > 1000000) this.add(c); System.out.println("ADD TO COLL " + c.name);}
  }
  public static void listCities(CityCollection cities) {
      for (City c : cities) {
         System.out.println(c.name + " " + c.country + " " + c.population); }
  }

  public static class City {
    String name = null;
    String country = null;
    int population = -1;
    public void setName(String n) { if (name == null) name = n; }
    public void setCountry(String code) { this.country = code; }
    // note: all PCDATA/CDATA values are strings!
    public void setPopulation(String pop) { this.population = new Integer(pop); }
  }
  // continue next page
```



```
// continue next page

public static void main(String[] args) {
  File mondial = new File("mondial.xml");
  final Digester digester = new Digester();
  digester.push(new CityCollection());
  System.out.println("INIT: " + digester.peek());

  // digester.addObjectCreate("*/city", City.class);
  digester.addRule("*/city", new ObjectCreateRule(City.class){
    public void begin(String namespace, String name, Attributes attrs) throws Exception {
        digester.push(new City());  // equivalent: super.begin(namespace, name, attrs);
        System.out.println("BEGIN: " + digester.peek() + digester.peek(1)); }
    public void end(String namespace, String name) throws Exception {
        System.out.println("END: " + digester.peek());
        // if (((City)digester.peek()).population > 1000000) {...}
        digester.pop();  // equivalent: super.end(namespace,name);
        System.out.println("ENDEND: " + digester.peek()); }
  } );

  // digester.addSetNext("*/city", "addCity");
  digester.addRule("*/city", new Rule(){
  public void begin(String namespace, String name, Attributes attrs) throws Exception {}
    public void end(String namespace, String name) throws Exception {
        System.out.println("ADD: " + digester.peek(0) + digester.peek(1));
        ((CityCollection)(digester.peek(1))).addCity((City)digester.peek(0)); }
  } );

  digester.addSetProperties("*/city", "country", "country");
  digester.addBeanPropertySetter("*/city/name");
  digester.addCallMethod("*/city/population", "setPopulation", 1);
  digester.addCallParam("*/city/population", 0);

  try {
     digester.setValidating(false);
```

```
    digester.addSetProperties("*/city", "country", "country");
    digester.addBeanPropertySetter("*/city/name");
    digester.addCallMethod("*/city/population", "setPopulation", 1);
    digester.addCallParam("*/city/population", 0);

    try {
        digester.setValidating(false);
        final CityCollection cities = digester.parse(mondial);
        System.out.println("####  now listing cities #### ");
        listCities(cities);
      } catch (Exception e) { e.printStackTrace(); }
  }
}
```

- addObjectCreate(...) replaced by addRule(...),
- addSetNext(...) replaced by addRule(...)

Rule Application Order

If multiple rules match in a situation:

- if startElement(), rules are applied in the order they have been added to the digester,
- if endElement(), rules are applied in *reverse* order.
- this guarantees if multiple rules push/pop, pop() is are applied in the correct order.

Digester: Producing (sysout) Streaming Output

- Rules can be designed to produce output immediately during the processing:

[Filename: java/Digester/PrintMillionCities.java]

```
import java.io.File;
import org.apache.commons.digester3.Digester;
import org.apache.commons.digester3.Rule;
import org.xml.sax.Attributes;   ###  note: import from SAX API ...

public class PrintMillionCities {

  public static class City {
    String name = null;
    String country = null;
    int population = -1;
    public void setName(String n) { if (name == null) name = n; }
    public void setCountry(String code) { this.country = code; }
    // note: all PCDATA/CDATA values are strings!
    public void setPopulation(String pop) { this.population = new Integer(pop); }
  }
  // continue next page
```



```
  public static void main(String[] args) {
    File mondial = new File("mondial.xml");
    final Digester digester = new Digester();

    digester.addObjectCreate("*/city", City.class);

    digester.addRule("*/city", new Rule(){
      public void begin(String namespace, String name, Attributes attrs) throws Exception {
        System.out.println("start city"); }
      public void end(String namespace, String name) throws Exception {
        City c = (City)(digester.peek(0));
        if (c.population > 1000000)
           System.out.println(c.name + " " + c.country + " " + c.population); }
    } );

    digester.addSetProperties("*/city", "country", "country");
    digester.addBeanPropertySetter("*/city/name");
    digester.addCallMethod("*/city/population", "setPopulation", 1);
    digester.addCallParam("*/city/population", 0);

    try { digester.setValidating(false);
          digester.parse(mondial);
      } catch (Exception e) { e.printStackTrace(); }   } }
```


Digester: A “Native” Stack Application - Evaluation of Artihmetic Term Trees

- Arithmetic terms as trees
- depth-first-evaluation: push values on the stack, operator: pop two values, compute the results, push it on the stack.
- Possible XML representation of terms:

```
<term><!-- ((90 div (19 - (3*8)))+3) -->
  <plus>
    <div>
      <val>90</val>
      <minus>
        <val>19</val>
        <mult>
          <val>3</val>
          <val>8</val>
        </mult>
      </minus>
    </div>
    <val>3</val>
  </plus>
</term>
```

[Filename: java/Digester/arithm-tree-example.java]

[Filename: java/Digester/ArithmTerms.java]

```
import java.io.File;
import org.apache.commons.digester3.Digester;
import org.apache.commons.digester3.Rule;

public class ArithmTerm {
  public static void main(String[] args) {
    File term = new File("arithm-tree-example.xml");
    final Digester digester = new Digester();
    //  Rule: push values as Integers on the stack
    digester.addRule("*/val", new Rule(){
     public void body(String namespace, String name, String text) throws Exception {
        digester.push(new Integer(Integer.parseInt(text)));
        System.out.println("value:" + text );
      }
    } );
    //  Rule: operators: combine the two top stack values accordingly:
    digester.addRule("*/plus", new Rule(){
      public void end(String namespace, String name) throws Exception {
        int n = (Integer)(digester.pop()) + (Integer)(digester.pop());
        digester.push(new Integer(n));
        System.out.println("plus: " + n); }
      } );
```



```
    digester.addRule("*/minus", new Rule(){
      public void end(String namespace, String name) throws Exception {
        int min = (Integer)(digester.pop());
        int n = (Integer)(digester.pop()) - min;
        digester.push(new Integer(n));
        System.out.println("minus: " + n ); }
      } );
    digester.addRule("*/mult", new Rule(){
      public void end(String namespace, String name) throws Exception {
        int n = (Integer)(digester.pop()) * (Integer)(digester.pop());
        digester.push(new Integer(n));
        System.out.println("mult: " + n); }
    } );
    digester.addRule("*/div", new Rule(){
      public void end(String namespace, String name) throws Exception {
        Integer div = (Integer)(digester.pop());
        int n = (Integer)(digester.pop()) / div;
        digester.push(new Integer(n));
        System.out.println("div: " + n); }
    } );
    try { digester.setValidating(false);
          Integer result = digester.parse(term);
  System.out.println("result: " + result);
      } catch (Exception e) { e.printStackTrace(); }   } }
```


11.9 Aside: Use of Date and Time Datatypes

- XML Schema: simple datatypes for dateTime
- represented by String literals in attribute values or text contents
  - xs:dateTime: *yyyy-mm-dd* T*hh:mm:ss*[.*xx*][{+|-}*hh:mm*]
  - xs:time: *hh:mm:ss* [{+|-}*hh:mm*]
  - xs:duration: P[*n*Y][*n*M][*n*D][T[*n*H][*n*M][*n*[.*n*]S]], where *n* can be any natural number
  - xs:dayTimeDuration, xs:yearMonthDuration: restrictions of xs:duration.
- for XQuery handling with specific operations (similar to those known from SQL) cf. Slide 309.
- map to appropriate classes for processing by Java (and by this e.g. with JDBC from and to SQL databases).

Aside: Date and Time in Java

Java provides several classes for handling date and time:

- Datatype: import java.util.GregorianCalendar;

Create from string representation:

- import java.text.DateFormat; import java.text.SimpleDateFormat;

```
public static String datedefaultpattern = "yyyy-MM-dd'T'HH:mm:ss";
// input: string s (following the XML Schema pattern)
DateFormat df = new SimpleDateFormat(datedefaultpattern);
GregorianCalendar typedvalue = df.parse(s);
// result: typedvalue as an object
```

- see java.util.GregorianCalendar for method documentation.
- JAXB: uses javax.xml.datatype.XMLGregorianCalendar class (see e.g. the generated gensrc/Organization.java)

11.10 Web Services (Overview)

- History: RPC (Remote Procedure Call)
  - call a specific procedure at a specific server
    (client stub→marshalling→message→unmarshalling→ server stub→ server).
- History: OMG Standard (Object Management Group) CORBA (1989, “Common Object Request Broker Architecture”; cf. Slides 38 ff.):
  - Middleware, usually applied in an Intranet,
  - central ORB bus where services can connect,
  - service registry (predecessor of WSDL and UDDI ideas),
  - description of service interfaces in object-oriented style
    (IDL - interface description language, similar to C++ declarations),
  - exchanging objects between services via OIF (Object Interchange Format),

  ⇒ RPC abstraction (call abstract functionality) by the ORB as a broker.
- XML-RPC and SOAP+WSDL+UDDI are XML-based variants of RPC+Corba.
- SOA (“Service-Oriented Architecture”).

HTTP: Hypertext Transfer Protocol (Overview)

- HTTP 0.9 (1991), HTTP 1.0 (1995), HTTP 1.1 (1996).
- Application Layer Protocol [OSI Level 7], based on a (reliable) transport protocol (usually TCP “Transmission Control Protocol” [ISI/OSI Level 4] that belongs to the “Internet Protocol Suite” (IP))
  [see Telematics lecture].
- Request-Response Protocol: open connection, send something, receive response (both can be streamed), close connection.
- well-known from Web browsing and HTML:
  send (HTTP GET) URL, get URL (=resource) contents
  ⇒ this is already a (very basic) Web Service
  also: send HTTP POST URL+Data (Web Forms) get answer
  ⇒ this is also a (still basic) Web Service; “Hidden Web”
- common protocol used for communication with and between Web Services ...

Infrastructure Architecture

Web Server

- hosts different things; amongst them
  - “simple” HTML pages, binaries (pdfs, pictures, movies, ...)
  - Web Services, i.e. software artifacts that implement some functionality.
- Example: Apache Web Server.
- not the topic of this lecture (→ technical infrastructure).

(Java) Servlet

- a piece of software that should be made available as a Web Service,
- implements the methods of the Servlet interface
  (Java: `javax.servlet.http.Servlet`, subclasses `GenericServlet`, `HttpServlet`)

Web (Service|Servlet) Container

- a piece of software that extends a Web Server with infrastructure to provide the runtime environment to run servlets as Web Services,
- hosts one or more Web Services that extend the container’s base URL

Web Servlet Container [incorrect: Web Service Container]

- Servlets are the pieces of software that are used to *provide* services.
- The servlets’ code must be accessible to the Web Servlet Container, usually located in a specific directory,
- WSC controls the lifecycle of the servlets: (`init()`, `destroy()`)
- maps the incoming communication from ports via the URLs to the appropriate servlet invocation.
  Container: method `service(`*httpContents*`)`, mapped to Servlets’ `doGet(`*httpContents*`)` `doPost(`*httpContents*`)`, (`doPut(`*httpContents*`)`), (`doDelete(`*httpContents*`)`).
- Example: Apache tomcat.
- standalone tomcat: one port (default 8080), one base URL;
- tomcat might be run in a Web Server (Apache), then, multiple base URLs can be mapped to the same tomcat.
- URL tails do not necessary belong to the same/different Servlets (see next slides)!

⇒ URL tails are just abstract names
(even the internal organization/implementation might change over time)

Abstraction Levels

Goal: abstract from internal software/programming structure of the projects against the externally visible URLs.

- a Web Service Container contains several “projects” (eclipse terminology) or “applications”:
  - from the programmer’s view, a “project” is an (e.g., eclipse) project,
    as a package it is a single .war file,
    at the end, it is a subdirectory in the container.
    Each project has an (internal) name (its directory name in the container), e.g. `xquery-demo` or `servletdemo`.
- Each project consists of one or more servlets:
  - each servlet has an (internal) name (relative to its directory name in the container), e.g. the `servletdemo` project contains three different servlets (just due to its programming as a “silly example”, nothing about efficiency)
    (nobody from the outside will see what are the actual names of these servlets)
  - each servlet’s code is a class that extends javax.servlet.http.HttpServlet;

Abstraction Levels: URL mapping

HTTP connections received by the servlet container are internally forwarded to the servlets.

- the Web Service Container has a base url;
  `http://www.semwebtech.org`.
  (actually, this is the base URL of an Apache that maps most things to a tomcat)
- Service URLs: `http://www.semwebtech.org/xquery-demo`,
  `http://www.semwebtech.org/servletdemo`,
  `http://www.semwebtech.org/services/2016/xml2sql` etc.
- the Web Service Container maps *relative paths* to projects (by tomcat’s server.xml):
  `/xquery-demo` to `xquery-demo`, and `/servletdemo` to `servletdemo`, and
  `/services/2016/xml2sql` to `xmlconverter`.
- each project’s configuration (in its web.xml) maps URL path tails to servlet ids, and servlet ids to servlet classes, e.g. for the servletdemo project
  `/sum` to `sum-servlet` to `org.semwebtech.servletdemo.SumServlet`,
  `/format`, `/all` and `/reset` to `format-servlet` to `org.s.s.FormatServlet`,
  `/makecalls` to `makecalls-servlet` to `org.s.s.MakeCallsServlet`, and `index.html` is the front page served for “/”.

⇒ internal software organization independent from externally visible URLs

Tomcat Basic Installation

- See course Web page for detailed instructions with servlet examples.
- Web Servlet Container with simple Web Server: Download and install Apache Tomcat
  - can *optionally, but not necessarily* be combined with the Apache Webserver,
  - can be installed in the CIP Pool
- set environment variable (catalina is tomcat’s Web Service Container)

```
export CATALINA_HOME=~/apache-tomcat-x.x.x
```

- configure server: edit

```
$CATALINA_HOME/conf/server.xml:
<!-- Define a non-SSL HTTP/1.1 Connector on port 8080 -->
<Connector port="8080" .../>
```

- start/stop tomcat:

```
$CATALINA_HOME/bin/startup.sh
$CATALINA_HOME/bin/shutdown.sh
```

- logging goes to

```
$CATALINA_HOME/logs/catalina.out
```


Tomcat: Servlet Deployment

- upon startup, tomcat deploys all servlets that are available in

  `$CATALINA_HOME/webapps`

  (considering path mappings etc. in `$CATALINA_HOME/conf/server.xml`)

Two alternatives how to make servlets available there:

- create a `myproject.war` file (web archive, similar to jar) and copy it into `$CATALINA_HOME/webapps`.
  (e.g. via build.xml targets "dist" and "deploy")
  (tomcat will unpack and deploy it upon startup)
  When replacing an old war file, delete the old unpacked stuff also.
- create a directory `myproject`, copy everything that is in the `WebRoot` directory there.
  (e.g. build.xml target "deploy"; cf. Demo-Servlet)

Tomcat's conf/server.xml

The URL paths to the projects can be defined to differ from the defaults (path name = webapps-directory name)

This is done in the <Host> element:

```
<Host>
  <Context path="/services/2016/xml2sql" reloadable="false" docBase="xmlconverter"/>
  :
</Host>
```

- if the project name is the same as the path (e.g. xquery-demo and servlet-demo), the entry can be omitted
  (usually, software projects do not have the same name, but distributed .war archive files can be renamed accordingly).
- reloadable: automatically reloads the servlet if the code is changed (e.g. a new .war archive).
  Should be done only during development.
- the path attribute is key. There can be multiple paths that are mapped to the same docBase.

General Servlet (Eclipse) Project Directory Structure

MyProject: project directory (anywhere outside tomcat)

MyProject/build.xml: the ant file for compiling and deploying – see later.

MyProject/src: the .java (and other) sources

MyProject/WebRoot: roughly, all this content is copied to the Servlet Container.
Plain HTML pages like index.html can be placed here.

MyProject/WebRoot/WEB-INF: the whole content of MyProject/WebRoot except WEB-INF is visible later (e.g., HTML pages can be placed here); the contents of WEB-INF is *used* by the Servlet Container.

MyProject/WebRoot/WEB-INF/web.xml: web application configuration,

MyProject/WebRoot/WEB-INF/classes: compiled binary stuff,

MyProject/WebRoot/WEB-INF/lib: used jars (except javax.servlet.jar – tomcat has own classes for servlets, this would create conflicts),

MyProject/lib: jars that are needed for building, but should not be copied to the Servlet Container (put javax.servlet.jar here),

build path: all jars in MyProject/lib + MyProject/WebRoot/WEB-INF/lib

Servlet-Demo Example

Basic demonstration of servlet programming [servletdemo.zip on course Web page]

- The basic functionality is simple:
  a form where the user enters two numbers, and the servlet computes the sum (SumServlet),
  [HTML form with simple HTTP GET from servlet, simple answer]
- The same (added to the same form): the result is presented in an HTML table (FormatServlet),
  [HTML page as an answer]
- The same again (added to the same form): the numbers are taken, submitted to the SumServlet, and all three are submitted to the FormatServlet and a HTML page is created as answer (MakeCallsServlet).
  [HTML form with simple HTTP POST to servlet, inter-Servlet HTTP POST]
- The Demo collects all formatted tables and can output them.
  [persistent information, multiple GETs in the same servlet]
- it can be reset.

The Project's web.xml (excerpt)

```
<web-app>
  <!-- Define servlet names and associate them with classfiles -->
  <servlet>
    <servlet-name>makecalls-servlet</servlet-name>
    <servlet-class>org.semwebtech.servletdemo.MakeCallsServlet</servlet-class>
    <init-param>
      <param-name>myURL</param-name>
      <param-value>http://localhost:8080/servletdemo/</param-value>
    </init-param>
  </servlet>
  <servlet> ... </servlet>
  <!-- define mapping of path tails to servlets -->
  <servlet-mapping>
    <servlet-name>makecalls-servlet</servlet-name>
    <url-pattern>/makecalls</url-pattern>
  </servlet-mapping>
  <servlet-mapping> ... </servlet-mapping>
  <!-- optionally: define default html page -->
  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Comments: web.xml

- <servlet>:
  - a short, abstract name (unique)
  - which java class
  - optional: init parameters that can be read in the `init(ServletConfig cfg)` method with `cfg.getInitParameter(param-name);`
    - allows some adaptation of “foreign” servlets by only editing the web.xml, without recompiling Java code (e.g. if a .war contains only binaries).
    - if servlets (like MakeCallsServlet) need to call other servlets or use files, they can be told about their actual URLs.
    - directories where files can be found locally can be specified:

```
<init-param>
  <param-name>examplesDir</param-name>
  <param-value>/home/may/teaching/ssd/XQuery/</param-value>
</init-param>
```

- <servlet-mapping>:
  - url-pattern: key, things like /∗ allowed,
  - multiple patterns can be mapped to the same servlet.

A Minimal Configuration (tomcat)

- A java class `Abc.java` which `extends HttpServlet`,
- implement a simple GET method for testing (HTML, Hello World)
- for compiling ist locally, put `.../tomcat/lib/servlet-api.jar` into the CLASSPATH,
- Directory structure:
  `tomcat/webapps/XYZ/WEB-INF/classes/Abc.class`
  `tomcat/webapps/XYZ/WEB-INF/web.xml`
- web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <servlet> <servlet-name>foobla</servlet-name>
           <servlet-class>Abc</servlet-class> </servlet>
  <servlet-mapping>  <servlet-name>foobla</servlet-name>
                     <url-pattern>/</url-pattern> </servlet-mapping>
</web-app>
```

- reachable via URL `localhost:8080/XYZ/`

Communication with Servlets: HTTP methods GET and POST

The servlets (virtually) run continuously in the Servlet Container and wait for incoming calls ...

HTTP GET and POST: request-response paradigm

HTTP GET should be used only if invocation does *not* change

- Request consists only of URL+parameters:
  `http://www.example.org/mondial?type=city&code=D&name=Berlin&province=Berlin`

HTTP POST should be used if it has side effects or changes the state of the Web Service

- Request URL consists only of the plain URL,
- parameters (e.g. queries using forms) or any other information is sent via a stream

⇒ often also queries use POST

Response: always as a stream.

- other HTTP methods PUT (resource), DELETE (resource) are used in REST (Representational State Transfer) “architectures”
  (e.g. the eXist XML database and document management system uses REST)

Content of the Response

- if the service is invoked via the browser (forms; e.g. the XQuery-Demo), the response contents is the HTML code that is shown as "Web page" to the user.
- The “page” that is shown initially:
  - static index.html in the WebRoot directory (servletdemo), or
  - *answer* dynamically generated by the servlet on the first GET request (HTTP GET http://www.semwebtech.org/xquery-demo).
- if the service is invoked by another Web Service, the answer contains data (this course: in XML form).

Simple GET: “Content” of the Request

- A simple GET (from filling a Web form) carries the parameters as extension to the URL:
  http://www.semwebtech.org/xquery-demo?query-text=//country[name='Germany']
  https://univz.uni-goettingen.de/qisserver/?search=3&raum.dtxt=2.101
  (simplified)

HTML Forms: invoking Web Services via Browser

The following elements (and several others) can be used in HTML pages:

`<form action="./sum" method="get">`	will call *thisURL*/sum, HTTP GET
`<input type="text" name="a"/>`	form field to type in parameter "a"
`<input type="text" name="b"/>`	form field to type in parameter "b"
`<input type="submit"/>`	click here to submit HTTP GET ...
`</form>`	with the parameters a and b

- the call URL is e.g. `http://localhost:8080/servletdemo/sum?a=4&b=5`
- an HTML page can contain multiple separate forms, to the same or different URLs/"actions",
- cf. servletdemo/WebRoot/index.html

Servlet Programming

- event-driven (cf. SAX): on incoming HTTP connections, the servlet container calls the servlets' doGet() (=react-on-get) and doPost() (=react-on-post) methods.

```
public class MyServlet extends HttpServlet
{ public void init(ServletConfig cfg) throws ServletException
    {  // initialization ...
       // read web.xml init params by  cfg.getInitParameter(...);
    }
  protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException  { ... }
  protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException  { ... }
}
```

- doGet() and doPost() both read the HttpServletRequest and write the HttpServletResponse object,
- the HttpServletRequest differs for GET (simpler) and POST (including a stream),
- the HttpServletRequest always provides a stream.

Recall:

- the distribution of connection URLs to projects is done according to tomcat's server.xml,
- the distribution inside of the project to servlets is done according to the project's web.xml,
- multiple URLs can be mapped to the same method (doGet/doPost) of the same servlet (Demo: FormatServlet)

⇒ must be analyzed in doGet() and doPost().

- Request.getPathInfo(): contains the tail of the URL path *after* the mapping by web.xml (non-null if `<url-pattern>/*</url-pattern>`)
- Request.getServletPath(): contains the tail of the URL path that is exploited for mapping according to the web.xml.

```
doGet/doPost(HttpServletRequest req, HttpServletResponse resp) throws ...
{ String path = req.getPathInfo();
  if (path == null) path = req.getServletPath();
  if (path.startsWith("/reset")) { ... }
  else if (path.startsWith("/format")) { ... }
  else if (path.startsWith("/all")) { ... }
}
```


Servlet Programming: Read Parameters and Contents

- GET and POST Requests can have parameters; POST can also have contents
- in doGet() and doPost() for accessing parameters:

```
java.util.Map<java.lang.String, java.lang.String[]> mymap =
                     req.getParameterMap();
String strA = req.getParameter("a");
```

(always Strings!)

- in doPost() for reading contents:

```
ServletInputStream in = req.getInputStream();
```

retrieves the body of the (POST) request (as binary data) using a ServletInputStream, where any Reader (e.g. a StAX XMLStreamReader) can be put on (usually, set reader's encoding to UTF-8).

`java.io.BufferedReader r = req.getReader();` retrieves the body of the (POST) request as character data (according to character encoding decl of the body) using a BufferedReader.

For instance, one can create a JDOM from the contents:

```
BufferedReader in = req.getReader();
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(in);
Element root = doc.getRootElement();
```

Servlet Programming: Write into a Response

- doGet() and doPost() provide the HttpServletResponse object of the HTTP connection,
- it consists mainly of a stream,
- The requesting service (Browser, Web Service) has a Reader waiting on the stream (see next slide).
- `PrintWriter out = resp.getWriter();`
  yields a Writer to the response – send character text (or XML events).
- `ServletOutputStream os = resp.getOutputStream();`
  yields an output stream that can directly fed with write(), print(), println() or can be connected to another stream. Don't forget os.flush() and os.close().

Invoking a new HTTP Connection (to a Web Service)

(servletdemo: MakeCallsServlet)

- (Http)UrlConnection object is created by invoking the openConnection method on a URL;
- below: urlstr is a string, in the GET case already with parameters.

```
HttpURLConnection.setFollowRedirects(true);   // static
HttpURLConnection con = (HttpURLConnection) new URL(urlstr).openConnection();
con.setRequestMethod("GET or POST");
  con.setDoInput(true);      // can be omitted - default is true
  con.setDoOutput(true);     // default is false(!), for "get" it's OK
  con.setRequestProperty("Connection", "keep-alive"); // is answer takes longer
  con.setRequestProperty("Content-type", ...);
  con.setRequestProperty("Accept", "text/xml");
con.connect();
-- use con.getOutputStream() to write contents of the request
con.getOutputStream().close();
-- use con.getInputStream() to read contents of the response
-- BufferedReader in =
     new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
con.getInputStream().close();
```


Code: HTTP GET

- Parameters given with the URL:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
public class HttpGetSimple {
  public static void main(String[] args) { try {
     BufferedReader br = null;
     URL inputURL = new URL("http://www.semwebtech.org/xquery-demo/" +
          "?action=query&query-text=//country[1]");
     HttpURLConnection con = (HttpURLConnection) inputURL.openConnection();
     con.setRequestMethod("GET");
     con.connect();
     String s = "";   StringBuffer res= new StringBuffer();
     br = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
     while ((s = br.readLine()) != null) { res.append(s+ "\n"); }
     br.close();
     System.out.println(res);
  } catch (Exception e) { e.printStackTrace(); } }}
```

[Filename: java/HttpGetSimple.java]

Code: HTTP POST – Parameters in the Request

- https://docs.oracle.com/javase/tutorial/networking/urls/readingWriting.html

```
import java.io.BufferedReader;        import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.net.HttpURLConnection;    import java.net.URL;
public class HttpPostSimple {
  public static void main(String[] args) { try {
     BufferedReader br = null;
     URL inputURL = new URL("http://www.semwebtech.org/xquery-demo/");
     String params = "action=query&query-text=//country[1]";
     HttpURLConnection con = (HttpURLConnection) inputURL.openConnection();
     con.setRequestMethod("POST");
     con.setDoOutput(true);    // default is false(!)
     con.connect();
       OutputStreamWriter wr = new OutputStreamWriter(con.getOutputStream());
       wr.write(params);
       wr.flush();  wr.close();
     String s = "";   StringBuffer res= new StringBuffer();
     br = new BufferedReader(new InputStreamReader(con.getInputStream(), "UTF-8"));
     while ((s = br.readLine()) != null) { res.append(s+ "\n"); }
     br.close();  System.out.println(res);
  } catch (Exception e) { e.printStackTrace(); } }}
```

[Filename: java/HttpPostSimple.java]

HTTP Access in the Data Management Area

- HTTP GET and POST are important means to access “Deep Web” data via queries against forms, and “Linked Open Data” (LOD) (RDF data, [Semantic Web lecture]).

Alternative: [not tested]

- Connection getContent() method:
  returns an Object whose type is determined by the the content-type header field of the response. Uses a ContentHandler to convert data based on its MIME type to the appropriate class of Java Object.
- maybe useful for binary types?
- or even URL.getContent() as a shortcut for openConnection().getContent();
  String foo = (String) url.getContent();
  seems to be useful for plain GET on HTML pages;
- for XML content, using the stream seems to be more useful
  (→ SAXBuilder→ JDOM, or → StAX)

Notes on Handling Character Encodings

- default for WebServices is ISO-8859-1 (covers german umlauts, swedish etc.)
- then, for HTML forms, set also
  `<form method="get/post" accept-charset="ISO-8859-1">`
- UTF-8 also covers chinese, persian, etc. (localnames in Mondial)
- Web Service side:
  - if HTTP GET is used, request character encoding can only be set *globally* (Apache tomcat: URIEncoding attribute of the <Connector port="..."> element in server.xml to UTF-8).
  - HTTP POST: request.setCharacterEncoding(“UTF-8”) before reading parameters or contents (e.g. DBIS XQuery and SQL Web Interfaces);
  - use also response.setCharacterEncoding(“UTF-8”)

Data Exchange: an Integrated XML Perspective

- HTTP connections are Unicode.
- exchanging XML via HTTP basically works on its serialization
  - explicitly working with Reader→String/StringBuffer and String/StringBuffer→Writer is possible, but often not necessary;
  - in:
    - let a SAXBuilder build a JDOM,
    - put SAX or an StAX XMLEventReader on the InputStream,
    - put a JAXB Unmarshaller on the InputStream,
    - put the Digester on the InputStream,
    - cf. Examples where these were put on the FileInputStream for mondial.xml.
  - out:
    - serialize XML by putting an XMLEventWriter on the OutputStream,
    - let JAXB write into it, ...

A Note on Multithreading

- servlets can be instantiated by the container permanently or on-demand.
- if multiple requests for the same servlet come in, the servlet container can run multiple threads on the same instance of a servlet.
  - be careful with instance variables,
  - implement mutual exclusion if necessary
- the servlet container can also create (and remove) additional instances of a servlet.

PHP in Tomcat

- Tomcat is Java-based,
- Embedded PHP in HTML files or pure PHP is not executed by default.
- Name HTML files that include embedded PHP (cf. Slide 186) *filename*.php,
- there are several implementations of PHP in Java,
- e.g. see `https://stackoverflow.com/questions/779246/run-a-php-app-using-tomcat/779319` and `http://www.studytrails.com/blog/php-on-a-java-app-server-apache-tomcat-using-quercus/`

Chapter 12
Between Relational Data and XML

Data integration between “Legacy Systems” and XML databases

- Note: “legacy” now means SQL ...

Mixing everything up ...

Access to data stored in relational databases by

- using an XML environment (e.g., saxon) and mapping relational data from a remote SQL database (e.g. Oracle) to XML, and working with it.
- using XML-world concepts and languages in an SQL environment, e.g. for exchanging data in XML format (again, e.g., Oracle).

(Note that IBM DB2/XML Extender and MS SQL Server provide similar functionality)

12.1 Publishing/Mapping Relational Data in XML

Several *generic* mappings are possible:

Consider country(name: “Germany”, code: “D”, population: 83536115, area: 356910)

- tables, rows, and subelements

```
<table name="country">
 <row><name>Germany</name><code>D</code>
      <population>83536115</population><area>356910</area></row>
    :
</table>
```

- tuples with subelements

```
<country><name>Germany</name><code>D</code>
         <population>83536115</population><area>356910</area></country>
    :
```

- analogous with XML attributes
- advantage with subelements (vs. attributes): SQL values can also be object types (mapped to XML) and XMLType contents!

Example: HTTP-based XML access to Oracle [Oracle 10/11, ∼2006]

The whole database is mapped (virtually) to XML:

```
<SCHMIDT>   -- user name as root element
  <COUNTRY>  -- all names are capitalized
    <ROW><NAME>Germany</NAME><CODE>D</CODE>
      <POPULATION>83536115</POPULATION><AREA>356910</AREA></ROW>
    :
  </COUNTRY>
  :
</SCHMIDT>
```

Access by extended URL notation:

- URL: *computer*:*port*/oradb/*user*/*tablename*/ROW[*condition*]
- capitalize user, table and attribute names
- URL must select a single element (whole table, or single row)

```
ap34.ifi...:8080/oradb/DUMMY/COUNTRY            %% show page source
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']/NAME
ap34.ifi...:8080/oradb/DUMMY/COUNTRY/ROW[CODE='D']/NAME/text()
```

Generic Mappings (Cont'd)

Up to now: mapping of materialized base tables.
Problem: how to map the result of a query with computed columns?

```
SELECT Name, Population/Area FROM country
```

- tables, rows, and subelements:
  the DTD is independent from the relational schema
  metadata is contained in the attributes
  ("JDBC-style" processing of result sets)

```
<table name="country">
  <row><column name="name">Germany</column>
       <column name="population/area">83536115</column>
       <column name="area">234.05473</column>
  </row>
   :
</table>
```

- another "most generic mapping" as (object, property, value) to be discussed later ...

Additionally: often, tools define their own access functionality ...

Access to SQL Databases with saxon-XSLT (saxonEE only)

- uses JDBC technology for remote access (at least for Java XSL tools)
- defines namespace "sql"
- <sql:connect> with attributes "database" (JDBC url), "driver" (JDBC driver)
  returns a JDBC connection object as a value of type "external object" that can be bound to a variable, e.g. $connection.
  Note: there can be several connections at the same time.
- <sql:query> with following attributes allows to state an SQL query whose result is generically mapped to XML:
  - connection
  - table: ... the "FROM" clause
  - column: ... the "SELECT" clause
  - where: optional condition
  - row-tag: tag to be used for rows (default: "row")
  - col-tag: tag to be used for columns (default: "col")

  result is a collection of <row> ... </row> elements that can e.g. be bound to a variable.

Administrative Parameters

```
<xsl:stylesheet
  xmlns:sql="http://saxon.sf.net/sql"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
  extension-element-prefixes="sql">

<!-- insert your database details here, or supply them in parameters -->
<xsl:param name="driver" select="'oracle.jdbc.driver.OracleDriver'"/>
<xsl:param name="database" select="'jdbc:oracle:thin:@IPADDRESS:1521/USER'"/>
<xsl:param name="user" select="'USER'"/>
<xsl:param name="password" select="'PASSWD'"/>

<xsl:variable name="connection" as="java:java.sql.Connection"
      xmlns:java="http://saxon.sf.net/java-type">
  <sql:connect driver="{$driver}" database="{$database}"
               user="{$user}" password="{$password}">
    <xsl:fallback>
      <xsl:message terminate="yes">SQL extensions not installed</xsl:message>
    </xsl:fallback>
  </sql:connect>
</xsl:variable>
</xsl:stylesheet>
```

[Filename: SaxonSQL/sql-administrativa-fake.xsl]

Example Access/Query

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
  xmlns:sql="http://saxon.sf.net/sql">
<xsl:include href="sql-administrativa.xsl"/>

<xsl:template match="*">
 <sql:query connection="$connection" table="country" column="*"/>
 <sql:close connection="$connection"/>
</xsl:template>
</xsl:stylesheet>
```

[Filename: SaxonSQL/sql-query.xsl]

uses a primitive mapping that relies on the order of columns.

Installation [saxonEE with SQL]

Put the following in a directory [here: SaxonSQL]:

- saxon9ee.jar, saxon9-sql.jar (both from the downloaded Saxon.zip)
- ojdbc7.jar (or any JDBC driver for the database)
- saxon-license.lic (the 30 days license)
- config.xml:

```
<configuration xmlns="http://saxon.sf.net/ns/configuration" edition="EE">
  <xslt>
    <extensionElement namespace="http://saxon.sf.net/sql"
                      factory="net.sf.saxon.option.sql.SQLElementFactory"/>
  </xslt>
</configuration>
```

[Filename: SaxonSQL/config.xml]

- some XML file "dummy.xml" (might even use the xsl stylesheet as dummy XML)
- `java -cp saxon9ee.jar:saxon9-sql.jar:ojdbc7.jar net.sf.saxon.Transform -config:config.xml -s:dummy.xml -xsl:sql-query.xsl`

Example Access/Query

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
  xmlns:sql="http://saxon.sf.net/sql"
  extension-element-prefixes="sql">
<xsl:include href="sql-administrativa.xsl"/>

<xsl:template match="*">
 <xsl:variable name="result">
 <sql:query connection="$connection" table="country,encompasses"
     where="country.code=encompasses.country"
     column="country.name,encompasses.continent,percentage"
     row-tag="country" column-tag="bla"/>
 </xsl:variable>
 <sql:close connection="$connection"/>
 <!-- and now use the variable somehow, here simply output it -->
 <xsl:copy-of select="$result"/>
</xsl:template>
</xsl:stylesheet>
```

[Filename: SaxonSQL/sql-query2.xsl]

Further Commands

- <sql:insert> with attribute
  - attribute: table="..."
  - children: <column name="..." select="..."/>
    value can also be given as contents of the column element; currently always interpreted as a string.
- <sql:close> with a "connection" attribute

See also `http://www.saxonica.com`.

12.2 The SQL/XML or SQLX standard

Goal: Coexistence between Relational Data and XML

- mapping relational data to XML
  - by a default mapping (previous section)
  - a (user-defined) XML views over relational data ("XML publishing")
- storing XML data in relational systems
  - data-centric XML: efficient, several possibilities
  - document-centric XML: problematic

SQL/XML

- draft for an ISO standard since 2003: `www.sqlx.org`
- an SQL datatype "XMLType" for storing XML as "value" in databases:

  Jim Melton (Oracle): *"SQL/XML includes an XML datatype for use inside SQL. This allows the SQL engine to automatically 'shred' data-oriented XML documents for storing some elements and contents in one place while keeping other elements in other places. Using indexes and joins, you can then bring this data back together very quickly."*

  – theory: an abstract datatype with operators/methods
    (cf. Computer Science I lecture)
  – API: like (user-defined) object types in object-relational databases
    (cf. SQL database lab)
    - ⋆ handled with special methods (constructors, selectors) in SQL
    - ⋆ can be exported to XML tools, e.g. by JDBC
      (either as DOM instance, or *serialized* as Unicode file/stream)
    - ⋆ libraries provide additional functions for processing XML in PL/SQL.
  – internal implementation: not seen by the user
    (i) shredding or (ii) storing as LOB (Large Object)

SQL/XML

Making XML data a first-class citizen in relational databases,
seamless flow from relational data to XML and back

SQL to XML

- XML generation from SQL queries (i.e., in the SELECT clause)
  (e.g., as packets for data exchange)
- define XMLType views over SQL data

SQL access and manipulation of XML inside the RDB

... use XPath expressions inside SQL (and rise the question what is actually the difference to XQuery):

- storing XML in RDB (e.g. if XML-data exchange packets came in),
- XPath-based extraction of XML content (SELECT clause),
- XPath-based query of XML content (WHERE clause),
- XPath-based update of XML content (in SQL),
- define XPath-based relational views over XML content.

"XML" AS AN SQL DATATYPE

XML/XMLType: an *SQL* datatype to hold XML data:

CREATE TABLE mondial OF XMLType;   use as Row Object Value
CREATE TABLE CountryXML OF XMLType;   use as Row Object Values

As column object type in relational tables:

```
CREATE TABLE CityXML
  (name XMLType,  province VARCHAR2(50),  country VARCHAR2(4),
   population XMLType,
   coordinates XMLType);
```

[Filename: SQLX/cityxmltable.sql]

- generation: INSERT INTO *table* VALUES (... , XMLType('*XML as unicode string*') ...)

```
INSERT INTO CityXML
VALUES(XMLType('<name>Clausthal</name>'), 'Niedersachsen', 'D',
 XMLType('<population year="2004">10000</population>'),
 XMLType('<coordinates><latitude>51.8</latitude><longitude>10.4</longitude>
          </coordinates>'));
```

[Filename: SQLX/cityxmltuple.sql]

HANDLING OF SQL XMLTYPE DATATYPE

- generate it by certain *constructors* ("XML Publishing Functions")
- storage: chosen by the database
  – "shredding" and distributing over suitable tables (of object-relational object types)
    (queries are translated into SQL joins/dereferencing)
  – storing it as VARCHAR, CLOB (Character Large Object), or as separate file
    (the remainder of this section uses CLOB)
  – storing it "natively"
- query it by XPath
- for export/exchange in Unicode:
  XMLSerialize: a function to serialize an XML value as a Unicode character string (not available in sqlplus, only in PL/SQL):
  XMLSerialize: XMLType $\rightarrow$ String
- additional methods provided by PL/SQL libraries,
- XML objects can also be used e.g., as documents or as stylesheets, applied to documents (by PL/SQL libraries).

How to get XMLType Instances

- by the *opaque* constructor
  XMLType: STRING → ELEMENT
  that generates an XMLType instance from a Unicode string
  - the inverse to Java's `to_string`,
  - nearly all datatypes have such an opaque constructor (e.g., for lists: list("[1,2,3,4,5]"));
- generate instances recursively by structural constructors that are closely related to the underlying *Abstract Datatype*
  (cf. binary trees, lists, stacks in Computer Science I) (see Slide 639 ff.);
- or load them from an XML file (that then actually contains the Unicode serialization and uses the opaque constructor).

12.2.1 Loading XMLType Instances (here: oracle-specific)

- This section describes two implementational variants that have to be configured and prepared by the admin.
- Oracle (here: 12c, 2016) has problems with UTF-8, maps it to what it calls "UTF8"; this causes problems with some characters [occured with "from Web" variant, seem not to occur for "from file" variant].
- Full UTF-8 is obtained by declaring the encoding as oracle-specific "AL32UTF8".

Loading XMLType from Local Files (Oracle, tested WM 27.7.2016 12.1.0.1.0)

- choose a directory on the same computer where the DBMS server (!) is installed (e.g. /tmp or /documents/xml),
- copy XML file (e.g. m.xml) into that directory
  - the XML file must not contain a reference to a DTD!
  - the file must be readable publicly or for the "oracle" user.
- tell Oracle where it finds XML files:
  - centrally by admin:

    ```
    admin: CREATE OR REPLACE DIRECTORY XMLDIR AS '/tmp' ;
           GRANT READ ON XMLDIR TO public (or to scott etc);
    ```
  - users might do it by themselves:

    ```
    admin: GRANT CREATE DIRECTORY TO scott;
    scott: CREATE OR REPLACE DIRECTORY XMLDIR AS '/tmp';
    ```
-

  ```
  insert into mondial
  values(xmltype(bfilename('XMLDIR', 'mondial.xml'),
                 nls_charset_id('AL32UTF8')));
  ```

- see documentation: an XML schema might be used; STORE AS OBJECT RELATIONAL

Loading XML Files: Local Solution

- for importing XML files, our local installation provides a method `system.getxml('`*http-url*`')`. (restricted to access dbis.informatik)

```
SELECT system.getxml(
  'http://www.dbis.informatik.uni-goettingen.de/Teaching/DBP/XML/mondial.xml')
FROM dual;
```

or, inserting it into a table:

```
INSERT INTO mondial VALUES( system.getxml(
  'http://www.dbis.informatik.uni-goettingen.de/Teaching/DBP/XML/mondial.xml'));
```

[Filename: SQLX/insertmondial.sql]

- the XML file must not contain a reference to a DTD!
- the file can e.g. reside in the local homedirectory or anywhere in the Web (the DB admin must configure the Oracle firewall to allow to access (certain) Web URLs).
- the file must be publicly readable – `chmod` *filename* `644`

```
SET LONG 10000;
SELECT * FROM mondial;
```

Aside: the getXML procedure

```
- execute as 'system' user (not by "CONNECT / AS SYSDBA"):
CREATE OR REPLACE FUNCTION getXML(url IN VARCHAR2)
RETURN XMLType
IS
 x UTL_HTTP.html_pieces;
 tempCLOB CLOB := NULL;
 s varchar2(2100) := null;  -- request pieces:
                            -- max length will be 2000
 s1 varchar2(2100) := null;
BEGIN
 x := UTL_HTTP.request_pieces(url, 10000);
 DBMS_LOB.createTemporary(tempCLOB, TRUE,
                          DBMS_LOB.SESSION);
 IF x.COUNT > 0 THEN
  -- In the xml encoding declaration, replace UTF-8 by AL32UTF8
  -- '' -> sqlplus escape of ' ; \1 references to matched  (...)
  s1 := REPLACE(x(1),
     '\s*(<?xml [^>]*encoding)\s*=\s*["'']UTF-8["'']',
     '\1="AL32UTF8"');
  -- remove DTD reference
  s := REGEXP_REPLACE(s1,'<!DOCTYPE [^>]+>','');
  DBMS_LOB.writeAppend(tempCLOB, LENGTH(s), s);
  FOR i IN 2..x.COUNT LOOP
   DBMS_LOB.writeAppend(tempCLOB, LENGTH(x(i)), x(i));
  END LOOP;
 END IF;
 RETURN XMLType(tempCLOB);
END;
/
GRANT EXECUTE ON getXML TO PUBLIC;
```


Aside: the getXML Procedure

- Allow users to use HTTP access (to certain URI patterns) via Access Control Lists (ACLs, admin only)

Aside: Notes

- UTF-8 encoding supports character sets of even exotic languages (local names of cities in Mondial).
- Thus, for any XML file somewhere in the Web whose encoding is declared "UTF-8", this must be changed into "AL32UTF8".
- Additionally, the DTD reference must be removed (here: 12c, 2016).

⇒ do this in the getXML procedure

- the HTTP stream is read piecewise (of 2000 chars per piece)

⇒ replace in the first piece.

12.2.2 SQL/XML: Generating XML by XML Publishing Functions

The SQL/XML Standard defines "XML publishing functions" that act as constructors (the name comes from the fact that they are also used to publish relational data in XML format):

- constructors of the recursively defined abstract datatype "XMLType",
- create fragments or instances of XMLType,
- usage in the same way as predefined or user-defined functions (e.g., in the SELECT clause),

Some Theory: the Abstract Datatype

... constructors of the recursively defined abstract datatype "XMLType":

Sub-datatypes:

- ELEMENT for element nodes
- ATTRIBUTE for attribute nodes
- QNAME for names of elements and attributes
  (restriction of STRING without whitespaces etc.)
- STRING for text values (text nodes and attribute values)
- TUPLE(*type*) for a tuple of instances of *type*
- TABLE(*type*) for a table of instances of *type*

Constructors are very similar to those of XQuery (in the return clause), e.g.,:

`element` *`name attrs-and-content`*

and those of XSLT: `<xsl:element name="...">` *content* `</xsl:element>`
and those of the ... DOM.

(always the same abstract datatype, but expressed with different syntaxes)

SQL/XML Publishing Functions: Overview

Basic constructors:

- XMLType: generates an XMLType instance from a Unicode string ("opaque constructor")
  XMLType: STRING $\rightarrow$ ELEMENT
- XMLElement: generates an XML element with a given name and content (either text (simple contents) or recursively created XML (complex contents) or mixed
  XMLElement: QNAME $\times$ (STRING $\cup$ ELEMENT $\cup$ ATTRIBUTE)* $\rightarrow$ ELEMENT
  XMLElement: QNAME $\rightarrow$ ELEMENT for empty elements
- XMLAttributes: generates a one or more attribute nodes from a sequence of name-value-pairs
  XMLAttributes: (QNAME $\times$ STRING)$^+$ $\rightarrow$ ATTRIBUTE$^+$

SQL/XML Publishing Functions: Overview (Cont'd)

Further constructors:

- XMLForest: a function to generate a sequence, called a "forest," of XML elements with simple contents from a *sequence* of name-value-pairs
  XMLForest: (QNAME $\times$ STRING)$^+$ $\rightarrow$ ELEMENT$^+$
  (note: the analogue to XMLAttributes for simple elements)
- XMLAgg: a function to group, or aggregate, XML data vertically from a column into a sequence of nodes
  XMLAgg: COLUMN(XMLTYPE) $\rightarrow$ XMLTYPE*
- XMLConcat: a function to concatenate the components of a (horizontal) SQL tuple into a sequence
  XMLConcat: TUPLE(XMLTYPE$^+$) $\rightarrow$ XMLTYPE*
  (note that a tuple is also different from a list as in XMLForest!)
- [XMLNamespaces: a function to declare namespaces in an XML element]

Constructing XML Elements from atomic SQL entries

Basic form: XMLElement

- XMLElement: Name $\times$ Element-Body $\rightarrow$ Element:
  - Element-Body: text or recursively generated (attributes, elements, mixed)

```
SELECT XMLElement(x) FROM DUAL;
```

(note: this result is not correct: `<X/>` is an empty Element, while `<X></X>` is an element with the empty string as contents!)

```
SELECT XMLElement("Country",'bla') FROM DUAL;
SELECT XMLElement(Country,'bla') FROM DUAL;
```

- note: using "..." to indicate non-capitalization (otherwise the whole name is capitalized). (note that single and double "..." must be used exactly as in the example).
- Note that the first argument is always interpreted as a string:
  `SELECT XMLElement(name, code) FROM Country;`
  yields `<NAME>AL</NAME>, <NAME>GR</NAME>` etc.

Elements with Non-Empty Content

- XMLElement: second argument contains the element body (attributes, subelements, text),
- XMLAttributes: list of name-value pairs that generate attributes.

```
SELECT XMLElement("Country",
  XMLAttributes(code AS "car_code", capital AS "capital"),
  name,
  XMLElement("Population",population),
  XMLElement("Area",area))
FROM country
WHERE area > 1000000;
```

[Filename: SQLX/xmlelement.sql]

A result element:

```
<Country car_code="R" capital="Moscow">
  Russia
  <Population>148178487</Population>
  <Area>17075200</Area>
</Country>
```

Optional Substructures

- XML as abstract datatype, *functional* constructors
- semistructured data: flexible and optional substructures

```
SELECT XMLElement("City",
  XMLAttributes(country AS country),
  XMLElement("Name",name),
  CASE WHEN latitude IS NULL THEN NULL
       ELSE XMLElement("Latitude",latitude) END,
  CASE WHEN longitude IS NULL THEN NULL
       ELSE XMLElement("Longitude",longitude) END
  )
FROM city;
```

[Filename: SQLX/xmlelement2.sql]

- Note: `CASE WHEN` *cond* `THEN` *a* `ELSE` *b* `END`
  is a *functional* construct
  (like in “`if`” in XQuery and `<xsl:if>` in XSLT)

CONSTRUCTING XML: SEQUENCES OF ELEMENTS

XMLForest: short form for simple elements

```
SELECT XMLElement("Country",
          XMLForest(name AS Name,
                    code AS car_code,
                    population AS "Population",
                    area AS "Area"))
FROM country
WHERE area > 1000000;
```

[Filename: SQLX/xmlforest.sql]

```
<Country>
  <NAME>Brazil</NAME>             <!-- note capitalization -->
  <CAR_CODE>BR</CAR_CODE>
  <Population>162661214</Population>
  <Area>8511965</Area>
</Country>
```

⇒ canonical mapping from tuples to XML elements with simple content.

Subqueries

Contents can also be generated by (correlated) Subqueries:

```
SELECT XMLElement("Country",
         XMLAttributes(code AS "car_code"),
         XMLElement("Name",name),
         XMLElement("NoOfCities",
                    (SELECT count(*)
                     FROM City
                     WHERE country=country.code)))
FROM country WHERE area > 1000000;

SELECT XMLElement("Country",
         XMLAttributes(code AS "car_code"),
         XMLElement("Name",name),
         (SELECT XMLElement("NoOfCities",count(*))
          FROM City
          WHERE country=country.code))
FROM country WHERE area > 1000000;
```

[Filename: SQLX/xmlsubquery.sql]

Constructed XML can then be used for filling tables:

FILLING A TABLE ... WITH XML ROW VALUES

```
CREATE TABLE CountryXML OF XMLType;

INSERT INTO CountryXML
(SELECT XMLElement("Country",
  XMLAttributes(code AS "Code",
                population AS "Population"),
  XMLElement("Name",name),
  XMLElement("Area",area),
  (SELECT XMLElement("Capital",
            XMLForest(name AS "Name",
                      population AS "Population"))
   FROM city
   WHERE country=country.code
     AND city.name=capital))
FROM country);
```

[Filename: SQLX/fillcountry.sql]

Filling a table: XML Column Values

```
CREATE TABLE CityXML
   (name XMLType,
    province VARCHAR2(50),
    country VARCHAR2(4),
    population XMLType,
    coordinates XMLType);
INSERT INTO CityXML
(SELECT XMLElement("name", name),  province, country,
  CASE WHEN population IS NULL THEN NULL
       ELSE XMLElement("population", XMLAttributes(2010 as year), population)
       END ,
  CASE WHEN longitude IS NULL THEN NULL
       ELSE XMLElement("coordinates",
              XMLElement("latitude", latitude),
              XMLElement("longitude", longitude))
       END
 FROM city);
```

[Filename: SQLX/fillcity.sql]

Constructing XML from XML-in-SQL: Restructuring

- Relational databases have
  - literals (and objects for object-relational tables),
  - tuples,
  - tables.
- XML structures have
  - sequences
  - nesting (generated by XMLElement or by nested subqueries)

Create XML structures from XML-in-SQL content nodes:

- horizontally SQL/XML-to-XML: create an XML node sequence from a tuple (with XML values) [note: XMLForest does not create a sequence from a tuple, but from a list that is generated from an SQL tuple], or
- vertically SQL/XML to XML: create an XML node sequence from a column (or some rows of that column) that holds XML nodes?

Constructing XML: Grouping into a Sequence

Aggregated Lists

- XMLAgg: is a new SQL aggregate function (like `count()` or `sum()`), that does not return a single value but the sequence of elements in that column.
- Note: XMLAgg is not applied to a *sequence* of XML elements, but to a *column* holding XML elements!

Simplest case: XMLAgg over a table of XMLType row objects:

```
SELECT XMLElement("Cities",
  (SELECT XMLAgg(name)
   FROM CityXML c))
FROM DUAL;
```

[Filename: SQLX/xmlagg.sql]

Result:

```
<Cities>
  <name>...</name>
  <name>...</name>
   :
</Cities>
```


Constructing XML: Grouping

Aggregated Lists

... another example

- create a sequence of XML nodes from XML nodes generated as 1-column query result (table):

```
SELECT XMLElement("Continents",
  (SELECT XMLAgg(XMLElement("Continent", XMLAttributes(name AS "Name",
                                                        area AS "Area")))
   FROM continent))
FROM DUAL;
```

[Filename: SQLX/xmlagg2.sql]

Result:

```
<Continents>
  <Continent Name="Europe" Area="..."/>
  <Continent Name="Asia" Area="..."/>
  :
</Continents>
```

Constructing XML: Nested Grouping

Grouping/Aggregation: Nested Lists

- XMLAgg: generate a collection from the tuples inside of a GROUP BY:
  In XML, this *list* of items can also be used!

... now we can have the number of cities in a country, together with a list of them:

```
SELECT XMLElement("Country",
        XMLAttributes(country AS car_code),
        XMLElement("NoOfCities", count(*)),
        XMLAgg(XMLElement("city",name) ORDER by population))
FROM city
GROUP BY country;
```

[Filename: SQLX/xmlgroupagg.sql]

Element of the result set:

```
<Country CAR_CODE="D">
  <NoOfCities>85</NoOfCities>
  <city>Erlangen</city> <city>Kaiserslautern</city> ... <city>Berlin</city>
</Country>
```


Constructing XML: Mapping Tuples into Sequences

XMLConcat

- takes a tuple of XML elements and transforms them into a sequence:
  (note: only elements allowed, cannot create mixed contents)

```
SELECT XMLElement("City", XMLConcat(name, population, coordinates))
--  XMLConcat(name, country -- varchar not allowed population, coordinates)
FROM CityXML
WHERE country='D';
```

[Filename: SQLX/xmlconcat.sql]

An element of the result set:

```
<City>
 <name>Berlin</name>
 <population>...</population>
 <coordinates><latitude>...</latitude><longitude>...</longitude></coordinates>
</City>
```


Same example with Mixed Content directly by XMLElement

```
SELECT XMLElement("City", name, country, population, coordinates)
FROM CityXML
WHERE country='D';
```

[Filename: SQLX/xmlconcat.sql]

An element of the result set:

```
<City>
 <name>Berlin</name>
 D
 <population>...</population>
 <coordinates><latitude>...</latitude><longitude>...</longitude></coordinates>
</City>
```

- XMLConcat directly in XMLElement does not make much sense, can be omitted.

Example: XMLConcat and XMLAgg

- the GROUP BY from Slide 653 can equivalently be expressed by using a (correlated) (Sub)query that returns a tuple for each country (consisting of the number and the aggregation of all cities):

```
SELECT XMLElement("Country",
  XMLAttributes(code AS code),
  XMLElement(name, name),
  (SELECT XMLConcat(
     XMLElement("NoOfCities", count(*)),
     XMLAgg(XMLElement("city",name)))
   FROM City
   WHERE country=code))
FROM country;
```

[Filename: SQLX/xmlconcatagg.sql]

12.2.3 Map XMLType to String for Data Exchange

- the "user interface" sqlplus automatically shows XMLType data in its serialized XML form.
- for transmitting XML data e.g. via HTTP as a unicode stream, it must first be serialized into a VARCHAR2 (PL/SQL fragment):

  SELECT XMLSerialize(CONTENT value(m)) FROM mondial m;

  (just looks "normal")

```
set serveroutput on;
declare s VARCHAR2(1000);
begin
  SELECT XMLSerialize(CONTENT c.name)
  INTO s
  FROM cityXML c
  WHERE country='MC';
  dbms_output.put_line(s);
end;
/
```

[Filename: SQLX/xmlserialize.sql]

12.2.4 Handling XML Data from within SQL

- recall: XMLType is defined as an abstract datatype.
- it also has *selectors* that provide an interface for standard XML languages
- signature:

  extract: XMLType $\times$ XPath_Expression $\rightarrow$ XMLType $\cup$ string
  extractValue: XMLType $\times$ XPath_Expression $\rightarrow$ string
  existsNode: XMLType $\times$ XPath_Expression $\rightarrow$ Boolean   for conditions
- implementation based on user-defined object types
  - cf. object-relational extensions to SQL
- above operations also available as member methods

```
SELECT extract(value(m), '//city[name="Berlin"]') FROM mondial m;
SELECT m.extract('//city[name="Berlin"]') FROM mondial m;
SELECT extractValue(value(m), '//country[@car_code="D"]/population[last()]')
FROM mondial m;
SELECT m.extractValue('//country[@car_code="D"]/population[last()]')
FROM mondial m;   -- buggy (since version 9 ... and still in 12c)!!!
```


SELECT: "Extract" Function

extract(XMLType_instance, XPath_string)
XMLType_instance.extract(XPath_string)

- First argument: selects an attribute with value of type "XMLType" in the current row (use value(.) function)
- Second argument: applies XPath_string to it
- Result: value of type XMLType or any other SQL type
  (multi-valued results are concatenated)

XML Row Values

Value of the row is of type XMLType: apply methods directly

```
SELECT extract(value(c), '/Country/@Code'),
       extract(value(c), '/Country/Capital/Name')
FROM CountryXML c;

SELECT c.extract('/Country/@Code'),
       c.extract('/Country/Capital/Name')
FROM CountryXML c;
```


Aside: Short Overview of XPath

(use the SQLX section in different lectures)

- Navigation as in Unix: /*step*/*step*/*step*
  /mondial/country/name
- capitalization is relevant!
- result: a sequence of XML nodes (not only values, but also trees):
  /mondial/country
- steps to deeper descendants: /mondial//city/name , //city/name
  (latter includes /mondial/country/city and /mondial/country/province/city)
- attributes: .../@*attributname*: /mondial/country/@area
- access to text contents: /mondial/country/name/text()
- evaluation of conditions during navigation:
  /mondial/country[@code='D']/@area
  /mondial/country[name/text()='Germany']/@area
- Comparisons automatically use only the text contents:
  /mondial/country[name='Germany']/@area

SELECT: "Extract" Function (Cont'd)

XML Column Values

Recall:

```
CREATE TABLE CityXML (name XMLType, province VARCHAR2(50),
    country VARCHAR2(4), population XMLType, coordinates XMLType);
```

CityXML.population is an XMLType column object:

```
SELECT extract(population,'/') FROM CityXML;
SELECT c.population.extract('/') FROM CityXML c;
SELECT name, extractValue(population,'/population/@YEAR'),
       extractValue(population,'/population')
FROM CityXML;
SELECT name, c.population.extract('/population/@YEAR').getNumberVal(),
       c.population.extract('/population/text()').getNumberVal()
FROM CityXML c
ORDER BY 3;
```

- exact capitalization in XPath argument!
- extractValue currently not implemented as member method (bug)
- use getNumberVal() and getStringVal() functions

Subqueries to XMLType in the WHERE clause

... for selecting and comparing values, use also extract():

```
SELECT name
FROM CityXML c
WHERE c.population.extract('/population/text()')
        .getNumberVal() > 1000000;

SELECT c.extract('/Country/Name/text()')
FROM CountryXML c
WHERE c.extract('/Country/@Population')
        .getNumberVal() > 1000000;
```

- Note: comparison takes place on the SQL level (WHERE) (→ join functionality when variables are used).
- Note: if the XPath expression returns a sequence of results, these are concatenated already during the evaluation of the extract() function ...
- ... thus, one has to use another way.

WHERE: "ExistsNode" Function

existsNode(XMLType_instance, XPath_string)

- Checks if item is of XMLType_instance, and XPath_string has a nonempty result set:
- note: the value for the comparison must be given in the XPath *string* – no joins on the SQL level possible.
- result: 1, if a node exists, 0 otherwise.

```
SELECT name, extractValue(population,'/population')
FROM CityXML
WHERE existsNode(population, '/population[text()>1000000]') = 1;

SELECT name, extractValue(population,'/population')
FROM CityXML c
WHERE c.population.existsNode('/population[text()>1000000]') = 1;
```


Updating XML Data

- the complete XMLType value is *changed*, not updated
- updateXML(...) as a (transformation) function!
  Note: the statement "SELECT updateXML(...) FROM ..." does not update the DB, but returns the value that would result from the update ("hypothetical update").

updateXML(XMLType_instance, (XPath_string, new_value)$^+$)

- first argument: SQL – selects an (SQL-)attribute of the current tuple (must result in an XMLType object),
- $2n$th argument: selects the node(s) to be modified by the value of the ...
- $2n + 1$th argument: new value,
- result: updates instance of type XMLType.
- Note: the expression "SELECT updateXML(...) FROM ..." does not change the database but returns only the value that *would* result from the update.

Updating XML Data (Cont'd)

```
SELECT updateXML(c.population,
                'population/text()','3600000',
                'population/@YEAR','2016')
FROM CityXML c WHERE extractValue(c.name,'name')='Berlin';

SELECT updateXML(value(c),
                '/Country/Name/text()','Fidschi')
FROM CountryXML c
WHERE extractValue(value(c),'Country/Name')='Fiji';
```

[Filename: SQLX/updatexml.sql]

Updating XML Data (Cont'd)

This function is then used in the SQL SET-Statement:

```
UPDATE CityXML c
SET c.population   -- an XMLType element
    = updateXML(c.population,
                'population/text()','3600000',
                'population/@YEAR','2016')
WHERE  extractValue(c.name,'name')='Berlin';

UPDATE CountryXML c
SET value(c) = updateXML(value(c),
                '/Country/Name/text()','Fidschi')
WHERE existsNode(value(c),'/Country[Name="Fiji"]') = 1
```



```
CREATE OR REPLACE FUNCTION xslexample RETURN CLOB IS
  xmldoc      CLOB;
  xsldoc      CLOB;
  myParser    dbms_xmlparser.Parser;
  indomdoc    dbms_xmldom.domdocument;
  xsltdomdoc  dbms_xmldom.domdocument;
  xsl         dbms_xslprocessor.stylesheet;
  outdomdocf  dbms_xmldom.domdocumentfragment;
  outnode     dbms_xmldom.domnode;
  proc        dbms_xslprocessor.processor;
  html        CLOB DEFAULT 'BLA';   -- must be initialized;
BEGIN
  -- Get the XML document as CLOB
  SELECT value(m).getClobVal() INTO xmldoc FROM mondial m;
  -- Get the XSL Stylesheet as CLOB
  SELECT s.stylesheet.getClobVal() INTO xsldoc
  FROM stylesheets s WHERE name='mondial-simple.xsl';

  -- Get the new xml parser instance
  myParser := dbms_xmlparser.newParser;
  -- Parse the XML document and get its DOM
  dbms_xmlparser.parseClob(myParser, xmldoc);
  indomdoc := dbms_xmlparser.getDocument(myParser);

  -- Parse the XSL document and get its DOM
  dbms_xmlparser.parseClob(myParser, xsldoc);
  xsltdomdoc := dbms_xmlparser.getDocument(myParser);

  xsl  := dbms_xslprocessor.newstylesheet(xsltdomdoc, '');
  -- Get the new xsl processor instance
  proc := dbms_xslprocessor.newProcessor;

  -- Apply stylesheet to DOM document
  outdomdocf := dbms_xslprocessor.processxsl(proc, xsl, indomdoc);
  outnode    := dbms_xmldom.makenode(outdomdocf);

  -- Write the transformed output to the CLOB
  dbms_xmldom.writetoCLOB(outnode, html);
  -- Return the transformed output
  return(html);
END;
/
SELECT xslexample FROM dual;
```

[Filename: SQLX/xslexample.sql]

12.3 XQuery Support in SQLX

SQL function XMLQuery()

- SQL function xmlquery('*query*' [*passing vars clause*] returning content)
- XQuery function ora:view(tablename) turns tables into sequences of XML elements:
  - relational tables: Every row is turned into a ROW element as shown on Slide 620,
  - object table of XMLType: sequence of the XML elements in the object table, comparable to XQuery's let
- the result is the sequence of nodes as returned by the XQuery statement (of type "XML content").

```
SELECT
xmlquery(
'for $c in ora:view("countryXML")/Country
 where $c/Capital[Population > 1000000]
 return $c/Name'
returning content)
from dual;
```

```
SELECT
XMLElement("result",
xmlquery(
'for $c in ora:view("countryXML")/Country
 where $c/Capital[Population > 1000000]
 return $c/Name'
returning content)) from dual;
```

Passing XML parameters to XMLQuery()

Instances of XMLType can be selected in the SQL environment and passed to XMLQuery:

- context node
- variables

```
SELECT
XMLElement("result",
 xmlquery(
  'for $c in ora:view("countryXML")/Country
   where $c/Capital[Population > $pop]
   return $c/Name'
  passing
    (SELECT population FROM City WHERE name='Tokyo') as "pop"
  returning content
 )) from dual;
```

- comma-separated value-as-varname-list
- without “as ...”: context node
- "*varname*" cares for capitalization ($POP and `... as pop` would also be correct)

Syntax example

- Select names of all countries
- from the only XML element stored in table mondial (used as context element)
- that have a city that has a higher population than Tokyo (obtained from an SQL query)

```
SELECT
XMLElement("result",
 xmlquery(
  'for $c in //country
   where $c//city[population[last()] > $POP]
   return $c/name'
  passing
    (SELECT value(m) from mondial m),
    (SELECT population FROM City WHERE name='Tokyo') as POP
  returning content
 )) from dual;
```


XMLTable(): from XML contents sequences back to relational tables

- XQuery returns a sequence of nodes, which is of XML type “content” that can be put in an element (see above).

Turn the sequence of nodes into a table of rows:

- SQL function xmltable('*query*' [*passing vars clause*] [COLUMNS *column def clause*])
- column-def-clause is a comma-separated list of (datatype, column name, XPath expr.),
- default: a single XMLType pseudo-column, named COLUMN_VALUE,
- the result of XMLTable can be used like a relational table.

```
SELECT column_value
FROM
xmltable ('
  for $j in //country
  return $j/name'
  passing
    (SELECT value(m) FROM mondial m));
```

every row of the table is of XMLType and contains a <name>...</name>element

```
SELECT column_value
FROM
XMLTABLE ('
  for $j in //country
  return $j/name'
  PASSING
    (SELECT value(m) FROM mondial m))
WHERE column_value LIKE '%fr%';
```

Note: “like” is applied to the (contents of the) element.

XMLTABLE columns specification

```
SELECT *
FROM XMLTable ('
  for $j in //country
  return $j/name'
  PASSING
    (SELECT value(m) FROM mondial m)
  COLUMNS
    result XMLTYPE PATH '.',
    x VARCHAR2(50) PATH 'text()');
```

returns <name>...</name>elements.

```
SELECT *
FROM XMLTable ('
  for $j in //country
  return $j'
  PASSING
    (SELECT value(m) FROM mondial m)
  COLUMNS
    name VARCHAR2(50) PATH 'name',
    area NUMBER PATH '@area',
    population NUMBER
      PATH 'population[position()=last()]');
```

casts automatically to numbers.

- note: the second example requires explicit “[position()=last()]” instead of only [last()].
- Additional specification of namespaces (for the paths): see documentation.

Back and Forth: an example

- the result of XMLTable(...) can be used like a relational table:

```
SELECT u.column_value, u.column_value.extract('//Name/text()')
FROM (
SELECT t.column_value
FROM
XMLTABLE ('
  for $j in $X/*
  return $j'
  PASSING
  (xmlquery(
    'for $c in ora:view("countryXML")/Country
     where $c/Capital[Population[last()] > $pop]
     return $c/Name'
    PASSING (SELECT population FROM city WHERE name='Berlin') as "pop"
    RETURNING content)
  ) AS X) t) u
WHERE  u.column_value.extract('//Name/text()') like '%ic%';
```

- or e.g. in insert: INSERT INTO ... (SELECT * FROM XMLTable(...)).

XQuery in SQLplus

- simple keyword “xquery”,
- returns the result of applying XMLTable (i.e., one row for each result of the xquery statement):

```
xquery
 for $c in ora:view("countryXML")/Country
 where $c/Capital[Population > 1000000]
 return $c/Name
/
```

In contrast to many XML tools, attribute nodes are output as string values:

```
xquery
for $i in ora:view("mondial")/mondial/country
return $i/@car_code
/
```


Namespaces and Function Declarations

- as usual in XQuery:

```
xquery
declare namespace local = "http://localhost:8080/defaultNS";
declare function local:density($pop, $area)
{
    $pop div $area
};
for $c in ora:view("mondial")//country
return local:density($c/population[last()],$c/@area)
/
```


Restrictions of Functionality

(Oracle version 12c)

- most XQuery/XPath functionality is supported (aggregation, context functions, some/every, string functions, path alternatives, ...)
- id(.) and idref(.) are not supported (recall that documents do not contain a DTD reference)

Indexes

- Indexes on XML data can be defined over any literal fields:

```
CREATE INDEX countrycodeindex
ON countryxml c
 (EXTRACTVALUE(value(c), '//Country/@Code'));
CREATE INDEX countrycapnameindex
ON countryxml c
 (EXTRACTVALUE(value(c), '//Country/Capital/Name'));
CREATE INDEX mondialcitynameindex
ON mondial m
 (EXTRACTVALUE(value(m), '//Country//City/Name'));
```


12.4 XML Storage in Oracle

- CLOB (Character Large Object): Default.
  XML is stored in its Unicode representation.
  Note: for content management/delivery (e.g., to a Web server or as a Web service that just requires to get a Unicode stream) this is optimal.
  Queries: XML is parsed internally and XPath/XQuery is applied.
- Binary XML

```
CREATE TABLE mondialBin OF XMLType
 XMLTYPE STORE AS BINARY XML;
INSERT INTO mondialBin VALUES(
 system.getxml(
 'http://www.dbis.informatik.uni-goettingen.de/Teaching/DBP/XML/mondial.xml'));
```

- object-relational (only possible, if an XML Schema with oracle-specific annotations is preloaded)

Storage: Performance Comparison

- BinaryXML is much faster
  this example: 16:1

```
SET PAUSE OFF;
SET TIMING ON;
```

```
select xmlquery('
 for $i in ora:view("mondial")/mondial
 let  $city := $i//city
 let $country := $i/country
 where $city/@country = $country/@car_code
   and $city/@id = $country/@capital
 return $city/name
 'returning content)
 from dual;
```

needs first time: 8.34,
then between 7.30 and 7.90

```
select xmlquery('
 for $i in ora:view("mondialbin")/mondial
 let  $city := $i//city
 let $country := $i/country
 where $city/@country = $country/@car_code
   and $city/@id = $country/@capital
 return $city/name
 'returning content)
 from dual;
```

needs first time: 0.42,
then between 0.28 and 0.30

Storage: Performance Comparison (Cont'd)

- join between two XPaths on a single XML table

```
SELECT * FROM XMLTABLE('
for $i in ora:view("mondial")//country
    $j in ora:view("mondial")//city,
where $i/@car_code = $j/@country
  and $i/@capital = $j/@id =
return $j/name/text()');
```

- without XMLTYPE STORE AS BINARY XML: to do
- with XMLTYPE STORE AS BINARY XML: to do

Storage: Performance Comparison (Cont'd)

```
CREATE TABLE mcity OF XMLType XMLTYPE STORE AS BINARY XML;
CREATE TABLE mcountry OF XMLType XMLTYPE STORE AS BINARY XML;
INSERT INTO mcity (SELECT COLUMN_VALUE FROM XMLTABLE(
    'for $c in ora:view("mondial")//city return $c'));
INSERT INTO mcountry (SELECT COLUMN_VALUE FROM XMLTABLE(
    'for $c in ora:view("mondial")//country return $c'));

CREATE INDEX mcountrycode ON mcountry c
  (EXTRACTVALUE(value(c), '//country/@car_code'));
CREATE INDEX mcountrycap ON mcountry c
  (EXTRACTVALUE(value(c), '//country/@capital'));
CREATE INDEX mcitycountry ON mcity c
  (EXTRACTVALUE(value(c), '//city/@country'));
CREATE INDEX mcityid ON mcity c
  (EXTRACTVALUE(value(c), '//city/@id'));

SELECT * FROM XMLTABLE('
for $i in ora:view("mcountry")/country,
    $j in ora:view("mcity")/city
where $i/@car_code = $j/@country and $i/@capital = $j/@id
return $j/name');   -- /text() -> error/bug
```


Storage: Performance Comparison (Cont'd)

- without XMLTYPE STORE AS BINARY XML: even for restricted size (cities $>$ 200000 inhabitants, countries with area $>$1000000) 26 minutes.
- with XMLTYPE STORE AS BINARY XML: 3 minutes
- without indexes: first run needs longer (e.g., 26min/20min); then nearly same time as with indexes.

12.5 Background: XMLType as Object Type

(cf. "Practical Training in SQL" course)

Since SQL3 Standard: Object(-relational) types

- user-definable: CREATE TYPE AS OBJECT ... / CREATE TYPE BODY
- stored as *row* or *column objects*
  CREATE TABLE cities OF CityObjectType;
- *member methods*
  - programmed in PL/SQL or recently also in Java
  - calls are embedded into SQL: SELECT *object.method(args)*
- reference attributes:
  CREATE TABLE COUNTRY (..., capital REF CityType, ...);
  SELECT c.capital ...;

⇒ now used for implementing XMLType

- as predefined internal classes/types
- can be used high-level from SQL, or low-level inside PL/SQL

XSLT in Oracle: "Transform" Member Method

Member Method of XMLType: *XML-instance*.transform(*Stylesheet-as-XMLValue*)
as SQL function: SELECT XMLTransform(*XML-instance*,*Stylesheet-as-XMLValue*)

```
CREATE TABLE stylesheets
 (name VARCHAR2(100),
  stylesheet XMLTYPE);

INSERT INTO stylesheets VALUES('mondial-simple.xsl',
 system.getxml('http://www.dbis.informatik.uni-goettingen.de' ||
   '/Teaching/DBP/XML/mondial-simple.xsl'));

SELECT value(m).transform(s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';

SELECT XMLTransform(value(m),s.stylesheet)
FROM mondial m, stylesheets s
WHERE s.name = 'mondial-simple.xsl';
```

[Filename: SQLX/applystylesheet.sql]

Using built-in DOM, Parser, and XSL Engine

Tools from several packages can be explicitly used inside PL/SQL procedures:

- dbms_xmldom: implements DOM (usually, XML is transformed into DOM for processing it in detail)
  PL/SQL call: dbms_xmldom.*dosomething*(*object,args*)
- dbms_xmlparser: parses documents from CLOB or URL, parses DTD from CLOB or URL (and stores the result);
  access to the DOM instance/DTD in the parser then by "getdocument" or "getdoctype"
- dbms_xslprocessor: processxsl(*different arguments*);
  clob2file/file2clob allows for reading/writing;
  selectnodes/selectsinglenode/valueof: XPath queries

... for details: Oracle Documentation, google ...

12.6 Storing XML Data in Database Systems

- "shredding" and distributing over suitable tables (of object-relational object types) (queries are translated into SQL joins/dereferencing)
  - Schema-based
  - Generic mapping of the graph structure
- storing it as VARCHAR, CLOB (Character Large Object), or as separate file with special functionality
- storing it "natively"/binary/model-based: internal object model

Literature

Klettke/Meyer "XML & Datenbanken" (dpunkt-Verlag), Ch. 8
Schöning: "XML und Datenbanken" (Hanser), Ch. 7,8
Chaudhri/Rashid/Zicari: XML Data Management

12.6.1 Mapping XML $\rightarrow$ Relational Model

two basic approaches:

- Schema-based: one or more "customized" tables for each element type
  ($\rightarrow$ similar to relational normalization theory)
  - (possibly) many null values
  - efficient access on data that belongs together
- one generic large table based on the graph structure:
  (element-id, name of the property, value/id of the property)
  - no null values
  - although memory-consuming (keys/names that are stored once in (1) are now stored for each occurrence)
  - data that belongs together is split over several tuples

$\Rightarrow$ in both cases, theory and efficiency of relational database systems can be exploited.

Schema-Based Storage

necessary: DTD or XML Schema of the instance.

1. For each element type that has children or attributes, define a table that contains
   - a column that holds the primary key of the parent,
   - a primary key column if the element type has a member that satisfies (1) or (2),
   - for each scalar attribute and child element type with text-only contents that appears at most once, a column that holds the text contents.
2. for each multi-valued attribute or text-only subelement type that occurs more than once for some element type, a separate table is created with the following columns:
   - key of the parent node,
   - the (attribute or text) value

   (similar to 1:n relationships in the relational model).

- for mixed content: possible solutions depend on the specific structure
- special treatment for infrequent properties (to avoid nulls): handling in a separate XMLType column that holds all these properties together.

Schema-Based Storage: Example

For Mondial countries, provinces and cities, the following relations are created:

- country: key(mondial), key, name, code, population, area, . . .
- border: ref(country), ref(other country), length
- language: ref(country), language, percentage
- province: ref(country), key, name, population, area
- city: ref(country), ref(province), key, name, latitude, longitude
- city-population: ref(city), year, value

Exercise

- give an excerpt of the instance
- translate some XPath/XQuery queries to SQL
- extended exercise: generate and populate the schema in SQL

Supported: Oracle (with augmented XML Schema), IBM DB2 (with DAD – Data Access Definition), MS SQL Server (extended Data-Reduced XML)

Object-Relational Approach: Internal Object Types

- "shredded storage" of XML data is in general not implemented by plain relational tables, but using object-relational technology:
  object types, collections, varrays etc ...
- collections/varrays: with value- and path indexes
- XPath expressions are rewritten into these structures

Integration of "legacy" object types

- application-dependent object types (as used in SQL3 in pre-XML times): standard mapping to XML (e.g. for data exchange)

Oracle & XML Schema

... register XMLSchema (must be typed in one single line!)

```
EXEC dbms_xmlschema.registerURI('http://mondial.de/m.xsd',
         'http://dbis.informatik.uni-goettingen.de/Mondial/mondial.xsd');
```

can be deleted with

```
EXEC dbms_xmlschema.deleteSchema('http://mondial.de/m.xsd',
      dbms_xmlschema.DELETE_CASCADE_FORCE);
```

- ... now, it knows http://mondial.de/m.xsd and created object tables for all root element types:
  SELECT * from ALL_XML_TABLES;

```
CREATE TABLE mondial2 OF XMLType
 XMLTYPE STORE AS OBJECT RELATIONAL
 XMLSCHEMA "http://mondial.de/m.xsd"
 ELEMENT "mondial";
```

```
INSERT INTO mondial2 VALUES(
system.getxml(
 'http://www.dbis.informatik.uni-goettingen.de/Teaching/DBP/XML/mondial.xml'));
SELECT XMLisValid(value(m)) FROM mondial2 m;
```


Graph-Structure-Based Storage

Without any schema knowledge, the graph structure can be represented in a single large table:

NodeNumber	ParentNode	[SiblingNo if ordered]	Name	Value

(see next page)

Alternatives

- separate table for elements and attributes (without node number and sibling number)
- separate between no-value, string value and numeric values for storing adequate types.
- previous-sibling and following-sibling columns instead of sibling-no (DOM style)

Querying

- requires recursive queries (PL/SQL; CONNECT BY)
- large joins (using the same large table several times)
- not implemented in any commercial system [according to Schöning 2003]

NodeNumber	ParentNode	SiblingNo	Name	Value
1	doc	1	mondial	
2	1	1	country	
3	2		@code	D
4	2		@membership	ref(eu)
:	:		:	:
41	2		@membership	ref(un)
42	2		@area	356910
43	2		@capital	ref(92)
44	2	1	name	Germany
45	2	2	population	83536115
:	:		:	:
90	2	47	province	
91	90	1	name	Berlin
92	90	2	city	
93	92		@country	ref(2)
94	92	1	name	Berlin
95	92	2	population	
96	95		@year	1995
97	95		text()	3472009
:	:		:	:

12.6.2 “Opaque” Storage

XML documents are stored as a whole as special datatype that can be used as row type or column data type (most commercial DBS; as described above for SQL/XML)

- approaches with text-based storage (CLOBs, files)
- specialized functionality for this datatype
  (cf. object-relational DBs: member functions)
  - XPath querying, XSLT support
  - validation
  - text search functions
- syntax embedded into SQL
- supported by indexes
  - full text indexes
  - path indexes/ “functional” indexes (user-defined, e.g. over //city/@country)
  - application and refinement of classical algorithms
- optimization of queries below the relational level!

12.6.3 “Native” Storage

Using “original” concepts of the database for storing XML (internal XML or object model) instead of mapping it or “simply” representing it as Unicode string.

- often based on existing object-oriented DB-systems with application of concepts from hierarchical and network-DBs
- no document transformation to another data model
- data model/classes based on the notions of “tree”, “element”, “attribute”, “document order”
- navigation
- XPath/XQuery/XSQL APIs

“Native” Storage: Systems and Products

Many early implementations came from the object-oriented area:

- XML-QL, based on the Strudel system
- LoPiX, based on F-Logic
- Lorel-XML, based on Lorel/OEM
- Tamino (Software AG, Darmstadt, founded 1969 (Adabas, hierarchical DB), Tamino 1999, with XQL, first native XML DBMS),
- Excelon (until 1998: ObjectDesign with “ObjectStore”; since 12.2002: acquired by Progress Software Corp.)
- POET (Hamburg, “Persistent Objects and Extended Database Technology”, product 1990, spin-off from BKS 1992, OQL interfaces, SGML Document Repository 1997, Content Management Suite since 1998, merger with Versant 2004)
- Infonyte (GMD IPSI XQL 1998, based on a “Persistent DOM”, 12.2000: spinoff TU Darmstadt/Fraunhofer-IPSI)

Generic Database Behavior for XML Databases

Everything that has been developed and dicussed for relational databases is also relevant for XML:

- physical storage + storage management
- optimization, evaluation algorithms
- multiuser operation, transactions (ACID), safety, access control
- ECA-rules, triggers

The algorithms and theoretical foundations are very similar.

Often, relational (or hierarchical) DB technology is actually used inside.

Coexistence of XML and Relational Data

- generating XML (views, data exchange packets, ...) from stored relational data
- relational (and object-relational) techniques used for efficiently storing data-centric XML
- storing text-oriented data in RDB with specialized “native” datatypes
- XPath is also accepted by SQL/XML
- additional XML processing functionality by packages and object types
- XQuery is still not the “winner” for data-oriented applications!
- is it the winner for document-oriented applications?
  `http://www.w3.org/TR/xquery-full-text/`

Chapter 13 Miscellaneous

Shortcomings in this lecture

- XML: namespaces (used e.g. for XSL, XMLSchema, XLink – but really interesting only in combination with RDF [$\rightarrow$ Semantic Web])
- XML: processing instructions
- theory: XML Data Model, XPath/XQuery Formal Semantics

Application areas

- Web Services (see lab course)
- document-oriented XML, document management
- Multimedia applications with XML
- Semantic Web (see separate lecture)

Epilogue

What should have been taught?

- knowledge for practical use of XML
- XML is more than only angle brackets
- the XML world provides examples for many basic concepts of computer science and their combination,
- illustrating how concepts in computer science evolve, and