Java Architecture for XML Binding (JAXB)

Mark Volkmann

Partner

Object Computing, Inc.

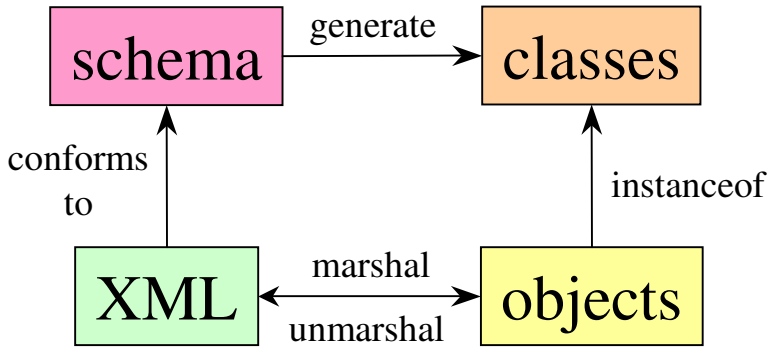
May 8, 2003

What Is XML Binding?

- Maps XML to in-memory objects
  - according to a schema
- Generates classes to represent XML elements
  - so developers don't have to write them
  - the "binding compiler" does this
  - the classes follow JavaBeans property access conventions
- Supports three primary operations
  - marshalling a tree of objects into an XML document
  - unmarshalling an XML document into a tree of objects
    - includes validation of the XML against the schema used to generate the classes of the objects
  - validation of object trees against the schema used to generate their classes
    - some constraints are enforced while working with the objects
    - others are only enforced when validation is requested (see p. 21)

XML Binding Relationships

- The relationships between the components involved in XML binding (a.k.a. data binding) are shown below



Why Use XML Binding?

- It's not necessary
  - everything that must be done with XML can be done with SAX and DOM
- It's easier
  - don't have to write as much code
  - don't have to learn SAX and/or DOM
- It's less error-prone
  - all the features of the schema are utilized
  - don't have to remember to manually implement them
- It can allow customization of the XML structure
  - unlike XMLEncoder and XMLDecoder in the java.beans package

JAXB

- An XML Binding implementation
- Developed through the Java Community Process (JCP)
  - see JSR 31 - XML Data Binding Specification
    - JSR approved on August 23, 1999
    - "Final" release on March 4, 2003
  - representatives from many companies contributed
    - BEA, HP, Intalio (Castor), Oracle, IBM, Sun and others
- Web site http://java.sun.com/xml/jaxb contains
  - reference implementation (RI)
    - bundled in the Java Web Services Developer Pack (JWSDP)
  - specification
  - javadoc
  - other documentation

JAXB Use Cases

- Create/Read/Modify XML using Java
  - but without using SAX or DOM
- Validate user input
  - using rules described in XML Schemas
- Use XML-based configuration files
  - access their values
  - write tools that creates and modifies these files

JAXB Goals

- Easy to use
  - require minimal XML knowledge
  - don't require SAX/DOM knowledge
- Customizable
  - can customize mapping of XML to Java
- Portable
  - can change JAXB implementation without changing source code
- Deliver soon
  - deliver core functionality ASAP
- Natural
  - follow standard design and naming conventions in generated Java
- Match schema
  - easy to identify generated Java components that correspond to schema features
- Hide plumbing
  - encapsulate implementation of unmarshalling, marshalling and validation
- Validation on demand
  - validate objects without requiring marshalling
- Preserve object equivalence (round tripping)
  - marshalling objects to XML and unmarshalling back to objects results in equivalent objects

JAXB Non-Goals

- Standardize generated Java
  - classes generated by different JAXB implementations may not be compatible with each other
- Preserve XML equivalence
  - unmarshalling XML to objects and marshalling back to XML may not result in equivalent XML
- Bind existing JavaBeans to schemas
  - can only marshal and unmarshal classes generated by JAXB
  - may be added later
- Schema evolution support
  - can't modify previously generated code to support schema changes
  - must generated new code
- Allow generated Java to access XML elements/attributes not described in initial schema
- Partial binding
  - unmarshalling only a subset of an XML document breaks round tripping
- Implement every feature of the schema language
  - it's tough to implement all of XML Schema!
- Support DTDs
  - focusing on XML Schema
  - DTDs were supported in an earlier version, but won't be anymore
  - tools for converting DTDs to XML Schemas exist

see spec. for parts of XML Schema that are supported

Installing JAXB

- Download
  - JWSDP 1.1 (for Windows the file is jwsdp-1_1-windows-i586.exe)
- Run installer
  - by double-clicking downloaded executable
  - creates a jwsdp-1_1 directory that contains a jaxb-1.0 directory
  - this contains directories bin, docs, examples and lib
- Environment variables
  - JAVA_HOME must point to root of JDK installation
    - for example, C:\j2sdk1.4.1
  - JAXB_HOME must point to root of JAXB installation
    - for example, C:\JWSDP\jwsdp-1_1\jaxb-1.0

Example XML Schema (cars.xsd)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.ociweb.com/cars"
  targetNamespace="http://www.ociweb.com/cars">
  <xs:complexType name="car">
    <xs:sequence>
      <xs:element name="make" type="xs:string"/>
      <xs:element name="model" type="xs:string"/>
      <xs:element name="color" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="year" type="xs:positiveInteger" use="required"/>
  </xs:complexType>
  <xs:element name="cars">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="car" type="car" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Example XML Document (cars.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<cars xmlns="http://www.ociweb.com/cars"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ociweb.com/cars cars.xsd">
  <car year="2001">
    <make>BMW</make>
    <model>Z3</model>
    <color>yellow</color>
  </car>
  <car year="2001">
    <make>Honda</make>
    <model>Odyssey</model>
    <color>green</color>
  </car>
  <car year="1997">
    <make>Saturn</make>
    <model>SC2</model>
    <color>purple</color>
  </car>
</cars>
```

Generating Java From XML Schema

- From command-line
  - Windows: `%JAXB_HOME%\bin\xjc cars.xsd`
  - UNIX: `%JAXB_HOME%/bin/xjc.sh cars.xsd`
  - these write generated files to current directory
- From Ant

```
<java jar="${env.JAXB_HOME}/lib/jaxb-xjc.jar" fork="yes">
  <arg line="-d ${gen.src.dir} cars.xsd"/>
</java>
```

may also want jaxb-api.jar in classpath to use DatatypeConverter (discussed later)

directory where generated files should be written

Use -p option **to override default package of generated classes** which is taken from the target namespace of the XML Schema. For example, the namespace "http://www.ociweb.com/cars" maps to the package "com.ociweb.cars".

Generated Files

- com/ociweb/cars directory
  - Car.java
    - interface representing the “car” complex type
    - only describes get and set methods for car properties
  - Cars.java
    - interface representing “cars” global element
    - extends CarsType and javax.xml.bind.Element (just a marker interface)
    - describes no additional methods
  - CarsType.java
    - interface representing anonymous complex type defined inside the “cars” global element
    - provides method to get collection of Car objects (as a java.util.List)
  - ObjectFactory.java
    - class used to create objects of the above interface types
    - extends DefaultJAXBContextImpl which extends **JAXBContext** ← more on this later

Generated Files (Cont'd)

- com/ociweb/cars directory (cont'd)
  - bgm.ser
    - **for internal use by RI**
    - a serialized object of type com.sun.msv.grammar.trex.TREXGrammar
    - can't find any documentation on this - don't know its purpose
  - jaxb.properties
    - **for internal use by RI**
    - sets a property that defines the class used to create JAXBContext objects

Generated Files (Cont'd)

- com/ociweb/cars/impl directory

Your code should not reference these classes! Instead, use the interfaces they implement.

  - CarImpl.java
    - class that implements Car
    - corresponds to the "car" XML Schema complexType
  - CarsTypeImpl.java
    - class that implements CarType
    - corresponds to the XML Schema anonymous type inside the "cars" element
  - CarsImpl.java
    - class that extends CarsTypeImpl and implements Cars
    - corresponds to the "cars" XML Schema element

These classes also implement
com.sun.xml.bind.unmarshaller.**UnmarshallableObject**, ← to read from XML
com.sun.xml.bind.serializer.**XMLSerializable** and ← to write to XML
com.sun.xml.bind.validator.**ValidatableObject** ← to validate content

Unmarshalling XML Into Java Objects

- Example

```
ObjectFactory factory = new ObjectFactory();
Unmarshaller u = factory.createUnmarshaller();
Cars cars = (Cars) u.unmarshal(new FileInputStream("cars.xml"));
```

used in subsequent examples too

- unmarshal method accepts
  - java.io.File
  - java.io.InputStream
  - java.net.URL
  - javax.xml.transform.Source
    - related to XSLT
  - org.w3c.dom.Node
    - related to DOM
  - org.xml.sax.InputSource
    - related to SAX

The **ObjectFactory** for a particular package only works with classes from that package.
When working with objects from multiple packages, create the factory like this instead:

```
JAXBContext factory =
    JAXBContext.newInstance
        ("colon-delimted-package-list");
```

Unmarshalling Java Objects Into XML (Cont'd)

- Other Unmarshaller methods
  - `void` **`setValidating`**`(boolean validating)`
    - true to enable validation during unmarshalling; false to disable (the default)
  - `boolean` **`setEventHandler`**`(ValidationEventHandler handler)`
    - **handleEvent** method of ValidationEventHandler is called if validation errors are encountered during unmarshalling
    - default handler terminates marshalling after first error
    - return true to continue unmarshalling
    - return false to terminate with UnmarshalException
    - see discussion of ValidationEventCollector later

Marshalling Java Objects Into XML

- Example

```
Marshaller m = factory.createMarshaller();
m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
Writer fw = new FileWriter("newCars.xml");
m.marshal(cars, fw);
```

objects being marshaled do not necessarily have to be "valid"

- marshal method accepts
  - java.io.OutputStream
  - java.io.Writer
  - javax.xml.transform.Result
    - related to XSLT
  - org.w3c.dom.Node
    - related to DOM
  - org.xml.sax.ContentHandler
    - related to SAX

Marshalling Java Objects Into XML (Cont'd)

- Other Marshaller methods
  - `boolean` **`setEventHandler`**`(ValidationEventHandler handler)`
    - same as use with Unmarshaller, but validation events are delivered during marshalling
  - `void` **`setProperty`**`(String name, Object value)`
    - supported properties are
      - `jaxb.encoding` - value is a String
        - » the **encoding** to use when marshalling; defaults to "UTF-8"
      - `jaxb.formatted.output` - value is a Boolean
        - » true to output **line breaks and indentation**; false to omit (the default)
      - `jaxb.schemaLocation` - value is a String
        - » to specify **xsi:schemaLocation attribute** in generated XML
      - `jaxb.noNamespaceSchemaLocation` - value is a String
        - » to specify **xsi:noNamespaceSchemaLocation attribute** in generated XML

doesn't associate generated XML with an XML Schema by default

Creating Java Objects From Scratch

- Example

```
Cars cars = factory.createCars();

Car car = factory.createCar();
car.setColor("blue");
car.setMake("Mazda");
car.setModel("Miata");
car.setYear
    (BigInteger.valueOf(2003));
cars.getCar().add(car);

car = factory.createCar();
car.setColor("red");
car.setMake("Ford");
car.setModel("Mustang II");
car.setYear
    (BigInteger.valueOf(1975));
cars.getCar().add(car);
```

must be an instance of **ObjectFactory** for the package of the object being created, not an instance of **JAXBContext** created using JAXBContext.newInstance()

returns a live List of the Car objects associated with the Cars object

Write your own convenience methods to make this easier!

```
public void addCar(Cars cars, String make,
    String model, int year, String color) {
    Car car = factory.createCar();
    car.setMake(make);
    car.setModel(model);
    car.setYear(BigInteger.valueOf(year));
    car.setColor(color);
    cars.getCar().add(car)
}
```

Validating Java Objects

- The graph of Java objects can contain invalid data
  - could occur when objects created by unmarshalling are modified
  - could occur when objects are created from scratch
- Use a Validator to validate the objects
- Example

```
Validator v = factory.createValidator();
try {
    v.validateRoot(cars);
    v.validate(car);
} catch (ValidationException e) {
    // Handle the validation error described by e.getMessage().
}
```

from unmarshalling example

to validate entire tree

to validate a subtree

in this case, the only validation specified in the XML Schema is to insure that years are positive integers

Validating Java Objects (Cont'd)

- Other Validator methods
  - `boolean setEventHandler(ValidationEventHandler handler)`
    - **handleEvent** method of ValidationEventHandler is called if validation errors are encountered
    - default handler terminates marshalling after first error
    - return true to continue validating
    - return false to terminate with ValidationException

Pass an instance of javax.xml.bind.util.**ValidationEventCollector** (in jaxb-api.jar) to setEventHandler to collect validation errors and query them later instead of handling them during validation.

```
ValidationEventCollector vec =
    new ValidationEventCollector();
v.setEventHandler(vec);
v.validate(cars);
ValidationEvent[] events = vec.getEvents();
```

Default Type Bindings

- The table below shows the default mappings from XML Schema types to Java types

XML Schema Type	Java Type
string	java.lang.String
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
integer	java.math.BigInteger
decimal	java.math.BigDecimal
unsignedByte	short
unsignedShort	int
unsignedInt	long
base64Binary	byte[]
hexBinary	byte[]
date	java.util.Date
time	java.util.Date
dateTime	java.util.Calendar
QName	javax.xml.namespace.QName
anySimpleType	java.lang.String

Customizing Type Bindings

- Default bindings can be overridden
  - at global scope
  - on case-by-case basis
- Customizations include
  - names of generated package, classes and methods
  - method return types
  - class property (field) types
  - which elements are bound to classes, as opposed to being ignored
  - class property to which each attribute and element declaration is bound

Customization Syntax

- Customizations can be specified in
  - the XML Schema (our focus)
  - a binding declarations XML document (not well supported by RI yet)
- The XML Schema must declare the JAXB namespace and version

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
  jxb:version="1.0">
```

- Customization elements are placed in annotation elements

```
<xsd:annotation>
  <xsd:appinfo>
    binding declarations
  </xsd:appinfo>
</xsd:annotation>
```

Customization Levels

- Customizations can be made at four levels
  - **global**
    - defined at "top level" in a **<jxb:globalBindings>** element
    - applies to all schema elements in the **source schema and** in all **included/imported schemas** (recursively)
  - **schema**
    - defined at "top level" in a **<jxb:schemaBindings>** element
    - applies to **all schema elements in the target namespace** of the source schema
  - **definition**
    - defined in a type or global declaration
    - applies to **all schema elements that reference the type or global declaration**
  - **component**
    - defined in **a particular schema element or attribute declaration**
    - applies only to it

globalBindings Attributes

(see spec. for more details)

- **collectionType**
  - "indexed" (uses array and provides methods to get/set elements) or fully-qualified-java-class-name (must implement java.util.List)
  - default is "java.util.ArrayList"
- **enableFailFastCheck**
  - "true" or "false" (default)
  - if true, invalid property values are reported as soon as they are set, instead of waiting until validation is requested
  - not implemented yet in RI
- **generateIsSetMethod**
  - "true" or "false" (default)
  - if true, generates isSet and unSet methods for the property
- **underscoreBinding**
  - "asCharInWord" or "asWordSeparator" (default)
  - if "asWordSeparator", underscores in XML names are removed and words are camel-cased to form Java name
  - for example, "gear_shift_knob" goes to "gearShiftKnob"

JAXB converts **XML names to camel-cased Java names** by breaking XML names into words using the following characters as **delimiters**: **hyphen**, **period**, **colon**, **underscore** and a few foreign characters. For example, the XML attribute name "`my-big.lazy_dog`" is converted to "`myBigLazyDog`". Setting **underscoreBinding** to "asCharInWord" changes this to "`myBigLazy_dog`".

globalBindings Attributes (Cont'd)

(see spec. for more details)

- bindingStyle (was modelGroupAsClass)
  - "modelGroupBinding" or "elementBinding" (default)
- choiceContentProperty
  - "true" or "false" (default)
  - allows objects to hold one of a number of property choices which may each have a different data type
- enableJavaNamingConventions
  - "true" (default) or "false"
- fixedAttributeAsConstantProperty
  - "true" or "false" (default)
  - if true, "fixed" attributes will be represented as constants
- typesafeEnumBase
  - "xsd:string", "xsd:decimal", "xsd:float", "xsd:double" or "xsd:NCName" (default)
  - defines field type used to represent enumerated values in generated typesafe enum class
- typesafeEnumMemberName
  - "generateName" or "generateError" (default)
  - specifies what to do if an enumerated value cannot be mapped to a valid Java identifier (for example, 3.14 and "int")
  - "generateName" generates names in the form VALUE_#
  - "generateError" reports an error

schemaBindings Syntax

- The syntax for the schemaBindings element is

```
<jxb:schemaBindings>

  <jxb:package [name="package-name"]>
    <jxb:javadoc> ... javadoc ... </jxb:javadoc>
  </package>

  <jxb:nameXmlTransform>
    <jxb:typeName prefix="prefix" suffix="suffix"/>
    <jxb:elementName prefix="prefix" suffix="suffix"/>
    <jxb:modelGroupName prefix="prefix" suffix="suffix"/>
    <jxb:anonymousTypeName prefix="prefix" suffix="suffix"/>
  </jxb:nameXmlTransform>

</jxb:schemaBindings>
```

package name to be used for all generated classes

supplies package-level javadoc; enclose all javadoc in CDATA sections

avoids name collisions between types and elements; can have any number of these, but why?

doesn't work in RI

specifies a prefix and/or suffix to be added to the names of **all generated type, element, modelGroup and anonymous type interfaces**

  - every element and attribute within schemaBindings is optional

when prefixes are used, the RI doesn't capitalize the first letter after the prefix!

Other Customizations

- To override a generated class/interface name so it differs from the corresponding XML type name and/or add javadoc

```
<jxb:class name="class-name"
       implClass="impl-class">
  <jxb:javadoc> ... javadoc ... </jxb:javadoc>
</jxb:class>
```

name to use for Java interface

name to use for Java implementation class (not supported in RI)

supplies class-level javadoc

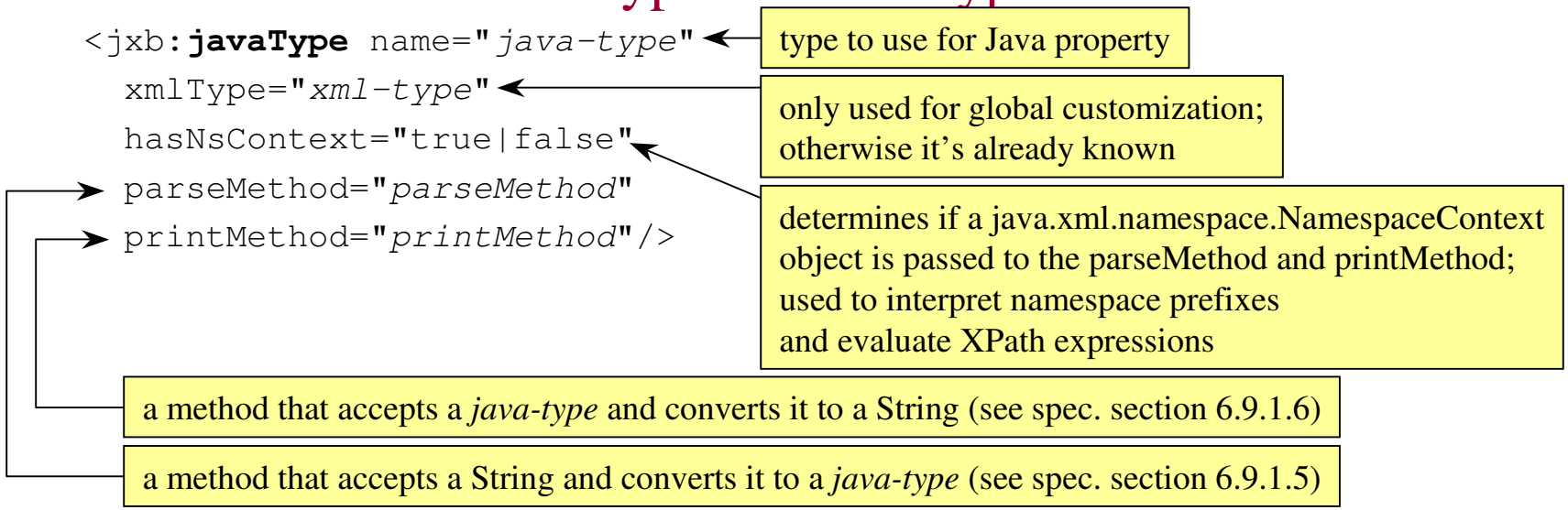
- To override a generated class/interface property name so it differs from the corresponding XML element/attribute name and/or add javadoc

```
<jxb:property name="property-name”>
  <jxb:javadoc> ... javadoc ... </jxb:javadoc>
</jxb:property>
```

name to use for Java property

Other Customizations (Cont'd)

- To override the default mapping from an XML Schema type to a Java type



Using **DatatypeConverter** simplifies this for certain XML Schema types.
For **example**, by default xs:positiveInteger maps to Java type java.math.BigInteger.
To map to int instead, use this.

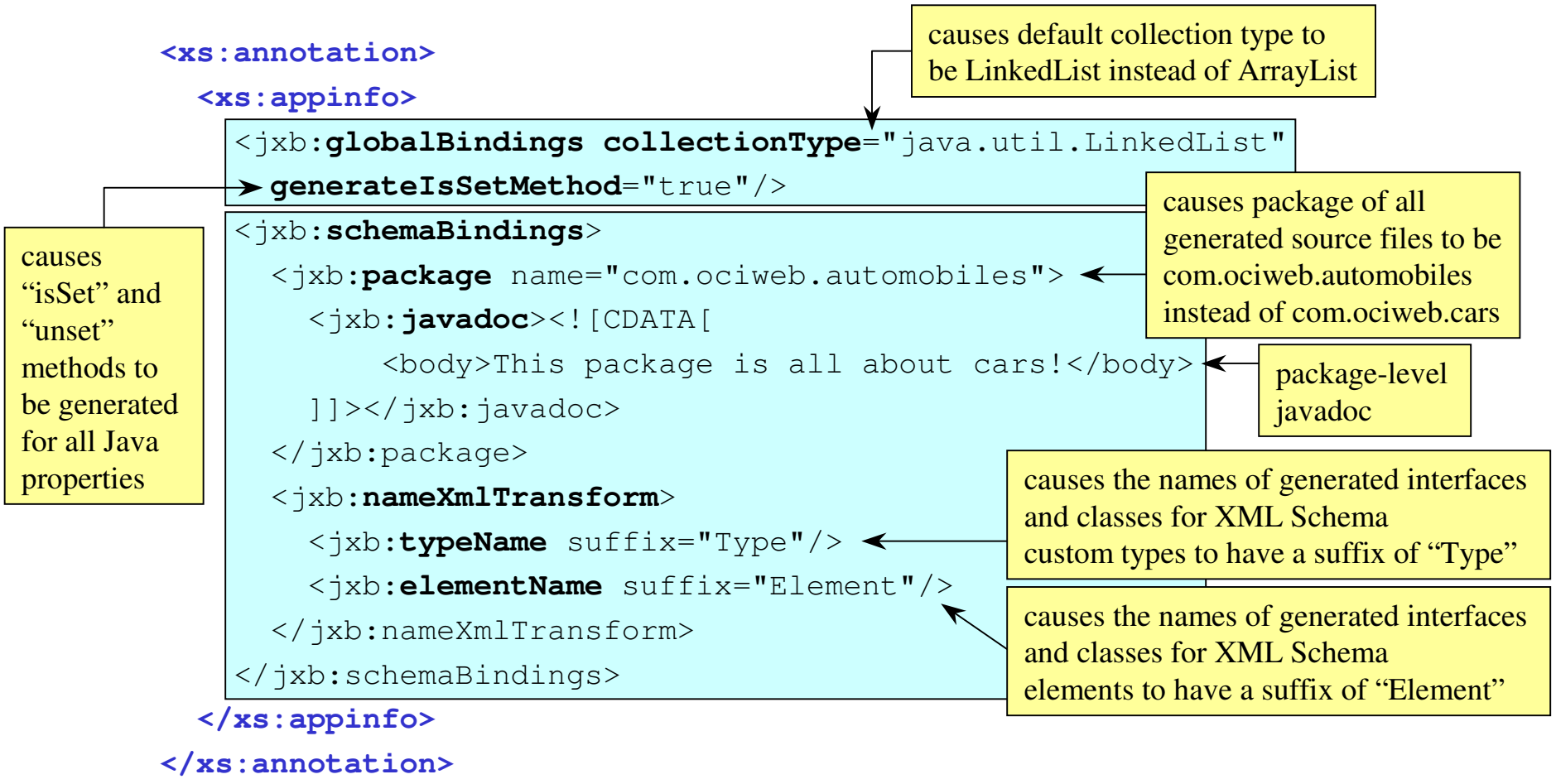
```
<jxb:javaType name="int"
  parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
  printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
```

Customization Example

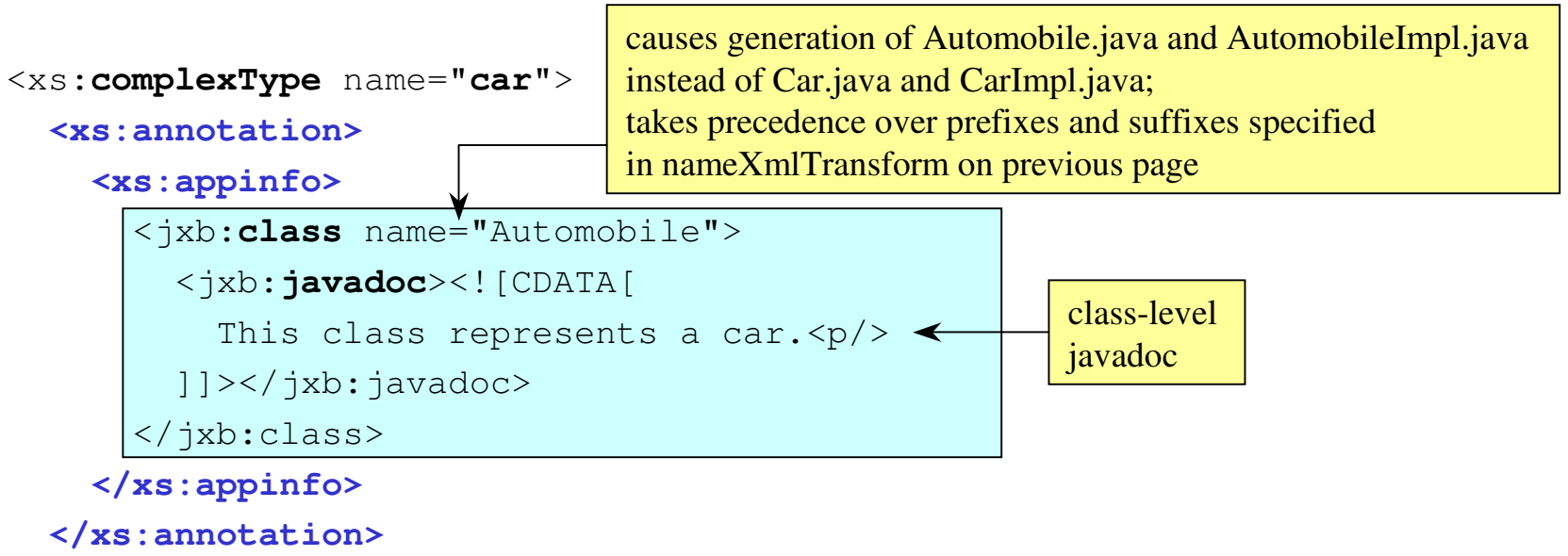
(modifying our earlier XML Schema)

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:jxb="http://java.sun.com/xml/ns/jaxb" jxb:version="1.0"
  xmlns="http://www.ociweb.com/cars"
  targetNamespace="http://www.ociweb.com/cars">
```

Customization Example (Cont'd)



Customization Example (Cont'd)



Customization Example (Cont'd)

```
<xs:sequence>
  <xs:element name="make" type="xs:string">
    <xs:annotation>
      <xs:appinfo>
        <jxb:property name="manufacturer"/>
      </xs:appinfo>
    </xs:annotation>
  </xs:element>
  <xs:element name="model" type="xs:string"/>
  <xs:element name="color" type="xs:string"/>
</xs:sequence>
```

causes the Java property to be named "manufacturer" instead of "make"

Customization Example (Cont'd)

```
<xs:attribute name="year" type="xs:positiveInteger"
  use="required">
  <xs:annotation>
    <xs:appinfo>
      <jxb:javaType name="int"
        parseMethod="javax.xml.bind.DatatypeConverter.parseInt"
        printMethod="javax.xml.bind.DatatypeConverter.printInt"/>
    </xs:appinfo>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
```

causes the Java property type to be "int" instead of "java.math.BigDecimal"

Customization Example (Cont'd)

```
  <xs:element name="cars">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="car" type="car" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

no changes were made to this part of the schema

XML Schema Support

- These XML Schema features are not currently supported
  - wildcard types
    - types any and anyAttribute
  - notations
    - notation element
  - declaration redefinition
  - **identity-constraint definition**
    - **key, keyref and unique elements**
  - substitution
    - attributes complexType.abstract, element.abstract, element.substitutionGroup
    - type substitution (derived types substituting for ancestor types)
      - xsi:type attribute
    - block feature to block substitutions
      - attributes complexType.block, complexType.final, element.block, element.final, schema.blockDefault, schema.finalDefault

Lab

- Steps
  - examine the provided XML Schema portfolio.xsd
  - create an XML document named portfolio.xml that conforms it
    - remember to associate the document with the schema using the schemaLocation attribute (see cars.xml for an example)
  - use JAXB to generate classes from the schema
  - write a class that does the following
    - use JAXB to unmarshal the XML document into Java objects
    - print a description of each stock in the portfolio to the command-line
    - use the generated ObjectFactory class to create a new Stock object
    - add the new Stock object to the list of Stock objects in the Portfolio
    - use JAXB to marshal the Portfolio to a file called newPortfolio.xml