

Oracle® Fusion Middleware

Developing JAXB Applications Using Oracle TopLink



14c (14.1.1.0.0)
F25532-01
March 2020

ORACLE®

Copyright © 2014, 2020, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience	viii
Documentation Accessibility	viii
Related Documents	viii
Conventions	ix

1 Introduction

About EclipseLink MOXy	1-1
Solving Object-XML Impedance Mismatch	1-1
About This Documentation	1-2
Other Resources	1-2

2 EclipseLink MOXy Runtime

Specifying the EclipseLink Runtime	2-1
Bootstrapping	2-1
Using the JAXBContext API	2-2
Bootstrapping from Classes	2-2
Bootstrapping from a Context Path	2-3
Using a jaxb.index File	2-3
Using an ObjectFactory	2-3
Using MetadataSource	2-4
Bootstrapping from EclipseLink XML Bindings	2-4
Combining Annotated Classes and XML Bindings	2-5
Using XML Bindings	2-6
Understanding the XML Bindings Format	2-6
Bootstrapping with XML Bindings	2-7
Using XML Bindings with Annotations	2-8
Using Multiple Bindings Documents	2-9
Understanding Override Rules	2-10
Using Complete Metadata	2-12
Using Virtual Mappings	2-13

Using MetadataSource	2-13
Implementing a MetadataSource	2-14
Using an XmlBindings Object	2-14
Specifying the MetadataSource	2-14
MetadataSource Example	2-15
Building XmlBindings Programatically	2-15
Generating an XML Schema	2-16
Validating Against an XML Schema	2-17
Using a ValidationEventHandler	2-18
Enabling Validation	2-19
Input (input.xml File)	2-20
Output	2-20
Understanding Events	2-21
Adding Event Listener Methods on JAXB Mapped Objects	2-21
Registering Listeners on Marshallers and Unmarshallers	2-22
Querying Objects by XPath	2-25
Binding to an Existing Document	2-25

3 Mapping Type Levels

Defining the Default Root Element	3-1
Customizing the Default Root Element	3-2
Understanding How EclipseLink Uses the Default Root Element	3-3
Setting Up Namespace Information	3-4
Qualifying at the Package Level	3-4
Qualifying at the Type Level	3-5
Qualifying at the Field/Property Level	3-6
Specifying Inheritance	3-7
Using xsi:type	3-7
Using Substitution Groups	3-8
Using @XmlDiscriminatorNode/@XmlDiscriminatorValue	3-8

4 Mapping Simple Values

Mapping Simple Values	4-1
Mapping to an Attribute	4-1
Mapping to a Text Node	4-2
Mapping to a Text Node in a Simple Sequence	4-2
Mapping to a Text Node in a Sub-element	4-4
Mapping to a Text Node by Position	4-5
Mapping to a Simple Text Node	4-7

Mapping to a Specified Schema Type	4-8
Using Java Type Adapters	4-9
Mapping with a Simple Type Translator	4-11
Mapping Collections of Simple Values	4-12
Mapping to Text Nodes	4-12
Mapping to Text Nodes with a Grouping Element	4-13
Mapping to a List Element	4-15
Mapping a Collection of XmlAttributes or XmlValues	4-16
Multiple Mappings for a Single Property	4-17
Example	4-17
XML Output	4-18
Mapping Enums	4-18
Mapping Enums using Constant Names	4-18
Mapping Enums to Custom XML Values	4-19

5 Mapping Special Schema Types

Mapping Dates and Times	5-1
Understanding the Generated Model	5-1
Using a Different Date (or Calendar) Property	5-2
Mapping to a Union Field	5-3
Understanding Conversion Order	5-5
Customizing Conversion Classes	5-5
Binary Types	5-6
Specifying Binary Formats Base64 and Hex	5-6
Understanding byte[] versus Byte[]	5-7
Working with SOAP Attachments	5-8
Using @XmlInlineBinaryData	5-9
Using @XmlMimeType	5-9

6 Privately Owned Relationships

Mapping Privately Owned One-to-One Relationships	6-1
Mapping to an Element	6-1
Using EclipseLink's @XmlPath Annotation	6-3
Mapping Privately Owned One-to-Many Relationships	6-4
Mapping to Elements	6-6
Grouping Elements using the @XmlElementWrapper Annotation	6-7

7 Mapping Shared Reference Relationships

Understanding Keys and Foreign Keys	7-1
-------------------------------------	-----

Mapping Single Key Relationships	7-1
Using @XmlList	7-2
Using the Embedded Key Class	7-3
Mapping Composite Key Relationships	7-5
Mapping Bidirectional Relationships	7-7
See also	7-8

8 Advanced Concepts

Refreshing Metadata	8-1
Customizing XML Name Conversions	8-2
Using the XMLNameTransformer	8-3
Example Model	8-4
Specifying the Naming Algorithm	8-4
XML Output	8-5
Using Virtual Access Methods	8-5
Configuring Virtual Access Methods	8-6
Example	8-7
Using XmlAccessType.FIELD and XmlTransient	8-8
Options	8-8
Specifying Alternate Accessor Methods	8-8
Specifying Schema Generation Options	8-9
Using Extensible MOXy	8-11
Using the @XmlVirtualAccessMethods Annotation	8-11
Creating Tenant 1	8-13
Creating Tenant 2	8-15
Mapping Using XPath Predicates	8-16
Mapping with XPath Predicates	8-17
Mapping Based on Position	8-18
Mapping Based on an Attribute Value	8-19
Creating "Self" Mappings	8-20
Using an XmlAdapter	8-21
Using java.util.Currency	8-22
Using java.awt.Point	8-23
Specifying Package-Level Adapters	8-25
Specifying Class-Level @XmlJavaTypeAdapters	8-25
Using XML Transformations	8-25
Using an AttributeTransformer	8-27
Using a FieldTransformer	8-27
Generating Java Classes from an XML Schema	8-29
Running the JAXB Compiler	8-29

9 Using Dynamic JAXB

Understanding Static and Dynamic Entities	9-1
Using Static MOXy	9-1
Using Dynamic MOXy	9-2
Using Dynamic Entities	9-2
Specifying the EclipseLink Runtime	9-3
Instantiating a DynamicJAXBContext	9-4
Bootstrapping from XML Schema (XSD)	9-5
Importing Other Schemas / EntityResolvers	9-7
Customizing Generated Mappings with XJC External Binding Customization Files	9-8
Bootstrapping from EclipseLink Metadata (OXM)	9-10
Example	9-11

10 Using JSON Documents

Understanding JSON Documents	10-1
Marshalling and Unmarshalling JSON Documents	10-1
Specifying JSON Bindings	10-2
Specifying JSON Data Types	10-4
Supporting Attributes	10-4
Supporting no Root Element	10-5
Using Namespaces	10-5
Using Collections	10-6
Mapping Root-Level Collections	10-7
Wrapping XML Values	10-7

Preface

Oracle TopLink, with its core features provided by EclipseLink, the open source persistence framework from the Eclipse Foundation, delivers a standards-based enterprise Java solution for all of your relational, XML, and JSON persistence requirements, based on high performance and scalability, developer productivity, and flexibility in architecture and design.

EclipseLink delivers a standards-based enterprise Java solution for all of your relational, XML, and JSON persistence requirements, based on high performance and scalability, developer productivity, and flexibility in architecture and design.

Audience

This document is intended for application developers who want to develop applications using TopLink with Java Architecture for XML Binding (JAXB). This document does not include details about related common tasks, but focuses on TopLink functionality.

Developers should be familiar with the concepts and programming practices of

- Java SE and Java EE.
- Java Architecture for XML Binding (JAXB) specification (<http://jcp.org/aboutJava/communityprocess/mrel/jsr222/index.html>)
- Java Persistence specification (<http://jcp.org/en/jsr/detail?id=338>)
- EclipseLink (<http://www.eclipse.org/eclipselink>)
- Eclipse IDE (<http://www.eclipse.org>)

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit <http://www.oracle.com/pls/topic/lookup?ctx=accid=info> or visit <http://www.oracle.com/pls/topic/lookup?ctx=accid=trs> if you are hearing impaired.

Related Documents

For more information, see the following documents:

- *Understanding Oracle TopLink*
- *Solutions Guide for Oracle TopLink*
- *Java API Reference for Oracle TopLink*
- *Release Notes for Oracle TopLink*
- EclipseLink Documentation Center at <http://wiki.eclipse.org/EclipseLink/UserGuide>

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
monospace	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction

JAXB (Java Architecture for XML Binding – JSR 222) is the standard for XML Binding in Java. JAXB covers 100% of XML Schema concepts and EclipseLink provides a JAXB implementation with many extensions. See <http://jcp.org/en/jsr/detail?id=222> for complete information on the JAXB specification.

Oracle TopLink includes EclipseLink, and its JAXB provider: EclipseLink MOXy.

The EclipseLink MOXy component enables developers to efficiently bind Java classes to XML schemas and JSON documents. MOXy implements JAXB, allowing developers to provide their mapping information through annotations *as well as providing support for storing the mappings in XML and JSON formats.*

This chapter includes the following sections:

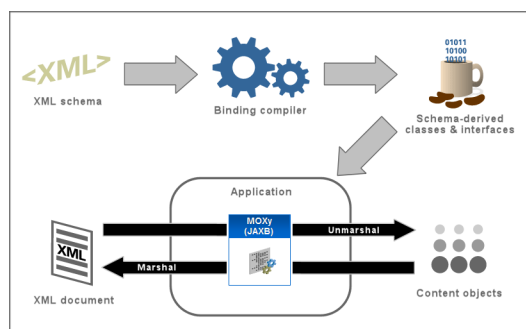
- [About EclipseLink MOXy](#)
- [Solving Object-XML Impedance Mismatch](#)

About EclipseLink MOXy

When using EclipseLink MOXy as the JAXB provider, no metadata is required to convert your existing object model to XML or JSON. You can supply metadata (using annotations, XML, or JSON) only when fine-tuning of the XML or JSON representation is required.

EclipseLink MOXy includes many advanced mappings that allows you to handle complex XML structures without having to mirror the schema in the Java class model.

Figure 1-1 Sample JAXB Architecture



Solving Object-XML Impedance Mismatch

Although XML is a common format for the exchange of data, for many applications *objects* are the preferred programmatic representation – not XML. In order to work at

the object-level, the XML data needs to be converted to object form. The mismatch between XML and objects is known as *object-xml impedance mismatch*.

JAXB allows you to interact with XML data by using domain-like objects. Unlike DOM objects, the JAXB content model provides insight into the XML document based on the XML schema. For example, if the XML schema defines XML documents that contain customer information, your content model will contain objects such as **Customer**, **Address**, and **PhoneNumber**. Each *type* in the XML schema will have a corresponding Java class.

With EclipseLink MOXy, you can efficiently bind Java classes to XML schemas or JSON documents. MOXy implements JAXB by allowing you to provide mapping information through annotations *as well as storing the mappings in XML or JSON*. By using the EclipseLink's advanced mappings, you can manage complex XML structures *without* having to mirror the schema in your Java class model.

With EclipseLink MOXy (included with TopLink), you can efficiently bind Java classes to XML schemas or JSON documents. MOXy implements JAXB by allowing you to provide mapping information through annotations *as well as storing the mappings in XML or JSON*. By using these mappings, you can manage complex XML structures *without* having to mirror the schema in your Java class model.

About This Documentation

Oracle TopLink includes EclipseLink, the reference implementation of the Java Persistence (JPA) specification, as its persistence provider. It also includes many enhancements and extensions.

EclipseLink is the reference implementation of the Java Persistence Architecture (JPA) specification. It also includes many enhancements and extensions.

Other Resources

For more information, see:

- Oracle TopLink on the Oracle Technology Network (OTN). <http://www.oracle.com/technetwork/middleware/toplink/overview/index.html>
- EclipseLink documentation center <http://www.eclipse.org/eclipselink/documentation/>
- The EclipseLink API reference documentation (Javadoc) for complete information on core JAXB plus the EclipseLink enhancements <http://www.eclipse.org/eclipselink/api/>
- Java Architecture for XML Binding (JAXB) specification <http://www.jcp.org/en/jsr/detail?id=222>
- Examples that display the use of a number of JAXB and EclipseLink MOXy features <http://wiki.eclipse.org/EclipseLink/Examples/MOXy>
- JAXB Tutorial <http://www.oracle.com/technetwork/articles/javase/index-140168.html>

2

EclipseLink MOXy Runtime

This chapter introduces and describes EclipseLink MOXy. EclipseLink MOXy is one implementation of the standard runtime defined by the JAXB specification. To specify EclipseLink MOXy as your JAXB provider:

- Add the JAXB APIs (included in Java SE 8) and `eclipselink.jar` on your classpath.
- Use a `jaxb.properties` file (in the same package as your domain classes).

This chapter includes the following sections:

- [Specifying the EclipseLink Runtime](#)
- [Bootstrapping](#)
- [Using XML Bindings](#)
- [Using MetadataSource](#)
- [Generating an XML Schema](#)
- [Validating Against an XML Schema](#)
- [Understanding Events](#)
- [Querying Objects by XPath](#)
- [Binding to an Existing Document](#)

Specifying the EclipseLink Runtime

To use EclipseLink MOXy as your JAXB implementation, identify the EclipseLink `JAXBContextFactory` in your `jaxb.properties` file.

1. Create a text file named `jaxb.properties`, specifying EclipseLink's `JAXBContextFactory` as the factory used to build new `JAXBContext`s:
2. Copy the file to the same package (directory) in which your model classes reside.
3. Use the standard `JAXBContext.newInstance(Class... classesToBeBound)` API to create a `JAXBContext`:

```
JAXBContext jaxbContext = JAXBContext.newInstance(Customer.class);
```

Because you do not need to change any application code, you can easily switch between different JAXB implementations.

For more information on different ways to create a `JAXBContext`, see "[Bootstrapping](#)".

Bootstrapping

EclipseLink MOXy offers several options when creating your `JAXBContext`. You have the option of bootstrapping from:

- A list of one or more JAXB-annotated classes
- A list of one or more EclipseLink XML bindings documents defining the mappings for your Java classes
- A combination of classes and XML bindings
- A list of context paths
- A list of session names, referring to EclipseLink sessions defined in `sessions.xml`

Using the JAXBContext API

The methods on `JAXBContext` (shown in [Example 2-1](#)) are used to create new instances.

`JAXBContext` accepts the following options:

- **classesToBeBound** – List of Java classes to be recognized by the new `JAXBContext`
- **contextPath** – List of Java package names (or EclipseLink session names) that contain mapped classes
- **classLoader** – The class loader used to locate the mapped classes
- **properties** – A map of additional properties.

The APIs in [Example 2-1](#) expect to find a `jaxb.properties` file in your Java package/context path. For more information see "[Specifying the EclipseLink Runtime](#)".

Example 2-1 JAXBContext Methods

```
public static JAXBContext newInstance(Class... classesToBeBound) throws JAXBException

public static JAXBContext newInstance(Class[] classesToBeBound, Map<String,?>
properties) throws JAXBException

public static JAXBContext newInstance(String contextPath) throws JAXBException

public static JAXBContext newInstance(String contextPath, ClassLoader classLoader)
throws JAXBException

public static JAXBContext newInstance(String contextPath, ClassLoader classLoader,
Map<String,?> properties) throws JAXBException
```

Bootstrapping from Classes

If you have a collection of Java classes annotated with JAXB annotations, you can provide a list of these classes directly:

```
JAXBContext context = JAXBContext.newInstance(Company.class, Employee.class);
```

Other classes that are reachable from the classes in the array (for example, referenced classes, super class) will automatically be recognized by the `JAXBContext`. Subclasses or classes marked as `@XmlTransient` will not be recognized.

Bootstrapping from a Context Path

Another way to bootstrap your JAXBContext is with a String, called the *context path*. This is a colon-delimited list of package names containing your mapped classes:

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

Using this approach, there are a few different ways that EclipseLink will discover your model classes:

- [Using a jaxb.index File](#)
- [Using an ObjectFactory](#)
- [Using MetadataSource](#)

Using a jaxb.index File

The context path could contain a file named `jaxb.index`, which is a simple text file containing the class names from the current package that will be brought into the JAXBContext:

```
src/example/jaxb.index:
```

Other classes that are reachable from the classes in list (for example, referenced classes, super class) will automatically be recognized by the JAXBContext. Subclasses or classes marked as `@XmlTransient` will not be recognized.

Example 2-2 Sample jaxb.index File

```
Employee  
PhoneNumber
```

Using an ObjectFactory

The context path could also contain a class called `ObjectFactory`, which is a special factory class that JAXB will look for. This class contains `create()` methods for each of the types in your model. Typically the `ObjectFactory` will be generated by the JAXB compiler, but one can be written by hand as well.

```
src/example/ObjectFactory.java:
```

Example 2-3 Sample ObjectFactory

```
@XmlRegistry  
public class ObjectFactory {  
    private final static QName _Employee_QNAME = new QName("", "employee");  
    private final static QName _PhoneNumber_QNAME = new QName("", "phone-number");  
    public ObjectFactory() {  
    }  
    public EmployeeType createEmployeeType() {  
        return new EmployeeType();  
    }  
    @XmlElementDecl(namespace = "", name = "employee")  
    public JAXBElement<EmployeeType> createEmployee(EmployeeType value) {  
        return new JAXBElement<EmployeeType>(_Employee_QNAME, EmployeeType.class,  
        null, value);  
    }  
}
```

```

    public PhoneNumberType createPhoneNumberType() {
        return new PhoneNumberType();
    }

    @XmlElementDecl(namespace = "", name = "phone-number")
    public JAXBElement<PhoneNumberType> createPhoneNumber(PhoneNumberType value) {
        return new JAXBElement<PhoneNumberType>(_PhoneNumber_QNAME,
            PhoneNumberType.class, null, value);
    }
}

```

Using MetadataSource

EclipseLink MOXy also has the ability to retrieve mapping information from an implementation of EclipseLink's `MetadataSource`. Using this approach, you are responsible for creating your own `XmlBindings`.

For information on using a `MetadataSource`, see ["Using MetadataSource"](#).

Example 2-4 Sample Metadata Source

```

package org.eclipse.persistence.jaxb.metadata;
public interface MetadataSource {

    /**
     * @param properties - The properties passed in to create the JAXBContext
     * @param classLoader - The ClassLoader passed in to create the JAXBContext
     *
     * @return the XmlBindings object representing the metadata
     */
    XmlBindings getXmlBindings(Map<String, ?> properties, ClassLoader classLoader);
}

```

Bootstrapping from EclipseLink XML Bindings

To have more control over how your classes will be mapped to XML, you can bootstrap from an EclipseLink XML bindings document. Using this approach, you can take advantage of EclipseLink's robust mappings framework and customize how each complex type in XML maps to its Java counterpart.

Links to the actual documents are passed in via the **properties** parameter, using a special key, `JAXBContextProperties.OXM_METADATA_SOURCE`:

For more information on the XML Bindings format, see ["Using XML Bindings"](#).

Example 2-5 Using an EclipseLink Bindings Document

```

InputStream iStream = myClassLoader.getResourceAsStream("example/xml-bindings.xml");

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, iStream);

JAXBContext context = JAXBContext.newInstance(new Class[]{ Customer.class },
    properties);

```

Combining Annotated Classes and XML Bindings

When bootstrapping from annotated classes, additional mapping information can be provided with an EclipseLink XML bindings document. For instance, you might annotate your model classes with JAXB-spec-only annotations, and put your EclipseLink-specific mapping customizations into an XML bindings document (negating the need to import EclipseLink annotations in your model classes).

For example, review the annotated Employee class in [Example 2-6](#).

You can customize the Employee to use an EclipseLink XMLAdapter for marshalling/unmarshalling PhoneNumbers by using the XML Bindings in [Example 2-7](#).

Finally, pass both the list of annotated classes and the link to the XML Bindings to the JAXBContext, as shown in [Example 2-8](#).

Example 2-6 Sample Java Class

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement

@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlElement(name="phone-number")
    private PhoneNumber phoneNumber;
    ...
}
```

Example 2-7 Using an XML Bindings Document

```
<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm">
  <java-types>
    <java-type name="example.Employee">
      <java-attributes>
        <xml-element java-attribute="phoneNumber">
          <xml-java-type-adapter value="example.util.PhoneNumberProcessor"/>
        </xml-element>
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>
```

Example 2-8 Sample Application Code

```
InputStream iStream = myClassLoader.getResourceAsStream("example/xml-bindings.xml");
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, iStream);

Class[] classes = new Class[] { Company.class, Employee.class };
JAXBContext context = JAXBContext.newInstance(classes, properties);
```

Using XML Bindings

In addition to standard JAXB annotations, EclipseLink offers another way of expressing your metadata: the EclipseLink XML Bindings document. Not only can XML Bindings separate your mapping information from your actual Java class, it can also be used for more advanced metadata tasks such as:

- Augmenting or overriding existing annotations with additional mapping information
- Specifying all mappings information externally, with no annotations in Java at all
- Defining your mappings across multiple Bindings documentsSpecifying "virtual" mappings that do not correspond to concrete Java fieldsand more..

This section describes the XML Bindings format and demonstrates some basic use cases.

Understanding the XML Bindings Format

An XML Bindings document is XML that specifies Java type information, mapping information, context-wide properties – everything you need to define your JAXB system. An example Bindings document is shown in [Example 2-9](#).

Table 2-1 Binding Document Attributes

Attribute	Description
<xml-bindings>	The root of the XML Bindings document. This is also where you can define top-level properties for your JAXB system, such as the package-name of your classes, specify a default xml-accessor-type, and so on.
<xml-schema>	Defines properties related to the schema-level of your JAXB system. Corresponds to the JAXB @XmlSchema annotation.
<java-types>	Defines mapping information for each of your Java classes.
<xml-enums>	Defines Java enumerations that can be used with your Java types.
<xml-registries>	Defines an ObjectFactory for use in your JAXB system.

Example 2-9 Sample Bindings Document

```
<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xs="http://
www.w3.org/2001/XMLSchema"
  package-name="example" xml-accessor-type="PUBLIC_MEMBER" xml-accessor-
order="ALPHABETICAL"
  xml-mapping-metadata-complete="false" xml-name-
transformer="example.NameGenerator"
  supported-versions="2.4" >

  <xml-schema element-form-default="QUALIFIED">
    <xml-ns prefix="ns1" namespace-uri="http://www.example.org/type" />
  </xml-schema>
  <java-types>
    <java-type name="Employee">
      <xml-type namespace="http://www.example.org/type" />
    </java-type>
  </java-types>
</xml-bindings>
```

```

        <java-attributes>
            <xml-attribute java-attribute="empId" xml-path="@id" />
            <xml-element java-attribute="empName" name="name" />
            <xml-element java-attribute="salary" />
            <xml-element java-attribute="type" type="EmployeeType" />
        </java-attributes>
    </java-type>
</java-type name="Company">
    <xml-root-element name="company" />
    <xml-attribute java-attribute="empId" xml-path="@id" />
    <xml-element java-attribute="empName" name="name" />
    <java-attributes>
        <xml-element java-attribute="employees" name="employee"
            type="example.Employee" container-type="java.util.ArrayList" />
    </java-attributes>
</java-type>
</java-types>

<xml-registries>
    <xml-registry name="example.ObjectFactory">
        <xml-element-decl java-method="createEmpleado"
            name="empleado" type="example.Employee" />
        <xml-element-decl java-method="createCorporacion"
            name="corporacion" type="example.Company" />
    </xml-registry>
</xml-registries>

<xml-enums>
    <xml-enum java-enum="EmployeeType" value="java.lang.String">
        <xml-enum-value java-enum-value="CONTRACT">CONTRACT</xml-enum-value>
        <xml-enum-value java-enum-value="PART_TIME">PART_TIME</xml-enum-value>
        <xml-enum-value java-enum-value="FULL_TIME">FULL_TIME</xml-enum-value>
    </xml-enum>
</xml-enums>
</xml-bindings>

```

Bootstrapping with XML Bindings

When instantiating a `JAXBContext`, links to Bindings documents are passed in via the properties parameter, using a special key, `JAXBContextProperties.OXM_METADATA_SOURCE`. The value of this key will be a handle to the Bindings document, in the form of one of the following:

- `java.io.File`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `javax.xml.stream.XMLStreamReader`
- `javax.xml.stream.XMLStreamReader`
- `javax.xml.transform.Source`
- `org.w3c.dom.Node`
- `org.xml.sax.InputSource`

To bootstrap from multiple XML Bindings documents:

- Maps of the above inputs are supported, keyed on Java package name.
- Lists of the above inputs are acceptable as well (<xml-bindings> must have package attribute).

Using XML Bindings with Annotations

The most typical use of an XML Bindings document is in conjunction with JAXB annotations. You may have situation where you are not permitted to edit your Java domain classes, but want to add additional mapping functionality. Or, you may wish to avoid importing any EclipseLink code into your domain model, but still take advantage of MOXy's advanced mapping features. When Bindings metadata is provided during context creation, its mapping information will be combined with any JAXB annotation information.

For example, consider the simple JAXB domain class and its default JAXB XML representation shown in [Example 2-10](#).

Now, assume that we would like to make the following mapping changes:

- Change the XML element name of custId to customer-id
- Change the root element name of the class to customer-info
- Write the picture to XML as picture-hex in hex binary format, and use our own custom converter, MyHexConverter.

We can specify these three customizations in an XML Bindings document as shown in [Example 2-11](#).

The Bindings must then be provided during JAXB context creation. Bindings information is passed in via the properties argument:

When providing Bindings, during JAXB context creation Oracle TopLink will:

1. Customer.class will be analyzed and JAXB mappings will be generated as usual.
2. The Bindings document is then analyzed, and the original JAXB mappings will be merged with the information in the Bindings document.

After applying the XML Bindings, we have the desired XML representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer-info customer-id="15">
  <name>Bob Dobbs</name>
  <salary>51727.61</salary>
  <picture-hex>020408102040</picture-hex>
</customer-info>
```

Example 2-10 Sample JAXB Domain Class and XML

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlAttribute
    private Integer custId;
    private String name;
```

```

    private Double salary;
    private byte[] picture;
    ...
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<customer custId="15">
    <name>Bob Dobbs</name>
    <salary>51727.61</salary>
    <picture>AgQIECBA</picture>
</customer>

```

Example 2-11 Customized XML Bindings

```

<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
    package-name="example">

    <java-types>
        <java-type name="Customer">
            <xml-root-element name="customer-info" />
            <java-attributes>
                <xml-attribute java-attribute="custId" name="customer-id" />
                <xml-element java-attribute="picture" name="picture-hex">
                    <xml-schema-type name="hexBinary" />
                    <xml-java-type-adapter
                        value="example.adapters.MyHexConverter" />
                </xml-element>
            </java-attributes>
        </java-type>
    </java-types>

</xml-bindings>

```

Example 2-12 Providing Bindings

```

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
InputStream iStream = classLoader.getResourceAsStream("metadata/xml-bindings.xml");

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, iStream);

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Customer.class },
    properties);

```

Using Multiple Bindings Documents

Starting with version 2.3, EclipseLink allows you to use mapping information from multiple XML Bindings documents. Using this approach, you can split your metadata up as you wish.

When using a List of Bindings documents, each one must define the package attribute of `<xml-bindings>`, to indicate the package for each set of Bindings.

Example 2-13 Using a List of XML Bindings:

```
...
FileReader file1 = new FileReader("base-bindings.xml");
FileReader file2 = new FileReader("override-bindings.xml");

List<Object> fileList = new ArrayList<Object>();
fileList.add(file1);
fileList.add(file2);

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, fileList);

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Customer.class },
properties);

...
```

Example 2-14 Using a Map for multiple packages:

```
...

FileReader fooFile1 = new FileReader("foo/base-bindings.xml");
FileReader fooFile2 = new FileReader("foo/override-bindings.xml");

List<Object> fooFileList = new ArrayList<Object>();
fooFileList.add(fooFile1);
fooFileList.add(fooFile2);

FileReader barFile1 = new FileReader("bar/base-bindings.xml");
FileReader barFile2 = new FileReader("bar/override-bindings.xml");

List<Object> barFileList = new ArrayList<Object>();
barFileList.add(barFile1);
barFileList.add(barFile2);

Map<String, List> metadataMap = new HashMap<String, List>();
metadataMap.put("foo", fooFileList);
metadataMap.put("bar", barFileList);

properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, metadataMap);

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Customer.class },
properties);

...
```

Understanding Override Rules

When multiple sources of metadata are encountered for the same package, a unified set of mappings will be created by merging the complete set of metadata. First, the annotations from the Java class will be processed, and then any XML Bindings information will be applied. The order that Bindings are specified is relevant; values in subsequent documents will override the ones defined in previous ones.

The following rules will be used for merging:

- xml-schema

- For values such as namespace, elementform, attributeform, the later file will override.
- The list of namespace declarations from XmlNs will be merged into a single list containing all entries from all files.

In the case of conflicting entries (the same prefix bound to multiple namespaces), the last file will override the declarations from previous files.
- java-types
 - The merged bindings will contain all unique java-type entries from all bindings files.
 - If the same java-type occurs in multiple files, any values that are set in the later file will override values from the previous file.
 - Properties on each java-type will be merged into a unified list. If the same property is referenced in multiple files, this will be an exception case.
 - Class-level XmlJavaTypeAdpater entries will be overridden if specified in a later bindings file.
 - Class-level XmlSchemaTypes will create a merged list. If an entry for the same type is listed in multiple bindings files at this level, the last file's entry will override all previous ones.
- xml-enums
 - The merged bindings will contain all unique xml-enum entries from all bindings files.
 - For any duplicated java-enums, a merged list of XmlEnumValues will be created. If an entry for the same enum facet occurs in multiple files, the last file will override the value for that facet.
- xml-java-type-adapters
 - Package-level Java type adapters will be merged into a single list. In the case that an adapter is specified for the same class in multiple files, the last file's entry will win.
- xml-registries
 - Each unique XmlRegistry entry will be added to the final merged list of XmlRegistries.
 - For any duplicated XmlRegistry entries, a merged list of XmlElementDecIs will be created.

In the case that an XmlElementDecl for the same XmlRegistry class appears in multiple bindings files, that XmlElementDecl will be replaced with the one from the later bindings.
- xml-schema-types
 - XmlSchemaType entries will be merged into a unified list.
 - In the case that an XmlSchemaType entry for the same java-type appears at the package level in multiple bindings files, the merged bindings will only contain the entry for the last one specified.

Using Complete Metadata

If you would like to store all of your metadata in XML Bindings and ignore any JAXB annotations in your Java class, you can include the `xml-mapping-metadata-complete` attribute in the `<xml-bindings>` element of your Bindings document. Default JAXB mappings will still be generated (the same as if you were using a completely unannotated class with JAXB), and then any mapping data defined in the XML Bindings will be applied.

This could be used, for example, to map the same Java class to two completely different XML representations: the annotations on the actual Java class would define the first XML representation, and then a second XML representation could be defined in an XML Bindings document with `xml-mapping-metadata-complete="true"`. This would essentially give you a "blank canvas" to remap your Java class.

If you would like to ignore the default mappings that JAXB generates, you can specify `xml-accessor-type="NONE"` in your `<java-type>` element. Using this approach, only mappings that are explicitly defined in Bindings document will be applied.

Using the **Customer** example from above, the following examples demonstrate the XML representations that will be generated when using `xml-mapping-metadata-complete`:

- Default JAXB mapping is generated for `custId` (note that `custId` is now an XML element, as if there were no annotation on the Java field)
- The name element has been renamed to `customer-name`
- Default JAXB mappings are generated for `picture` and `salary`
- Specifying `xml-accessor-type="NONE"` will prevent any default mappings from being generated
- The XML representation contains only the mappings defined in the XML Bindings document

Example 2-15 Sample Customer Class

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlAttribute
    private Integer custId;
    private String name;
    private Double salary;
    private byte[] picture;
    ...
}
```

Example 2-16 XML Bindings

```
<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
    package-name="example" xml-mapping-metadata-complete="true">
```

```

<java-types>
  <java-type name="Customer">
    <xml-root-element />
    <java-attributes>
      <xml-attribute java-attribute="name" name="customer-name" />
    </java-attributes>
  </java-type>
</java-types>

</xml-bindings>

```

Example 2-17 XML Representation

```

<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <custId>15</custId>
  <customer-name>Bob Dobbs</customer-name>
  <picture>AgQIECBA</picture>
  <salary>51727.61</salary>
</customer>

```

Example 2-18 XML Bindings (with xml-accessor-type="NONE")

```

<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
  package-name="example" xml-mapping-metadata-complete="true">

  <java-types>
    <java-type name="Customer" xml-accessor-type="NONE">
      <xml-root-element />
      <java-attributes>
        <xml-attribute java-attribute="name" name="customer-name" />
      </java-attributes>
    </java-type>
  </java-types>

</xml-bindings>

```

Example 2-19 XML Representation

```

<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <customer-name>Bob Dobbs</customer-name>
</customer>

```

Using Virtual Mappings

XML Bindings can also be used to specify virtual mappings – mappings that do not correspond to a concrete Java field. For example, you might want to use a `HashMap` as the underlying structure to hold data for certain mappings.

Using MetadataSource

The `MetadataSource`, introduced in EclipseLink 2.3, is responsible for serving up EclipseLink metadata. This allows you to store mapping information outside of your application and have it retrieved when the application's `JAXBContext` is being created or refreshed.

Implementing a MetadataSource

To implement your own MetadataSource, you can:

- Create a new class that implements the `org.eclipse.persistence.jaxb.metadata.MetadataSource` interface.
- Create a new class that extends the `org.eclipse.persistence.jaxb.metadata.MetadataSourceAdapter` class. Using this method is preferred, as it will insulate you from future additions to the interface.

In either case, you will be responsible for implementing the following method:

Example 2-20 Implementing the XmlBindings Method

```
/**
 * Retrieve XmlBindings according to the JAXBContext bootstrapping information.
 *
 * @param properties - The properties passed in to create the JAXBContext
 * @param classLoader - The ClassLoader passed in to create the JAXBContext
 * @return the XmlBindings object representing the metadata
 */
XmlBindings getXmlBindings(Map<String, ?> properties, ClassLoader classLoader);
```

Using an XmlBindings Object

Internally, EclipseLink metadata is stored in an `XmlBindings` object, which itself is mapped with JAXB. This means that you can actually use a JAXB unmarshaller to read external metadata and create an `XmlBindings` from it:

Example 2-21 Sample XmlBindings Object

```
package example;

import org.eclipse.persistence.jaxb.xmlmodel.XmlBindings;
...
JAXBContext xmlBindingsContext =
    JAXBContext.newInstance("org.eclipse.persistence.jaxb.xmlmodel");
FileReader bindingsFile = new FileReader("xml-bindings.xml");
XmlBindings bindings = (XmlBindings)
    xmlBindingsContext.createUnmarshaller().unmarshal(bindingsFile);
```

Specifying the MetadataSource

To use a MetadataSource in creating a JAXBContext, add it to the properties map with the key `JAXBContextProperties.OXM_METADATA_SOURCE`:

Example 2-22 Adding MetadataSource to the Properties Map

```
MetadataSource metadataSource = new MyMetadataSource();

Map<String, Object> properties = new HashMap<String, Object>(1);
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, metadataSource);

JAXBContext jc = JAXBContext.newInstance(new Class[] { Customer.class }, properties);
```

MetadataSource Example

The following example creates an `XmlBindings` object by unmarshalling from a URL:

Example 2-23 Sample `XmlBindings` Object

```
package example;

import java.net.URL;
import java.util.Map;

import javax.xml.bind.JAXBContext;

import org.eclipse.persistence.jaxb.metadata.MetadataSourceAdapter;
import org.eclipse.persistence.jaxb.xmlmodel.XmlBindings;

public class MyMetadataSource extends MetadataSourceAdapter {

    private JAXBContext bindingsContext;
    private URL bindingsUrl;

    private final String XML_BINDINGS_PACKAGE =
"org.eclipse.persistence.jaxb.xmlmodel";
    private final String METADATA_URL = "http://www.example.com/private/metadata/xml-
bindings.xml";
    public MyMetadataSource() {
        try {
            bindingsContext = JAXBContext.newInstance(XML_BINDINGS_PACKAGE);
            bindingsUrl = new URL(METADATA_URL);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    @Override
    public XmlBindings getXmlBindings(Map<String, ?> properties, ClassLoader
classLoader) {
        try {
            Unmarshaller u = bindingsContext.createUnmarshaller();
            XmlBindings bindings = (XmlBindings) u.unmarshal(bindingsUrl);
            return bindings;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

Building `XmlBindings` Programatically

You also have the option of building your own `XmlBindings` object from scratch in code. The example below modifies the `pCode` field of the **Address** class to use a locale-specific name:

Example 2-24 Sample `XmlBindings` Object

```
package example;

import java.util.Locale;
```

```

import java.util.Map;

import org.eclipse.persistence.jaxb.metadata.MetadataSourceAdapter;
import org.eclipse.persistence.jaxb.xmlmodel.JavaType;
import org.eclipse.persistence.jaxb.xmlmodel.JavaType.JavaAttributes;
import org.eclipse.persistence.jaxb.xmlmodel.ObjectFactory;
import org.eclipse.persistence.jaxb.xmlmodel.XmlBindings;
import org.eclipse.persistence.jaxb.xmlmodel.XmlBindings.JavaTypes;
import org.eclipse.persistence.jaxb.xmlmodel.XmlElement;

public class AddressMetadataSource extends MetadataSourceAdapter {

    private ObjectFactory factory;
    private XmlBindings xmlBindings;

    public AddressMetadataSource() {
        factory = new ObjectFactory();

        xmlBindings = new XmlBindings();
        xmlBindings.setPackageName("example");
        xmlBindings.setJavaTypes(new JavaTypes());
    }

    @Override
    public XmlBindings getXmlBindings(Map<String, ?> properties, ClassLoader
classLoader) {
        JavaType javaType = new JavaType();
        javaType.setName("Address");
        javaType.setJavaAttributes(new JavaAttributes());

        XmlElement pCodeElement = new XmlElement();
        pCodeElement.setJavaAttribute("pCode");

        String country = Locale.getDefault().getCountry();
        if (country.equals(Locale.US.getCountry())) {
            pCodeElement.setName("zip-code");
        } else if (country.equals(Locale.UK.getCountry())) {
            pCodeElement.setName("post-code");
        } else if (country.equals(Locale.CANADA.getCountry())) {
            pCodeElement.setName("postal-code");
        }

        javaType.getJavaAttributes().getJavaAttribute().add(factory.createXmlElement(
pCodeElement));

        xmlBindings.getJavaTypes().getJavaType().add(javaType);
        return xmlBindings;
    }
}

```

Generating an XML Schema

To generate an XML schema from a Java object model:

1. Create a class that extends `javax.xml.bind.SchemaOutputResolver`.

```

private class MySchemaOutputResolver extends SchemaOutputResolver {

    public Result createOutput(String uri, String suggestedFileName) throws

```

```
IOException {
    File file = new File(suggestedFileName);
    StreamResult result = new StreamResult(file);
    result.setSystemId(file.toURI().toURL().toString());
    return result;
}
}
```

2. Use an instance of this class with JAXBContext to capture the generated XML Schema.

```
Class[] classes = new Class[4];
classes[0] = org.example.customer_example.AddressType.class;
classes[1] = org.example.customer_example.ContactInfo.class;
classes[2] = org.example.customer_example.CustomerType.class;
classes[3] = org.example.customer_example.PhoneNumber.class;
JAXBContext jaxbContext = JAXBContext.newInstance(classes);

SchemaOutputResolver sor = new MySchemaOutputResolver();
jaxbContext.generateSchema(sor);
```

Validating Against an XML Schema

If you would like to validate your objects against an XML Schema during marshalling and unmarshalling, you can make use of JAXB's `ValidationEventHandler`.

Notice the following constraints:

- The customer's name cannot be longer than five (5) characters.
- The customer cannot have more than two (2) phone numbers.

The **Customer** class is shown below. Notice that the class *does not* contain any validation code.

Example 2-25 Sample XML Schema

In this example we would like to validate our objects against the following XML schema:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="customer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="name" type="stringMaxSize5"/>
        <xs:element ref="phone-number" maxOccurs="2"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="phone-number">
    <xs:complexType>
      <xs:sequence/>
    </xs:complexType>
  </xs:element>

  <xs:simpleType name="stringMaxSize5">
    <xs:restriction base="xs:string">
```

```

        <xs:maxLength value="5"/>
    </xs:restriction>
</xs:simpleType>

</xs:schema>

```

Example 2-26 Sample Customer Class

```

package example;

import java.util.ArrayList;
import java.util.List;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer {
    private String name;

    private List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlElement(name="phone-number")
    public List<PhoneNumber> getPhoneNumbers() {
        return phoneNumbers;
    }

    public void setPhoneNumbers(List<PhoneNumber> phoneNumbers) {
        this.phoneNumbers = phoneNumbers;
    }
}

```

Using a ValidationEventHandler

You can receive JAXB validation events by providing your own subclass of `ValidationEventHandler`. The event is represented as an instance of `ValidationEvent`, and provides many details about the issue. The data is similar to what is available from a `SAXParseException`.

- Returning **false** from the `handleEvent` method will cause the JAXB operation to stop.
- Returning **true** will allow the method to continue, if possible.

In [Example 2-27](#), we will simply print out an event's data when one is received:

Example 2-27 Sample ValidationEventHandler

```

package example;

import javax.xml.bind.ValidationEvent;
import javax.xml.bind.ValidationEventHandler;

```

```
public class MyValidationEventHandler implements ValidationEventHandler {

    public boolean handleEvent(ValidationEvent event) {
        System.out.println("\nEVENT");
        System.out.println("SEVERITY: " + event.getSeverity());
        System.out.println("MESSAGE: " + event.getMessage());
        System.out.println("LINKED EXCEPTION: " + event.getLinkedException());
        System.out.println("LOCATOR");
        System.out.println("    LINE NUMBER: " +
event.getLocator().getLineNumber());
        System.out.println("    COLUMN NUMBER: " +
event.getLocator().getColumnNumber());
        System.out.println("    OFFSET: " + event.getLocator().getOffset());
        System.out.println("    OBJECT: " + event.getLocator().getObject());
        System.out.println("    NODE: " + event.getLocator().getNode());
        System.out.println("    URL: " + event.getLocator().getURL());
        return true;
    }
}
```

Enabling Validation

In addition to providing an implementation of `ValidationEventHandler`, an instance of `Schema` must be set on the `Marshaller` or `Unmarshaller`.

Example 2-28 Sample Java Code

```
package example;

import java.io.File;
import javax.xml.XMLConstants;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.Unmarshaller;
import javax.xml.validation.Schema;
import javax.xml.validation.SchemaFactory;

public class UnmarshalDemo {

    public static void main(String[] args) throws Exception {
        SchemaFactory sf =
SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
        Schema schema = sf.newSchema(new File("customer.xsd"));

        JAXBContext jc = JAXBContext.newInstance(Customer.class);

        Unmarshaller unmarshaller = jc.createUnmarshaller();
        unmarshaller.setSchema(schema);
        unmarshaller.setEventHandler(new MyValidationEventHandler());
        Customer customer = (Customer) unmarshaller.unmarshal(new File("input.xml"));
    }
}
```

Input (input.xml File)

```
<customer>
  <name>Jane Doe</name>
  <phone-number/>
  <phone-number/>
  <phone-number/>
</customer>
```

Output

The validation performed during the unmarshal raised three events. The first two events are related to the text value of the **name** element being too long. The third event is related to the extra **phone-number** element.

```
EVENT
SEVERITY: 1
MESSAGE: cvc-maxLength-valid: Value 'Jane Doe' with length = '8' is not facet-valid
with respect
    to maxLength '5' for type 'stringWithMaxSize5'.
LINKED EXCEPTION: org.xml.sax.SAXParseException: cvc-maxLength-valid: Value 'Jane
Doe' with length = '8'
    is not facet-valid with respect to maxLength '5' for type
'stringWithMaxSize5'.
LOCATOR
    LINE NUMBER: 3
    COLUMN NUMBER: 25
    OFFSET: -1
    OBJECT: null
    NODE: null
    URL: null

EVENT
SEVERITY: 1
MESSAGE: cvc-type.3.1.3: The value 'Jane Doe' of element 'name' is not valid.
LINKED EXCEPTION: org.xml.sax.SAXParseException: cvc-type.3.1.3: The value 'Jane
Doe' of element
    'name' is not valid.
LOCATOR
    LINE NUMBER: 3
    COLUMN NUMBER: 25
    OFFSET: -1
    OBJECT: null
    NODE: null
    URL: null

EVENT
SEVERITY: 1
MESSAGE: cvc-complex-type.2.4.d: Invalid content was found starting with element
'customer'. No child
    element '{phone-number}' is expected at this point.
LINKED EXCEPTION: org.xml.sax.SAXParseException: cvc-complex-type.2.4.d: Invalid
content was found starting
    with element 'customer'. No child element '{phone-number}' is
expected at this point.
LOCATOR
    LINE NUMBER: 7
    COLUMN NUMBER: 12
```

```
OFFSET:  -1
OBJECT:  null
NODE:    null
URL:     null
```

Understanding Events

JAXB offers several mechanisms to get event callbacks during the marshalling and unmarshalling processes. You can specify callback methods directly on your mapped objects, or define separate Listener classes and register them with the JAXB runtime.

Adding Event Listener Methods on JAXB Mapped Objects

On any of your objects you have mapped with JAXB, you have the option of specifying some special methods to allow you to receive event notification when that object is marshalled or unmarshalled. The methods must have the following signatures:

```
/**
 * Invoked by Marshaller after it has created an instance of this object.
 */
void beforeMarshal(Marshaller m);

/**
 * Invoked by Marshaller after it has marshalled all properties of this object.
 */
void afterMarshal(Marshaller m);

/**
 * This method is called immediately after the object is created and before the
 * unmarshalling of this
 * object begins. The callback provides an opportunity to initialize JavaBean
 * properties prior to unmarshalling.
 */
void beforeUnmarshal(Unmarshaller u, Object parent);

/**
 * This method is called after all the properties (except IDREF) are unmarshalled
 * for this object,
 * but before this object is set to the parent object.
 */
void afterUnmarshal(Unmarshaller u, Object parent);
```

The following example shows how to write to a log every time a **Company** object is processed.

Example 2-29 Sample Event Listener

```
package example;

import java.util.Date;
import java.util.logging.Logger;
import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Company {

    @XmlAttribute
```

```

    private String id;

    void beforeMarshal(Marshaller m) {
        Logger.getLogger("example").info("COMPANY:[id=" + id + "]" +
            Thread.currentThread());
    }

    void afterMarshal(Marshaller m) {
        Logger.getLogger("example").info("COMPANY:[id=" + id + "]" +
            Thread.currentThread());
    }

    void beforeUnmarshal(Unmarshaller u, Object parent) {
        Logger.getLogger("example").info("COMPANY:[id=" + id + "]" +
            Thread.currentThread());
    }

    void afterUnmarshal(Unmarshaller u, Object parent) {
        Logger.getLogger("example").info("COMPANY:[id=" + id + "]" +
            Thread.currentThread());
    }
}

```

Registering Listeners on Marshallers and Unmarshallers

JAXB's `Marshaller` and `Unmarshaller` interfaces both define a `setListener()` method, which allows you to set your own custom `Listener` to intercept marshal and unmarshal events.

```

package javax.xml.bind;

public interface Marshaller {
    ...
    public static abstract class Listener {
        /**
         * Callback method invoked before marshalling from source to XML.
         *
         * This method is invoked just before marshalling process starts to marshal
         source.
         * Note that if the class of source defines its own beforeMarshal method,
         * the class specific callback method is invoked just before this method is
         invoked.
         *
         * @param source instance of JAXB mapped class prior to marshalling from it.
         */
        public void beforeMarshal(Object source) {}

        /**
         * Callback method invoked after marshalling source to XML.
         *
         * This method is invoked after source and all its descendants have been
         marshalled.
         * Note that if the class of source defines its own afterMarshal method,
         * the class specific callback method is invoked just before this method is
         invoked.
         *
         * @param source instance of JAXB mapped class after marshalling it.
         */
        public void afterMarshal(Object source) {}
    }
}

```

```

    }
}

package javax.xml.bind;

public interface Unmarshaller {
    ...
    public static abstract class Listener {

        /**
         * Callback method invoked before unmarshalling into target.
         *
         * This method is invoked immediately after target was created and
         * before the unmarshalling of this object begins. Note that
         * if the class of target defines its own beforeUnmarshasha method,
         * the class specific callback method is invoked before this method is invoked.
         *
         * @param target non-null instance of JAXB mapped class prior to unmarshalling
         into it.
         * @param parent instance of JAXB mapped class that will eventually reference
         target.
         *
         * null when target is root element.
         */
        public void beforeUnmarshal(Object target, Object parent) {}

        /**
         * Callback method invoked after unmarshalling XML data into target.
         *
         * This method is invoked after all the properties (except IDREF) are
         unmarshalled into target,
         * but before target is set into its parent object.
         * Note that if the class of target defines its own afterUnmarshal method,
         * the class specific callback method is invoked before this method is invoked.
         *
         * @param target non-null instance of JAXB mapped class prior to unmarshalling
         into it.
         * @param parent instance of JAXB mapped class that will reference target.
         *
         * null when target is root element.
         */
        public void afterUnmarshal(Object target, Object parent) {}
    }
}

```

This example performs the same logging as above, but using generic Listener classes. This makes it easier to log all JAXB objects in the system.

The following code sets up the listeners:

```

Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setListener(new MarshalLogger());

Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
unmarshaller.setListener(new UnmarshalLogger());

```

An example of a typical marshal/unmarshal example, showing both the class-level and Marshaller/Unmarshaller-level event output:

```

Jun 2, 2011 6:31:59 PM example.Company beforeMarshal
INFO: COMPANY:[id=Zoltrix] Thread[main,5,main]

```

```

Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger beforeMarshal
INFO: example.Company@10e790c Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger beforeMarshal
INFO: example.Employee@1db7df8 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger afterMarshal
INFO: example.Employee@1db7df8 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger beforeMarshal
INFO: example.Employee@3570b0 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger afterMarshal
INFO: example.Employee@3570b0 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger beforeMarshal
INFO: example.Employee@79717e Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger afterMarshal
INFO: example.Employee@79717e Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Company afterMarshal
INFO: COMPANY:[id=Zoltrix] Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$MarshalLogger afterMarshal
INFO: example.Company@10e790c Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Company beforeUnmarshal
INFO: COMPANY:[id=null] Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger beforeUnmarshal
INFO: example.Company@f0c0d3 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger beforeUnmarshal
INFO: example.Employee@4f80d6 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger afterUnmarshal
INFO: example.Employee@4f80d6 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger beforeUnmarshal
INFO: example.Employee@1ea0252 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger afterUnmarshal
INFO: example.Employee@1ea0252 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger beforeUnmarshal
INFO: example.Employee@3e89c3 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger afterUnmarshal
INFO: example.Employee@3e89c3 Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Company afterUnmarshal
INFO: COMPANY:[id=Zoltrix] Thread[main,5,main]
Jun 2, 2011 6:31:59 PM example.Tester$UnmarshalLogger afterUnmarshal
INFO: example.Company@f0c0d3 Thread[main,5,main]

```

Example 2-30 Logging with the Listener Class

```

package example;

import java.util.logging.Logger;

private class MarshalLogger extends Marshaller.Listener {
    @Override
    public void afterMarshal(Object source) {
        Logger.getLogger("example").info(source + " " + Thread.currentThread());
    }

    @Override
    public void beforeMarshal(Object source) {
        Logger.getLogger("example").info(source + " " + Thread.currentThread());
    }
}

package example;

import java.util.logging.Logger;

```

```

private class UnmarshalLogger extends Unmarshaller.Listener {
    @Override
    public void afterUnmarshal(Object target, Object parent) {
        Logger.getLogger("example").info(target + " " + Thread.currentThread());
    }

    @Override
    public void beforeUnmarshal(Object target, Object parent) {
        Logger.getLogger("example").info(target + " " + Thread.currentThread());
    }
}

```

Querying Objects by XPath

In addition to using conventional Java access methods to get and set your object's values, EclipseLink MOXy also allows you to access values using an XPath statement. There are special APIs on EclipseLink's JAXBContext to allow you to get and set values by XPath.

For example, consider the following XML document:

```

<customer id="1141">
  <first-name>Jon</first-name>
  <last-name>Smith</last-name>
  <phone-number>
    <area-code>515</area-code>
    <number>2726652</number>
  </phone-number>
</customer>

```

Typical application code might look something like this:

```

Customer customer = (Customer)
jaxbContext.createUnmarshaller().unmarshal(instanceDoc);
...
int customerId = customer.getId();
customer.setFirstName("Bob");
customer.getPhoneNumber().setAreaCode("555");
...
jaxbContext.createMarshaller().marshal(customer, System.out);

```

You could instead use XPath to access these values:

```

Customer customer = (Customer)
jaxbContext.createUnmarshaller().unmarshal(instanceDoc);
...
int customerId = jaxbContext.getValueByXPath(customer, "@id", null, Integer.class);
jaxbContext.setValueByXPath(customer, "first-name/text()", null, "Bob");
jaxbContext.setValueByXPath(customer, "phone-number/area-code/text()", null, "555");
...
jaxbContext.createMarshaller().marshal(customer, System.out);

```

Binding to an Existing Document

The JAXB Binder interface (introduced in JAXB 2.0) allows you to preserve an entire XML document, even if only some of the items are mapped.

Normally, when using an Unmarshaller to load the XML document into objects, and a Marshaller to save the objects back to XML, the unmapped content will be lost.

A Binder can be created from the JAXBContext to interact with the XML in the form of a DOM. The Binder maintains an association between the Java objects and their corresponding DOM nodes.

The Binder applies the changes to the original DOM instead of creating a new XML document, thereby preserving the entire XML infoset.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<customer>
  <UNMAPPED_ELEMENT_1/>
  <name>Jane Doe</name>
  <!-- COMMENT #1 -->
  <address>
    <UNMAPPED_ELEMENT_2/>
    <street>2 NEW STREET</street>
    <!-- COMMENT #2 -->
    <UNMAPPED_ELEMENT_3/>
    <city>Any Town</city>
  </address>
  <!-- COMMENT #3 -->
  <UNMAPPED_ELEMENT_4/>
  <phone-number type="home">555-HOME</phone-number>
  <!-- COMMENT #4 -->
  <phone-number type="cell">555-CELL</phone-number>
  <phone-number type="work">555-WORK</phone-number>
  <UNMAPPED_ELEMENT_5/>
  <!-- COMMENT #5 -->
</customer>
```

Example 2-31 Using a Binder

```
import java.io.File;
import javax.xml.bind.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import org.w3c.dom.*;

public class BinderDemo {

    public static void main(String[] args) throws Exception {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        DocumentBuilder db = dbf.newDocumentBuilder();
        File xml = new File("input.xml");
        Document document = db.parse(xml);

        JAXBContext jc = JAXBContext.newInstance(Customer.class);

        Binder<Node> binder = jc.createBinder();
        Customer customer = (Customer) binder.unmarshal(document);
        customer.getAddress().setStreet("2 NEW STREET");
        PhoneNumber workPhone = new PhoneNumber();
        workPhone.setType("work");
        workPhone.setValue("555-WORK");
        customer.getPhoneNumbers().add(workPhone);
        binder.updateXML(customer);
    }
}
```

```
TransformerFactory tf = TransformerFactory.newInstance();
Transformer t = tf.newTransformer();
t.transform(new DOMSource(document), new StreamResult(System.out));
}
}
```

3

Mapping Type Levels

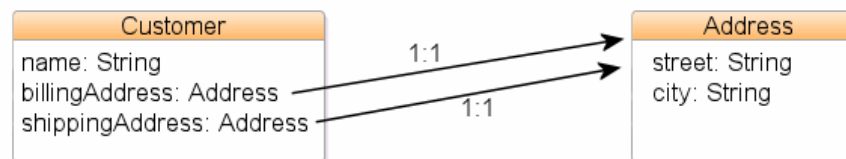
This chapter includes the following sections:

- [Defining the Default Root Element](#)
- [Setting Up Namespace Information](#)
- [Specifying Inheritance](#)

Defining the Default Root Element

At least one of your mapped classes must have a default root element defined. This tells EclipseLink what the top-level root of your XML document will be. Consider the **Customer** and **Address** classes shown in [Figure 3-1](#):

Figure 3-1 Sample Mapped Classes



These classes correspond to the XML schema shown in [Example 3-1](#). The schema contains a top-level element of type customer-type, therefore our **Customer** class will need to have a default root element specified.

[Example 3-2](#) shows how to annotate your Java class to specify a default root element. All that is needed is the standard JAXB `@XmlElement` annotation.

[Example 3-3](#) shows how to specify a default root element in EclipseLink's OXM metadata format.

Example 3-1 Sample XML Schema

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="address-type">
    <xsd:sequence>
      <element name="street" type="xsd:string"/>
      <element name="city" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
```

```

        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="billing-address" type="address-type"/>
            <xsd:element name="shipping-address" type="address-type"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>

```

Example 3-2 Using the @XmlRootElement Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
public class Customer {
    private String name;

    @XmlElement(name="billing-address")
    private Address billingAddress;

    @XmlElement(name="shipping-address")
    private Address shippingAddress;

    ...
}

```

Example 3-3 Specifying a Default Root Element

```

...
<java-type name="Customer">
    <xml-root-element/>
    <java-attributes>
        <xml-element java-attribute="name"/>
        <xml-element java-attribute="billingAddress" name="billing-address"/>
        <xml-element java-attribute="shippingAddress" name="shipping-address"/>
    </java-attributes>
</java-type>
...

```

Customizing the Default Root Element

In [Example 3-2](#), our class is called **Customer**, and our root element name in XML is `customer`. By default, when `@XmlRootElement` is specified, the name of the class has its first letter lower-cased, and this is set as the root element name. If, however, the element name in XML is different from the Java class name, a `name` attribute can be included in the annotation (as in [Example 3-4](#)) or OXM metadata (as in [Example 3-5](#)):

For more information on JAXB name-binding algorithms, see "Appendix D: Binding XML Names to Java Identifiers" of the Java Architecture for XML Binding (JAXB) Specification (<http://jcp.org/en/jsr/detail?id=222>).

Example 3-4 Using Annotations

```

package example;

import javax.xml.bind.annotation.*;

```

```

@XmlRootElement(name="my-customer")
public class Customer {
    private String name;

    @XmlElement(name="billing-address")
    private Address billingAddress;

    @XmlElement(name="shipping-address")
    private Address shippingAddress;

    ...
}

```

Example 3-5 Using OXM Metadata

```

...
<java-type name="Customer">
  <xml-root-element name="my-customer"/>
  <java-attributes>
    <xml-element java-attribute="name"/>
    <xml-element java-attribute="billingAddress" name="billing-address"/>
    <xml-element java-attribute="shippingAddress" name="shipping-address"/>
  </java-attributes>
</java-type>
...

```

Understanding How EclipseLink Uses the Default Root Element

When an instance of the **Customer** class is persisted to XML, the EclipseLink runtime performs the following:

- Gets the default root element. The **Customer** class instance corresponds to the root of the XML document. The EclipseLink runtime uses the default root element (customer) specified in either annotations or OXM to start the XML document. EclipseLink then uses the mappings on the class to marshal the object's attributes.

```

<customer>
  <name>...</name>
</customer>

```

- When the EclipseLink runtime encounters an object attribute such as `billingAddress`, it checks the mapping associated with it to determine with what element (`billing-address`) to continue.

```

<customer>
  <name>...</name>
  <billing-address/>
</customer>

```

- The EclipseLink runtime checks the mapping's reference descriptor (**Address**) to determine what attributes to persist.

```

<customer>
  <name>...</name>
  <billing-address>
    <street>...</street>

```

```

        <city>...</city>
    </billing-address>
</customer>

```

Setting Up Namespace Information

Most XML documents are qualified with a namespace. You can namespace-qualify elements of your Java class at the following levels:

- [Qualifying at the Package Level](#)
- [Qualifying at the Type Level](#)
- [Qualifying at the Field/Property Level](#)

In most cases, package-level annotation is sufficient. You can use the other levels to customize your document. Use the `@XmlSchema` annotation to specify the namespace.

Qualifying at the Package Level

Use the `@XmlSchema` annotation on the package to set a default namespace and specify that all elements in the package are qualified with the namespace. This information is specified in a special Java source file, `package-info.java`.

This can be defined in EclipseLink XML Bindings as follows:

Using a simple **Customer** class, [Example 3-6](#) and [Example 3-7](#) will produce the following XML:

```

<customer xmlns="http://www.example.org/package">
    <name>Jane Doe</name>
    <account>36328721</account>
</customer>

```

All elements are qualified with the `http://www.example.org/package` namespace.

Example 3-6 Using Annotations

```

@XmlSchema(
    namespace="http://www.example.org/package",
    elementFormDefault=XmlNsForm.QUALIFIED)
package example;

```

```

import javax.xml.bind.annotation.XmlNsForm;
import javax.xml.bind.annotation.XmlSchema;

```

Example 3-7 Using OXM Metadata

```

<?xml version="1.0" encoding="UTF-8"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm">
    <xml-schema
        element-form-default="QUALIFIED"
        namespace="http://www.example.org/package">
    </xml-schema>

    <java-types>
        <java-type name="Customer">
            ...

```

```
</xml-bindings>
```

Qualifying at the Type Level

Type level annotations will override the package level namespace.

This can be defined in EclipseLink XML Bindings as follows:

This will produce the following XML:

```
<custom xmlns="http://www.example.org/package" xmlns:ns0="http://www.example.org/type">
  <ns0:name>Bob</ns0:name>
  <ns0:account>1928712</ns0:account>
</custom>
```

Elements inside the **Customer** type are qualified with the **http://www.example.org/type** namespace.

Example 3-8 Using Annotations

```
package example;

@XmlRootElement
@XmlType(namespace="http://www.example.org/type")
public class Customer {
    private String name;

    private String account;

    ...
}
```

Example 3-9 Using XML Bindings File

```
<?xml version="1.0" encoding="UTF-8"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm">
  <xml-schema
    element-form-default="QUALIFIED"
    namespace="http://www.example.org/package">
  </xml-schema>

  <java-types>
    <java-type name="Customer">
      <xml-type namespace="http://www.example.org/type" />
      <java-attributes>
        <xml-element java-attribute="name" />
        <xml-element java-attribute="account" />
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>
```

Qualifying at the Field/Property Level

You can override the package or type namespaces at the property/field level. All attribute and element annotations accept the **namespace** parameter.

This can be defined in EclipseLink XML Bindings as follows:

This will produce the following XML:

```
<custom xmlns="http://www.example.org/package" xmlns:ns1="http://www.example.org/property"
  xmlns:ns0="http://www.example.org/type">
  <ns0:name>Bob</ns0:name>
  <ns1:account>1928712</ns1:account>
</custom>
```

Only the **account** element is qualified with the **http://www.example.org/property** namespace.

Example 3-10 Overriding the Namespace

```
package example;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(namespace="http://www.example.org/type")
public class Customer {
    private String name;

    @XmlElement(namespace="http://www.example.org/property")
    private String account;

    ...
}
```

Example 3-11 Sample Bindings File

```
<?xml version="1.0" encoding="UTF-8"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm">
  <xml-schema
    element-form-default="QUALIFIED"
    namespace="http://www.example.org/package">
  </xml-schema>

  <java-types>
    <java-type name="Customer">
      <xml-type namespace="http://www.example.org/type" />
      <java-attributes>
        <xml-element java-attribute="name" />
        <xml-element java-attribute="account" namespace="http://
www.example.org/property" />
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>
```

Specifying Inheritance

EclipseLink MOXy provides several ways to represent your inheritance hierarchy in XML:

- [Using xsi:type](#)
- [Using Substitution Groups](#)
- [Using @XmlDiscriminatorNode/@XmlDiscriminatorValue](#)

Using xsi:type

By default, EclipseLink will use the `xsi:type` attribute to represent inheritance in XML.

In this example an abstract super class (**ContactInfo**) contains all types of contact information. **Address** and **PhoneNumber** are the concrete implementations of **ContactInfo**.

Because the **Customer** object can have different types of contact information, its property refers to the superclass.

```
@XmlRootElement
public class Customer {

    private ContactInfo contactInfo;
    ...

}
```

Marshalling an example **Customer** would produce the following XML:

```
<customer>
  <contactInfo
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:type="address">
    <street>323 Main Street</street>
  </contactInfo>
</customer>
```

Note the `xsi:type` attribute on the `contactInfo` element.

Example 3-12 Sample Java Classes

```
public abstract class ContactInfo {
}

public class Address extends ContactInfo {

    private String street;
    ...

}

public class PhoneNumber extends ContactInfo {

    private String number;
    ...

}
```

```
}
```

Using Substitution Groups

Another way to model inheritance in XML is through XML Schema's substitution groups functionality. Using this approach, the element name *itself* determines which subclass to use.

Taking the same [Example 3-12](#), we will add `@XmlRootElement` annotations to each of the subclasses, which will act as the inheritance indicator.

Using this approach, marshalling an example **Customer** would produce the following XML:

```
<customer>
  <address>
    <street>323 Main Street</street>
  </address>
</customer>
```

Note that the **Address** object is marshalled to the address element.

Example 3-13 Using `@XmlRootElement` Annotation

```
public abstract class ContactInfo {
}

@XmlRootElement
public class Address extends ContactInfo {

    private String street;
    ...

}

@XmlRootElement
public class PhoneNumber extends ContactInfo {

    private String number;
    ...

}
```

Using `@XmlDiscriminatorNode`/`@XmlDiscriminatorValue`

You can also use the MOXY-specific `@XmlDiscriminatorNode` and `@XmlDiscriminatorValue` annotations (introduced in EclipseLink 2.2) to represent inheritance. With this approach, you can select the attribute to represent the subtype.

Using [Example 3-13](#), the **ContactInfo** class uses the `@XmlDiscriminatorNode` annotation to specify the XML attribute (classifier) that will hold the subclass indicator. **Address** and **PhoneNumber** are annotated with `@XmlDiscriminatorValue`, indicating that class' indicator name (address-classifier and phone-number-classifier).

Example 3-14 produces the following XML:

```
<customer>
  <contactInfo classifier="address-classifier">
    <street>323 Main Street</street>
  </contactInfo>
</customer>
```

Notice that **Address** is marshalled to the `contactInfo` element. Its classifier attribute contains the discriminator node value `address-classifier`.

Example 3-14 Using the @XmlDiscriminatorNode and @XmlDiscriminatorValue Annotations

```
@XmlDiscriminatorNode("@classifier")
public abstract class ContactInfo {
}

@XmlDiscriminatorValue("address-classifier")
public class Address extends ContactInfo {

    private String street;
    ...

}

@XmlDiscriminatorValue("phone-number-classifier")
public class PhoneNumber extends ContactInfo {

    private String number;
    ...

}
```

4

Mapping Simple Values

This chapter includes the following sections:

- [Mapping Simple Values](#)

Mapping Simple Values

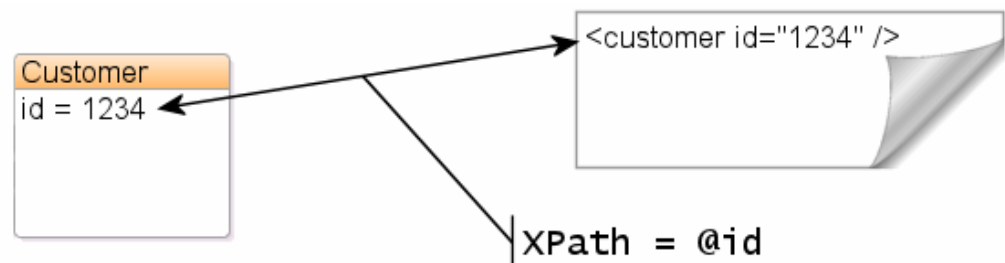
This section demonstrates several ways to map simple Java values directly to XML text nodes.

- [Mapping to an Attribute](#)
- [Mapping to a Text Node](#)
- [Mapping to a Specified Schema Type](#)
- [Mapping with a Simple Type Translator](#)

Mapping to an Attribute

Given the XML schema in [Example 4-1](#), [Figure 4-1](#) illustrates an XML direct mapping to an attribute in a corresponding XML document.

Figure 4-1 XML Direct Mapping to an Attribute



[Example 4-2](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. All that is needed is the standard JAXB `@XmlAttribute` annotation.

[Example 4-3](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-1 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>
```

```
<xsd:complexType name="customer-type">
  <xsd:attribute name="id" type="xsd:integer"/>
</xsd:complexType>

</xsd:schema>
```

Example 4-2 Using the @XmlAttribute Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlAttribute
    private Integer id;

    ...
}
```

Example 4-3 Defining the Mapping

```
...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-attribute java-attribute="id"/>
  </java-attributes>
</java-type>
...
```

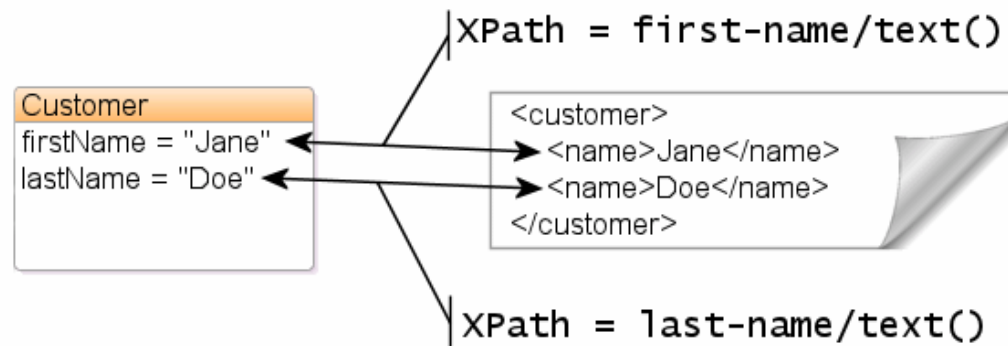
Mapping to a Text Node

This section describes using an XML direct mapping when doing the following:

- [Mapping to a Text Node in a Simple Sequence](#)
- [Mapping to a Text Node in a Sub-element](#)
- [Mapping to a Text Node by Position](#)
- [Mapping to a Simple Text Node](#)

Mapping to a Text Node in a Simple Sequence

Given the XML schema in [Example 4-4](#), [Figure 4-2](#) illustrates an XML direct mapping to individual text nodes in a sequence in a corresponding XML document.

Figure 4-2 XML Direct Mapping to Individual Text Nodes

Example 4-1 shows how to annotate your Java class to obtain this mapping with EclipseLink. Here, the standard JAXB `@XmlElement` annotation is used, with a customized element name.

In **Example 4-5**, we have explicitly specified the XML element names for the mapped attributes. This is an optional configuration – without an explicit name set, the XML element will simply match the Java attribute name; we would see `<firstName>Jane</firstName>` in XML. For more information on JAXB name-binding algorithms, see "Appendix D: Binding XML Names to Java Identifiers" of the Java Architecture for XML Binding (JAXB) Specification (<http://jcp.org/en/jsr/detail?id=222>).

Example 4-6 shows how to define your mapping information in EclipseLink's OXM metadata format. To specify a custom element name, the `name` attribute is used.

Example 4-4 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="first-name" type="xsd:string"/>
      <xsd:element name="last-name" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Example 4-5 Using the @XmlElement Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="first-name")
    private String firstName;
```

```

@XmlElement(name="last-name")
private String lastName;

...
}

```

Example 4-6 Using the name Attribute

```

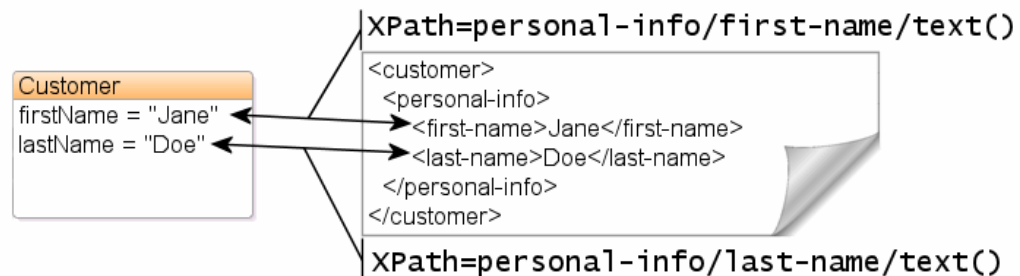
...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-element java-attribute="firstName" name="first-name"/>
    <xml-element java-attribute="lastName" name="last-name"/>
  </java-attributes>
</java-type>
...

```

Mapping to a Text Node in a Sub-element

Given the XML schema in [Example 4-7](#), [Figure 4-3](#) illustrates an XML direct mapping to a text node in a subelement in a corresponding XML document.

Figure 4-3 XML Direct Mapping to a Text Element in a Subnode



[Example 4-8](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. Here, because we are going beyond a simple element name customization and are actually introducing new XML structure, EclipseLink's `@XmlPath` annotation is used.

[Example 4-9](#) shows how to define your mapping information in EclipseLink's OXM metadata format. Here, the customized XML path is defined in the `xml-path` attribute.

Example 4-7 Sample XML Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="personal-info">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element name="first-name" type="xsd:string"/>

```

```

        <xsd:element name="last-name" type="xsd:string"/>
        <xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:sequence>
</xsd:complexType>

</xsd:schema>

```

Example 4-8 Using the @XmlPath Annotation

```

package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("personal-info/first-name/text()")
    private String firstName;

    @XmlPath("personal-info/last-name/text()")
    private String lastName;

    ...
}

```

Example 4-9 Using the xml-path Attribute

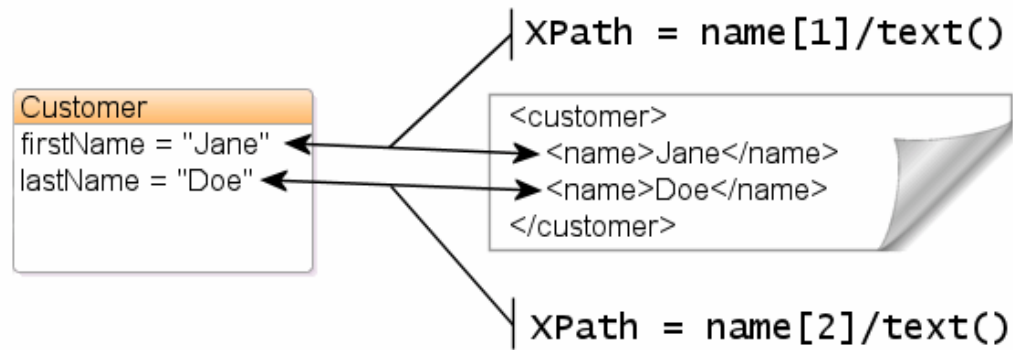
```

...
<java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
        <xml-element java-attribute="firstName" xml-path="personal-info/first-name/
text()"/>
        <xml-element java-attribute="lastName" xml-path="personal-info/last-name/
text()"/>
    </java-attributes>
</java-type>
...

```

Mapping to a Text Node by Position

Given the XML schema in [Example 4-10](#), [Figure 4-4](#) illustrates an XML direct mapping to a text node by position in a corresponding XML document.

Figure 4-4 XML Direct Mapping to a Text Node

[Example 4-11](#) shows how to configure this mapping in Java. Again, for more complex XML path customization, EclipseLink's `@XmlPath` annotation is used.

[Example 4-12](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-10 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="name" type="xsd:string" maxOccurs="2"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Example 4-11 Using the @XmlPath Annotation

```
package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("name[1]/text()")
    private String firstName;

    @XmlPath("name[2]/text()")
    private String lastName;

    ...
}
```

Example 4-12 Defining the Mapping

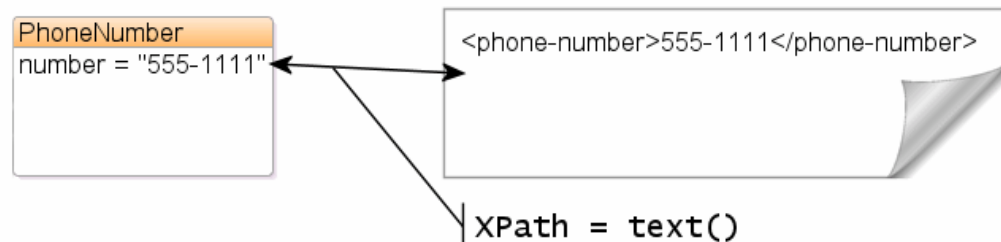
```

...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-element java-attribute="firstName" xml-path="name[1]/text()"/>
    <xml-element java-attribute="lastName" xml-path="name[2]/text()"/>
  </java-attributes>
</java-type>
...

```

Mapping to a Simple Text Node

Given the XML schema in [Example 4-13](#), [Figure 4-5](#) illustrates an XML direct mapping to a simple text node in a corresponding XML document.

Figure 4-5 XML Direct Mapping to a Simple Text Node

[Example 4-14](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. In this case, the `@XmlValue` annotation will be used.

[Example 4-15](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-13 Sample XML Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="phone-number" type="xsd:string"/>

</xsd:schema>

```

Example 4-14 Using the @XmlValue Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement(name="phone-number")
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
  @XmlValue
  private String number;
}

```

```
...
}
```

Example 4-15 Defining the Mapping

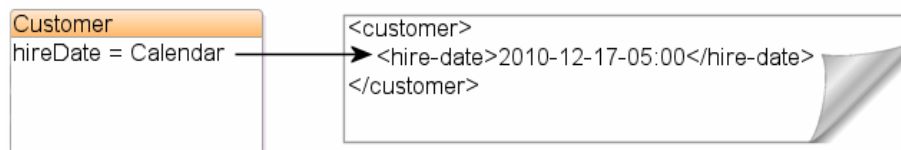
```
...
<java-type name="PhoneNumber">
  <xml-root-element name="phone-number"/>
  <java-attributes>
    <xml-value java-attribute="number"/>
  </java-attributes>
</java-type>
...
```

Mapping to a Specified Schema Type

In most cases, EclipseLink can determine the target format in the XML document. However, there are cases where you must specify which one of a number of possible targets EclipseLink should use. For example, a `java.util.Calendar` could be marshalled to a schema date, time, or dateTime node, or a `byte[]` could be marshalled to a schema `hexBinary` or `base64Binary` node.

The XML schema in [Example 4-16](#) and [Figure 4-6](#) an XML direct mapping from a **Calendar** object in Java to a **date** field in XML.

Figure 4-6 XML Direct Mapping to a Specific Schema Type



[Example 4-17](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. Here, the `@XmlSchemaType` is used to specify the datatype that will appear in the marshalled XML.

[Example 4-18](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-16 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="hire-date" type="xsd:date"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Example 4-17 Using the @XmlSchemaType Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="hire-date")
    @XmlSchemaType(name="date")
    private Calendar hireDate;

    ...
}

```

Example 4-18 Defining the Mapping

```

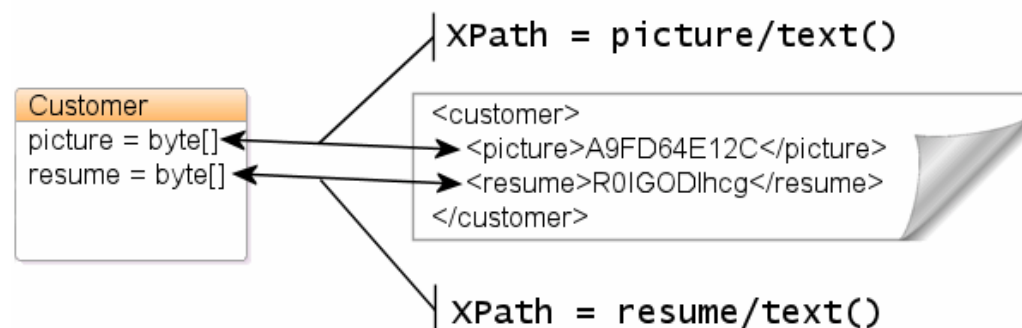
...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-element java-attribute="hireDate" name="hire-date">
      <xml-schema-type name="date"/>
    </xml-element>
  </java-attributes>
</java-type>
...

```

Using Java Type Adapters

[Example 4-19](#) and [Figure 4-7](#) illustrate XML direct mappings to two different text nodes of different binary types.

Figure 4-7 XML Direct Mappings to Different Text Nodes of Different Binary Types



[Example 4-20](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. By default, JAXB will marshal `byte[]` to `base64Binary`, so nothing special is needed for the resume mapping. To map to a **hexBinary** field, the `@XmlSchemaType` annotation specifies the XML type, while `@XmlJavaTypeAdapter` specifies the adapter class that will be responsible for converting the value (in this case, the built-in JAXB `HexBinaryAdapter`).

[Example 4-21](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-19 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="customer" type="customer-type"/>

    <xsd:complexType name="customer-type">
        <xsd:sequence>
            <xsd:element name="resume" type="xsd:base64Binary"/>
            <xsd:element name="picture" type="xsd:hexBinary"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Example 4-20 Using the @XmlSchemaType and @XmlJavaTypeAdapter Annotations

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    private byte[] resume;

    @XmlSchemaType(name="hexBinary")
    @XmlJavaTypeAdapter(HexBinaryAdapter.class)
    private byte[] picture;

    ...
}
```

Example 4-21 Defining the Mapping

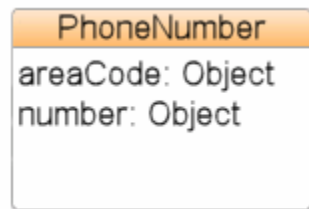
```
...
<java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
        <xml-element java-attribute="resume"/>
        <xml-element java-attribute="picture">
            <xml-schema-type name="hexBinary"/>
            <xml-java-type-adapter
value="javax.xml.bind.annotation.adapters.HexBinaryAdapter"/>
        </xml-element>
    </java-attributes>
</java-type>
...
```

Mapping with a Simple Type Translator

If the type of a node is not defined in your XML schema, you can configure an XML direct mapping to use the `xsi:type` attribute to provide type information.

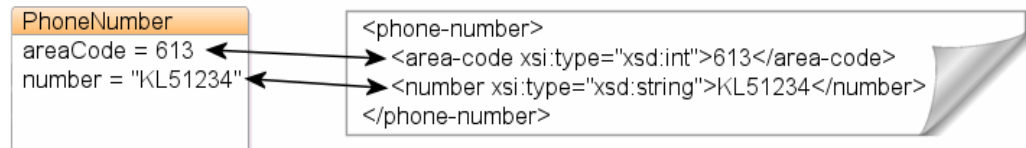
Given the XML schema in [Example 4-22](#), [Figure 4-8](#) illustrates a Java class that can be mapped to a corresponding XML document.

Figure 4-8 Sample Java Class



[Figure 4-9](#) illustrates an XML direct mapping with a simple type translator in an XML document that conforms to the schema above.

Figure 4-9 XML Direct Mapping with Simple Type Translator



[Example 4-23](#) shows how to annotate your Java class to obtain this mapping with EclipseLink.

[Example 4-24](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-22 Sample XML Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="phone-number" type="phone-number-type"/>

  <xsd:complexType name="phone-number-type">
    <xsd:sequence>
      <xsd:element name="area-code" type="anySimpleType"/>
      <xsd:element name="number" type="anySimpleType"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
  
```

Example 4-23 Sample Mapping

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement(name="phone-number")
public class PhoneNumber {
    @XmlElement(name="area-code")
    private Object areaCode;

    private Object number;

    ...
}
```

Example 4-24 Sample XML Mapping

```
...
<java-type name="PhoneNumber">
  <xml-root-element name="phone-number"/>
  <java-attributes>
    <xml-element java-attribute="areaCode" name="area-code"/>
    <xml-element java-attribute="number"/>
  </java-attributes>
</java-type>
...
```

Mapping Collections of Simple Values

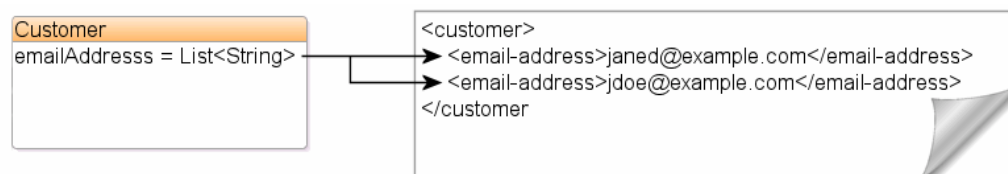
This section demonstrates several ways to map a collection of simple Java values directly to XML text nodes.

- [Mapping to Text Nodes](#)
- [Mapping to Text Nodes with a Grouping Element](#)
- [Mapping to a List Element](#)
- [Mapping a Collection of XmlAttributes or XmlValues](#)

Mapping to Text Nodes

Given the XML schema in [Example 4-25](#), [Figure 4-10](#) illustrates the mapping of a Java collection to elements in a corresponding XML document.

Figure 4-10 XML Direct Collection Mapping to Text Nodes



[Example 4-26](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. All that is needed is the standard JAXB `@XmlElement` annotation.

[Example 4-27](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-25 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="customer" type="customer-type"/>

    <xsd:complexType name="customer-type">
        <xsd:sequence>
            <xsd:element name="email-address" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

Example 4-26 Using the @XmlElement Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="email-address")
    private List<String> emailAddress;

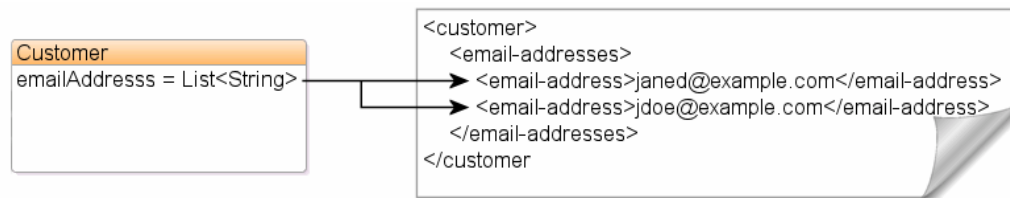
    ...
}
```

Example 4-27 Sample XML Mapping

```
...
<java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
        <xml-element java-attribute="emailAddresses" name="email-address"/>
    </java-attributes>
</java-type>
...
```

Mapping to Text Nodes with a Grouping Element

Given the XML schema in [Example 4-28](#), [Figure 4-11](#) illustrates the mapping of a Java collection to elements in a corresponding XML document, using a grouping element to organize the elements of the collection.

Figure 4-11 XML Direct Collection Mapping to Text Nodes with a Grouping Element

Example 4-29 shows how to annotate your Java class to obtain this mapping with EclipseLink. We specify the grouping element with the `@XmlElementWrapper` annotation.

Example 4-30 shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-28 Sample XML Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="email-address" type="xsd:string" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
  
```

Example 4-29 Using the `@XmlElementWrapper` Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="email-address")
    @XmlElementWrapper(name="email-addresses")
    private List<String> emailAddresses;

    ...
}
  
```

Example 4-30 Sample XML Mapping

```

...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-element java-attribute="emailAddresses" name="email-address">
  
```

```

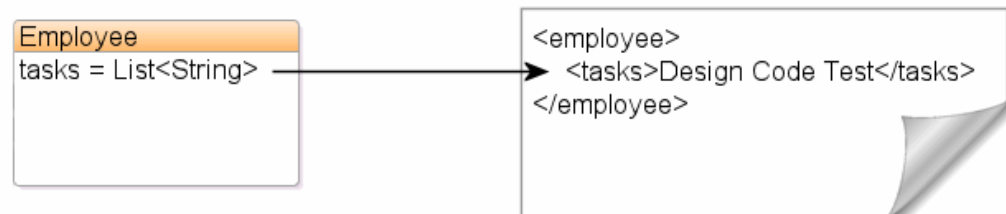
        <xml-element-wrapper name="email-addresses"/>
    </xml-element>
</java-attributes>
</java-type>
...

```

Mapping to a List Element

Given the XML schema in [Example 4-31](#), [Figure 4-12](#) illustrates a mapping to an `xsd:list` type in a corresponding XML document. Using this mapping, you can represent the collection of simple Java objects as a String of white space delimited tokens in XML. Here, the tasks list contains three entries: **Design**, **Code** and **Test**.

Figure 4-12 XML Direct Mapping to a List Field



[Example 4-32](#) shows how to annotate your Java class to obtain this mapping with EclipseLink.

[Example 4-33](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 4-31 Sample XML Schema

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="employee" type="employee-type"/>

    <xsd:complexType name="employee-type">
        <xsd:sequence>
            <xsd:element name="tasks" type="tasks-type"/>
        </xsd:sequence>
    </xsd:complexType>

    <xsd:simpleType name="tasks-type">
        <xsd:list itemType="xsd:string"/>
    </xsd:simpleType>

</xsd:schema>

```

Example 4-32 Using the @XmlList Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement

```

```

@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlList
    private List<String> tasks;

    ...
}

```

Example 4-33 Sample XML Mapping

```

...
<java-type name="Employee">
  <xml-root-element name="employee"/>
  <java-attributes>
    <xml-element java-attribute="tasks" xml-list="true"/>
  </java-attributes>
</java-type>
...

```

Mapping a Collection of XmlAttributes or XmlValues

`@XmlList` can also be used in conjunction with `@XmlAttribute` or `@XmlValue`, as shown in [Example 4-34](#). The collection will be represented as a space-separated string in the attribute.

Example 4-34 Java Annotations

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlAttribute
    @XmlList
    private List<Integer> ids;

    ...
}

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement(name="phone-numbers")
@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumbers {
    @XmlValue
    @XmlList
    private List<String> numbers;

    ...
}

```

Example 4-35 EclipseLink OXM Metadata

```

...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-attribute java-attribute="ids" xml-list="true"/>
  </java-attributes>
</java-type>
...

...
<java-type name="PhoneNumbers">
  <xml-root-element name="phone-numbers"/>
  <java-attributes>
    <xml-value java-attribute="numbers" xml-list="true"/>
  </java-attributes>
</java-type>
...

```

Example 4-36 Example XML Documents

```

<customer ids="726 1982 1989 2991"/>

<phone-numbers>6132883982 6139828817 18882982298</phone-numbers>

```

Multiple Mappings for a Single Property

Standard JAXB can have at most one mapping per Java field. Since EclipseLink MOXY 2.3, multiple mappings can be created for a single property using OXM metadata, with the caveat that at most one mapping will be readable (the rest will be "write-only").

Example

This example will use the following Java class:

[Example 4-38](#) shows how to define multiple mappings for the `currencyPairCode` in EclipseLink's OXM metadata format. Note that the second mapping has specified `write-only="true"`.

Example 4-37 Sample Java Class

```

package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class CustomQuoteRequest {

    private int requestId;

    private String currencyPairCode;

}

```

Example 4-38 Sample XML Schema

```

...
<java-type name="CustomQuoteRequest">
  <xml-root-element/>
  <java-attributes>
    <xml-element java-attribute="requestId" name="id"/>
    <xml-attribute java-attribute="currencyPairCode" xml-path="req/info/instrmt/
@sym"/>
    <xml-attribute java-attribute="currencyPairCode" xml-path="req/info/leg/token/
@sym" write-only="true"/>
  </java-attributes>
</java-type>
...

```

XML Output

[Example 4-39](#) shows an example CustomQuoteRequest marshalled to XML.

Example 4-39 Resulting XML

```

<?xml version="1.0" encoding="UTF-8"?>
<customQuoteRequest>
  <id>881</id>
  <req>
    <info>
      <instrmt sym="CAD/USD"/>
      <leg>
        <token sym="CAD/USD"/>
      </leg>
    </info>
  </req>
</customQuoteRequest>

```

Mapping Enums

Java enums can be mapped to XML using the `@XmlEnum` and `@XmlEnumValue` annotations.

Mapping Enums using Constant Names

The following schema demonstrates an XML enumeration:

If your enum constant names themselves are sufficient for the XML representation, you can simply use the `@XmlEnum` annotation:

[Example 4-42](#) shows how to define this mapping in EclipseLink OXM Metadata:

When marshalled to XML, a **Customer** would look like this:

```

<customer>
  <type>NEW_CUSTOMER</type>
</customer>

```

Example 4-40 Sample XML Schema

```
<xs:simpleType name="CustomerType">
  <xs:restriction base="xs:string"/>
    <xs:enumeration value="PROMO_CUSTOMER"/>
    <xs:enumeration value="NEW_CUSTOMER"/>
    <xs:enumeration value="VIP"/>
    <xs:enumeration value="NORMAL"/>
  </xs:restriction>
</xs:simpleType>
```

Example 4-41 Using the @XmlEnum Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    private CustomerType type = CustomerType.NEW_CUSTOMER;

    @XmlEnum(String.class)
    private enum CustomerType { PROMO_CUSTOMER, NEW_CUSTOMER, VIP, NORMAL }

    ...
}
```

Example 4-42 Sample XML Mapping

```
...
<xml-enums>
  <xml-enum java-enum="CustomerType" value="java.lang.String">
    <xml-enum-value java-enum-value="PROMO_CUSTOMER">PROMO_CUSTOMER</xml-enum-
value>
    <xml-enum-value java-enum-value="NEW_CUSTOMER">NEW_CUSTOMER</xml-enum-value>
    <xml-enum-value java-enum-value="VIP">VIP</xml-enum-value>
    <xml-enum-value java-enum-value="NORMAL">NORMAL</xml-enum-value>
  </xml-enum>
</xml-enums>

<java-types>
  <java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
      <xml-element java-attribute="type" type="CustomerType"/>
    </java-attributes>
  </java-type>
</java-types>
...
```

Mapping Enums to Custom XML Values

Given the following XML schema enumeration:

To model this in Java, the `@XmlEnumValue` annotation is used to provide an XML value for each enum constant:

[Example 4-45](#) shows how to define this mapping in EclipseLink OXM Metadata:

When marshalled to XML, a new **Customer** would look like this:

```
<customer>
  <type>2</type>
</customer>
```

Example 4-43 Sample XML Schema

```
<xs:simpleType name="CustomerType">
  <xs:restriction base="xs:int">
    <xs:enumeration value="1"/>
    <xs:enumeration value="2"/>
    <xs:enumeration value="3"/>
    <xs:enumeration value="4"/>
  </xs:restriction>
</xs:simpleType>
```

Example 4-44 Using the @XmlEnumValue Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    private CustomerType type = CustomerType.NEW_CUSTOMER;

    @XmlEnum(Integer.class)
    private enum CustomerType {
        @XmlEnumValue("1") PROMO_CUSTOMER,
        @XmlEnumValue("2") NEW_CUSTOMER,
        @XmlEnumValue("3") VIP,
        @XmlEnumValue("4") NORMAL;
        ...
    }
}
```

Example 4-45 Sample XML Mapping

```
...
<xml-enums>
  <xml-enum java-enum="CustomerType" value="java.lang.Integer">
    <xml-enum-value java-enum-value="PROMO_CUSTOMER">1</xml-enum-value>
    <xml-enum-value java-enum-value="NEW_CUSTOMER">2</xml-enum-value>
    <xml-enum-value java-enum-value="VIP">3</xml-enum-value>
    <xml-enum-value java-enum-value="NORMAL">4</xml-enum-value>
  </xml-enum>
</xml-enums>

<java-types>
  <java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
      <xml-element java-attribute="type" type="CustomerType"/>
    </java-attributes>
  </java-type>
```

```
</java-types>  
...
```

5

Mapping Special Schema Types

This chapter includes the following sections:

- [Mapping Dates and Times](#)
- [Mapping to a Union Field](#)
- [Binary Types](#)

Mapping Dates and Times

You can use the `@XmlSchemaType` annotation to customize the XML representation of date and time information. Additionally, EclipseLink MOXy supports the following types which are not covered in the JAXB specification (JSR-222):

- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`

The following XML schema contains a **date-of-birth** element of type `xsd:date`:

Example 5-1 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="date-of-birth" type="xsd:date" minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Understanding the Generated Model

The JAXB XML Schema to Java compiler (XJC) can be used to generate a class model from the sample schema. For example:

```
> xjc -d output-dir -p example date.xsd
```

will generate the following **Customer** class:

Notice that:

- The `dateOfBirth` property is of type `javax.xml.datatype.XMLGregorianCalendar`
- The `dateOfBirth` property uses the `@XmlSchemaType` annotation

Some Java data types (like **XMLGregorianCalendar**) have multiple XML representations (such as `xsd:date`, `xsd:time`, or **`xsd:dateTime`**). Use `@XmlSchemaType` to select the appropriate representation.

Example 5-2 Sample Customer Class

```
package example;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlSchemaType;
import javax.xml.bind.annotation.XmlType;
import javax.xml.datatype.XMLGregorianCalendar;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {"dateOfBirth"})
@XmlRootElement(name = "customer")
public class Customer {

    @XmlElement(name = "date-of-birth")
    @XmlSchemaType(name = "date")
    protected XMLGregorianCalendar dateOfBirth;

    public XMLGregorianCalendar getDateOfBirth() {
        return dateOfBirth;
    }

    public void setDateOfBirth(XMLGregorianCalendar value) {
        this.dateOfBirth = value;
    }
}
```

Using a Different Date (or Calendar) Property

By default, the JAXB XML schema to Java compiler (XJC) generates a property of type `XMLGregorianCalendar`. However, you can easily change this to `java.util.Date` or `java.util.Calendar`, as shown in [Example 5-3](#):

Example 5-3 Using `java.util.Date`

```
package blog.date;

import java.util.Date;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlSchemaType;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name = "customer")
public class Customer {

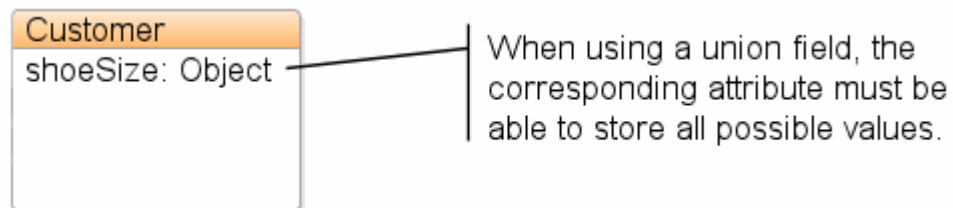
    @XmlElement(name = "date-of-birth")
    @XmlSchemaType(name = "date")
```

```
protected Date dateOfBirth;  
  
public Date getDateOfBirth() {  
    return dateOfBirth;  
}  
  
public void setDateOfBirth(Date value) {  
    this.dateOfBirth = value;  
}  
}
```

Mapping to a Union Field

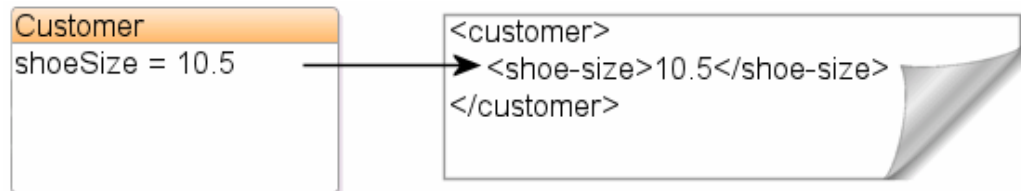
The following XML schema and class diagram show a typical use of an XML Schema Union:

Figure 5-1 Mapping to a Union Field

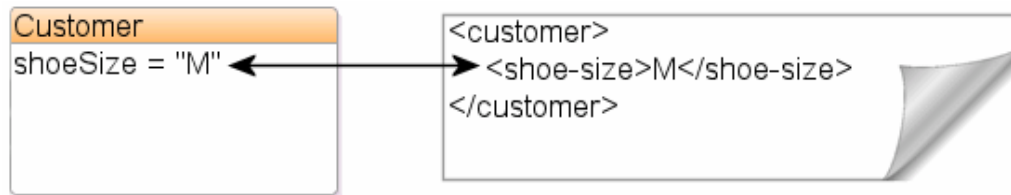


[Figure 5-2](#) illustrates a mapping to a union field in an XML document that conforms to the example schema. When EclipseLink unmarshalls the XML document, it tries each of the union types until it can make a successful conversion. The first schema type in the union is `xsd:decimal`. Because **10.5** is a valid decimal, EclipseLink converts the value to the appropriate type.

Figure 5-2 Mapping to a Union Field in an XML Document



In [Figure 5-3](#), the value **M** is *not* a valid `xsd:decimal` type, so the next union type is tried, `xsd:string`.

Figure 5-3 Mapping to a Union Field

Currently, EclipseLink does not support the mapping of Unions using Annotations or OXM Metadata. However, an EclipseLink XML Customizer can be used to create the mapping.

First, we annotate the shoeSize attribute with `@XmlTransient`, to avoid automatically generating a mapping for it. We also include an `@XmlCustomizer` annotation; the `CustomerCustomizer` class will create the Union mapping in code.

The `CustomerCustomizer` class can be used to manually add a mapping to the shoeSize attribute. In [Example 5-6](#), an `XMLUnionField` is configured on the mapping, and the possible Union member types are added by calling `addSchemaType()`:

Example 5-4 XML Schema Union

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="customer" type="customer-type" />
  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="shoe-size" type="size-type" />
    </xsd:sequence>
  </xsd:complexType>
  <xsd:simpleType name="size-type">
    <xsd:union memberTypes="xsd:decimal xsd:string" />
  </xsd:simpleType>
</xsd:schema>
```

Example 5-5 Using an EclipseLink Customizer

```
package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
@XmlCustomizer(CustomerCustomizer.class)
public class Customer {
    @XmlTransient
    private Object shoeSize;

    ...
}
```

Example 5-6 Mapping a Union Field

```

package example;

import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.xml.*;

public class CustomerCustomizer implements DescriptorCustomizer {

    @Override
    public void customize(ClassDescriptor descriptor) throws Exception {
        XMLDirectMapping shoeSizeMapping = new XMLDirectMapping();
        shoeSizeMapping.setAttributeName("shoeSize");

        XMLUnionField shoeSizeField = new XMLUnionField();
        shoeSizeField.setXPath("shoe-size/text()");
        shoeSizeField.addSchemaType(XMLConstants.DECIMAL_QNAME);
        shoeSizeField.addSchemaType(XMLConstants.STRING_QNAME);

        shoeSizeMapping.setField(shoeSizeField);

        descriptor.addMapping(shoeSizeMapping);
    }
}

```

Understanding Conversion Order

The order of the calls to `addSchemaType()` is important; when converting an XML value into Java, EclipseLink will attempt the conversions in the order that they were added to the field, and return as soon as a successful conversion is made. For example, when unmarshalling a shoeSize of **10.5**:

```

...
shoeSizeField.addSchemaType(XMLConstants.DECIMAL_QNAME);
shoeSizeField.addSchemaType(XMLConstants.STRING_QNAME);
...

```

A `BigDecimal` will be created to store the value. If, however, your `XMLUnionField` was set up like this:

```

...
shoeSizeField.addSchemaType(XMLConstants.STRING_QNAME);
shoeSizeField.addSchemaType(XMLConstants.DECIMAL_QNAME);
...

```

The shoeSize value will be a String ("**10.5**").

Customizing Conversion Classes

EclipseLink uses a set of default conversions to create a value for the Java attribute (in this case, `xsd:decimal` will be converted into a `BigDecimal`). You can override this behavior in Java code using the `XMLUnionField` method `addConversion`. For example, if you want your Java object to store shoeSize as a `Float`:

```
shoeSizeField.addConversion(XMLConstants.DECIMAL_QNAME, Float.class);
```

Binary Types

There are additional items to consider when mapping to binary type fields, such as `byte[]` or `Byte[]`.

Specifying Binary Formats Base64 and Hex

EclipseLink supports marshalling and unmarshalling binary data in two different representation formats: `base64Binary` (default) and `hexBinary`. You can specify the desired binary format using the `@XmlSchemaType` annotation, or `<xml-schema-type>` element in EclipseLink OXM. The examples below shows the result of marshalling the same `byte[]` to each of these formats.

```
BinaryData b = new BinaryData();
b.hexBytes = new byte[] {2,4,8,16,32,64};
b.base64Bytes = b.hexBytes;

jaxbContext.createMarshaller().marshal(b, System.out);
```

Example 5-7 Annotations

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
public class BinaryData {

    @XmlSchemaType(name="hexBinary")
    public byte[] hexBytes;

    @XmlSchemaType(name="base64Binary")
    public byte[] base64Bytes;

}
```

Example 5-8 EclipseLink OXM

```
...
<java-type name="example.BinaryData">
  <xml-root-element/>
  <java-attributes>
    <xml-element java-attribute="hexBytes">
      <xml-schema-type name="hexBinary"/>
    </xml-element>
    <xml-element java-attribute="base64Bytes">
      <xml-schema-type name="base64Binary"/>
    </xml-element>
  </java-attributes>
</java-type>
...
```

Example 5-9 Output

```
<?xml version="1.0" encoding="UTF-8"?>
<binaryData>
  <hexBytes>020308102040</hexBytes>
  <base64Bytes>AgMIECBA</base64Bytes>
</binaryData>
```

Understanding byte[] versus Byte[]

Unlike other Java primitive/wrapper types, EclipseLink differentiates between byte[] (primitive) and Byte[] (wrapper) data types. By default, byte[] will marshal to a single element or attribute, whereas Byte[] will marshal each byte as its own element, as illustrated by the following example:

```
BinaryData b = new BinaryData();
b.primitiveBytes = new byte[] {34,45,56,67,78,89,89,34,23,12,12,11,2};
b.byteObjects = new Byte[] {23,1,112,12,1,64,1,14,3,2};

jaxbContext.createMarshaller().marshal(b, System.out);
```

Example 5-10 Using byte[] and Byte[]

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
public class BinaryData {

    public byte[] primitiveBytes;
    public Byte[] byteObjects;

}
```

Example 5-11 Output

```
<?xml version="1.0" encoding="UTF-8"?>
<binaryData>
  <primitiveBytes>Ii04Q05ZWSIXDAwLAG==</primitiveBytes>
  <byteObjects>23</byteObjects>
  <byteObjects>1</byteObjects>
  <byteObjects>112</byteObjects>
  <byteObjects>12</byteObjects>
  <byteObjects>1</byteObjects>
  <byteObjects>64</byteObjects>
  <byteObjects>1</byteObjects>
  <byteObjects>14</byteObjects>
  <byteObjects>3</byteObjects>
  <byteObjects>2</byteObjects>
</binaryData>
```

Working with SOAP Attachments

If you are using EclipseLink MOXy in a Web Services environment, certain types of binary data may be created as an MTOM/XOP Attachment, instead of written directly into an XML element or attribute. This is done as an optimization for large amounts of binary data.

The following table shows the Java types that are automatically treated as Attachments, along with their corresponding MIME type:

Table 5-1 Java Attributes Treated as Attachments

Java Type	MIME Type
java.awt.Image	image/gif
java.awt.Image	image/jpeg
javax.xml.transform.Source	text/xml
application/xml	*
javax.activation.DataHandler	*/*



Note:

For more information on the basics of SOAP Attachments, see "Appendix H: Enhanced Binary Data Handling" of the Java Architecture for XML Binding (JAXB) Specification (<http://jcp.org/en/jsr/detail?id=222>).

The following Java class contains two binary fields: a simple `byte[]`, and a `java.awt.Image`. In a Web Services environment, the Image data will automatically be created as an attachment.

Marshalling the object in [Example 5-12](#) in a Web Services environment would look something like [Example 5-13](#) (the actual appearance will depend on your application server's implementation of AttachmentMarshaller):

Example 5-12 Sample Java Class

```
package example;

import java.awt.Image;

import javax.xml.bind.annotation.*;

@XmlRootElement
public class BinaryData {

    public byte[] bytes;

    public Image photo;

}
```

Example 5-13 Resulting XML

```
<?xml version="1.0" encoding="UTF-8"?>
<binaryData>
  <bytes>Ii04Q05ZWSIXDAwLAg==</bytes>
  <photo>
    <xop:Include href="cid:1" xmlns:xop="http://www.w3.org/2004/08/xop/include"/>
  </photo>
</binaryData>
```

Using @XmlInlineBinaryData

If you would like to force your binary data to be written as an inline string in your XML, you can annotate the field with the `@XmlInlineBinaryData` annotation:

This will result in an XML document like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<binaryData>
  <bytes>Ii04Q05ZWSIXDAwLAg==</bytes>
  <photo>/9j/4AAQSkZJRgABAgAAQAABAAD/2wBDAAgGBgcGBQgHB ... Af/2Q==</photo>
</binaryData>
```

Example 5-14 Using the @XmlInlineBinaryData Annotation

```
package example;

import java.awt.Image;

import javax.xml.bind.annotation.*;

@XmlRootElement
public class BinaryData {

    public byte[] bytes;

    @XmlInlineBinaryData
    public Image photo;

}
```

Using @XmlMimeType

You can explicitly set the MIME Type for an binary field using the `@XmlMimeType` annotation. Your application's `AttachmentMarshaller` and `AttachmentUnmarshaller` will be responsible for processing this information.

Example 5-15 Using the @XmlMimeType Annotation

```
package example;

import java.awt.Image;

import javax.xml.bind.annotation.*;

@XmlRootElement
```

```
public class BinaryData {  
    public byte[] bytes;  
  
    @XmlMimeType("image/gif")  
    public Image photo;  
}
```

6

Privately Owned Relationships

This chapter includes the following sections:

- [Mapping Privately Owned One-to-One Relationships](#)
- [Mapping Privately Owned One-to-Many Relationships](#)

Mapping Privately Owned One-to-One Relationships

This section demonstrates several ways to map a one-to-one relationship between objects. By default, one-to-one relationships are privately-owned in JAXB – their XML content will appear nested inside the owning element. For example, a **Customer** with a one-to-one mapping to a **PhoneNumber** would be marshalled as:

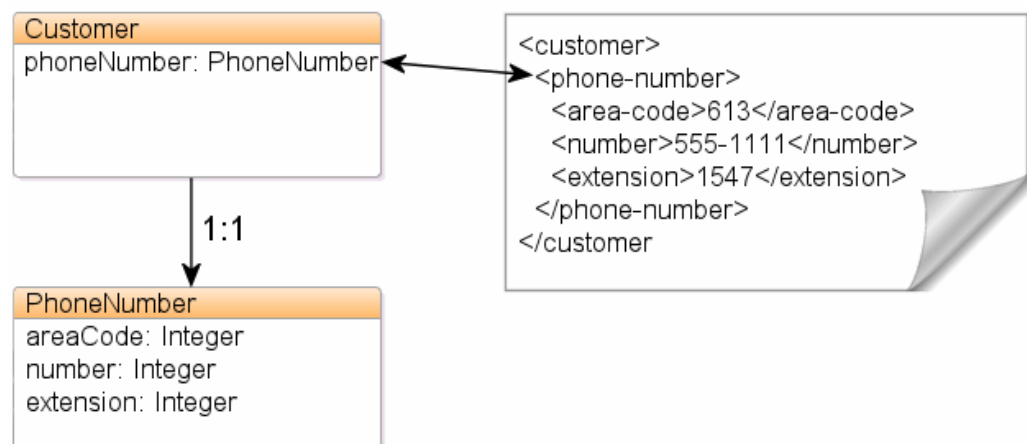
Example 6-1 Sample XML Mapping

```
<customer>
  <name>Bob Smith</name>
  <id>1982812</id>
  <phone-number>
    <area-code>613</area-code>
    <number>5550210</number>
    <extension>20016</extension>
  </phone-number>
</customer>
```

Mapping to an Element

Given the XML schema in [Example 6-2](#), [Figure 6-1](#) illustrates a one-to-one (1:1) relationship between two complex types.

Figure 6-1 One-to-one Relationship



Example 6-3 shows how to annotate your Java class to obtain this mapping with EclipseLink. The standard JAXB `@XmlElement` annotation can be used to indicate that the associated Java field should be mapped to an XML element.



Note:

By default, JAXB will assume all fields on your Java object are `@XmlElement`s, so in many cases the annotation itself is not required. If, however, you want to customize the Java field's XML name, you can specify an `@XmlElement` annotation with a name argument.

Example 6-4 shows how to define your mapping information in an EclipseLink's XML Bindings document.

Example 6-2 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:element name="customer" type="customer-type"/>

    <xsd:complexType name="customer-type">
        <xsd:element name="phone-number" type="phone-type"/>
    </xsd:complexType>

    <xsd:complexType name="phone-type">
        <xsd:element name="area-code" type="xsd:int"/>
        <xsd:element name="number" type="xsd:int"/>
        <xsd:element name="extension" type="xsd:int"/>
    </xsd:complexType>

</xsd:schema>
```

Example 6-3 Using the `@XmlElement` Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="phone-number")
    private PhoneNumber phoneNumber;

    ...
}

package example;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
    @XmlElement(name="area-code")
    private Integer areaCode;
```

```

        private Integer number;

        private Integer extension;

        ...
    }

```

Example 6-4 Sample XML Mapping

```

...
<java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
        <xml-element java-attribute="phoneNumber" name="phone-number"
type="PhoneNumber"/>
    </java-attributes>
</java-type>

<java-type name="PhoneNumber">
    <java-attributes>
        <xml-value java-attribute="areaCode" name="area-code"
type="java.lang.Integer"/>
        <xml-value java-attribute="number" type="java.lang.Integer"/>
        <xml-value java-attribute="extension" type="java.lang.Integer"/>
    </java-attributes>
</java-type>
...

```

Using EclipseLink's @XmlPath Annotation

By default, your Java attributes will be mapped to XML based on their attributes Java name, or by a name specified in an `@XmlElement` annotation. This mapping is based on XPath, and EclipseLink's `@XmlPath` annotation allows you to customize this mapping. For example, you can use it to control the nesting of your elements in XML:

This will produce the following XML:

```

<customer>
    <contact-info>
        <phone-number>
            <number>555-631-2124</number>
        </phone-number>
    </contact-info>
</customer>

```

You can also use `@XmlPath` to map to different occurrences of the same element in XML, by index. For example:

will produce the following XML:

```

<customer>
    <contact-info>
        <phone>
            <number>555-631-2124</number>
        </phone>
        <phone>
            <number>555-631-8298</number>
        </phone>
    </contact-info>
</customer>

```

```

        </phone>
    </contact-info>
</customer>

```

For information on using XPath in your mappings, see [Mapping Using XPath Predicates](#).

Example 6-5 Using the @XmlPath Annotation

```

package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("contact-info/phone-number")
    private PhoneNumber phoneNumber;

    ...
}

```

Example 6-6 Using EclipseLink XML Bindings

```

...
<java-type name="Customer">
    <xml-root-element name="customer"/>
    <java-attributes>
        <xml-element java-attribute="phoneNumber" name="phone-number"
            type="PhoneNumber" xml-path="contact-info/phone-number"/>
    </java-attributes>
</java-type>
...

```

Example 6-7 Using the @XmlPath Annotation

```

package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("contact-info/phone[1]")
    private PhoneNumber homePhone;
    @XmlPath("contact-info/phone[2]")
    private PhoneNumber workPhone;

    ...
}

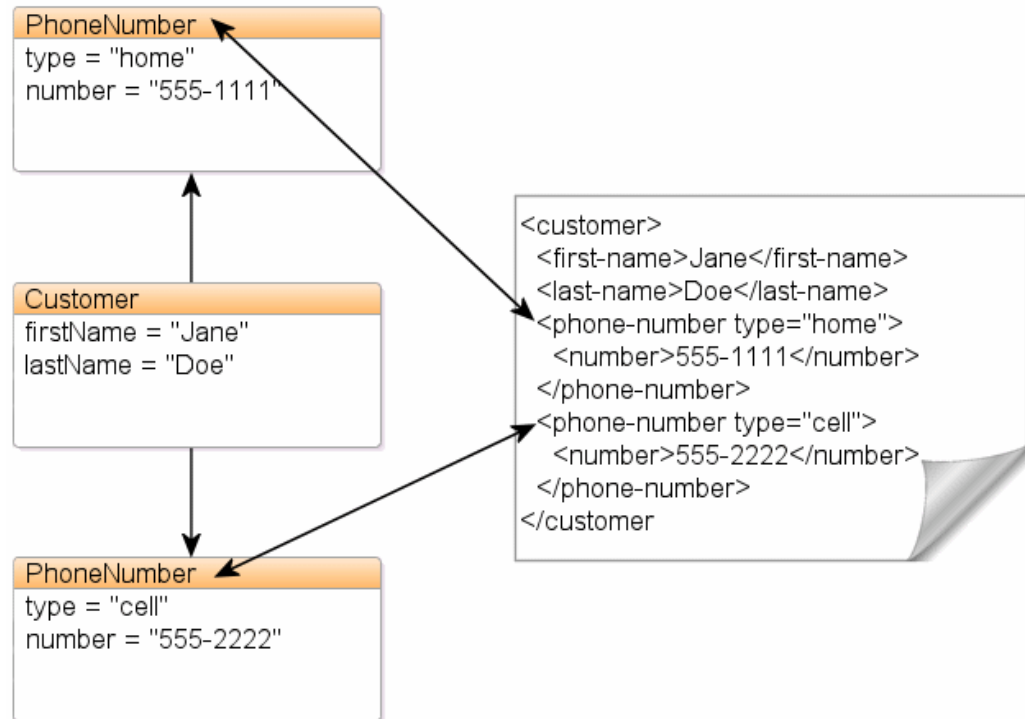
```

Mapping Privately Owned One-to-Many Relationships

This section illustrates how to map one-to-many relationships with EclipseLink.

The schema in [Example 6-8](#) a typical one-to-many (1:M) relationship between Customer and PhoneNumber, as shown in [Figure 6-2](#).

Figure 6-2 One-to-many Relationship



Example 6-8 Sample XML Mapping

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:element name="customer" type="customer-type"/>

  <xsd:complexType name="customer-type">
    <xsd:sequence>
      <xsd:element name="first-name" type="xsd:string"/>
      <xsd:element name="last-name" type="xsd:string"/>
      <xsd:element name="phone-number" type="phone-type" minOccurs="0"
maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>

  <xsd:complexType name="phone-type">
    <xsd:sequence>
      <xsd:attribute name="type" type="xsd:string"/>
      <xsd:element name="number" type="xsd:int"/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Mapping to Elements

[Example 6-9](#) shows how to annotate your Java class to obtain this mapping with EclipseLink. The standard JAXB `@XmlElement` annotation, when used on a Collection or array field, can achieve this.

[Example 6-10](#) shows how to define your mapping information in EclipseLink's OXM metadata format.

Example 6-9 Using the `@XmlElement` Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name="first-name")
    private String firstName;
    @XmlElement(name="last-name")
    private String lastName;
    @XmlElement(name="phone-number")
    private List<PhoneNumber> phoneNumbers;

    ...
}

package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
    @XmlAttribute
    private String type;
    private Integer number;

    ...
}
```

Example 6-10 Sample XML Mapping

```
...
<java-type name="Customer">
  <xml-root-element name="customer"/>
  <java-attributes>
    <xml-element java-attribute="firstName" name="first-name"
type="java.lang.String"/>
    <xml-element java-attribute="lastName" name="last-name"
type="java.lang.String"/>
    <xml-element java-attribute="phoneNumbers" name="phone-number"
type="PhoneNumber" container-type="java.util.ArrayList"/>
  </java-attributes>
</java-type>

<java-type name="PhoneNumber">
  <java-attributes>
```

```
        <xml-attribute java-attribute="type" type="java.lang.String"/>
        <xml-value java-attribute="number" type="java.lang.Integer"/>
    </java-attributes>
</java-type>
...
```

Grouping Elements using the @XmlElementWrapper Annotation

To make the elements of the Collection appear inside a grouping element, you can use `@XmlElementWrapper`:

This will produce the following XML:

```
<customer>
  <first-name>Bob</first-name>
  <last-name>Smith</last-name>
  <phone-numbers>
    <phone-number type="Home">
      <number>5559827222</number>
    </phone-number>
    <phone-number type="Work">
      <number>5558872216</number>
    </phone-number>
  </phone-numbers>
</customer>
```

Example 6-11 Using the @XmlElementWrapper Annotation

```
package example;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlElement(name = "phone-number")
    @XmlElementWrapper(name="phone-numbers")
    private List<PhoneNumber> phoneNumbers;

    ...
}
```

7

Mapping Shared Reference Relationships

This chapter includes the following sections:

- [Understanding Keys and Foreign Keys](#)
- [Mapping Single Key Relationships](#)
- [Mapping Composite Key Relationships](#)
- [Mapping Bidirectional Relationships](#)

Understanding Keys and Foreign Keys

EclipseLink supports shared reference keys and foreign keys through:

- Single key
- Composite Key
- Embedded Key Class

Mapping Single Key Relationships

To model non-privately-owned relationships, your "target" objects must have IDs (keys) defined, and your "source" object must use these IDs to map the relationship.

Relationships represented with keys use the `@XmlID` and `@XmlIDREF` annotations. Although the JAXB specification requires that the property marked with `@XmlID` be a String, MOXy JAXB does not enforce this restriction.

In [Example 7-1](#), each **Employee** has one **manager** but multiple **reports**.

The following example shows how to define this mapping information in EclipseLink's OXM metadata format.

This would produce the following XML:

```
<company>
  <employee id="1" name="Jane Doe">
    <report>2</report>
    <report>3</report>
  </employee>
  <employee id="2" name="John Smith">
    <manager>1</manager>
  </employee>
  <employee id="3" name="Anne Jones">
    <manager>1</manager>
  </employee>
</company>
```

The **manager** and **reports** elements contain the IDs of the **Employee** instances they are referencing.

Example 7-1 Using the @XmlID and @XmlIDREF Annotations

```

package example;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlAttribute
    @XmlID
    private Integer id;

    @XmlAttribute
    private String name;

    @XmlIDREF
    private Employee manager;

    @XmlElement(name="report")
    @XmlIDREF
    private List<Employee> reports;

    ...
}

```

Example 7-2 Sample XML Mapping

```

...
<java-type name="Employee">
  <java-attributes>
    <xml-attribute java-attribute="id" type="java.lang.Integer" xml-id="true"/>
    <xml-attribute java-attribute="name" type="java.lang.String"/>
    <xml-element java-attribute="manager" type="mypackage.Employee" xml-
idref="true"/>
    <xml-element java-attribute="reports" type="mypackage.Employee" container-
type="java.util.ArrayList" xml-idref="true"/>
  </java-attributes>
</java-type>
...

```

Using @XmlList

Because the @XmlIDREF annotation is also compatible with the @XmlList annotation, the **Employee** object could be modeled as:

This would produce the following XML:

```

<company>
  <employee id="1" name="Jane Doe">
    <reports>2 3</reports>
  </employee>
  <employee id="2" name="John Smith">
    <manager>1</manager>
  </employee>
  <employee id="3" name="Anne Jones">
    <manager>1</manager>
  </employee>
</company>

```

Example 7-3 Using the @XmlList Annotation

```

package example;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlID
    @XmlAttribute
    private Integer id;

    @XmlAttribute
    private String name;

    @XmlIDREF
    private Employee manager;

    @XmlIDREF
    @XmlList
    private List<Employee> reports;

    ...
}

```

Using the Embedded Key Class

With JAXB, you can derive an XML representation from a set of JPA entities, when a JPA entity has an embedded ID class.

In [Example 7-4](#), the `EmployeeId` is the embedded ID of the **Employee** class:

For the JAXB bindings, the XML accessor type will be set to **FIELD** for all the model classes. This can be set as a package level JAXB annotation, as shown here:

```

@XmlAccessorType(XmlAccessType.FIELD)
package com.example.model;

import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;

```

[Example 7-5](#) uses the EclipseLink extension `@XmlCustomizer` which extends the JAXB specification. Because the contact attribute is a bidirectional relationship, it includes the EclipseLink extension `@XmlInverseReference`.

To embed the content of the **EmployeeId** class in the complex type corresponding to the **Employee** class, change the XPath on the mapping for the `id` property to be *self* or `..`. Then specify the XPath to the XML nodes which represent the ID.

If the target object had a single ID then we would use `@XmlIDREF`. Since the target object has a compound key, we will mark the field `@XmlTransient`, and use the EclipseLink extension `@XmlCustomizer` to set up the mapping.

An `XMLObjectReferenceMapping` will be created. The mapping will include multiple key mappings.

```

import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;

```

```
import org.eclipse.persistence.oxm.mappings.XMLObjectReferenceMapping;

public class PhoneNumberCustomizer implements DescriptorCustomizer {

    public void customize(ClassDescriptor descriptor) throws Exception {
        XMLObjectReferenceMapping contactMapping = new XMLObjectReferenceMapping();
        contactMapping.setAttributeName("contact");
        contactMapping.setReferenceClass(Employee.class);
        contactMapping.addSourceToTargetKeyFieldAssociation("contact/@eID", "eID/
text()");
        contactMapping.addSourceToTargetKeyFieldAssociation("contact/@country",
"country/text()");
        descriptor.addMapping(contactMapping);
    }

}
```

Example 7-4 Sample Embedded ID

```
@Entity
public class PhoneNumber {

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
        @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
    })
    private Employee contact;

}

@Entity
@IdClass(EmployeeId.class)
public class Employee {

    @EmbeddedId
    private EmployeeId id;

    @OneToMany(mappedBy="contact")
    private List<PhoneNumber> contactNumber;

}

@Embeddable
public class EmployeeId {

    @Column(name="E_ID")
    private BigDecimal eId;

    private String country;

}
```

Example 7-5 Using the @XmlCustomizer Annotation

```
@Entity
@IdClass(EmployeeId.class)
@XmlCustomizer(EmployeeCustomizer.class)
public class Employee {
```

```

    @EmbeddedId
    private EmployeeId id;

    @OneToMany(mappedBy="contact")
    @XmlInverseReference(mappedBy="contact")
    private List<PhoneNumber> contactNumber;
}

```

Example 7-6 Changing the XPath

```

import org.eclipse.persistence.config.DescriptorCustomizer;
import org.eclipse.persistence.descriptors.ClassDescriptor;
import org.eclipse.persistence.oxm.mappings.XMLCompositeObjectMapping;

public class EmployeeCustomizer implements DescriptorCustomizer {

    public void customize(ClassDescriptor descriptor) throws Exception {
        XMLCompositeObjectMapping idMapping =
            (XMLCompositeObjectMapping) descriptor.getMappingForAttributeName("id");
        idMapping.setXPath(".");

        descriptor.addPrimaryKeyFieldName("eId/text()");
        descriptor.addPrimaryKeyFieldName("country/text()");
    }
}

```

Example 7-7 Using the @XmlTransient Annotation

```

@Entity
@XmlCustomizer(PhoneNumberCustomizer.class)
public class PhoneNumber {

    @ManyToOne
    @JoinColumns({
        @JoinColumn(name="E_ID", referencedColumnName = "E_ID"),
        @JoinColumn(name="E_COUNTRY", referencedColumnName = "COUNTRY")
    })
    @XmlTransient
    private Employee contact;
}

```

Mapping Composite Key Relationships

If the objects that you want to map have multi-part keys (that is, a combination of fields that determines uniqueness), you can use EclipseLink's `@XmlKey` and `@XmlJoinNodes` to set up this relationship.

One or more `@XmlKey` annotations can be used to declare the primary keys in a given class. For a single key, either `@XmlID` or `@XmlKey` can be used. For composite primary keys, multiple `@XmlKey` annotations can be used, or a single `@XmlID` can be combined with one or more `@XmlKey` annotations.

**Note:**

Composite Keys can be useful when using JAXB to map JPA entities. For more information see [Converting JPA entities to/from XML \(via JAXB\)](#).

In [Example 7-8](#), each **Employee** has one **manager** but multiple **reports**, and **Employees** are uniquely identified by the combination of their **id** and **name** fields.

[Example 7-9](#) shows how to define this mapping information in EclipseLink's OXM metadata format.

This would produce the following XML:

```
<company>
  <employee id="1" name="Jane Doe">
    <report id="2" name="John Smith"/>
    <report id="3" name="Anne Jones"/>
  </employee>
  <employee id="2" name="John Smith">
    <manager id="1" name="Jane Doe"/>
  </employee>
  <employee id="3" name="Anne Jones">
    <manager id="1" name="Jane Doe"/>
  </employee>
</company>
```

Example 7-8 Using the @XmlKey and @XmlJoinNodes Annotations

```
package example;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.oxm.annotations.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    @XmlID
    @XmlAttribute
    private Integer id;

    @XmlKey
    @XmlAttribute
    private String name;

    @XmlJoinNodes( {
        @XmlJoinNode(xmlPath = "manager/@id", referencedXmlPath = "@id"),
        @XmlJoinNode(xmlPath = "manager/@name", referencedXmlPath = "@name") })
    public Employee manager;

    @XmlJoinNodes( {
        @XmlJoinNode(xmlPath = "report/@id", referencedXmlPath = "@id"),
        @XmlJoinNode(xmlPath = "report/@name", referencedXmlPath = "@name") })
    public List<Employee> reports = new ArrayList<Employee>();

    ...
}
```

Example 7-9 Sample XML Mapping

```

...
<java-type name="Employee">
  <java-attributes>
    <xml-attribute java-attribute="id" xml-id="true" />
    <xml-attribute java-attribute="name" xml-key="true" />
    <xml-join-nodes java-attribute="manager">
      <xml-join-node xml-path="manager/@id" referenced-xml-path="@id" />
      <xml-join-node xml-path="manager/@name" referenced-xml-path="@name" />
    </xml-join-nodes>
    <xml-join-nodes java-attribute="reports" container-
type="java.util.ArrayList">
      <xml-join-node xml-path="report/@id" referenced-xml-path="@id" />
      <xml-join-node xml-path="report/@name" referenced-xml-path="@name" />
    </xml-join-nodes>
  </java-attributes>
</java-type>
...

```

Mapping Bidirectional Relationships

In order to map bidirectional relationships in EclipseLink MOXy, the back-pointer must be annotated as an `@XmlInverseReference`. Without this annotation, the cyclic relationship will result in an infinite loop during marshalling.

`@XmlInverseReferences` must specify the `mappedBy` attribute, which indicates the property on the opposite side of the relationship.

In [Example 7-11](#), an `Employee` has a collection of `PhoneNumbers`, and each `PhoneNumber` has a back-pointer back to its `Employee`:

[Example 7-11](#) shows how to define this mapping in EclipseLink's OXM metadata format:

```

Employee emp = new Employee();
emp.setName("Bob Smith");

PhoneNumber p = new PhoneNumber();
p.setNumber("555-1212");

emp.getPhones().add(p);

// Not Necessary
// p.setEmployee(emp);

```

`@XmlInverseReference` back-pointers can be used with the following types of mappings:

- One-To-One Relationships (see ["Mapping Privately Owned One-to-One Relationships"](#))
- One-To-Many Relationships (see ["Mapping Privately Owned One-to-Many Relationships"](#))
- Single Key Relationships (see ["Mapping Single Key Relationships"](#))
- Composite Key Relationships (see ["Mapping Composite Key Relationships"](#))

@XmlInverseReference can be particularly useful when mapping JPA entities to XML (see [Using XML Bindings](#)).

Example 7-10 Using the @XmlInverseReference Annotation

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Employee {
    private String name;
    private List<PhoneNumber> phones = new ArrayList<PhoneNumber>();
    ...
}

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {
    private String number;
    @XmlInverseReference(mappedBy="phones")
    private Employee employee;
    ...
}
```

Example 7-11 Sample XML Mapping

```
...
<java-type name="Employee">
  <java-attributes>
    <xml-element java-attribute="name" type="java.lang.String"/>
    <xml-element java-attribute="phones" type="PhoneNumber" container-
type="java.util.ArrayList"/>
  </java-attributes>
</java-type>

<java-type name="PhoneNumber">
  <java-attributes>
    <xml-element java-attribute="number" type="java.lang.String"/>
    <xml-inverse-reference java-attribute="employee" type="Employee" mapped-
by="phones" />
  </java-attributes>
</java-type>
...
```

In addition, when using @XmlInverseReference, it is not necessary to explicitly set the back-pointer in your Java code; EclipseLink will do this for you automatically:

See also

For more information, see:

- Binding JPA Relationships to XML <http://wiki.eclipse.org/EclipseLink/Examples/MOxy/JPA/Relationships>

8

Advanced Concepts

This chapter includes the following sections:

- [Refreshing Metadata](#)
- [Customizing XML Name Conversions](#)
- [Using Virtual Access Methods](#)
- [Using Extensible MOXy](#)
- [Mapping Using XPath Predicates](#)
- [Using an XmlAdapter](#)
- [Using XML Transformations](#)
- [Generating Java Classes from an XML Schema](#)
- [Customizing Generated Mappings](#)

Refreshing Metadata

Introduced in EclipseLink MOXy 2.3, you can refresh the JAXBContext metadata at runtime. This allows you to make changes to existing mappings in a live application environment and see those changes immediately without having to create a new JAXBContext.

In order to use the **Metadata Refresh** feature, your metadata information must be provided in one of the following formats:

- `javax.xml.transform.Source`
- `org.w3c.dom.Node`
- `org.eclipse.persistence.jaxb.metadata.MetadataSource`

The JAXBContext is created in the standard way:

```
...
ClassLoader classLoader = ClassLoader.getSystemClassLoader();

DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
InputStream metadataStream = classLoader.getResourceAsStream("example/eclipselink-oxm.xml");
Document metadataDocument = db.parse(metadataStream);
metadataStream.close();

Map<String, Object> props = new HashMap<String, Object>(1);
props.put(JAXBContextProperties.OXM_METADATA_SOURCE, metadataDocument);
JAXBContext context = JAXBContextFactory.createContext(new Class[] { Root.class },
props);
```

...

At this point, if we were to marshal a `Root` object to XML, it would look like this:

```
<root>
  <orig-name>RootName</orig-name>
</root>
```

For this example, we will modify the metadata Document directly to change the XML name for the name field. We can then refresh the metadata using the `refreshMetadata()` API:

```
...
Element xmlElementElement = (Element)
metadataDocument.getElementsByTagNameNS("http://www.eclipse.org/eclipselink/xsds/
persistence/oxm", "xml-element").item(0);
xmlElementElement.setAttribute("name", "new-name");
JAXBHelper.getJAXBContext(jc).refreshMetadata();
...
```

After refreshing metadata, the same `Root` object will be marshalled as follows:

```
<root>
  <new-name>RootName</new-name>
</root>
```

Example 8-1 Refreshing Metadata

This example will be bootstrapped from the following EclipseLink OXM file:

```
<?xml version="1.0"?>
<xml-bindings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
  package-name="example">
  <java-types>
    <java-type name="Root">
      <java-attributes>
        <xml-element java-attribute="name" name="orig-name"/>
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>
```

Customizing XML Name Conversions

JAXB has well-established rules for converting Java names to XML names, which can be overridden through the use of annotations. This can become burdensome if your names follow common rules (such as making everything upper-case). Starting with EclipseLink MOXy 2.3, you can override this default naming algorithm.

This example will create an implementation of `XMLNameTransformer` to provide a naming algorithm to MOXy.

Using the XMLNameTransformer

The XMLNameTransformer interface defines several methods for customizing name generation:

- transformElementName – called when creating an element from a Java field or method
- transformAttributeName – called when creating an attribute from a Java field or method
- transformTypeName – called when creating a simple type or complex type from a Java class
- transformRootElementName – called when creating a (root) simple type or complex type from a Java class

Example 8-2 defines an XMLNameTransformer that does the following:

- Root element will be the unqualified Java class name
- Other types will be named (unqualified Java class name) + "Type"
- Camel-case element names will be converted to lower-case, hyphenated names
- XML attributes will appear in all upper-case

Example 8-2 Using an XMLNameTransformer

```
package example;

public class NameGenerator implements org.eclipse.persistence.oxm.XMLNameTransformer
{
    // Use the unqualified class name as our root element name.
    public String transformRootElementName(String name) {
        return name.substring(name.lastIndexOf('.') + 1);
    }

    // The same algorithm as root element name plus "Type" appended to the end.
    public String transformTypeName(String name) {
        return transformRootElementName(name) + "Type";
    }

    // The name will be lower-case with word breaks represented by '-'.
    // Note: A capital letter in the original name represents the start of a new
    word.
    public String transformElementName(String name) {
        StringBuilder strBldr = new StringBuilder();
        for (char character : name.toCharArray()) {
            if (Character.isUpperCase(character)) {
                strBldr.append('-');
                strBldr.append(Character.toLowerCase(character));
            } else {
                strBldr.append(character);
            }
        }
        return strBldr.toString();
    }

    // The original name converted to upper-case.
    public String transformAttributeName(String name) {
```

```

        return name.toUpperCase();
    }
}

```

Example Model

The domain model in [Example 8-3](#) will be used. To save space, the accessors have been omitted.

Example 8-3 Customer

```

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlType(propOrder={"fullName", "shippingAddress"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlAttribute
    private long id;

    private String fullName;

    private Address shippingAddress;

}

```

Example 8-4 Address.java

```

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    @XmlAttribute
    private String type;

    private String street;

}

```

Specifying the Naming Algorithm

Our implementation of the naming algorithm can be provided via the `@XmlNameTransformer` annotation (package or type level) or via the external bindings file in XML.

1. At the type level:

```

@XmlNameTransformer(example.NameGenerator.class)
public class Customer

```

2. At the package level (package-info.java):

```

@XmlNameTransformer(example.NameGenerator.class)
package example;

```

3. External bindings file:

```
<?xml version='1.0' encoding='UTF-8'?>
<xml-bindings xmlns='http://www.eclipse.org/eclipselink/xsds/persistence/oxm'
xml-name-transformer='example.NameGenerator'>
  <xml-schema/>
  <java-types/>
</xml-bindings>
```

XML Output

Without any customization, JAXB's default naming algorithm will produce XML that looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<customer id="123">
  <fullName>Jane Doe</fullName>
  <shippingAddress type="residential">
    <street>1 Any Street</street>
  </shippingAddress>
</customer>
```

By leveraging our customized naming algorithm we can get the following output without specifying any additional metadata on our domain classes:

```
<?xml version="1.0" encoding="UTF-8"?>
<Customer ID="123">
  <full-name>Jane Doe</full-name>
  <shipping-address TYPE="residential">
    <street>1 Any Street</street>
  </shipping-address>
</Customer>
```

Using Virtual Access Methods

In addition to standard JAXB properties (represented by Java fields and accessor methods), EclipseLink MOXy 2.3 introduced the concept of virtual properties and virtual access methods, which instead rely on special `get()` and `set()` methods to maintain mapping data. For example, you might want to use a `HashMap` as the underlying structure to hold data for certain mappings. The mappings that use virtual method access must be defined in EclipseLink OXM metadata.

In order to add virtual properties to an entity:

- the Java class must be marked with an `@XmlVirtualAccessMethods` annotation, or `<xml-virtual-access-methods>` element in OXM
- the Java class must contain getter and setter methods to access virtual property values:
 - `public <ValueType> get(String propertyName)`
 - `public void set(String propertyName, <ValueType> value)`
 - * method names are configurable
 - * `<ValueType>` can be `Object`, or any other Java type (if you would like to use a particular type of value class in the method signature)

**Note:**

By default, EclipseLink will look for methods named `set` and `get`. To customize accessor method names, see ["Specifying Alternate Accessor Methods"](#).

For an example of using virtual properties in a multi-tenant architecture, see ["Using Extensible MOXy"](#).

Configuring Virtual Access Methods

Virtual Access Methods can be configured either by through Java annotations (see [Example 8-5](#)) or EclipseLink OXM metadata (see [Example 8-6](#)).

Example 8-5 Using Java Annotations

```
package example;

import java.util.Map;
import java.util.HashMap;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlVirtualAccessMethods;

@XmlRootElement
@XmlVirtualAccessMethods
@XmlAccessorType(XmlAccessType.PROPERTY)
public class Customer {

    private int id;

    private String name;

    private Map<String, Object> extensions = new HashMap<String, Object>();

    public Object get(String name) {
        return extensions.get(name);
    }

    public void set(String name, Object value) {
        extensions.put(name, value);
    }

    @XmlAttribute
    public int getId() {
        ...
    }
}
```

Example 8-6 Using OXM Metadata

```
...
<java-types>
  <java-type name="Customer">
    <xml-virtual-access-methods />
  </java-type>
</java-types>
```

```

    <java-attributes>
      <xml-attribute java-attribute="id" />
      <xml-element java-attribute="name" />
    </java-attributes>
  </java-type>
...

```

Example

For this example we will use the **Customer** class ([Example 8-3](#)), along with an EclipseLink OXM file to define our virtual mappings. Any property encountered in this file that does not have a corresponding Java attribute will be considered a virtual property and will be accessed through the virtual access methods. Because there is no associated Java field, the type information must also be provided.

When creating the JAXBContext, we pass in the virtualprops metadata along with our **Customer** class.

To set the values for virtual properties, we will use the aforementioned `set()` method.

```

InputStream oxm = classLoader.getResourceAsStream("virtualprops-oxm.xml");
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, oxm);

Class[] classes = new Class[] { Customer.class };
JAXBContext ctx = JAXBContext.newInstance(classes, properties);

Customer c = new Customer();
c.setId(7761);
c.setName("Bob Smith");
c.set("discountCode", "SIUB372JS7G2IUDS7");

ctx.createMarshaller().marshal(c, System.out);

```

This will produce the following XML:

```

<customer id="7761">
  <name>Bob Smith</name>
  <discount-code>SIUB372JS7G2IUDS7</discount-code>
</customer>

```

Conversely, we use the `get(String)` method to access virtual properties:

```

...
Customer c = (Customer) ctx.createUnmarshaller().unmarshal(CUSTOMER_URL);

// Populate UI
customerWindow.getTextField(ID).setText(String.valueOf(c.getId()));
customerWindow.getTextField(NAME).setText(c.getName());
customerWindow.getTextField(DCODE).setText(c.get("discountCode"));
...

```

Example 8-7 The virtualprops-oxm.xml File

```

...
<java-types>
  <java-type name="Customer">

```

```

        <java-attributes>
            <xml-element java-attribute="discountCode" name="discount-code"
                type="java.lang.String" />
        </java-attributes>
    </java-type>
</java-types>
...

```

Using XmlAccessType.FIELD and XmlTransient

If you are using an `@XmlAccessorType` of `XmlAccessType.FIELD`, you will need to mark your virtual properties Map attribute to be `@XmlTransient`, to prevent the Map itself from being bound to XML:

Example 8-8 Marking the Map Attribute

```

package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlVirtualAccessMethods;

@XmlRootElement
@XmlVirtualAccessMethods
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlTransient
    private Map<String, Object> extensions;
    ...
}

```

Options

This section includes the following topics:

- [Specifying Alternate Accessor Methods](#)
- [Specifying Schema Generation Options](#)

Specifying Alternate Accessor Methods

To use different method names as your virtual method accessors, specify them using the `getMethod` and `setMethod` attributes on `@XmlVirtualAccessMethods`:

In OXM:

Example 8-9 Using Alternate Accessor Methods

```

package example;

import java.util.Properties;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlVirtualAccessMethods;

@XmlRootElement
@XmlVirtualAccessMethods(getMethod = "getCustomProps", setMethod = "putCustomProps")

```

```

@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlAttribute
    private int id;

    private String name;

    @XmlTransient
    private Properties<String, Object> props = new Properties<String, Object>();

    public Object getCustomProps(String name) {
        return props.getProperty(name);
    }

    public void putCustomProps(String name, Object value) {
        props.setProperty(name, value);
    }
}

```

Example 8-10 Using the xml-virtual-access-methods Element

```

...
<java-types>
  <java-type name="Customer">
    <xml-virtual-access-methods get-method="getCustomProps" set-
method="putCustomProps" />
    <java-attributes>
      <xml-attribute java-attribute="id" />
      <xml-element java-attribute="name" />
      <!-- virtual -->
      <xml-element java-attribute="discountCode" name="discount-code"
type="java.lang.String" />
    </java-attributes>
  </java-type>
...

```

Specifying Schema Generation Options

You can configure how virtual properties should appear in generated schemas using the schema attribute on `@XmlVirtualAccessMethods`. EclipseLink offers two options. Virtual properties can be:

- written as individual nodes, or
- consolidated into a single `<any>` element.

Virtual Properties as Individual Nodes

This is EclipseLink's default behavior, or can be specified explicitly as an override as follows:

For example:

Example 8-11 Mapping as Individual Nodes

```
package example;
```

```

@XmlRootElement
@XmlVirtualAccessMethods(schema = XmlVirtualAccessMethodsSchema.NODES)
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    ...

```

Example 8-12 Original Customer Schema

```

<xs:schema ...>

    <xs:element name="customer">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="first-name" type="xs:string" />
                <xs:element name="last-name" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>

```

Example 8-13 Generated Schema (After adding middle-initial and phone-number)

```

<xs:schema ...>

    <xs:element name="customer">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="first-name" type="xs:string" />
                <xs:element name="last-name" type="xs:string" />
                <xs:element name="middle-initial" type="xs:string" />
                <xs:element name="phone-number" type="xs:string" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>

```

Virtual Properties in an <any> Element

EclipseLink can also use an <any> element to hold all of the virtual properties in one node:

From [Example 8-14](#), a newly generated schema using this approach would look like:

Example 8-14 Using an <any> Element

```

package example;

@XmlRootElement
@XmlVirtualAccessMethods(schema = XmlVirtualAccessMethodsSchema.ANY)
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    ...

```

Example 8-15 Generated Schema

```

<xs:schema ...>

  <xs:element name="customer">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first-name" type="xs:string" />
        <xs:element name="last-name" type="xs:string" />
        <xs:any minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>

```

Using Extensible MOXy

In a multi-tenant architecture, a single application runs on a server serving multiple client organizations (tenants). Good multi-tenant applications allow per-tenant customizations. When these customizations are made to data, it can be difficult for the binding layer to handle them.

JAXB is designed to work with domain models that have *real* fields and properties. EclipseLink MOXy virtual properties provide a way to extend a class without modifying the source.

Using the @XmlVirtualAccessMethods Annotation

The @XmlVirtualAccessMethods annotation is used to specify that a class is extensible. An extensible class is required to have a get method that returns a value by property name, and a set method that stores a value by property name. The default names for these methods are get and set, and can be overridden with the @XmlVirtualAccessMethods annotation.

Since we will have multiple extensible classes in this example, we'll configure a base class for this behavior that extensible classes can extend. We will use the @XmlTransient annotation to prevent ExtensibleBase from being mapped as an inheritance relationship. The *real* properties represent the parts of the model that will be common to all tenants. The per-tenant extensions will be represented as *virtual* properties.

Example 8-16 Sample ExtensibleBase

```

package examples.virtual;

import java.util.HashMap;
import java.util.Map;

import javax.xml.bind.annotation.XmlTransient;
import org.eclipse.persistence.xml.annotations.XmlVirtualAccessMethods;

@XmlTransient
@XmlVirtualAccessMethods(setMethod="put")
public class ExtensibleBase {

    private Map<String, Object> extensions = new HashMap<String, Object>();

```

```
    public <T> T get(String property) {  
        return (T) extensions.get(property);  
    }  
  
    public void put(String property, Object value) {  
        extensions.put(property, value);  
    }  
}
```

Example 8-17 Customer

The **Customer** class will be extensible since it inherits from a domain class that has been annotated with `@XmlVirtualAccessMethods`.

```
package examples.virtual;  
  
import javax.xml.bind.annotation.XmlRootElement;  
  
@XmlRootElement  
public class Customer extends ExtensibleBase {  
  
    private String firstName;  
    private String lastName;  
    private Address billingAddress;  
  
    public String getFirstName() {  
        return firstName;  
    }  
  
    public void setFirstName(String firstName) {  
        this.firstName = firstName;  
    }  
  
    public String getLastName() {  
        return lastName;  
    }  
  
    public void setLastName(String lastName) {  
        this.lastName = lastName;  
    }  
  
    public Address getBillingAddress() {  
        return billingAddress;  
    }  
  
    public void setBillingAddress(Address billingAddress) {  
        this.billingAddress = billingAddress;  
    }  
}
```

Example 8-18 Address

It is not necessary to have every class in your model be extensible. In this example the **Address** class will not have any virtual properties.

```
package examples.virtual;  
  
public class Address {
```

```
private String street;

public String getStreet() {
    return street;
}

public void setStreet(String street) {
    this.street = street;
}
}
```

Example 8-19 PhoneNumber

Like **Customer**, **PhoneNumber** will be an extensible class.

```
package examples.virtual;

import javax.xml.bind.annotation.XmlValue;

public class PhoneNumber extends ExtensibleBase {

    private String number;

    @XmlValue
    public String getNumber() {
        return number;
    }

    public void setNumber(String number) {
        this.number = number;
    }
}
```

Creating Tenant 1

The first tenant is an online sporting goods store that requires the following extensions to the model:

- Customer ID
- Customer's middle name
- Shipping address
- A collection of contact phone numbers
- Type of phone number (i.e. home, work, or cell)

The metadata for the virtual properties is supplied through MOXy's XML mapping file. Virtual properties are mapped in the same way as real properties. Some additional information is required including type (since this cannot be determined via reflection), and for collection properties a container type.

The virtual properties defined in [Example 8-20](#) for **Customer** are: **middleName**, **shippingAddress**, and **phoneNumbers**. For **PhoneNumber** the virtual property is the type property.

The get/set methods are used on the domain model to interact with the *real* properties and the accessors defined on the @XmlVirtualAccessMethods annotation are used to interact with the *virtual* properties. The normal JAXB mechanisms are used for marshal and unmarshal operations:

```
Customer customer = new Customer();

//Set Customer's real properties
customer.setFirstName("Jane");
customer.setLastName("Doe");

Address billingAddress = new Address();
billingAddress.setStreet("1 Billing Street");
customer.setBillingAddress(billingAddress);

//Set Customer's virtual 'middleName' property
customer.put("middleName", "Anne");

//Set Customer's virtual 'shippingAddress' property
Address shippingAddress = new Address();
shippingAddress.setStreet("2 Shipping Road");
customer.put("shippingAddress", shippingAddress);

List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();
customer.put("phoneNumbers", phoneNumbers);

PhoneNumber workPhoneNumber = new PhoneNumber();
workPhoneNumber.setNumber("555-WORK");
//Set the PhoneNumber's virtual 'type' property
workPhoneNumber.put("type", "WORK");
phoneNumbers.add(workPhoneNumber);

PhoneNumber homePhoneNumber = new PhoneNumber();
homePhoneNumber.setNumber("555-HOME");
//Set the PhoneNumber's virtual 'type' property
homePhoneNumber.put("type", "HOME");
phoneNumbers.add(homePhoneNumber);

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, "examples/virtual/binding-tenant1.xml");
JAXBContext jc = JAXBContext.newInstance(new Class[] {Customer.class,
Address.class}, properties);

Marshaller marshaller = jc.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.out);
```

Example 8-20 binding-tenant1.xml

```
<?xml version="1.0"?>
<xml-bindings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
  package-name="examples.virtual">
  <java-types>
    <java-type name="Customer">
      <xml-type prop-order="firstName middleName lastName billingAddress
shippingAddress phoneNumbers"/>
    <java-attributes>
      <xml-attribute
```

```

        java-attribute="id"
        type="java.lang.Integer"/>
    <xml-element
        java-attribute="middleName"
        type="java.lang.String"/>
    <xml-element
        java-attribute="shippingAddress"
        type="examples.virtual.Address"/>
    <xml-element
        java-attribute="phoneNumbers"
        name="phoneNumber"
        type="examples.virtual.PhoneNumber"
        container-type="java.util.List"/>
    </java-attributes>
</java-type>
<java-type name="PhoneNumber">
    <java-attributes>
        <xml-attribute
            java-attribute="type"
            type="java.lang.String"/>
        </java-attributes>
    </java-type>
</java-types>
</xml-bindings>

```

Example 8-21 Output

```

<?xml version="1.0" encoding="UTF-8"?>
<customer>
    <firstName>Jane</firstName>
    <middleName>Anne</middleName>
    <lastName>Doe</lastName>
    <billingAddress>
        <street>1 Billing Street</street>
    </billingAddress>
    <shippingAddress>
        <street>2 Shipping Road</street>
    </shippingAddress>
    <phoneNumber type="WORK">555-WORK</phoneNumber>
    <phoneNumber type="HOME">555-HOME</phoneNumber>
</customer>

```

Creating Tenant 2

The second tenant is a streaming media provider that offers on-demand movies and music to its subscribers. It requires a different set of extensions to the core model: a single contact phone number

For this tenant we will also leverage the mapping file to customize the mapping of the real properties, as shown in [Example 8-22](#):

```

Customer customer = new Customer();
customer.setFirstName("Jane");
customer.setLastName("Doe");

Address billingAddress = new Address();
billingAddress.setStreet("1 Billing Street");
customer.setBillingAddress(billingAddress);

```

```

PhoneNumber phoneNumber = new PhoneNumber();
phoneNumber.setNumber("555-WORK");
customer.put("phoneNumber", phoneNumber);

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, "examples/virtual/binding-tenant2.xml");
JAXBContext jc = JAXBContext.newInstance(new Class[] {Customer.class, Address.class}, properties);

Marshaller marshaller = jc.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
marshaller.marshal(customer, System.out);

```

Example 8-22 binding-tenant2.xml

```

<?xml version="1.0"?>
<xml-bindings
  xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
  package-name="examples.virtual">
  <xml-schema namespace="urn:tenant1" element-form-default="QUALIFIED"/>
  <java-types>
    <java-type name="Customer">
      <xml-type prop-order="firstName lastName billingAddress phoneNumber"/>
      <java-attributes>
        <xml-attribute java-attribute="firstName"/>
        <xml-attribute java-attribute="lastName"/>
        <xml-element java-attribute="billingAddress" name="address"/>
        <xml-element
          java-attribute="phoneNumber"
          type="examples.virtual.PhoneNumber"/>
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>

```

Example 8-23 Output

Note that even though both tenants share several real properties, the corresponding XML representation can be quite different due to virtual properties:

```

<?xml version="1.0" encoding="UTF-8"?>
<customer xmlns="urn:tenant1" firstName="Jane" lastName="Doe">
  <address>
    <street>1 Billing Street</street>
  </address>
  <phoneNumber>555-WORK</phoneNumber>
</customer>

```

Mapping Using XPath Predicates

By default, JAXB will use the Java field name as the XML element name:

Or, the XML name can be customized using the name attribute of the `@XmlElement` annotation:

However, sometimes elements need to be mapped based on their position in the document, or based on an attribute value of an element:

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
  <name>Jane</name>
  <name>Doe</name>
  <node name="address">
    <node name="street">123 A Street</node>
  </node>
  <node name="phone-number" type="work">555-1111</node>
  <node name="phone-number" type="cell">555-2222</node>
</node>
```

For cases like this, EclipseLink MOXy allows you to use XPath predicates to define an expression that will specify the XML element's name.

Example 8-24 Sample Java Code and XML Schema

```
public class Customer {
    @XmlElement
    private String firstName;
    @XmlElement
    private String lastName;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <firstName>Bob</firstName>
  <lastName>Roberts</lastName>
</customer>
```

Example 8-25 Sample Java Code and XML Schema

```
public class Customer {
    @XmlElement(name="f-name")
    private String firstName;
    @XmlElement(name="l-name")
    private String lastName;
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <f-name>Bob</f-name>
  <l-name>Roberts</l-name>
</customer>
```

Mapping with XPath Predicates

An XPath predicate represents an expression that will be evaluated against the element specified. For example, the XPath statement:

```
node[2]
```

Would match the second occurrence of the node element ("DEF"):

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <node>ABC</node>
  <node>DEF</node>
</data>
```

Predicates can also match based on an attribute value:

```
node[@name='foo']
```

Would match the **node** element with the attribute **name="foo"** (that is, **ABC**). It would not match the node that contains **DEF**.

```
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <node name="foo">ABC</node>
  <node name="bar">DEF</node>
</data>
```



Note:

For more information on XPath Predicates, see "2.4 Predicates" of the XML Path Language (XPath) specification (<http://www.w3.org/TR/xpath>).

Mapping Based on Position

In the following example, our XML contains two **name** elements; the first occurrence of **name** should represent the **Customer's** first name, and the second **name** will be their last name. To map this, we will specify XPath expressions for each property that will match the appropriate XML element. Note that we also use `@XmlType(propOrder)` to ensure that our elements will always be in the proper positions.

This same configuration can be expressed in an EclipseLink XML Bindings document as follows:

This will give us the desired XML representation:

Example 8-26 Using the `@XmlType(propOrder)` Annotation

```
package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlRootElement
@XmlType(propOrder={"firstName", "lastName"})
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
    @XmlPath("name[1]/text()")
    private String firstName;

    @XmlPath("name[2]/text()")
    private String lastName;
```

```
    ...
}
```

Example 8-27 Using the prop-order Attribute

```
...
<java-type name="Customer">
  <xml-root-element/>
  <xml-type prop-order="firstName lastName"/>
  <java-attributes>
    <xml-element java-attribute="firstName" xml-path="name[1]/text()"/>
    <xml-element java-attribute="lastName" xml-path="name[2]/text()"/>
  </java-attributes>
</java-type>
...
```

Example 8-28 Resulting XML

```
<?xml version="1.0" encoding="UTF-8"?>
<customer>
  <name>Bob</name>
  <name>Smith</name>
</customer>
```

Mapping Based on an Attribute Value

Since EclipseLink MOXy 2.3, you can also map to an XML element based on an Attribute value. In this example, all of our XML elements are named **node**, differentiated by the value of their **name** attribute:

We can use an XPath in the form of `element-name[@attribute-name='value']` to map each Java field:

Example 8-29 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
  <node name="first-name">Bob</node>
  <node name="last-name">Smith</node>
  <node name="address">
    <node name="street">123 A Street</node>
  </node>
  <node name="phone-number" type="work">555-1111</node>
  <node name="phone-number" type="cell">555-2222</node>
</node>
```

Example 8-30 Sample Mappings

```
package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlRootElement(name="node")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {
```

```

    @XPath("node[@name='first-name']/text()")
    private String firstName;

    @XPath("node[@name='last-name']/text()")
    private String lastName;

    @XPath("node[@name='address']")
    private Address address;

    @XPath("node[@name='phone-number']")
    private List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();

    ...
}

package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XPath;

@XmlAccessorType(XmlAccessType.FIELD)
public class Address {

    @XPath("node[@name='street']/text()")
    private String street;

    ...
}

package example;

import javax.xml.bind.annotation.*;

@XmlAccessorType(XmlAccessType.FIELD)
public class PhoneNumber {

    @XmlAttribute
    private String type;

    @XmlValue
    private String number;

    ...
}

```

Creating "Self" Mappings

EclipseLink allows you to configure your one-to-one mappings so the data from the target object will appear inside the source object's XML element. Expanding on the previous example, we could map the **Address** information so that it would appear directly under the **customer** element, and *not* wrapped in its own element. This is referred to as a "self" mapping, and is achieved by setting the target object's XPath to `.` (dot).

[Example 8-31](#) demonstrates a self mapping declared in annotations.

Using a self mapping, EclipseLink produces the desired XML. The **street** data is stored in the root **node**.

```
<?xml version="1.0" encoding="UTF-8"?>
<node>
  <node name="first-name">Bob</node>
  <node name="last-name">Smith</node>
  <node name="street">123 A Street</node>
  <node name="phone-number" type="work">555-1111</node>
  <node name="phone-number" type="cell">555-2222</node>
</node>
```

Example 8-31 Self Mapping Example

```
package example;

import javax.xml.bind.annotation.*;

import org.eclipse.persistence.oxm.annotations.XmlPath;

@XmlRootElement(name="node")
@XmlAccessorType(XmlAccessType.FIELD)
public class Customer {

    @XmlPath("node[@name='first-name']/text()")
    private String firstName;

    @XmlPath("node[@name='last-name']/text()")
    private String lastName;

    @XmlPath(".")
    private Address address;

    @XmlPath("node[@name='phone-number']")
    private List<PhoneNumber> phoneNumbers = new ArrayList<PhoneNumber>();

    ...
}
```

Using an XmlAdapter

Some Java classes may not be well suited for use with JAXB and at first glance may seem "unmappable" – for example, classes that do not have a default no-arg constructor, or classes for which an XML representation cannot be automatically determined. Using JAXB's `XmlAdapter`, you can define custom code to convert the unmappable class into something that JAXB can handle. Then, you can use the `@XmlJavaTypeAdapter` annotation to indicate that your adapter should be used when working with the unmappable class.

`XmlAdapter` uses the following terminology:

- **ValueType** – The type that JAXB knows how to handle out of the box.
- **BoundType** – The type that JAXB doesn't know how to handle. An adapter is written to allow this type to be used as an in-memory representation through the **ValueType**.

The outline of an `XmlAdapter` class is as follows:

Example 8-32 XmlAdapter Class Outline

```

public class AdapterName extends XmlAdapter<ValueType, BoundType> {

    public BoundType unmarshal(ValueType value) throws Exception {
        ...
    }

    public ValueType marshal(BoundType value) throws Exception {
        ...
    }
}

```

Using java.util.Currency

Our first example will use the following domain class:

Here, the **Currency** cannot be automatically mapped with JAXB because it does not contain a no-argument constructor. However, we can write an adapter that will convert the **Currency** into something that JAXB does know how to handle – a simple `String`. Luckily, in this case the **Currency's** `toString()` method returns the currency code, which can also be used to create a new **Currency**:

To indicate that our adapter should be used for the **Currency** property, we annotate it with `@XmlJavaTypeAdapter` and provide the class name of our adapter:

Example 8-33 Sample Domain Class

```

package example;

import java.util.Currency;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class PurchaseOrder {

    private Double amount;

    private Currency currency;

    ...
}

```

Example 8-34 Using an Adapter

```

package example;

import java.util.Currency;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class CurrencyAdapter extends XmlAdapter<String, Currency> {

    /*
     * Java => XML
     * Given the unmappable Java object, return the desired XML representation.
     */
}

```

```

    */
    public String marshal(Currency val) throws Exception {
        return val.toString();
    }

    /*
     * XML => Java
     * Given an XML string, use it to build an instance of the unmappable class.
     */
    public Currency unmarshal(String val) throws Exception {
        return Currency.getInstance(val);
    }
}

```

Example 8-35 Using the @XmlJavaTypeAdapter Annotation

```

package example;

import java.util.Currency;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class PurchaseOrder {

    private Double amount;

    @XmlJavaTypeAdapter(CurrencyAdapter.class)
    private Currency currency;

    ...
}

```

Using java.awt.Point

Sometimes the best way to handle an unmappable class is to write a "stand-in" class which *can* be mapped with JAXB, and convert between the two classes in the XmlAdapter. In this example, we want to use the **Point** class. Because of that class' getLocation() method (which JAXB will pickup automatically and map), an infinite loop will occur during marshalling. Because we cannot change the **Point** class, we will write a new class, **MyPoint**, and use it in the adapter.

Finally, our **Point** properties are marked with @XmlJavaTypeAdapter:

Example 8-36 Using java.awt.Point

```

package example;

public class MyPoint {

    private int x, y;

    public MyPoint() {
        this(0, 0);
    }
}

```

```
    public MyPoint(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() {
        return x;
    }

    ...
}

package example;

import java.awt.Point;

import javax.xml.bind.annotation.adapters.XmlAdapter;

public class MyPointAdapter extends XmlAdapter<MyPoint, Point> {

    /*
     * Java => XML
     */
    public MyPoint marshal(Point val) throws Exception {
        return new MyPoint((int) val.getX(), (int) val.getY());
    }

    /*
     * XML => Java
     */
    public Point unmarshal(MyPoint val) throws Exception {
        return new Point(val.getX(), val.getY());
    }
}
```

Example 8-37 Using the @XmlJavaTypeAdapter Annotation

```
package example;

import java.awt.Point;

import javax.xml.bind.annotation.*;

@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Zone {

    private String name;

    @XmlJavaTypeAdapter(MyPointAdapter.class)
    private Point startCoord;

    @XmlJavaTypeAdapter(MyPointAdapter.class)
    private Point endCoord;

    ...
}
```

Specifying Package-Level Adapters

In [Example 8-37](#), we annotated both **Point** properties with the `@XmlJavaTypeAdapter` annotation. If you have many of these types of properties – for example, in other domain classes – it can be more convenient to specify the `@XmlJavaTypeAdapters` at the package level.

We could define both of the adapter classes in **package-info.java**, and would no longer have to annotate any further **Currency** or **Point** properties:

```
@XmlJavaTypeAdapters({
    @XmlJavaTypeAdapter(value=CurrencyAdapter.class, type=Currency.class),
    @XmlJavaTypeAdapter(value=MyPointAdapter.class, type=Point.class)
})
package example;
```

Specifying Class-Level @XmlJavaTypeAdapters

If you have a Java class and you would like to always use an `XmlAdapter` during marshalling and unmarshalling, then you can specify the `@XmlJavaTypeAdapter` directly at the class level:

```
package example;

import javax.xml.bind.annotation.adapters.XmlJavaTypeAdapter;

@XmlJavaTypeAdapter(DataStructureAdapter.class)
public class DataStructure {

    ...

}
```

Now, any object that has a **DataStructure** property will automatically use the `DataStructureAdapter`, without the need for an annotation on the property itself.

Using XML Transformations

In many cases, you can use MOXy's `@XmlTransformation` annotation to give you considerably more control over the marshalling and unmarshalling of your objects. `@XmlTransformation` can be used to create a custom mapping where one or more XML nodes can be used to create the value for the Java attribute.

To handle the custom requirements at marshal (write) and unmarshal (read) time, `@XmlTransformation` uses instances of `org.eclipse.persistence.mappings.transformers` (such as `AttributeTransformer` and `FieldTransformer`), providing a non-intrusive solution that avoids the need for domain objects to implement any 'special' interfaces.

For example, if you wanted to map the following XML to objects and combine the values of `DATE` and `TIME` into a single `java.util.Date` object, you can use an `@XmlTransformation`:

```

<ELEM_B>
  <B_DATE>20100825</B_DATE>
  <B_TIME>153000</B_TIME>
  <NUM>123</NUM>
  <C_DATE>20100825</C_DATE>
  <C_TIME>154500</C_TIME>
</ELEM_B>

```

Note:

Ordinarily, you would use `@XmlAdapter`. However:

- Although the **DATE/TIME** pairings are repeated throughout the document, the element name changes each time (such as **B_DATE/B_TIME**, **C_DATE/C_TIME**, and so on).
- Because each pairing is missing a grouping element, you would need to adapt the *entire* **ElemB** class.

Because of these issues, MOXy's transformation mapping is much easier to implement:

Example 8-38 Mapping Example

```

package example;

import java.util.Date;

import javax.xml.bind.annotation.*;
import org.eclipse.persistence.xml.annotations.*;

@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name="ELEM_B")
public class ElemB {

    @XmlTransformation
    @XmlReadTransformer(transformerClass=DateAttributeTransformer.class)
    @XmlWriteTransformers({
        @XmlWriteTransformer(xmlPath="B_DATE/text()",
transformerClass=DateFieldTransformer.class),
        @XmlWriteTransformer(xmlPath="B_TIME/text()",
transformerClass=TimeFieldTransformer.class),
    })
    private Date bDate;

    @XmlElement(name="NUM")
    private int num;

    @XmlTransformation
    @XmlReadTransformer(transformerClass=DateAttributeTransformer.class)
    @XmlWriteTransformers({
        @XmlWriteTransformer(xmlPath="C_DATE/text()",
transformerClass=DateFieldTransformer.class),
        @XmlWriteTransformer(xmlPath="C_TIME/text()",
transformerClass=TimeFieldTransformer.class),
    })
    private Date cDate;
}

```

```
}
```

Using an AttributeTransformer

Use an `AttributeTransformer` to construct the Java attribute value:

Example 8-39 Sample AttributeTransformer

```
package example;

import java.text.ParseException;
import java.text.SimpleDateFormat;

import org.eclipse.persistence.internal.helper.DatabaseField;
import org.eclipse.persistence.mappings.foundation.AbstractTransformationMapping;
import org.eclipse.persistence.mappings.transformers.AttributeTransformer;
import org.eclipse.persistence.sessions.Record;
import org.eclipse.persistence.sessions.Session;

public class DateAttributeTransformer implements AttributeTransformer {

    private AbstractTransformationMapping mapping;
    private SimpleDateFormat yyyyMMddHHmmss = new SimpleDateFormat("yyyyMMddHHmmss");

    public void initialize(AbstractTransformationMapping mapping) {
        this.mapping = mapping;
    }

    public Object buildAttributeValue(Record record, Object instance, Session
session) {
        try {
            String dateString = null;
            String timeString = null;

            for (DatabaseField field : mapping.getFields()) {
                if (field.getName().contains("DATE")) {
                    dateString = (String) record.get(field);
                } else {
                    timeString = (String) record.get(field);
                }
            }
            return yyyyMMddHHmmss.parseObject(dateString + timeString);
        } catch (ParseException e) {
            throw new RuntimeException(e);
        }
    }
}
```

Using a FieldTransformer

Use a `FieldTransformer` to construct the XML field value from the Java object.

Each transformation mapping may have multiple write transformers. In this example, you will need two:

- The first write transformer writes the year, month, and day in **yyMMdd** format:

- The second write transformer writes out the hour, minutes, and seconds in **HHmmss** format.

Example 8-40 First Write Transformer

```
package example;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.eclipse.persistence.mappings.foundation.AbstractTransformationMapping;
import org.eclipse.persistence.mappings.transformers.FieldTransformer;
import org.eclipse.persistence.sessions.Session;

public class DateFieldTransformer implements FieldTransformer {

    private AbstractTransformationMapping mapping;
    private SimpleDateFormat yyyyMMdd = new SimpleDateFormat("yyyyMMdd");

    public void initialize(AbstractTransformationMapping mapping) {
        this.mapping = mapping;
    }

    public Object buildFieldValue(Object instance, String XPath, Session session) {
        Date date = (Date) mapping.getAttributeValueFromObject(instance);
        return yyyyMMdd.format(date);
    }

}
```

Example 8-41 Second Write Transformer

```
package example;

import java.text.SimpleDateFormat;
import java.util.Date;

import org.eclipse.persistence.mappings.foundation.AbstractTransformationMapping;
import org.eclipse.persistence.mappings.transformers.FieldTransformer;
import org.eclipse.persistence.sessions.Session;

public class TimeFieldTransformer implements FieldTransformer {

    private AbstractTransformationMapping mapping;
    private SimpleDateFormat HHmmss = new SimpleDateFormat("HHmmss");

    public void initialize(AbstractTransformationMapping mapping) {
        this.mapping = mapping;
    }

    public Object buildFieldValue(Object instance, String XPath, Session session) {
        Date date = (Date) mapping.getAttributeValueFromObject(instance);
        return HHmmss.format(date);
    }

}
```

Generating Java Classes from an XML Schema

Use the JAXB Compiler to generate Java classes from an XML schema. The generated classes will contain JAXB annotations that represent the XML binding metadata.

Running the JAXB Compiler

Use the **.sh** or **.cmd** file to run the JAXB Compiler:

```
<ECLIPSELINK_HOME>/eclipselink/bin/jaxb-compiler.sh <source-file.xsd> [-options]
```

or

```
<ECLIPSELINK_HOME>\eclipselink\bin\jaxb-compiler.cmd <source-file.xsd> [-options]
```

The JAXB Compiler supports the following options:

Option	Description
-nv	Do not perform <i>strict</i> validation of the input schemas.
-extension	Allow vender-specific extensions; do not strictly follow the Compatibility Rules in Appendix E.2 of the JAXB specification
-b <file/directory>	Specify the external binding files to process. Note: Each file must use its own -b flag.
-d <directory>	Specify the output directory for the generated files.
-p <package>	Specify the target package.
-classpath <arg>	Specify where to find user class files.
-verbose	Enable additional compiler output, such as informational messages.
-quiet	Disable compiler output.
-version	Display the compiler version.

For example:

```
jaxb-compiler.sh -d jaxb-compiler-output config/Customer.xsd
```

To display a complete list of compiler options, use:

```
jaxb-compiler.sh -help
```

Customizing Generated Mappings

When bootstrapping from an XML Schema (or an EclipseLink project from `sessions.xml`), you can customize the mappings that EclipseLink generates by using your own EclipseLink OXM Bindings file. This file contains your additional mappings and allows you to combine OXM with XSD bootstrapping. This means that you can use EclipseLink mappings to customize an existing XML schema.

This section shows how to override mappings defined in the schema. Although the schema defines **addresses** in Canadian format (with province and postal code), you can use XML that contains the **address** in USA format (with state and zip code).

First, you must create an `eclipselink-oxm.xml` file that contains the mapping overrides. In [Example 8-42](#), we modify the XPath expressions for province and postalCode:

When you create a `DynamicJAXBContext`, use the `properties` argument to pass this binding file to the `DynamicJAXBContextFactory` (in addition to the Schema):

```
// Load Schema
InputStream xsdStream = myClassLoader.getResourceAsStream("example/
resources/xsd/customer.xsd");

// Load OXM with customizations, put into Properties
InputStream oxmStream = myClassLoader.getResourceAsStream("example/resources/
eclipselink/eclipselink-oxm.xml");
Map<String, Object> props = new HashMap<String, Object>();
props.put(JAXBContextProperties.OXM_METADATA_SOURCE, oxmStream);

// Create Context
DynamicJAXBContext dContext =
DynamicJAXBContextFactory.createContextFromXSD(inputStream, null, myClassLoader,
props);
```

Example 8-42 Sample `eclipselink-oxm.xml` File

```
<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
package-name="example">
  <java-types>
    <java-type name="Address">
      <java-attributes>
        <xml-element java-attribute="province" xml-path="state/text()"/>
        <xml-element java-attribute="postalCode" xml-path="zip-code/text()"/>
      </java-attributes>
    </java-type>
  </java-types>
</xml-bindings>
```

9

Using Dynamic JAXB

This chapter includes the following sections:

- [Understanding Static and Dynamic Entities](#)
- [Specifying the EclipseLink Runtime](#)
- [Bootstrapping from XML Schema \(XSD\)](#)
- [Bootstrapping from EclipseLink Metadata \(OXM\)](#)

Understanding Static and Dynamic Entities

There are two high-level ways to use EclipseLink JAXB: using pre-existing Java classes ([Using Static MOXy](#)), or using EclipseLink-generated in-memory Java classes ([Using Dynamic MOXy](#)).

Using Static MOXy

The most common way to use EclipseLink JAXB is with existing Java classes, mapped to XML using Java annotations and/or EclipseLink OXM metadata. These classes might be ones that you have written yourself, or they could be generated from an XML Schema using the XJC compiler tool.

Using this approach, you will be dealing with your actual domain objects when converting to and from XML. [Example 9-1](#) shows a simple Java class that can be used with JAXB.

Note:

When using static classes with JAXB, you can take advantage of JAXB's defaulting behavior and only annotate things which differ from the default. For example, all fields on a Java class will default to being mapped to an XML element, so no annotation is needed on the name field. We want the id field, however, to map to an XML attribute, so have annotated it as such.

[Example 9-2](#) demonstrates how to unmarshal, modify, and marshal an object using static JAXB:

Example 9-1 Sample Java Class

```
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Customer {
    @XmlAttribute
```

```
private long id;

private String name;

// ...
// get() and set() methods
// ...
}
```

Example 9-2 Marshalling and Unmarshalling Example

```
JAXBContext jaxbContext = JAXBContext.newInstance(Customer.class, Address.class);
Customer customer = (Customer)
jaxbContext.createUnmarshaller().unmarshal(instanceDoc);

Address address = new Address();
address.setStreet("1001 Fleet St.");

customer.setAddress(address);

jaxbContext.createMarshaller().marshal(customer, System.out);
```

Using Dynamic MOXy

With EclipseLink Dynamic MOXy, you can bootstrap a JAXBContext from a variety of metadata sources and use existing JAXB APIs to marshal and unmarshal data... *without having compiled Java class files on the classpath*. This allows you to alter the metadata as needed, without having to update and recompile the previously-generated Java source code.

You should consider using Dynamic MOXy when:

- You want EclipseLink to generate mappings from an XML schema (XSD).
- You do not want to deal with concrete Java domain classes.

Using Dynamic Entities

Instead of using actual Java classes (such as Customer.class or Address.class), Dynamic MOXy uses a simple get(propertyName)/set(propertyName, propertyValue) API to manipulate data. EclipseLink generates (in memory) a DynamicType associated with each DynamicEntity.



Note:

DynamicTypes are similar to Java classes; whereas DynamicEntities can be thought of as instances of a DynamicType.

[Example 9-3](#) demonstrates how to unmarshal, modify, and marshal an object using dynamic JAXB:

Example 9-3 Marshalling and Unmarshalling Example

```

DynamicJAXBContext dynamicJAXBContext =
DynamicJAXBContextFactory.createContextFromXSD(xsdInputStream, null, myClassLoader,
null);
DynamicEntity customer = (DynamicEntity)
dynamicJAXBContext.createUnmarshaller().unmarshal(instanceDoc);

String lastName = customer.get("lastName");
List orders = customer.get("orders");
...
DynamicEntity address = dynamicJAXBContext.newDynamicEntity("mynamespace.Address");
address.set("street", "1001 Fleet St.");

customer.set("lastName", lastName + "Jr.");
customer.set("address", address);

dynamicJAXBContext.createMarshaller().marshal(customer, System.out);

```

 Note:

XML names found in the metadata (complex type names, element names, attribute names) will be translated to Java identifiers according to the algorithms described in "Appendix D: Binding XML Names to Java Identifiers" of the Java Architecture for XML Binding (JAXB) 2.3 Specification (<http://jcp.org/en/jsr/detail?id=222>).

In [Example 9-3](#), last-name in XML was translated to lastName for the Java object.

Specifying the EclipseLink Runtime

In order to use EclipseLink Dynamic MOXy as your JAXB implementation, you must identify the EclipseLink DynamicJAXBContextFactory in your jaxb.properties file.

1. Create a text file named jaxb.properties, specifying DynamicJAXBContextFactory as the factory used to build new JAXBContexts.
2. Copy the jaxb.properties file to the context path used to create the JAXBContext.
3. Use the standard JAXBContext.newInstance(String contextPath) API to create a DynamicJAXBContext.

```

DynamicJAXBContext jaxbContext = (DynamicJAXBContext)
JAXBContext.newInstance("org.example.mypackage");

```

Because you do not need to change any application code, you can easily switch between different JAXB implementations.

Instantiating a DynamicJAXBContext

The following methods on `JAXBContext` can be used to create new instances of `DynamicJAXBContext`s:

```
public static JAXBContext newInstance(String contextPath) throws JAXBException
public static JAXBContext newInstance(String contextPath, ClassLoader classLoader)
    throws JAXBException
public static JAXBContext newInstance(String contextPath, ClassLoader classLoader,
    Map<String,?> properties) throws JAXBException
```

- **contextPath** – Location of `jaxb.properties` file.
- **classLoader** – The application's current class loader, which will be used to first lookup classes to see if they exist before new `DynamicTypes` are generated.
- **properties** – A map of properties to use when creating a new `DynamicJAXBContext`. This map must contain one of the following two keys:
 - `org.eclipse.persistence.jaxb.JAXBContextFactory.XML_SCHEMA_KEY`, which can have several possible values:
 - * One of the following, pointing to your XML Schema file:
 - * `java.io.InputStream`
 - * `org.w3c.dom.Node`
 - * `javax.xml.transform.Source`
 - `org.eclipse.persistence.jaxb.JAXBContextProperties.OXM_METADATA_SOURCE`, which can have several possible values:
 - * One of the following, pointing to your OXM file:
 - * `java.io.File`
 - * `java.io.InputStream`
 - * `java.io.Reader`
 - * `java.net.URL`
 - * `javax.xml.stream.XMLStreamReader`
 - * `javax.xml.stream.XMLStreamReader`
 - * `javax.xml.transform.Source`
 - * `org.w3c.dom.Node`
 - * `org.xml.sax.InputSource`



Note:

If using one of these options, a package-name element must be defined in the `xml-bindings` element of your OXM file.

- * A List of objects from the set above.

 Note:

If using this option, a package-name element must be defined in the xml-bindings element of your OXM file.

- * A Map<String, Object>, where String is a package name, and Object is the pointer to the OXM file, from the set of possibilities above.

Bootstrapping from XML Schema (XSD)

With EclipseLink MOXy, you can provide an existing XML schema from which to create a DynamicJAXBContext. EclipseLink will parse the schema and generate DynamicTypes for each complex type. This is achieved by use of the DynamicJAXBContextFactory class. A DynamicJAXBContext cannot be instantiated directly; it must be created through the factory API.

You can pass the XML Schema to DynamicJAXBContextFactory by using:

- java.io.InputStream
- org.w3c.dom.Node
- javax.xml.transform.Source

 Note:

EclipseLink MOXy uses Sun's XJC (XML-to-Java Compiler) APIs to parse the schema into an in-memory representation and generate dynamic types and mappings. When bootstrapping from XSD, you will need to include jaxb-xjc.jar (from the JAXB reference implementation) on your **CLASSPATH**.

The APIs used to create a DynamicJAXBContext are as follows:

 Note:

The classLoader parameter is your application's current class loader, and will be used to first lookup classes to see if they exist before new DynamicTypes are generated. The user may pass in null for this parameter, and Thread.currentThread().getContextClassLoader() will be used instead.

This example shows how to create and marshall a new object using Dynamic MOXy.

The code snippet in [Example 9-6](#):

- Passes the XML Schema to DynamicJAXBContextFactory to create a DynamicJAXBContext
- Creates new DynamicEntities and sets their properties

- Creates a JAXBMarshaller and marshals the Java objects to XML

Example 9-4 Creating a DynamicJAXBContext

```
/**
 * Create a DynamicJAXBContext, using XML Schema as the metadata source.
 *
 * @param schemaStream
 *     java.io.InputStream from which to read the XML Schema.
 * @param resolver
 *     An org.xml.sax.EntityResolver, used to resolve schema imports. Can be null.
 * @param classLoader
 *     The application's current class loader, which will be used to first lookup
 *     classes to see if they exist before new DynamicTypes are generated. Can be
 *     null, in which case Thread.currentThread().getContextClassLoader() will be used.
 * @param properties
 *     Map of properties to use when creating a new DynamicJAXBContext. Can be null.
 *
 * @return
 *     A new instance of DynamicJAXBContext.
 *
 * @throws JAXBException
 *     if an error was encountered while creating the DynamicJAXBContext.
 */
public static DynamicJAXBContext createContextFromXSD(java.io.InputStream schemaStream,
EntityResolver resolver,
    ClassLoader classLoader, Map<String, ?> properties) throws JAXBException

public static DynamicJAXBContext createContextFromXSD(org.w3c.dom.Node schemaDOM, EntityResolver
resolver,
    ClassLoader classLoader, Map<String, ?> properties) throws JAXBException

public static DynamicJAXBContext createContextFromXSD(javax.xml.transform.Source schemaSource,
EntityResolver resolver,
    ClassLoader classLoader, Map<String, ?> properties) throws JAXBException
```

Example 9-5 Sample XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="example" xmlns:myns="example" xmlns:xs="http://
www.w3.org/2001/XMLSchema"
    attributeFormDefault="qualified" elementFormDefault="qualified">

    <xs:element name="customer" type="myns:customer"/>

    <xs:complexType name="customer">
        <xs:sequence>
            <xs:element name="first-name" type="xs:string"/>
            <xs:element name="last-name" type="xs:string"/>
            <xs:element name="address" type="myns:address"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="address">
        <xs:sequence>
            <xs:element name="street" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="province" type="xs:string"/>
            <xs:element name="postal-code" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
```

```
</xs:schema>
```

Example 9-6 Sample Application Code

```
InputStream inputStream = myClassLoader.getResourceAsStream("example/
resources/xsd/customer.xsd");
DynamicJAXBContext dContext =
DynamicJAXBContextFactory.createContextFromXSD(inputStream, null, myClassLoader,
null);

DynamicEntity newCustomer = dContext.newDynamicEntity("example.Customer");
newCustomer.set("firstName", "George");
newCustomer.set("lastName", "Jones");

DynamicEntity newAddress = dContext.newDynamicEntity("example.Address");
newAddress.set("street", "227 Main St.");
newAddress.set("city", "Toronto");
newAddress.set("province", "Ontario");
newAddress.set("postalCode", "M5V1E6");

newCustomer.set("address", newAddress);

dContext.createMarshaller().marshal(newCustomer, System.out);
```

Importing Other Schemas / EntityResolvers

If the XML schema that you use to bootstrap imports other schemas, you must configure an `org.xml.sax.EntityResolver` to resolve the locations of the imported schemas. You can then pass the `EntityResolver` to the `DynamicJAXBContextFactory`.

In [Example 9-7](#), each type is defined in its own schema:

You must supply an `EntityResolver` implementation to resolve the location of the imported schema.

[Example 9-8](#) illustrates the `EntityResolver`:

When you create the `DynamicJAXBContext`, pass the `EntityResolver` to it, as shown here:

```
InputStream inputStream = ClassLoader.getResourceAsStream("com/foo/sales/xsd/
customer.xsd");
DynamicJAXBContext dContext =
DynamicJAXBContextFactory.createContextFromXSD(inputStream, new MyEntityResolver(),
null, null);
```

If you encounter the following exception when importing another schema:

```
Internal Exception: org.xml.sax.SAXParseException: schema_reference.4:
Failed to read schema document '<imported-schema-name>', because 1) could
not find the document; 2) the document could not be read; 3) the root
element of the document is not <xsd:schema>.
```

You should disable XJC's schema correctness check option, either in code:

```
System.setProperty("com.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCo
rrectnessCheck", "true")
```

or from the command line:

```
-Dcom.sun.tools.xjc.api.impl.s2j.SchemaCompilerImpl.noCorrect
```

Example 9-7 Sample XML Schema

```
<!-- customer.xsd -->

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:myns="example" xmlns:add="addressNamespace"
  xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="example">

  <xs:import namespace="addressNamespace" schemaLocation="address.xsd"/>

  <xs:element name="customer" type="myns:customer"/>

  <xs:complexType name="customer">
    <xs:sequence>
      <xs:element name="first-name" type="xs:string"/>
      <xs:element name="last-name" type="xs:string"/>
      <xs:element name="address" type="add:address"/>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Example 9-8 Sample Application Code

```
class MyEntityResolver implements EntityResolver {

    public InputSource resolveEntity(String publicId, String systemId) throws
    SAXException, IOException {
        // Imported schemas are located in ext\appdata\xsd\

        // Grab only the filename part from the full path
        String filename = new File(systemId).getName();

        // Now prepend the correct path
        String correctedId = "ext/appdata/xsd/" + filename;

        InputSource is = new
        InputSource(ClassLoader.getResourceAsStream(correctedId));
        is.setSystemId(correctedId);

        return is;
    }
}
```

Customizing Generated Mappings with XJC External Binding Customization Files

When bootstrapping from an XSD, you have the option to customize the mappings that will be generated through the use of XJC's External Binding Customization file format (.xjb). In the example below, the package name of the dynamic classes has been overridden, and the name attribute has been renamed to last-name-comma-first-name.

For complete information on the External Binding Customization file format, please see http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/tutorial/doc/JAXBUsing4.html.

Note:

If you wish to use External Binding Customization files, you will need to use **Source** objects to point to your XML Schema. **Sources** are used to load the .xjb files as well, and they must all have the same System ID set.

Example 9-10 illustrates bootstrapping from an XSD, and customizing the mapping generation using two separate .xjb files.

The value of EXTERNAL_BINDINGS_KEY can be either a single Source or a List<Source>, pointing to your External Binding Customization file(s).

Example 9-9 custom1.xjb File

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

    <!-- Customize the package name that is generated for each schema -->
    <jxb:schemaBindings>
      <jxb:package name="com.acme.internal"/>
    </jxb:schemaBindings>

    <!-- Rename the 'name' element to 'last-name-comma-first-name' -->
    <jxb:bindings node="//xs:complexType[@name='person']">
      <jxb:bindings node="//xs:element[@name='name']">
        <jxb:property name="last-name-comma-first-name"/>
      </jxb:bindings>
    </jxb:bindings>

  </jxb:bindings>
</jxb:bindings>
```

Example 9-10 Bootstrapping Example

```
ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
String xsd = "example/resources/xsd/employee.xsd";
String xjb1 = "example/resources/xsd/custom1.xjb";
String xjb2 = "example/resources/xsd/custom2.xjb";

InputStream xsdStream = classLoader.getResourceAsStream(xsd);
Source xsdSource = new StreamSource(xsdStream);
// Set SYSTEM_ID to the filename part of the XSD
xsdSource.setSystemId("employee.xsd");

InputStream xjbStream = classLoader.getResourceAsStream(xjb1);
Source xjbSource = new StreamSource(xjbStream);
// Set SYSTEM_ID to be the same as the XSD
xjbSource.setSystemId(xsdSource.getSystemId());

InputStream xjbStream2 = classLoader.getResourceAsStream(xjb2);
Source xjbSource2 = new StreamSource(xjbStream2);
```

```
// Set SYSTEM_ID to be the same as the XSD
xjbSource2.setSystemId(xsdSource.getSystemId());

ArrayList<Source> xjbFiles = new ArrayList<Source>(2);
xjbFiles.add(xjbSource);
xjbFiles.add(xjbSource2);

// Put XSD and XJBs into Properties
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(DynamicJAXBContextFactory.XML_SCHEMA_KEY, xsdSource);
properties.put(DynamicJAXBContextFactory.EXTERNAL_BINDINGS_KEY, xjbFiles);

// Create Context
DynamicJAXBContext jaxbContext = (DynamicJAXBContext)
JAXBContext.newInstance("example", classLoader, properties);
```

Bootstrapping from EclipseLink Metadata (OXM)

If you would like to have more control over how your `DynamicEntities` will be mapped to XML, you can instead bootstrap from an EclipseLink OXM Metadata file. Using this approach, you can take advantage of EclipseLink's robust mappings framework and customize how each complex type in XML maps to its Java counterpart. The following API on `DynamicJAXBContextFactory` can be used to bootstrap from an OXM file:

Links to the actual OXM files are passed in via the `properties` parameter, using a special key, `JAXBContextProperties.OXM_METADATA_SOURCE`. The value of this key will be a handle to the OXM metadata file, in the form of one of the following:

- `java.io.File`
- `java.io.InputStream`
- `java.io.Reader`
- `java.net.URL`
- `javax.xml.stream.XMLEventReader`
- `javax.xml.stream.XMLStreamReader`
- `javax.xml.transform.Source`
- `org.w3c.dom.Node`
- `org.xml.sax.InputSource`

Lists of the above inputs are acceptable as well, to bootstrap from multiple OXM files. For more information, see the documentation on the `DynamicJAXBContextFactory` class.

In the following example, we will obtain our OXM file as a resource from our `ClassLoader`, and use the resulting `InputStream` to bootstrap a `DynamicJAXBContext`:

```
InputStream iStream = myClassLoader.getResourceAsStream("example/resources/
eclipselink/eclipselink-oxm.xml");

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, iStream);

DynamicJAXBContext jaxbContext =
DynamicJAXBContextFactory.createContextFromOXM(myClassLoader, properties);
```

Example 9-11 Creating a DynamicJAXBContext

```

/**
 * Create a <tt>DynamicJAXBContext</tt>, using an EclipseLink OXM file as the metadata source.
 *
 * @param classLoader
 *     The application's current class loader, which will be used to first lookup classes to
 *     see if they exist before new <tt>DynamicTypes</tt> are generated. Can be <tt>null</tt>,
 *     in which case <tt>Thread.currentThread().getContextClassLoader()</tt> will be used.
 * @param properties
 *     Map of properties to use when creating a new <tt>DynamicJAXBContext</tt>. This map must
 *     contain a key of JAXBContext.ECLIPSELINK_OXM_XML_KEY, with a value of... (see below)
 *
 * @return
 *     A new instance of <tt>DynamicJAXBContext</tt>.
 *
 * @throws JAXBException
 *     if an error was encountered while creating the <tt>DynamicJAXBContext</tt>.
 */
public static DynamicJAXBContext createContextFromOXM(ClassLoader classLoader, Map<String, ?>
properties) throws JAXBException {

```

Example

Using the sample OXM in [Example 9-12](#), we will show an example of how to create and marshall a new object using Dynamic MOXy. It is important to note the type attributes. Because there is no underlying Java class, the types of each property must be explicitly specified.

The code in [Example 9-13](#) demonstrates:

- Passing the OXM file to DynamicJAXBContextFactory to create a DynamicJAXBContext
- Creating new DynamicEntities and setting their properties
- Creating a JAXBMarshaller and marshalling the Java objects to XML

Example 9-12 Sample XML Schema

```

<?xml version="1.0" encoding="US-ASCII"?>
<xml-bindings xmlns="http://www.eclipse.org/eclipselink/xsds/persistence/oxm"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xs="http://www.w3.org/2001/XMLSchema" package-name="example">

  <java-types>
    <java-type name="Customer">
      <xml-root-element name="customer"/>
      <java-attributes>
        <xml-element java-attribute="firstName" type="java.lang.String"/>
        <xml-element java-attribute="lastName" type="java.lang.String"/>
        <xml-element java-attribute="address" type="example.Address"/>
      </java-attributes>
    </java-type>

    <java-type name="Address">
      <java-attributes>
        <xml-element java-attribute="street" type="java.lang.String"/>
        <xml-element java-attribute="city" type="java.lang.String"/>
      </java-attributes>
    </java-type>
  </java-types>

```

```
        <xml-element java-attribute="province" type="java.lang.String"/>
        <xml-element java-attribute="postalCode" type="java.lang.String"/>
    </java-attributes>
</java-type>
</java-types>

</xml-bindings>
```

Example 9-13 Sample Application Code

```
InputStream iStream = myClassLoader.getResourceAsStream("example/resources/
eclipselink/eclipselink-oxm.xml");

Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.OXM_METADATA_SOURCE, iStream);

DynamicJAXBContext jaxbContext =
DynamicJAXBContextFactory.createContextFromOXM(myClassLoader, properties);

DynamicEntity newCustomer = dContext.newDynamicEntity("example.Customer");
newCustomer.set("firstName", "George");
newCustomer.set("lastName", "Jones");

DynamicEntity newAddress = dContext.newDynamicEntity("example.Address");
newAddress.set("street", "227 Main St.");
newAddress.set("city", "Toronto");
newAddress.set("province", "Ontario");
newAddress.set("postalCode", "M5V1E6");

newCustomer.set("address", newAddress);

dContext.createMarshaller().marshal(newCustomer, System.out
```

10

Using JSON Documents

Starting in Release 2.4, Oracle TopLink MOXy supports the ability to convert objects to and from JSON (JavaScript Object Notation). This feature is useful when creating RESTful services; JAX-RS services can accept both XML and JSON messages.

This chapter includes the following sections:

- [Understanding JSON Documents](#)
- [Marshalling and Unmarshalling JSON Documents](#)
- [Specifying JSON Bindings](#)

Understanding JSON Documents

TopLink supports all MOXy object-to-XML options when reading and writing JSON, including:

- TopLink's advanced and extended mapping features (in addition to the JAXB specification)
- Storing mappings in external bindings files
- Creating dynamic models with Dynamic JAXB
- Building extensible models that support multitenant applications

Marshalling and Unmarshalling JSON Documents

Use the **eclipselink.media-type** property on your JAXB Marshaller or Unmarshaller to produce and use JSON documents with your application, as shown in [Example 10-1](#).

You can also specify the `eclipselink.media-type` property in the Map of the properties used when you create the `JAXBContext`, as shown in [Example 10-2](#).

You can also configure your application to use JSON documents by using the `MarshallerProperties`, `UnmarshallerProperties`, and `MediaType` constants, as shown in [Example 10-3](#).

Example 10-1 Marshalling and Unmarshalling

```
...

Marshaller m = jaxbContext.createMarshaller();
m.setProperty("eclipselink.media-type", "application/json");

Unmarshaller u = jaxbContext.createUnmarshaller();
u.setProperty("eclipselink.media-type", "application/json");

...
```

Example 10-2 Using a Map

```
import org.eclipse.persistence.jaxb.JAXBContextProperties;
import org.eclipse.persistence.oxm.MediaType;

Map<String, Object> properties = new HashMap<String, Object>();
properties.put("eclipselink.media-type", "application/json");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Employee.class },
properties);
Marshaller jsonMarshaller = ctx.createMarshaller();
Unmarshaller jsonUnmarshaller = ctx.createUnmarshaller();
```

When specified in a Map, the Marshallers and Unmarshallers created from the JAXBContext will automatically use the specified media type.

Example 10-3 Using MarshallerProperties and UnmarshallerProperties

```
import org.eclipse.persistence.jaxb.MarshallerProperties;
import org.eclipse.persistence.jaxb.UnmarshallerProperties;
import org.eclipse.persistence.oxm.MediaType;

m.setProperty(MarshallerProperties.MEDIA_TYPE, MediaType.APPLICATION_JSON);
u.setProperty(UnmarshallerProperties.MEDIA_TYPE, MediaType.APPLICATION_JSON);
...
```

Specifying JSON Bindings

[Example 10-4](#) shows a basic JSON binding that does not require compile time dependencies in addition to those required for normal JAXB usage. This example shows how to unmarshal JSON from a StreamSource into the user object SearchResults, add a new Result to the collection, and then marshal the new collection to System.out.

You can also write MOXy External Bindings files as JSON documents. [Example 10-5](#) shows how to use bindings.json to map **Customer** and **PhoneNumber** classes to JSON.

[Example 10-6](#) shows how to use the JSON file (created in [Example 10-5](#)) when bootstrapping a JAXBContext.

Example 10-4 Using Basic JSON Binding

```
package org.example;

import org.example.model.Result;
import org.example.model.SearchResults;

import java.util.Date;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBElement;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import javax.xml.transform.stream.StreamSource;

public class Demo {

    public static void main(String[] args) throws Exception {
        JAXBContext jc = JAXBContext.newInstance(SearchResults.class);
```

```

        Unmarshaller unmarshaller = jc.createUnmarshaller();
        unmarshaller.setProperty("eclipselink.media-type", "application/json");
        StreamSource source = new StreamSource("http://search.twitter.com/
search.json?q=jaxb");
        JAXBElement<SearchResults> jaxbElement = unmarshaller.unmarshal(source,
SearchResults.class);

        Result result = new Result();
        result.setCreatedAt(new Date());
        result.setFromUser("bsmith");
        result.setText("You can now use EclipseLink JAXB (MOXy) with JSON :");
        jaxbElement.getValue().getResults().add(result);

        Marshaller marshaller = jc.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
        marshaller.setProperty("eclipselink.media-type", "application/json");
        marshaller.marshal(jaxbElement, System.out);
    }
}

```

Example 10-5 Using External Bindings

```

{
  "package-name" : "org.example",
  "xml-schema" : {
    "element-form-default" : "QUALIFIED",
    "namespace" : "http://www.example.com/customer"
  },
  "java-types" : {
    "java-type" : [ {
      "name" : "Customer",
      "xml-type" : {
        "prop-order" : "firstName lastName address phoneNumbers"
      },
      "xml-root-element" : {},
      "java-attributes" : {
        "xml-element" : [
          {"java-attribute" : "firstName", "name" : "first-name"},
          {"java-attribute" : "lastName", "name" : "last-name"},
          {"java-attribute" : "phoneNumbers", "name" : "phone-number"}
        ]
      }
    }
  ],
  {
    "name" : "PhoneNumber",
    "java-attributes" : {
      "xml-attribute" : [
        {"java-attribute" : "type"}
      ],
      "xml-value" : [
        {"java-attribute" : "number"}
      ]
    }
  }
]
}

```

Example 10-6 Using JSON to Bootstrap a JAXBContext

```
Map<String, Object> properties = new HashMap<String, Object>(2);
properties.put("eclipselink-oxm-xml", "org/example/binding.json");
properties.put("eclipselink.media-type", "application/json");
JAXBContext context = JAXBContext.newInstance("org.example",
Customer.class.getClassLoader() , properties);

Unmarshaller unmarshaller = context.createUnmarshaller();
StreamSource json = new StreamSource(new File("src/org/example/input.json"));
...
```

Specifying JSON Data Types

Although XML has a single datatype, JSON differentiates between strings, numbers, and booleans. TopLink supports these datatypes automatically, as shown in [Example 10-7](#)

Example 10-7 Using JSON Data Types

```
public class Address {

    private int id;
    private String city;
    private boolean isMailingAddress;

}

{
    "id" : 1,
    "city" : "Ottawa",
    "isMailingAddress" : true
}
```

Supporting Attributes

JSON does not use attributes; anything mapped with a `@XmlAttribute` annotation will be marshalled as an element. By default, TopLink triggers *both* the attribute and element events, thereby allowing either the mapped attribute or element to handle the value.

You can override this behavior by using the `JSON_ATTRIBUTE_PREFIX` property to specify an attribute prefix, as shown in [Example 10-8](#). TopLink prepends the prefix to the attribute name during marshal and will recognize it during unmarshal.

In the example below the number field is mapped as an attribute with the prefix `@`.

You can also set the `JSON_ATTRIBUTE_PREFIX` property in the Map used when creating the JAXBContext, as shown in [Example 10-9](#). Allmarshallers and unmarshallers created from the context will use the specified prefix.

Example 10-8 Using a Prefix

```
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_ATTRIBUTE_PREFIX, "@");
jsonMarshaller.setProperty(MarshallerProperties.JSON_ATTRIBUTE_PREFIX, "@") ;
```

```
{
  "phone" : {
    "area-code" : "613",
    "@number" : "1234567"
  }
}
```

Example 10-9 Setting a Prefix in a Map

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.JSON_ATTRIBUTE_PREFIX, "@");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Phone.class }, properties);
```

Supporting no Root Element

TopLink supports JSON documents without a root element. By default, if no `@XmlRootElement` annotation exists, the marshalled JSON document will not have a root element. You can override this behavior (that is omit the root element from the JSON output, even if the `@XmlRootElement` is specified) by setting the `JSON_INCLUDE_ROOT` property when marshalling a document, as shown in [Example 10-10](#).

When unmarshaling a document with no root elements, you should set the `JSON_INCLUDE_ROOT` property as shown in [Example 10-10](#).

Note:

If the document has no root element, you must specify the class to unmarshal to.

Example 10-10 Marshalling no Root Element Documents

```
marshaller.setProperty(MarshallerProperties.JSON_INCLUDE_ROOT, false);
```

Example 10-11 Unmarshalling no Root Element Documents

```
unmarshaller.setProperty(UnmarshallerProperties.JSON_INCLUDE_ROOT, false);
JAXBElement<SearchResults> jaxbElement = unmarshaller.unmarshal(source,
SearchResults.class);
```

Using Namespaces

Because JSON does not use namespaces, by default all namespaces and prefixes are ignored when marshaling and unmarshaling. In some cases, this may be an issue if you have multiple mappings with the same local name – there will be no way to distinguish between the mappings.

With TopLink, you can supply a Map of namespace-to-prefix (or an instance of `NamespacePrefixMapper`) to the `Marshaller` and `Unmarshaller`. The namespace prefix will appear in the marshalled document prepended to the element name. TopLink will recognize the prefix during an unmarshal operation and the resulting Java objects will be placed in the proper namespaces.

[Example 10-12](#) shows how to use the `NAMESPACE_PREFIX_MAPPER` property.

The `MarshallerProperties.NAMESPACE_PREFIX_MAPPER` applies to *both* XML and JSON; `UnmarshallerProperties.JSON_NAMESPACE_PREFIX_MAPPER` is a *JSON-only* property. XML unmarshalling can obtain the namespace information directly from the document.

When JSON is marshalled, the namespaces will be given the prefix from the Map separated by a dot (.):

```
{
  "ns1.employee" : {
    "ns2.id" : 123
  }
}
```

The dot separator can be set to any custom character by using the `JSON_NAMESPACE_SEPARATOR` property. Here, a colon (:) will be used instead:

```
jsonMarshaller.setProperty(MarshallerProperties.JSON_NAMESPACE_SEPARATOR, ':');
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_NAMESPACE_SEPARATOR, ':');
```

Example 10-12 Using Namesapces

```
Map<String, String> namespaces = new HashMap<String, String>();
namespaces.put("namespace1", "ns1");
namespaces.put("namespace2", "ns2");
jsonMarshaller.setProperty(MarshallerProperties.NAMESPACE_PREFIX_MAPPER, namespaces);
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_NAMESPACE_PREFIX_MAPPER,
namespaces);
```

Using Collections

By default, when marshalling to JSON, TopLink marshals empty collections as `[ ]`, as shown in [Example 10-13](#).

Use the `JSON_MARSHAL_EMPTY_COLLECTIONS` property to override this behavior (so that empty collections are not marshalled at all).

```
jsonMarshaller.setProperty(MarshallerProperties.JSON_MARSHAL_EMPTY_COLLECTIONS,
Boolean.FALSE) ;
```

```
{
  "phone" : {
  }
}
```

Example 10-13

```
{
  "phone" : {
    "myList" : [ ]
  }
}
```

Mapping Root-Level Collections

If you use the `@XmlRootElement(name="root")` annotation to specify a root level, the JSON document can be marshaled as:

```
marshaller.marshall(myListOfRoots, System.out);
```

```
[ {
  "root" : {
    "name" : "aaa"
  }, {
    "root" : {
      "name" : "bbb"
    }
} ]
```

Because the root element *is* present in the document, you can unmarshal it using:

```
unmarshaller.unmarshal(json);
```

If the class *does not* have an `@XmlRootElement` (or if `JSON_INCLUDE_ROOT = false`), the marshal would produce:

```
[ {
  "name": "aaa"
}, {
  "name": "bbb"
} ]
```

Because the root element *is not* present, you must indicate the class to unmarshal to:

```
unmarshaller.unmarshal(json, Root.class);
```

Wrapping XML Values

JAXB supports one or more `@XmlAttributes` on `@XmlValue` classes, as shown in [Example 10-14](#)

To produce a valid JSON document, `TopLink` uses a value wrapper, as shown in [Example 10-15](#).

By default, `TopLink` uses **value** as the name of the wrapper. Use the `JSON_VALUE_WRAPPER` property to customize the name of the value wrapper, as shown in [Example 10-16](#).

You can also specify the `JSON_VALUE_WRAPPER` property in the Map of the properties used when you create the `JAXBContext`, as shown in [Example 10-17](#).

When specified in a Map, the Marshallers and Unmarshallers created from the `JAXBContext` will automatically use the specified value wrapper.

Example 10-14 Using @XmlAttributes

```
public class Phone {
```

```

@XmlValue
public String number;

@XmlAttribute
public String areaCode;

public Phone() {
    this("", "");
}

public Phone(String num, String code) {
    this.number = num;
    this.areaCode = code;
}
}

```

Example 10-15 Using a value Wrapper

```

{
  "employee" : {
    "name" : "Bob Smith",
    "mainPhone" : {
      "areaCode" : "613",
      "value" : "555-5555"
    },
    "otherPhones" : [ {
      "areaCode" : "613",
      "value" : "123-1234"
    }, {
      "areaCode" : "613",
      "value" : "345-3456"
    } ]
  }
}

```

Example 10-16

```

jsonMarshaller.setProperty(MarshallerProperties.JSON_VALUE_WRAPPER, "$");
jsonUnmarshaller.setProperty(UnmarshallerProperties.JSON_VALUE_WRAPPER, "$");

```

Would produce:

```

{
  "employee" : {
    "name" : "Bob Smith",
    "mainPhone" : {
      "areaCode" : "613",
      "$" : "555-5555"
    },
    "otherPhones" : [ {
      "areaCode" : "613",
      "$" : "123-1234"
    }, {
      "areaCode" : "613",
      "$" : "345-3456"
    } ]
  }
}

```

Example 10-17 Using a Map

```
Map<String, Object> properties = new HashMap<String, Object>();
properties.put(JAXBContextProperties.JSON_VALUE_WRAPPER, "$");

JAXBContext ctx = JAXBContext.newInstance(new Class[] { Employee.class },
properties);
Marshaller jsonMarshaller = ctx.createMarshaller();
Unmarshaller jsonUnmarshaller = ctx.createUnmarshaller();
```