The Java Architecture For XML Binding (JAXB)

Motivation for JAXB

- The main purpose of XML Schema is:
  - Validation of XML documents
- Any other purposes?
  - Hint 1: determinism requirement
  - Hint 2: the "**default**" attribute has **nothing** to do with validation

```
<xs:attribute name="country" use="optional"
        default="Israel" />
```

- Answer: Given a valid XML document, its schema defines a unique data model

Motivation for JAXB

- Problem: How to manipulate this data model?
- DOM (data object model) solution:
  - Pros: simple, general (a schema is not even required)
  - Cons: no types, no compile-time checking

DOM pseudo-code example

```
root.getChild("Address").getChild("Number").getText()
```

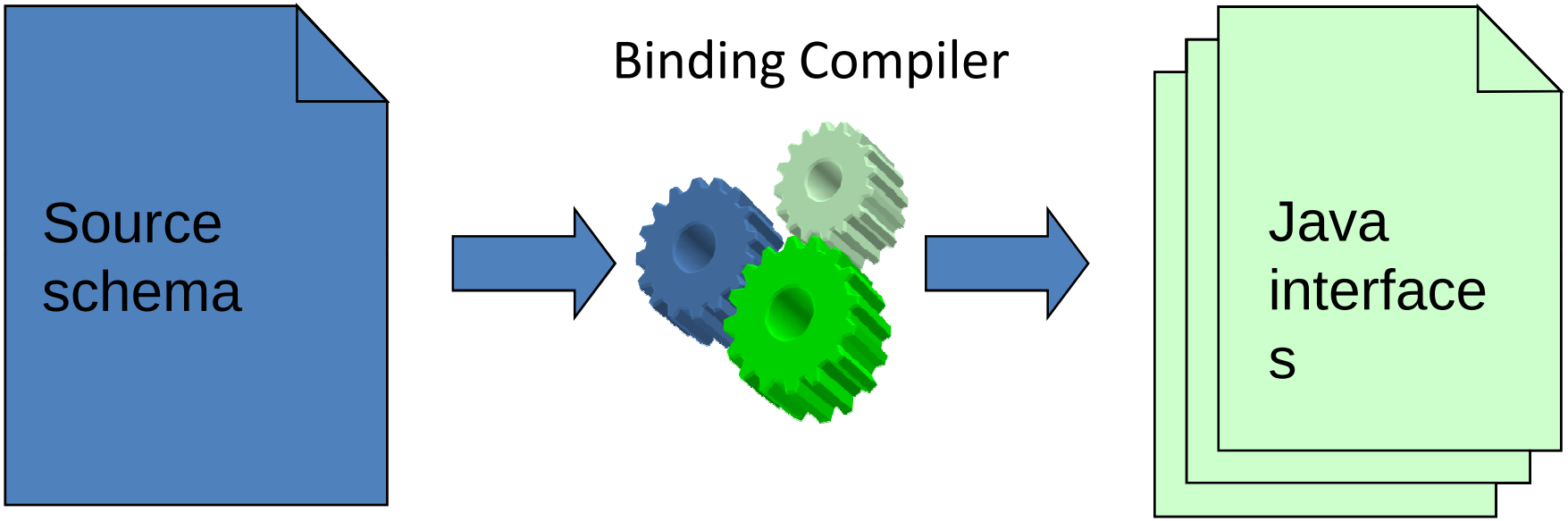
returns a string

- I wish to write …

```
root.getAddress().getNumber()
```

returns a number

JAXB solution:
Mapping XML Schema to Java interfaces



Pros: preserve types, compile-time checking
Cons: complex, specific to a certain schema

```
<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:element name="Number" type="xs:unsignedInt"/>
    <xs:element name="Street" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Binding Compiler

```
public interface AddressType {
    long getNumber();
    void setNumber(long value);

    String getStreet();
    void setStreet(String value);
}
```

Must be non-negative

Must be non-null

Mapping XML Schema to Java interfaces

- Unmarshal: xml → objects
- Create / Read / Update / Delete objects
- Validate objects
- Marshal: objects → xml
- No roundtrip guarantees
  - Marshal(Unmarshal(xml)) ≠ xml
  - We found that order is not always preserved
  - But usually roundtrip holds

Step 1: Create XML Schema

Demo.xsd

```
<xs:element name="Person" type="PersonType"/>
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="Address" type="AddressType"
       minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:element name="Number" type="xs:unsignedInt"/>
    <xs:element name="Street" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

Step 2: Create XML Document

Demo.xml

```
<Person
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="C:\JAXB Demo\demo.xsd">
       <Name>Sharon Krisher</Name>
       <Address>
              <Street>Iben Gevirol</Street>
              <Number>57</Number>
       </Address>
       <Address>
              <Street>Moshe Sharet</Street>
              <Number>89</Number>
       </Address>
</Person>
```

Check that your XML conforms to the Schema

Step 3: Run the binding compiler

- `%JWSDP_HOME%\jaxb\bin\xjc -p` **`demo`** `demo.xsd`
  - A package named **`demo`** is created (in the directory demo)
  - The package contains (among other things):
    - interface **`AddressType`**
    - interface **`PersonType`**

AddressType and PersonType

```
public interface AddressType {
    long getNumber();
    void setNumber(long value);

    String getStreet();
    void setStreet(String value);
}
```

Must be non-negative

Must be non-null

```
public interface PersonType {
    String getName();
    void setName(String value);

    /* List of AddressType */
    java.util.List getAddress();
}
```

Must be non-null

Must contain at least one item

In Java1.5: `List<AddressType>`

Step 4: Create Context

- The context is the entry point to the API
- Contains methods to create Marshaller, Unmarshaller and Validator instances

```
JAXBContext context = JAXBContext.newInstance("demo");
```

The package name is **demo**

Step 5: Unmarshal: xml -> objects

```
Unmarshaller unmarshaller =
        context.createUnmarshaller();

unmarshaller.setValidating(true);

PersonType person =
        (PersonType) unmarshaller.unmarshal(
                      new FileInputStream("demo.xml") );
```

Enable validation of xml according to the schema while unmarshalling

Step 6: Read

```
System.out.println("Person name=" +
        person.getName() );

AddressType address = (AddressType)
        person.getAddress().get(0);

System.out.println("First Address: " +
        " Street=" + address.getStreet() +
        " Number=" + address.getNumber() );
```

Step 7: Manipulate objects

```
// Update
person.setName("Yoav Zibin");

// Delete
List addressList = person.getAddress();
addressList.clear();

// Create
ObjectFactory objectFactory = new ObjectFactory();
AddressType newAddr = objectFactory.createAddressType();
newAddr.setStreet("Hanoter");
newAddr.setNumber(5);
addressList.add( newAddr );
```

part of the demo package

uses the factory pattern

Step 8: Validate on-demand

```
Validator validator = context.createValidator();

validator.validate(newAddr);
```

Check that we have set **Street** and **Number**, and that **Number** is non-negative

```
validator.validate(person);
```

Check that we have set **Name**, and that **Address** contains at least one item

Step 9: Marshal: objects -> xml

```
Marshaller marshaller = context.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
        Boolean.TRUE);
marshaller.marshal(person,
        new FileOutputStream("output.xml"));
```

output.xml

```
<Person>
  <Name>Yoav Zibin</Name>
  <Address>
    <Street>Hanoter</Street>
    <Number>5</Number>
  </Address>
</Person>
```

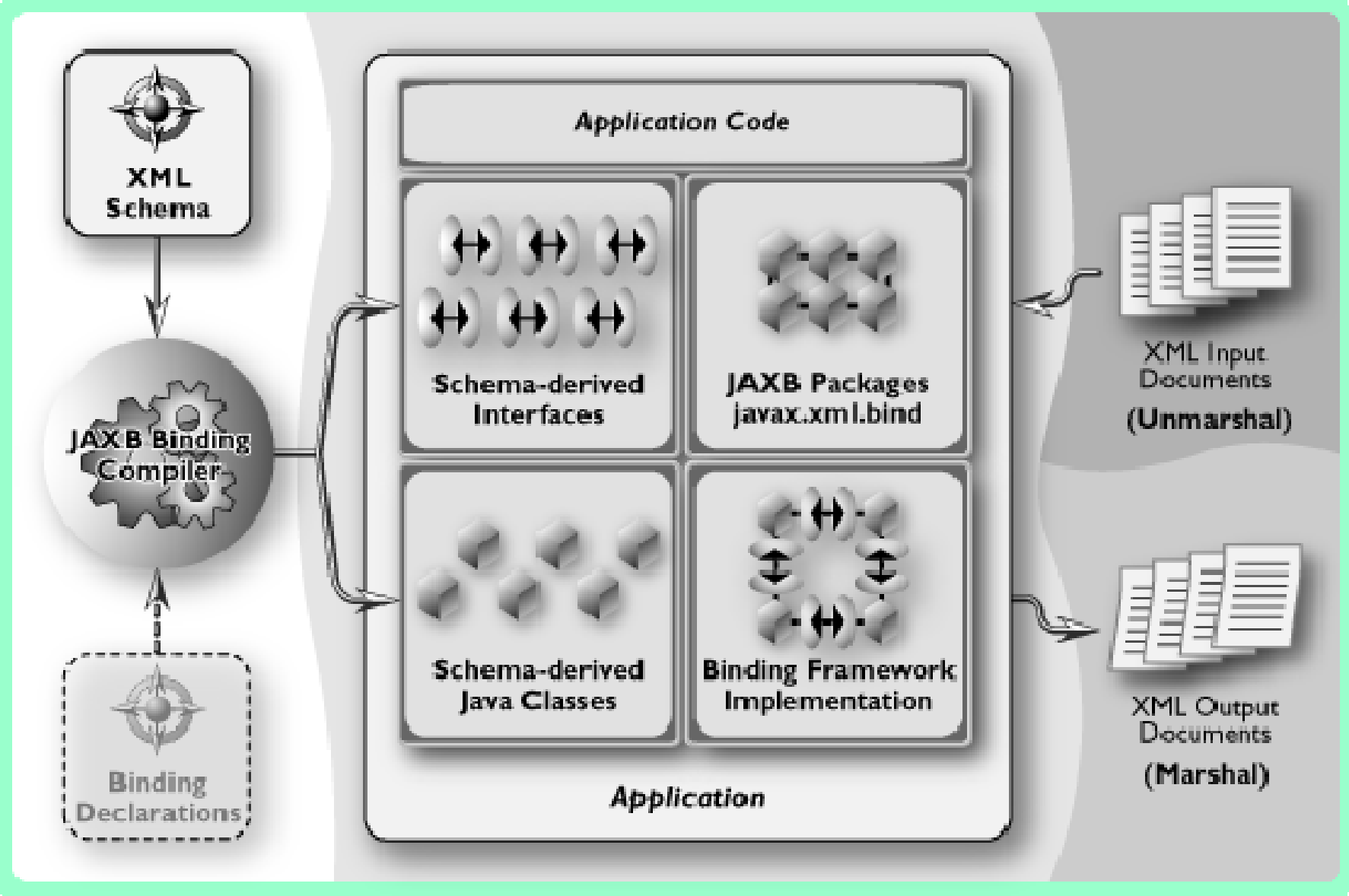
Modeling Database in XML

1. Construct XML document from the relational table by mapping the fields of the table as XML elements
2. Create a schema using DTD/XMLSchema
3. Define JAXB binding schema which is an XML document that contains instructions on how to bind a DTD/XMLSchema to a Java Class
4. Generate Java source files based on DTD and JAXB Schemas
5. The classes in source code contains private data members corresponding to XML elements, public get/set methods, validate() method to verify if contents of the object is according to DTD, methods for marshal() and unmarshal()
6. The objects of such class is known as Data Access Object (DAO) that provides access to the backend database.

DAO Design Pattern

- Data Access Object (DAO) design pattern provides a higher level of abstraction for database access
- DAO encapsulates JDBC and SQL calls
- It converts result set into a collection of objects which model the database data
- When database schema or database vendor changes, DAO alone is modified without requiring any changes in the client program

Summary


XML Schema
JAXB Binding Compiler
Binding Declarations
Application Code
Schema-derived Interfaces
JAXB Packages javax.xml.bind
Schema-derived Java Classes
Binding Framework Implementation
Application
XML Input Documents
(Unmarshal)
XML Output Documents
(Marshal)