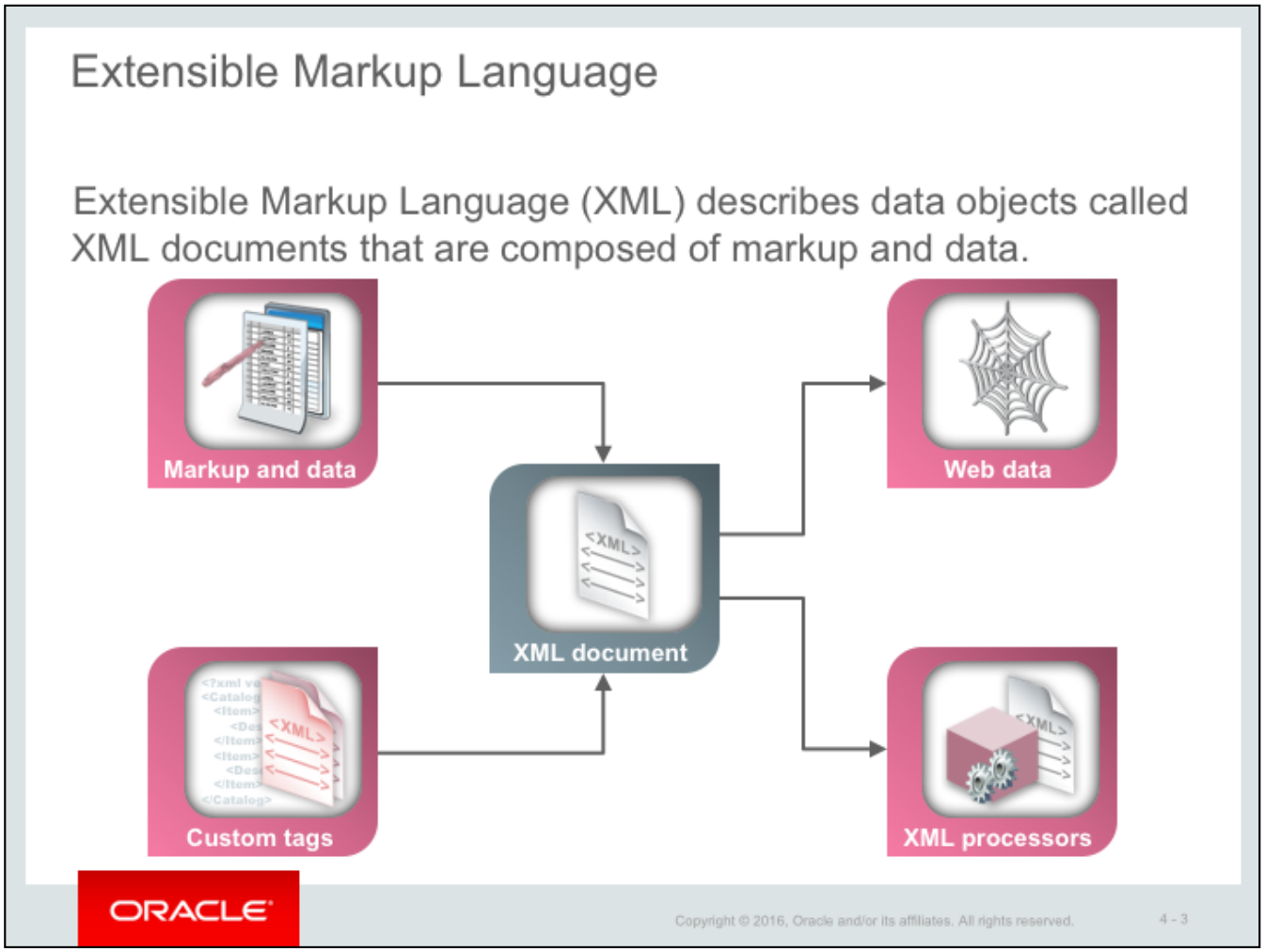
4

XML Programming with JAXB

Objectives

After completing this lesson, you should be able to do the following:

- Describe the benefits of XML
- Describe the components of an XML document
- Define a simple XML Schema
- Define the benefits of JAXB
- Create XML data by using JAXB
- Unmarshal XML data
- Marshal XML data

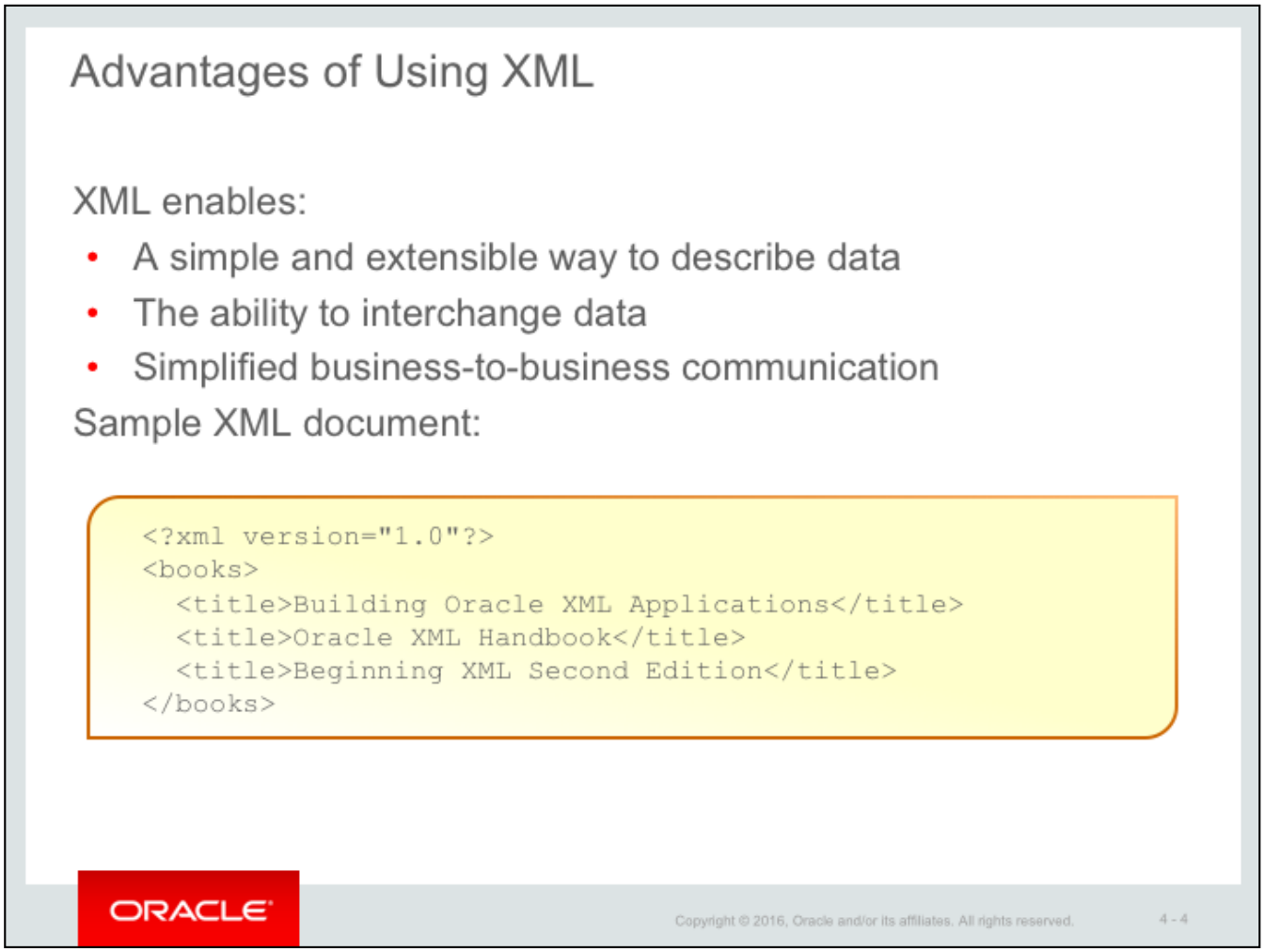


XML describes data objects called XML documents that:

- Are composed of markup language for structuring the document data
- Support custom tags for data definition, transmission, validation, and interpretation
- Have become a standard way to describe data on the web
- Are processed by XML processors

XML was developed by an XML working group headed by the World Wide Web Consortium (W3C) with the following design goals:

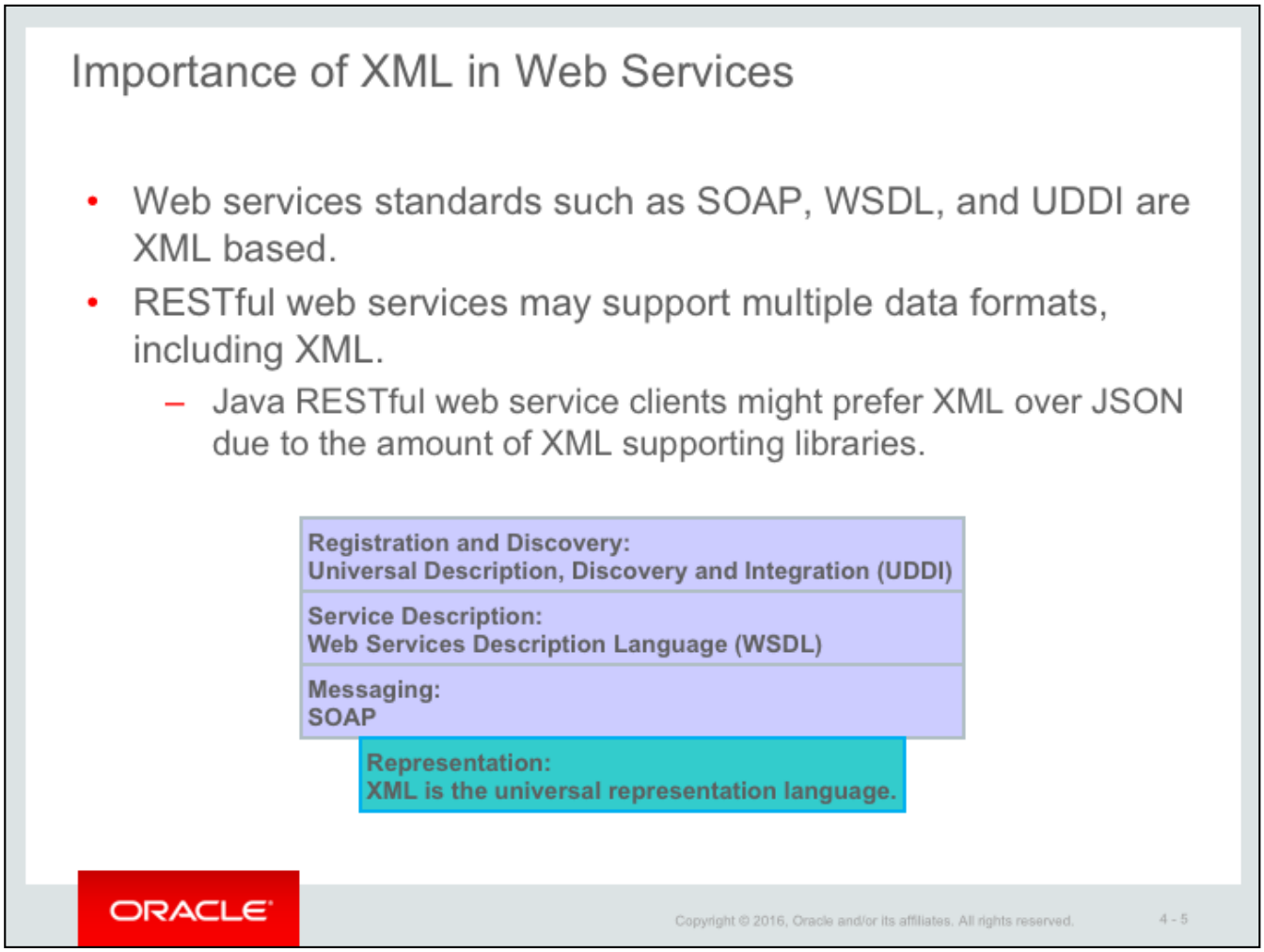
- XML is usable over the Internet.
- It supports a wide variety of applications.
- It is compatible with Standard Generalized Markup Language (SGML).
- XML can be processed by using easy-to-write programs.
- It has a minimum number of optional features.
- XML is human-legible and reasonably clear.
- XML enables quick design preparation.
- It enables formal and concise design.
- XML documents are easy to create.
- XML documents can be verbose.



XML's strongest point is its ability to perform data interchange. Because different groups of people rarely standardize on a single set of tools, it takes a significant amount of work for two groups to communicate. XML makes it easy to send structured data across the web so that nothing gets lost in translation.

When using XML, you can receive XML-tagged data from your system and you can receive XML-tagged data from another system. Neither of the users has to know how the other user's system is organized. If another partner or supplier teams up with your organization, you do not have to write code to exchange data with their system. You simply require them to follow the document rules defined in the document type definition (DTD). You can also transform those documents by using XSLT.

The slide shows a simple example of an XML document that contains a list of book titles. The XML document can be embellished or restructured depending on how it will be used.

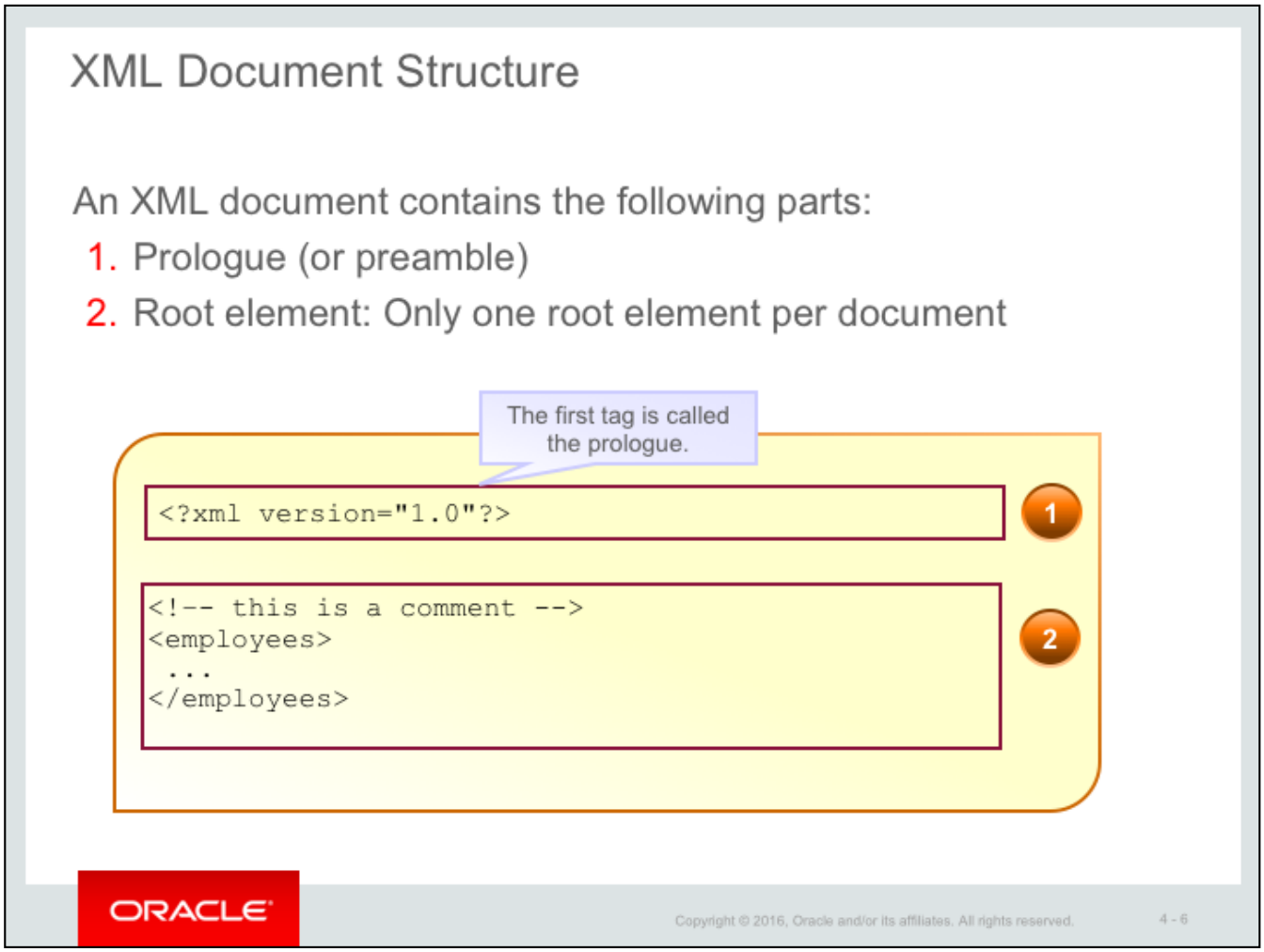


Many businesses today are becoming e-businesses. The greatest barrier to this transformation is application-to-application communication. An enterprise system usually has multiple applications, each running on different hardware, and sometimes implemented in different languages. These applications often need to exchange data with one another, and developers must write additional code to make this integration possible. A web service is a technology (based on a set of standards) that provides a solution for application-to-application communication and enterprise application integration.

Using web services, clients and servers can communicate over standard Internet protocols, regardless of the platform or programming language. Developers can write their business functionality in any language and on any platform.

To be exposed and accessed as web services, the applications must be developed according to the standards used by web services. Web services standards use the core concept of XML-based representation to carry out information exchange between service providers and requestors.

Simple Object Access Protocol (SOAP) is a lightweight, XML-based protocol for exchanging data in a distributed environment. The definition of SOAP is in XML and places no restriction on the format.



An XML document contains the following parts:

- The prologue, which may contain the following information:
  - XML declaration (optional in XML 1.0, mandatory in XML 1.1)
  - Document type definition (DTD), which is required only to validate the document structure, e.g.:

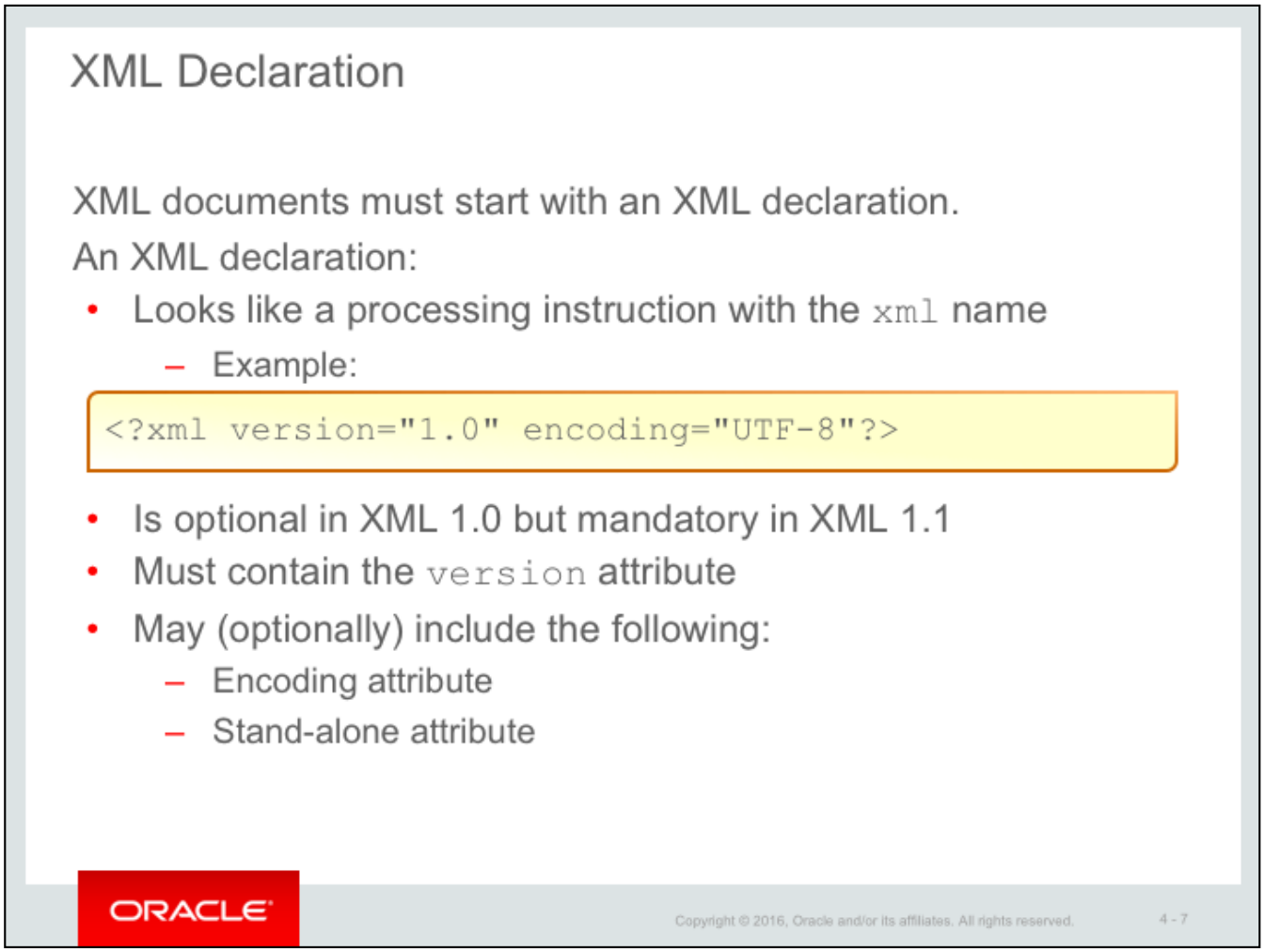
```
<!DOCTYPE employees [
  <!ELEMENT employee (#PCDATA)>
]>
```

  - Processing instructions and comments, which are optional
- The root element, which is also called the “document element” and contains all other elements

Processing instructions give commands or information to an application that processes the XML data.

Processing instructions have the format `<?target instructions?>`, where target is the name of the application that is expected to do the processing, and the instructions consist of a string of characters that embody the information or commands for the application to process.

Processing instructions can be written in the prologue, or root element of an XML document.

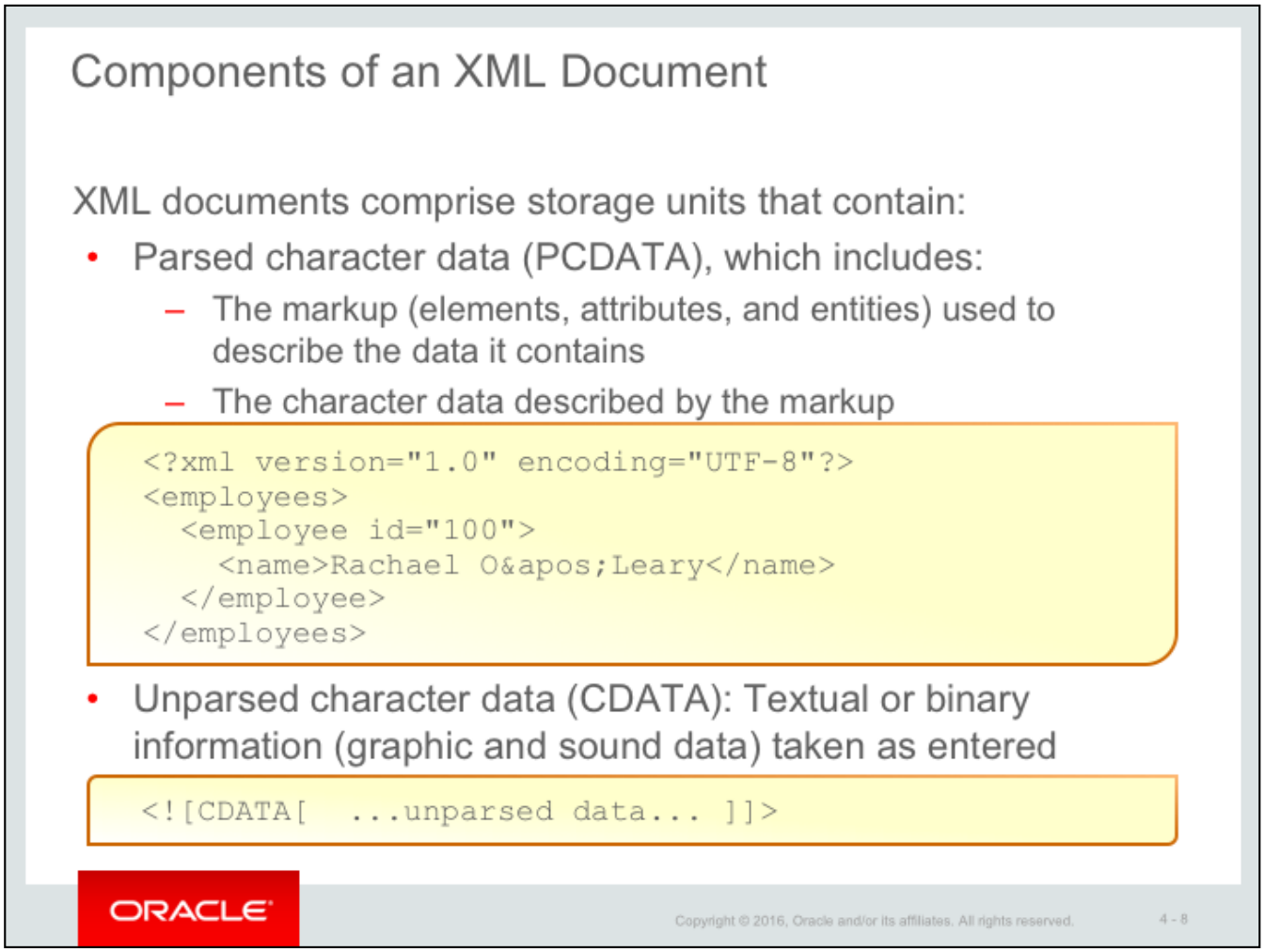


It is generally recommended that you start an XML document with an XML declaration. If it is present, it must be the first line in the document. Although the XML declaration resembles a processing instruction, it is really a simple declaration. In the XML 1.0 specifications, an XML document does not require the XML declaration. However, in the XML 1.1 specifications, the XML declaration is mandatory.

Note: Not all tools and XML Parsers support the XML 1.1 syntax.

The only attribute that is mandatory in the declaration is the `version` attribute. Additional attributes that can be added to the XML declaration include the following:

- `encoding` attribute, which is optional. By default, XML documents are encoded in the UTF-8 format of the Unicode character set.
- `standalone` attribute, which is optional and may be set to the `yes` or `no` value. If omitted, the value is assumed to be `no`. Set the `standalone` value to:
  - No, if an application requires an external DTD to determine proper document structure
  - Yes, when the document does not have a DTD, does not change the document content, or if the DTD is internal

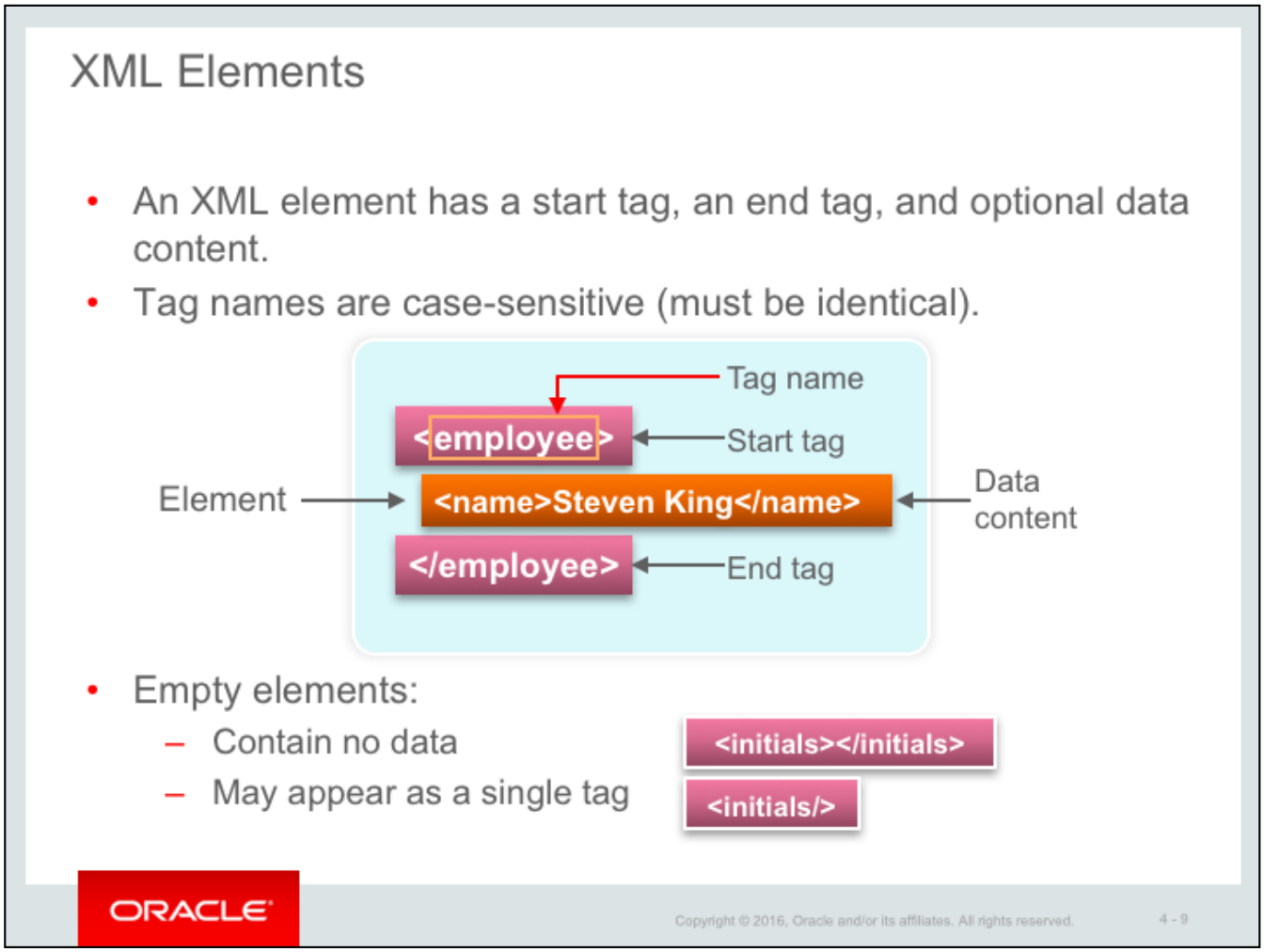


An XML document comprises storage units that contain parsed or unparsed data.

Parsed character data (`PCDATA`) is textual information comprising the following:

- The markup that describes the data it contains. Markup includes:
  - Elements to describe the data it contains such as the root element (`employees`) and its child elements (`employee, name`)
  - Attributes, which are name and value pairs (`id="100"`) included in the start tag of an element
  - The entities (`&apos;`) representing any character data substituted in place of their appearance
- The character data described by the markup components, for example:
  - The value `100` assigned to the ID attribute
  - The data Rachael O'Leary that is described by the `<name>` element
  - The `&apos;` entity, which represents the apostrophe (`'`) character

Note: The element tree in an XML document defines its layout and logical structure.



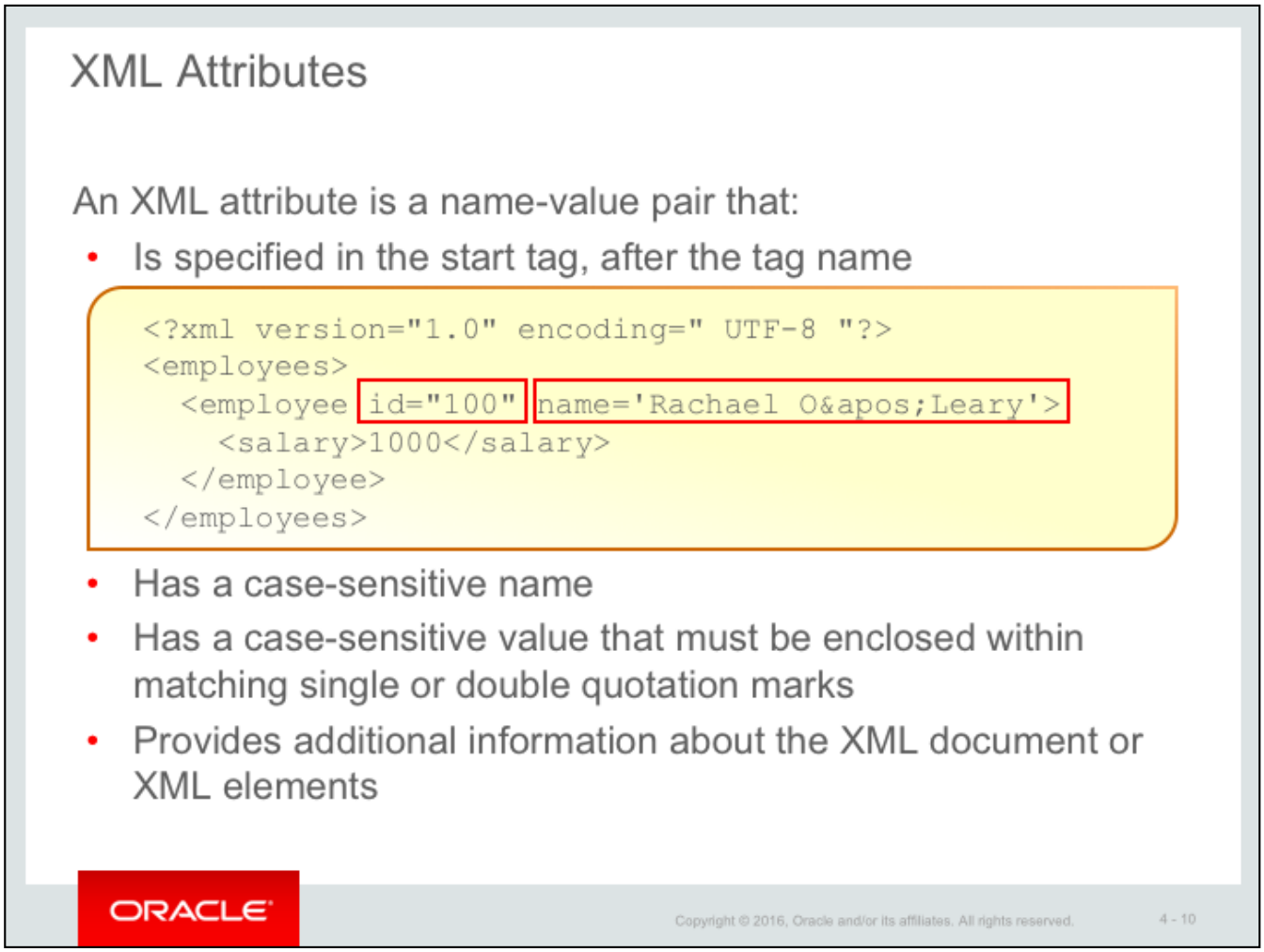
There is one root element, which is sometimes called the top-level or document element.

All elements:

- Must have matching start and end tags, or be a self-closing tag (an empty element)
- Can contain nested elements so that their tags do not overlap
- Have case-sensitive tag names that are subject to naming conventions (starting with a letter, no spaces, and not starting with the letters xml)
- May contain white space (spaces, tabs, new lines, and combinations of them) that is considered part of the element data content

Markup Rules for Elements

Every XML document must contain one root element (top-level or document element). XML documents are hierarchical in structure with elements nested within others to form a document tree. The start and end tags for elements must not overlap.



Attributes are simple name-value pairs that are associated with a particular element. XML attributes must be specified after the start tag of an element or after the tag name of an empty element.

Example: `<employee id="100" email="SKING"/>`

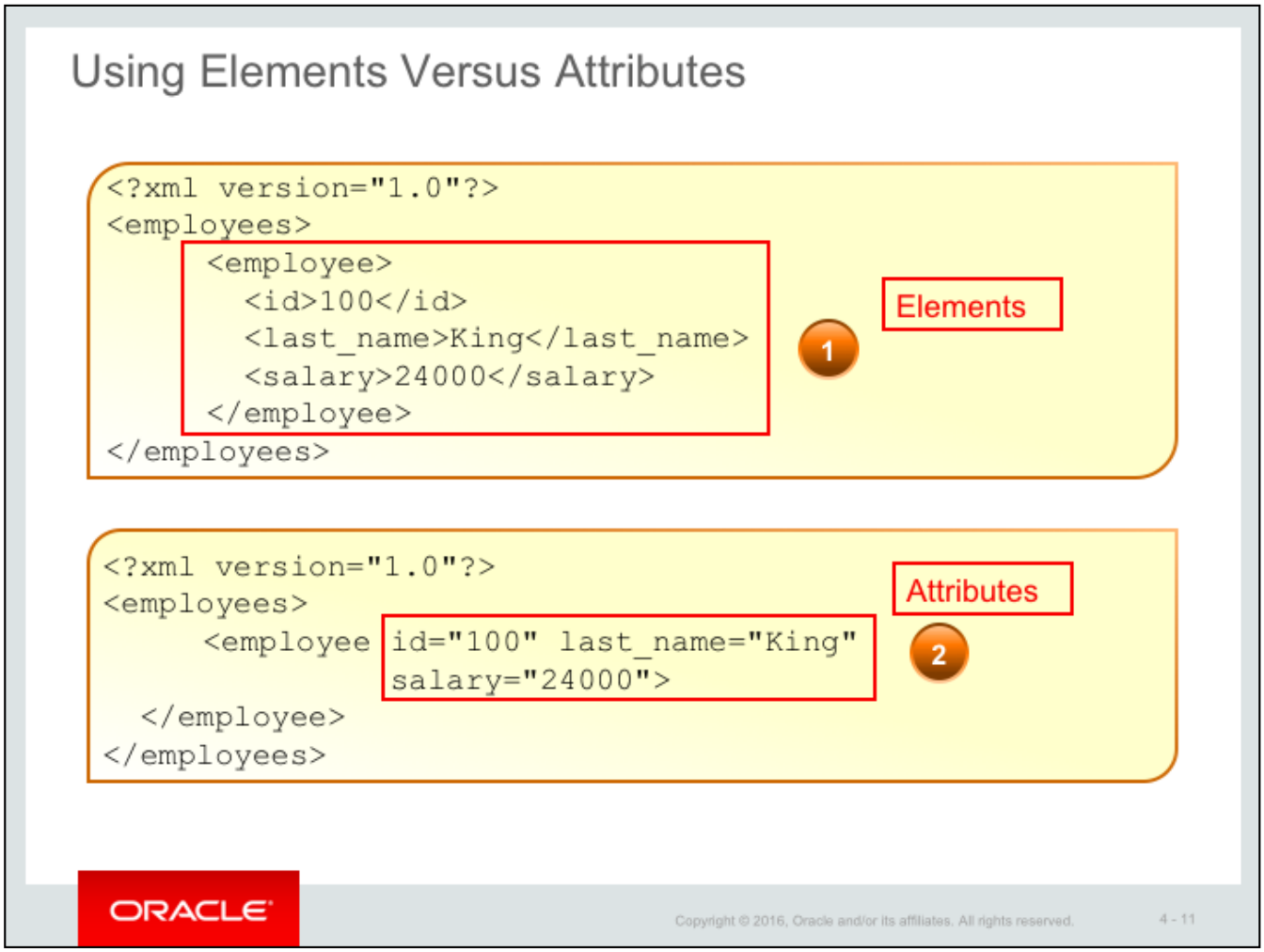
Attribute names are case-sensitive and follow the naming rules that apply to element names. In general, spaces are not used, but are allowed on either side of the equal sign. Attribute names should be unique within the start tag.

Attribute values must be within matching quotation marks, either single or double.

Naming Convention Rules for Elements and Attribute Names

- Can contain the letters `A–Z` and `a–z`
- Can contain the numbers `0–9`
- Can contain ideograms and non-English characters
- Can contain _ (underscore), - (dash), and . (period)
- Must start with letters or an "_"
- Cannot start with a number, punctuation character, or the letters xml (or XML, Xml, and so on)
- Cannot contain spaces

- Can use any name. No words are reserved.



In the slide, you see two examples:

1. In Example 1, `id`, `last_name`, and `salary` are defined as elements within the `employee` element, whose parent is the `employees` element.
2. In Example 2, `id`, `last_name`, and `salary` are attributes of the `employee` element, whose root element is `employees`.

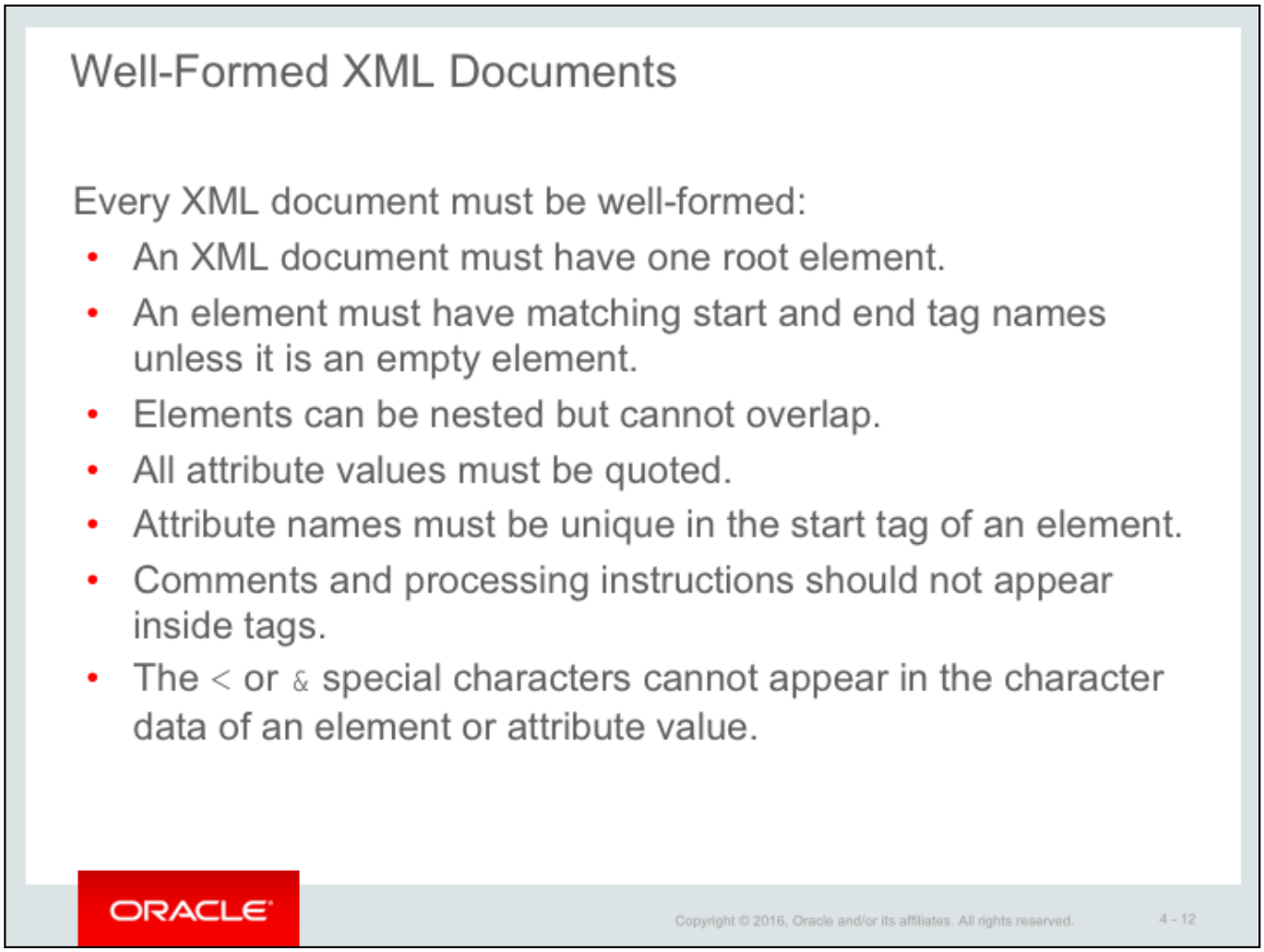
Elements and attributes accomplish approximately the same thing; that is, they describe data. The examples shown in the slide are both valid and communicate the same basic information about an employee.

Note: In general, elements are more robust and flexible to use than attributes.

The choice to use elements or attributes is often a matter of preference, but it takes some time and experience to determine the best style to use for different contexts. The easiest and most consistent approach is to use elements for data that the user of the document wants to see, and attributes for metadata representing additional information about the data that the document may require. The following are additional points to consider:

- Elements are more easily added as your requirements expand.
- Elements may be ordered, but attributes are returned unordered from the document.
- Elements can be structured and may contain other elements.

- Attributes are atomic; that is, they cannot be nested or have attributes of their own.



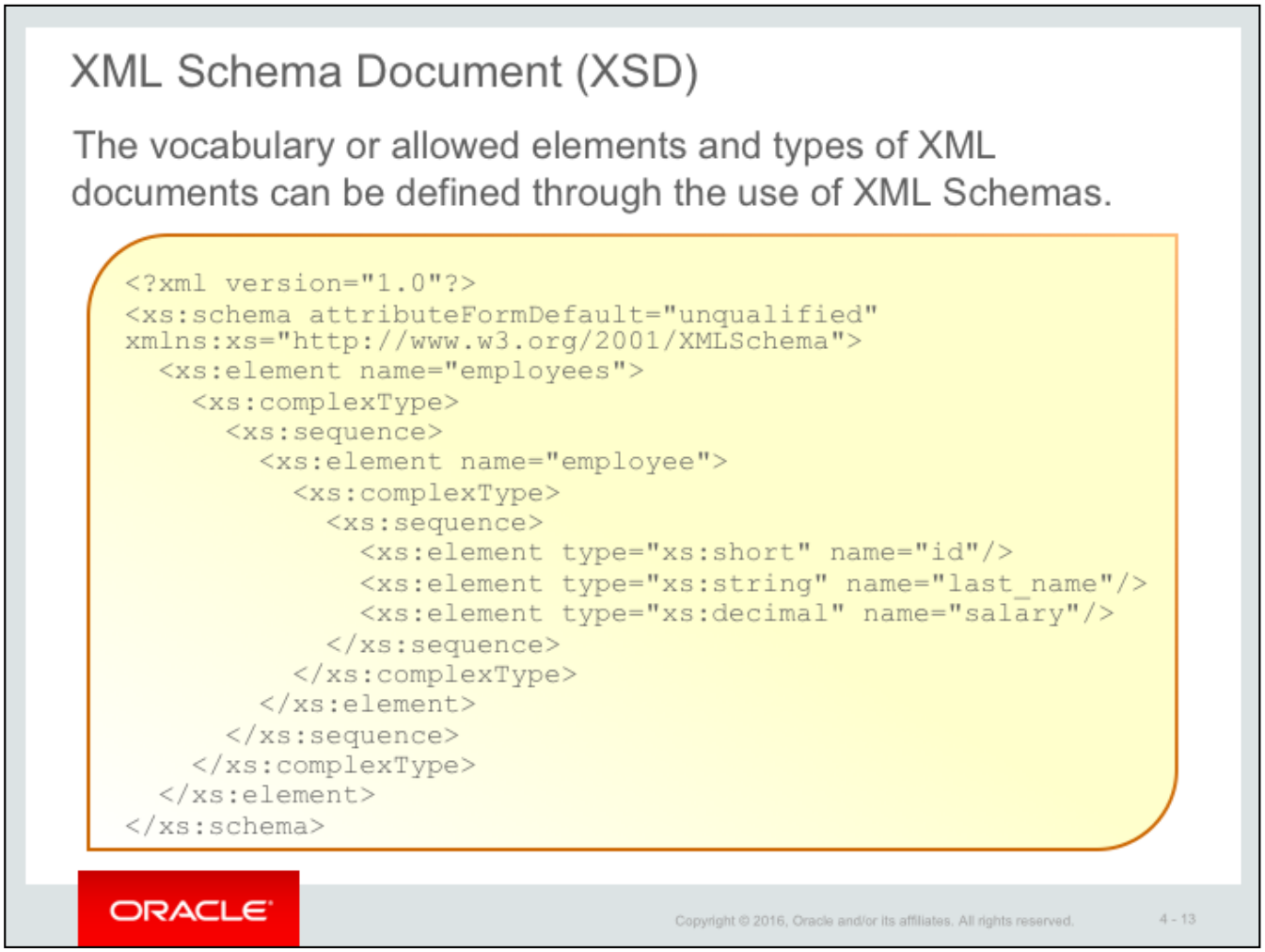
An XML document must be well formed to guarantee that it is correctly structured and adheres to the rules defined in the slide. The slide contains a list of the most common rules for well-formed documents, but it is not a complete list.

Note: The less-than (`<`) and ampersand (`&`) characters are special in XML and cannot appear as themselves within the character data part of an element, or the value of an attribute. In character data and attribute values, an XML Parser recognizes:

- The less-than (`<`) sign as a character that introduces the start tag of another element
- The ampersand (`&`) as an escape character before an entity name terminated by a semicolon (`;`)

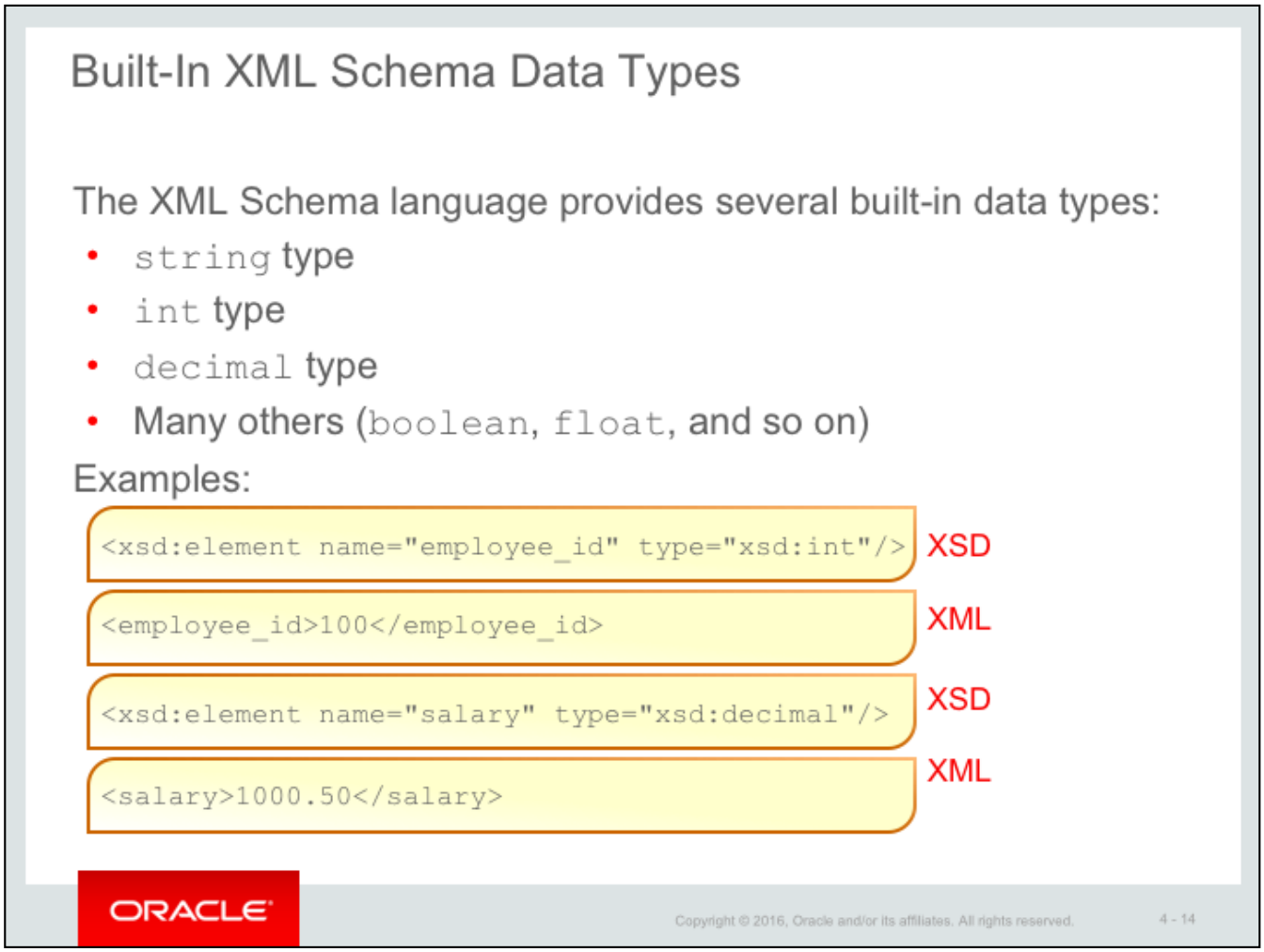
Therefore, to include special characters in the character data of an element or attribute value, you must use built-in XML entity names for the special characters. For example, use `<` to include the less-than character, and `&` for the ampersand character. The XML Parser replaces the entity reference with its textual or binary representation, as discussed earlier in this lesson.

Note: The XML 1.1 specification requires the XML declaration to appear at the beginning of the document. However, the declaration is optional in XML 1.0.



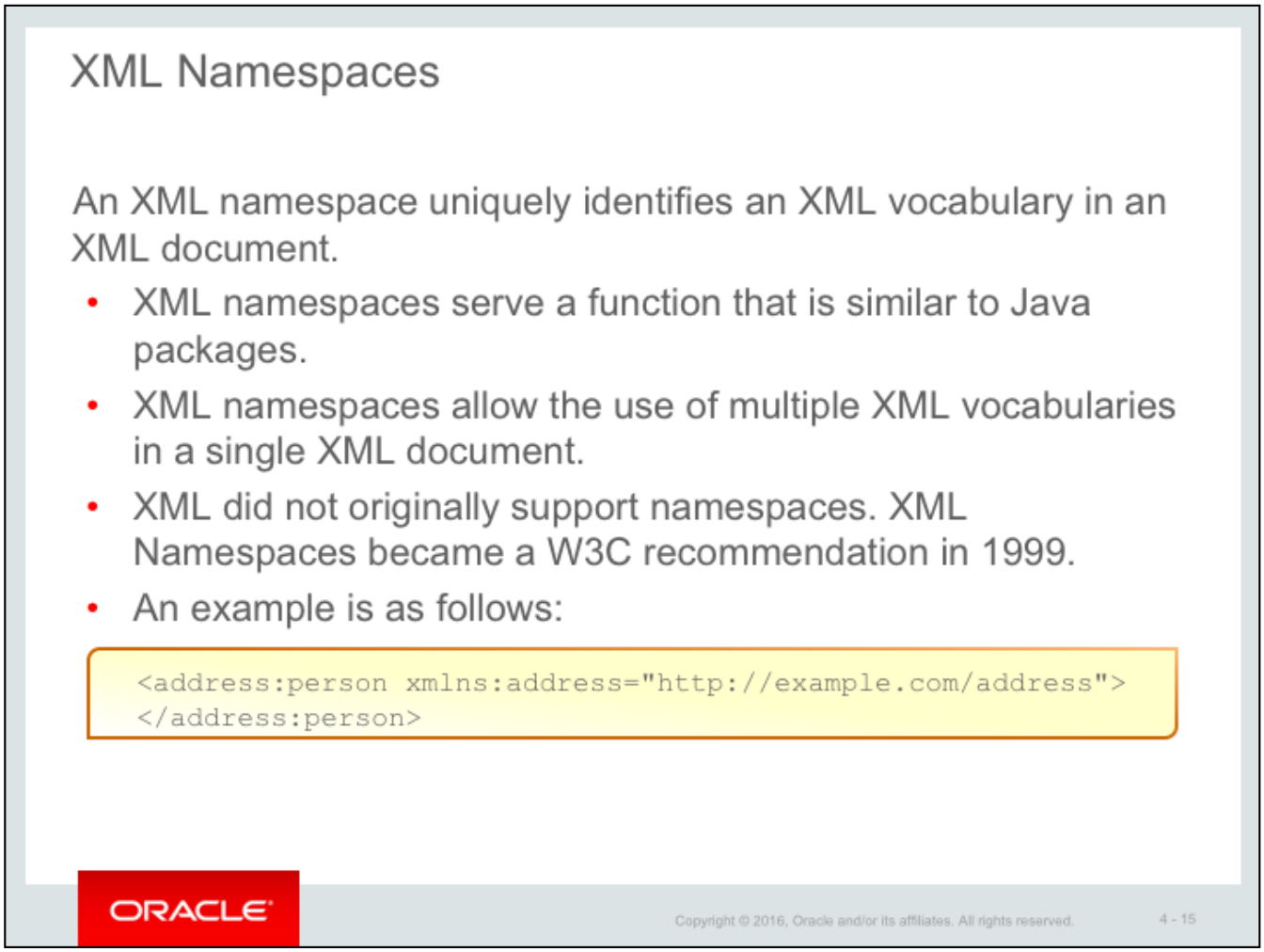
An XML document need not follow any rules beyond the well-formedness criteria laid out in the XML 1.0 specification. To exchange documents in a meaningful way, however, requires that their structure and content be described and constrained so that the various parties involved will interpret them correctly and consistently. This can be accomplished through the use of a schema.

- XML Schemas define the allowed elements and attributes along with their types.
- XML Schemas are themselves XML documents.
- XML documents specify the schemas they are constrained by.
- Developer created XML schemas, being XML documents, can be constrained by an XML schema – a schema for schemas. http://www.w3.org/2012/04/datatypes.xsd



The built-in data types that can be specified in the `type` attribute are listed in the following table:

SimpleType	Example or Explanation
string	“this is a string”
boolean	true, false
float	single-precision floating point
double	double-precision 64-bit floating point
decimal	123.45
Integer	-12345; 0; 12345
non-positive-integer	12345; 0
negative-integer	-12345
int	123456789
short	12345



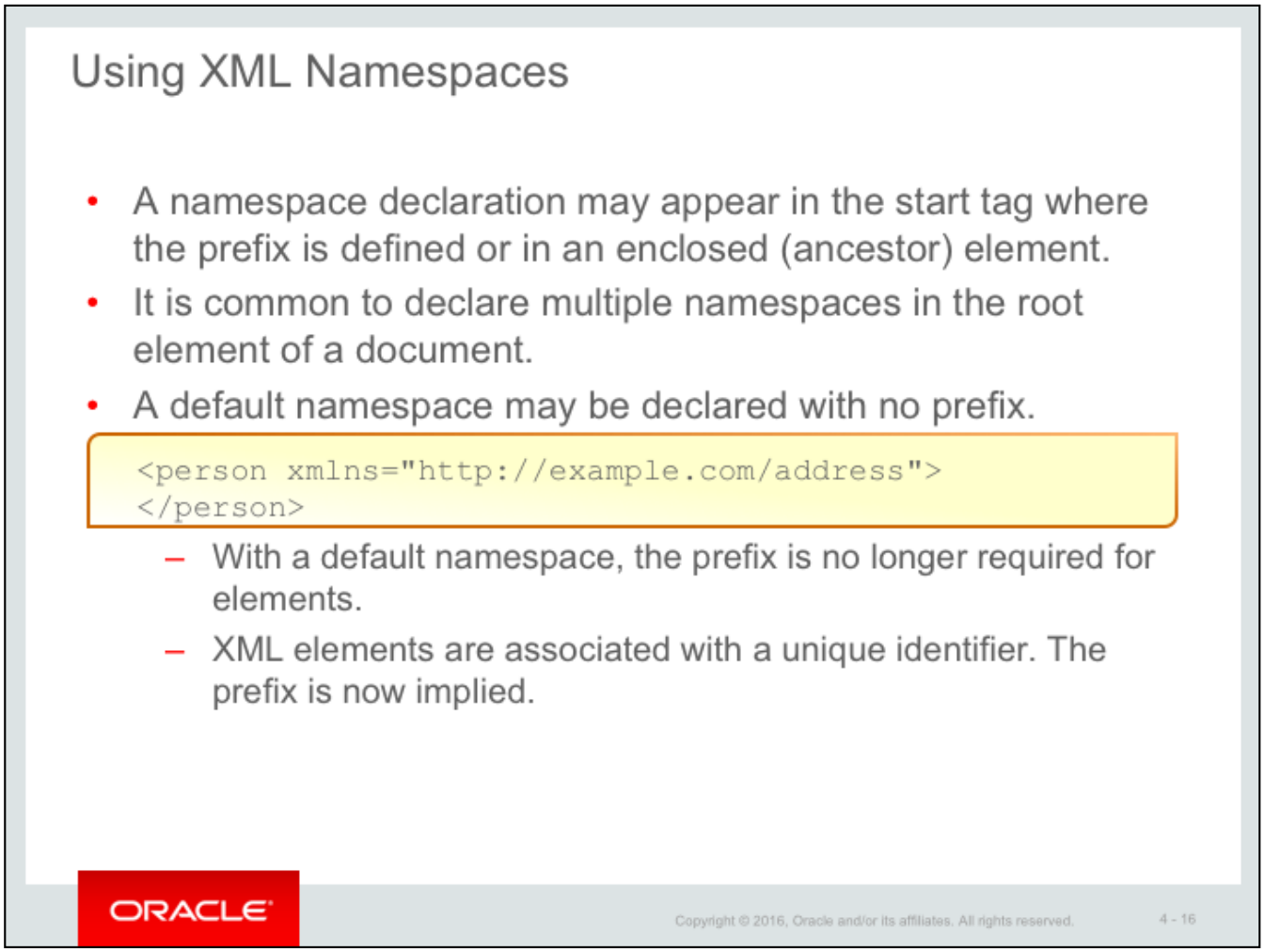
The example in the slide defines an XML namespace. The component parts can be defined as:

- `xmlns`: Is a special attribute that is used to define a namespace. The attribute can be used more than once in a definition as long as the prefix is different for each definition.
- `address`: Defines the prefix that is used for the vocabulary. Any elements that use this prefix are part of this vocabulary.
- `http://example.com/address`: Is a unique identifier that defines this vocabulary. This could be any string, for example: 123-46-6789. However, by convention, this is typically a URL. Generally, information about the vocabulary can be found at the URL provided.

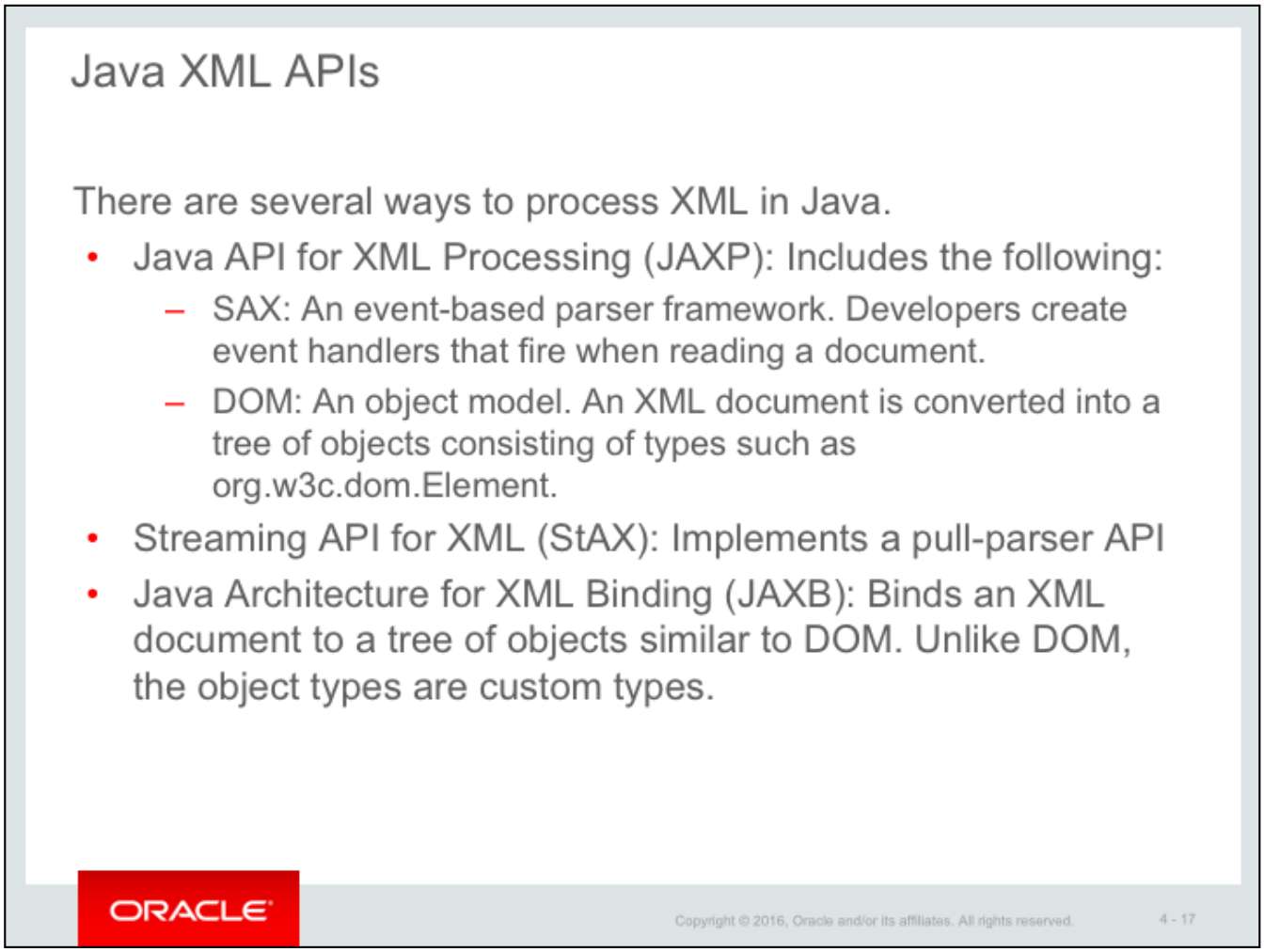
When the XML namespace is used, each tag name is preceded by the prefix: `<address:person>`

The tag name that follows the colon (person in the preceding example) is known as the local name. The <prefix:local> form creates a qualified name or QName.

The prefix should describe the vocabulary used or provide an acronym for the vocabulary. In this example, an address-related vocabulary is defined.

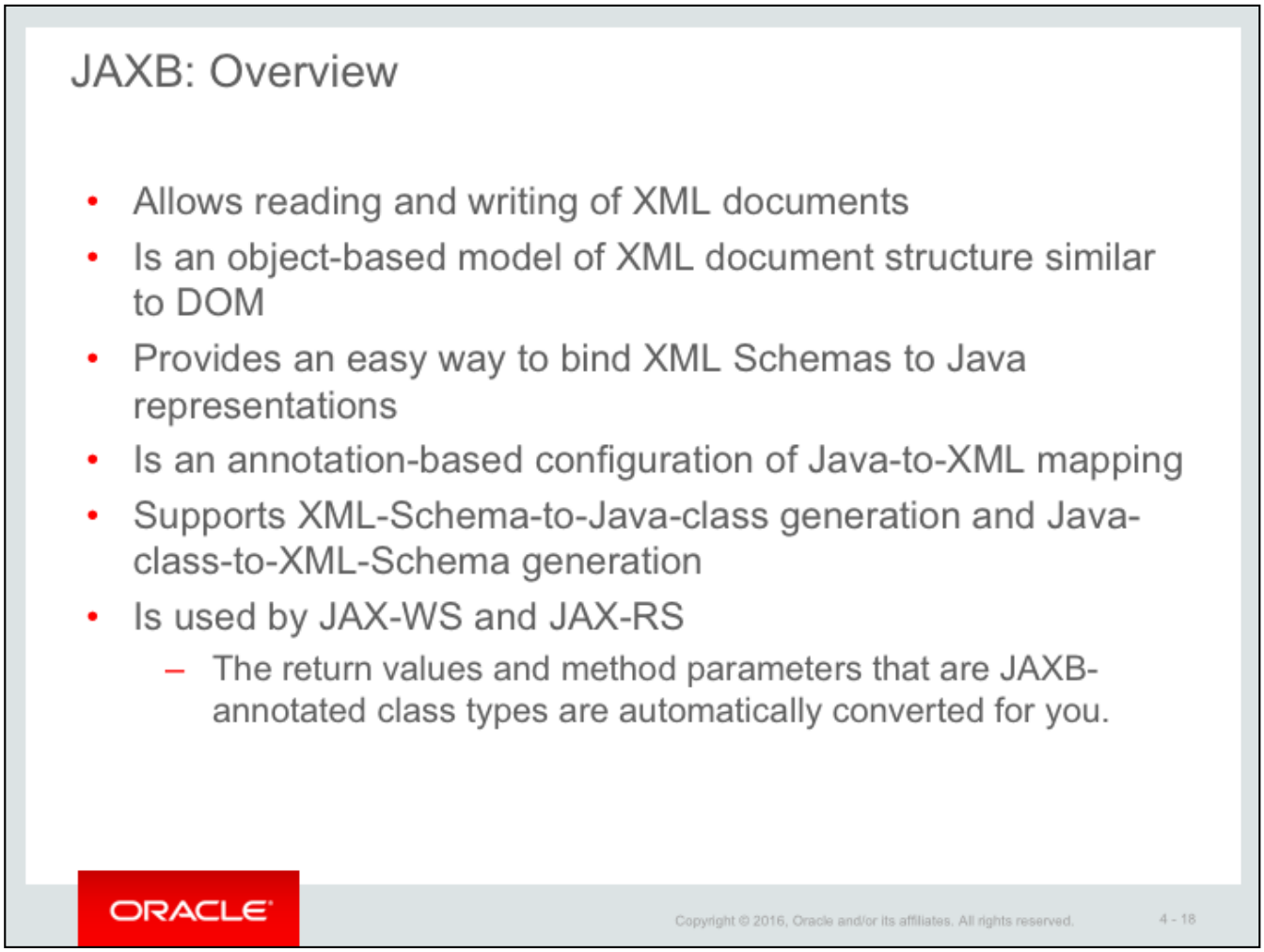


The best practice is to place all your namespace declarations at the top of your documents.



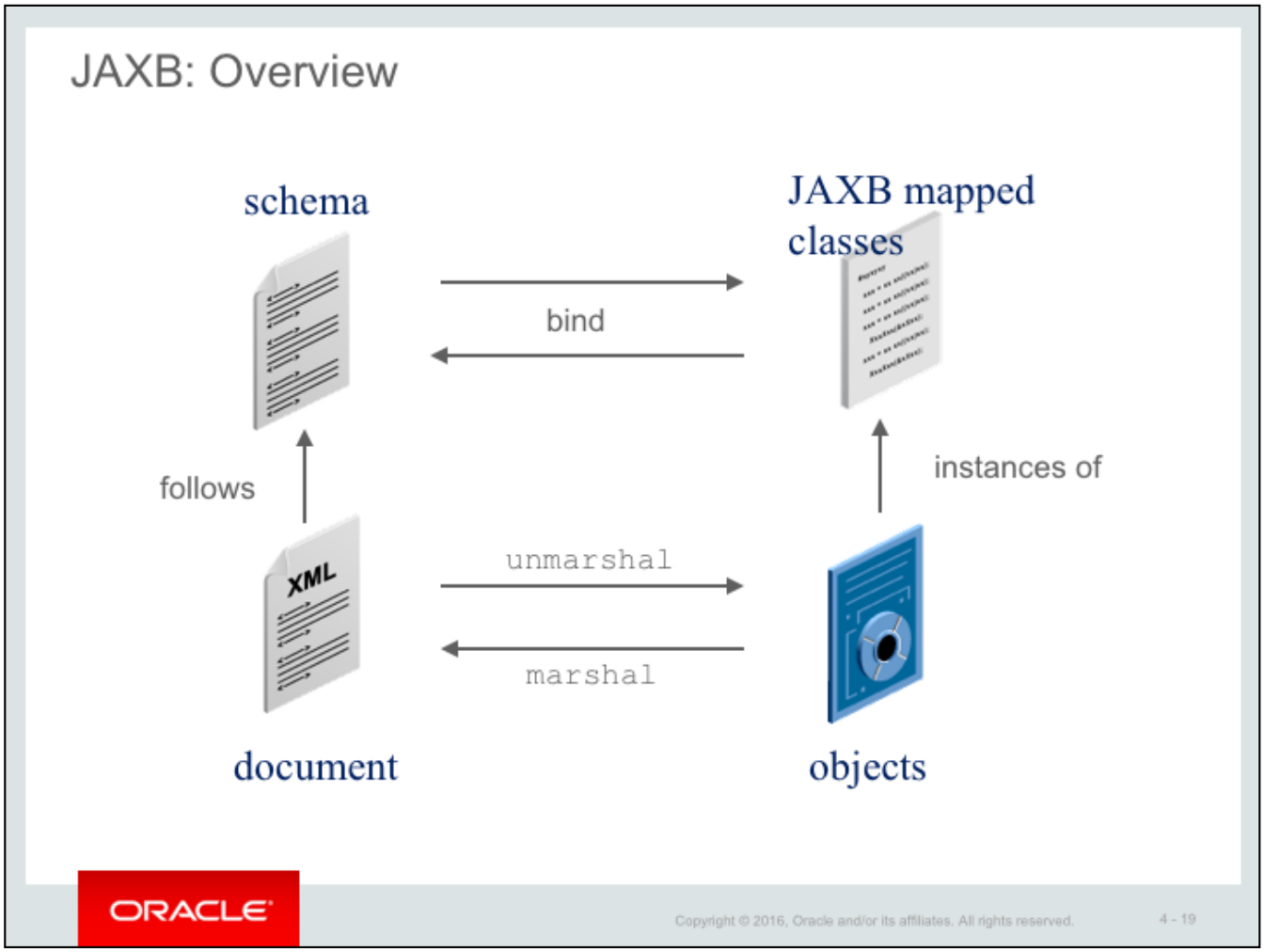
Both the SAX and DOM processing models are low level and can result in very verbose code. If you want to read XML documents and cannot use JAXB (because the XML syntax is not known at compile time), using StAX should be your first consideration.

- SAX (a streaming push parser model) is extremely fast and has a low memory footprint, but is probably the most challenging method to process XML in Java.
- DOM provides an object tree in memory and is easier to program against than SAX, but consumes more memory.
- StAX (a streaming pull parser model) is different than SAX in that instead of having the parser fire events, you request the next event by calling `next()`.



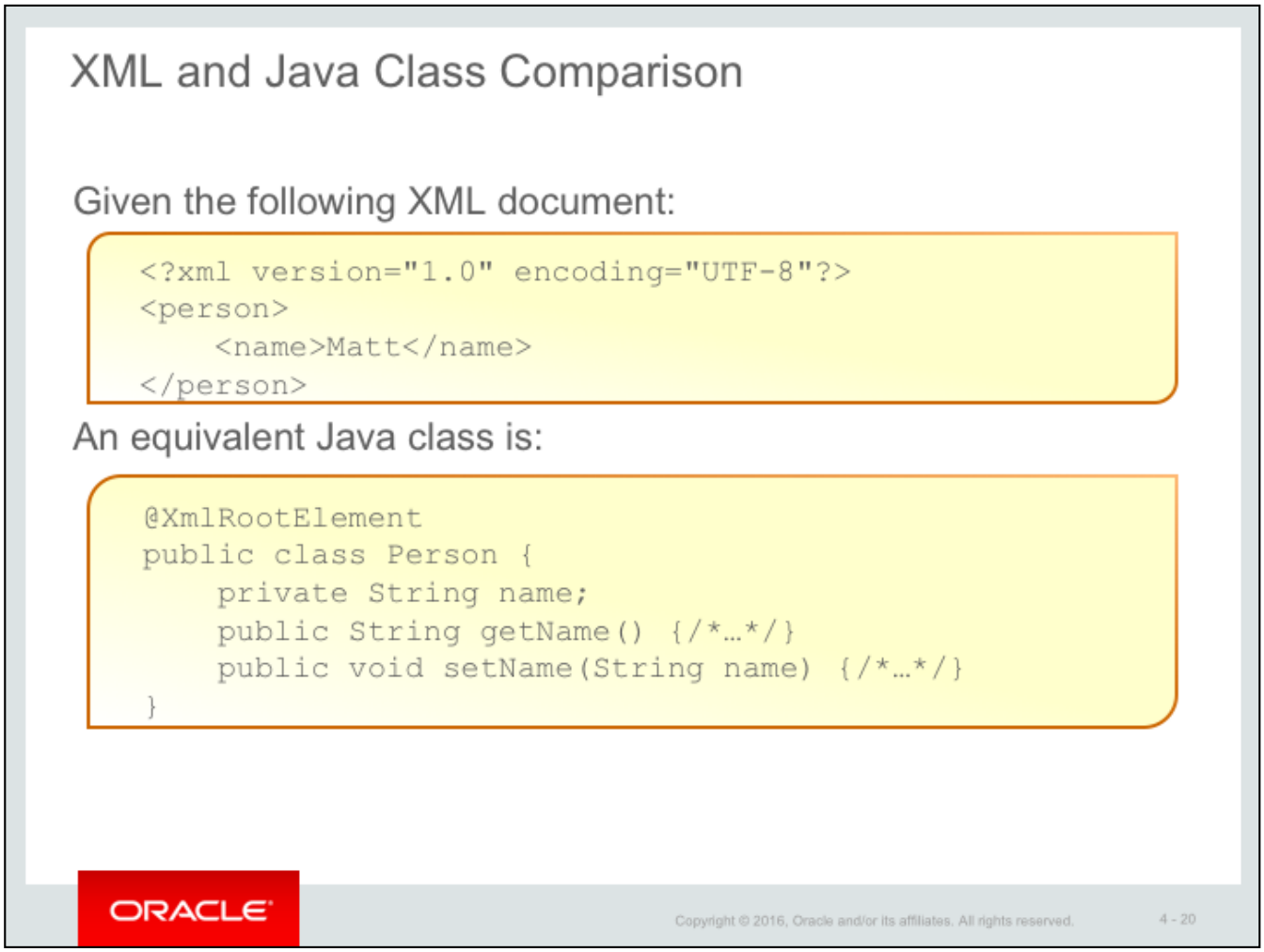
Binding

Binding a schema means generating a set of Java classes that represents the schema. All JAXB implementations provide a tool called a binding compiler to bind a schema (the way the binding compiler is invoked can be implementation-specific).



The general steps in the JAXB data binding process are:

- **Generate classes:** An XML Schema is used as input to the JAXB binding compiler to generate JAXB classes based on that schema.
- **Unmarshal:** The XML documents that are written according to the constraints in the source schema are unmarshalled by the JAXB binding framework.
- **Generate content tree:** The unmarshalling process generates a content tree of data objects instantiated from the generated JAXB classes. This content tree represents the structure and content of the source XML documents.
- **Validate (optional):** The unmarshalling process involves validation of the source XML documents before generating the content tree. Note that if you modify the content tree in the Process Content step, you can also use the JAXB Validate operation to validate the changes before marshalling the content back to an XML document.
- **Process content:** The client application can modify the XML data represented by the Java content tree by using the interfaces generated by the binding compiler.
- **Marshal:** The processed content tree is marshalled out to one or more XML output documents. The content may be validated before marshalling.



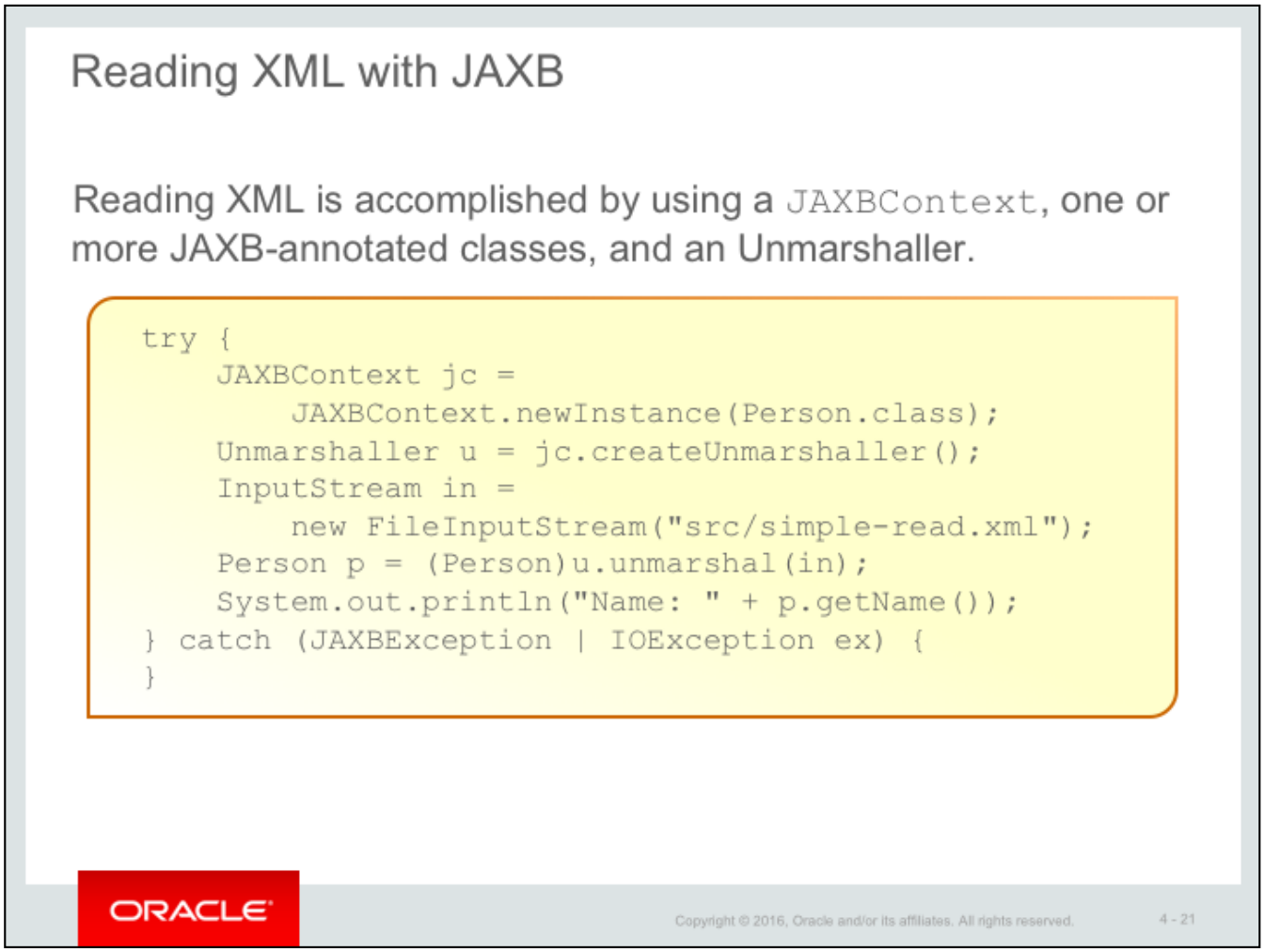
`@XmlRootElement`

The `@XmlRootElement` annotation is used to indicate that a class is used as a global (root) XML element.

```
@XmlRootElement(name="human")
public class Person { /* … */
```

Corresponds to an XML Schema of:

```
<xs:element name="human" type="person"/>
<xs:complexType name="person">
  <!-- ... -->
</xs:complexType>
```

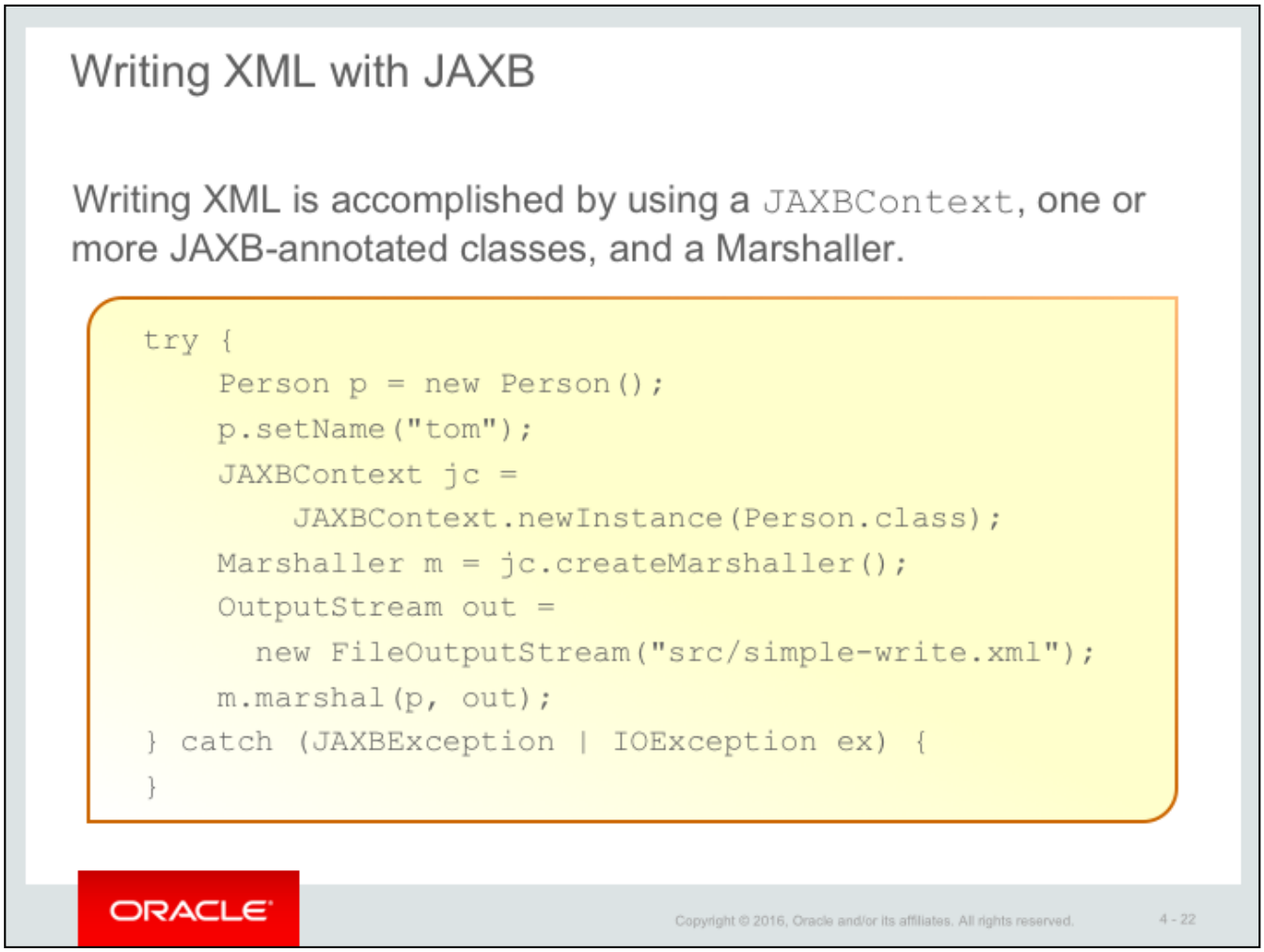


Unmarshalling the Document

Unmarshalling an XML document means creating a tree of content objects that represents the content and organization of the document. Unmarshalling provides a client application the ability to convert XML data into JAXB-derived Java objects.

To unmarshal an XML document, perform the following steps:

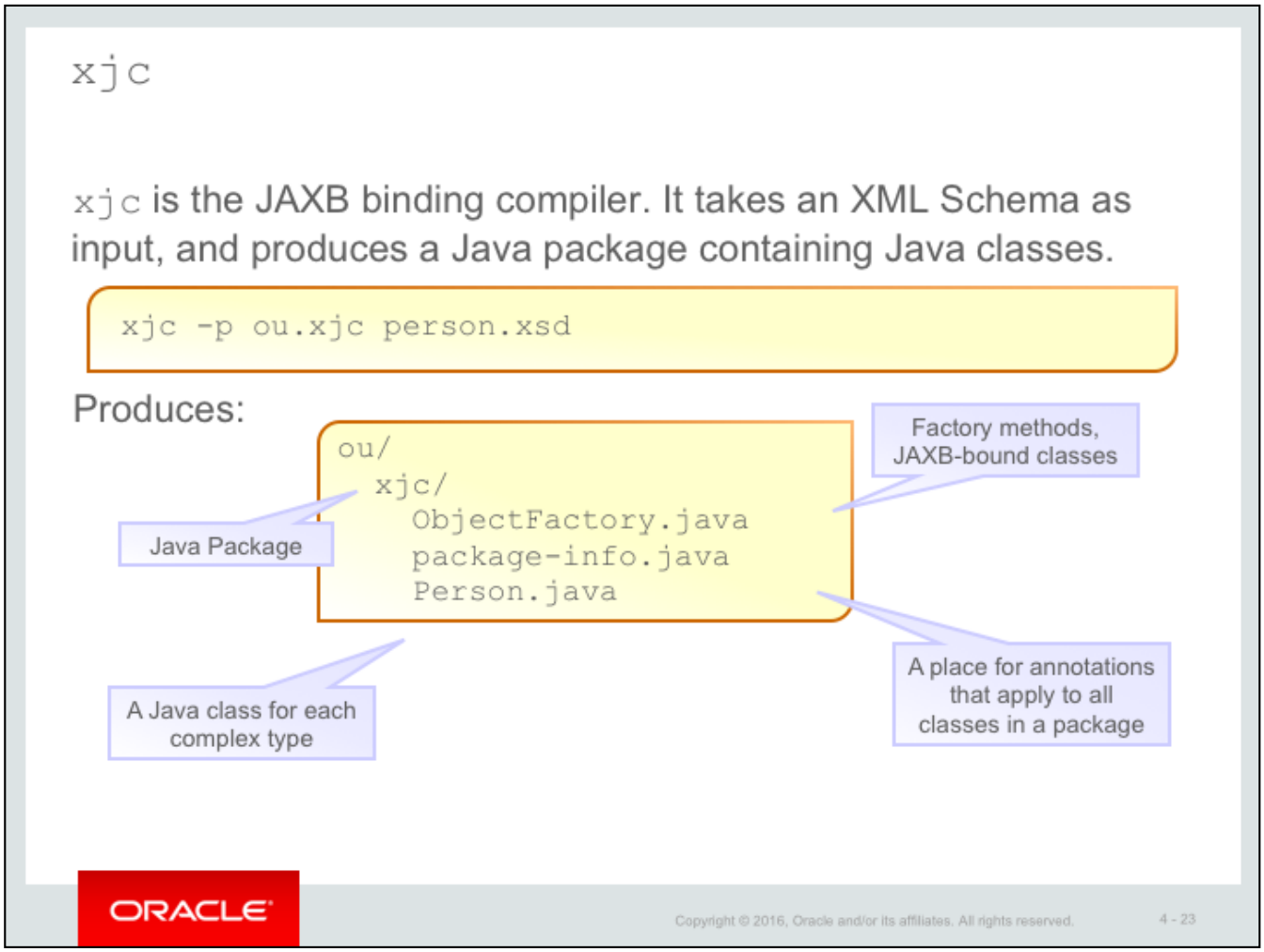
1. Create a `JAXBContext` object. This object provides the entry point to the JAXB API. When you create the object, you need to specify a context path. This is a list of one or more package names that contain interfaces generated by the binding compiler.
2. Create an `Unmarshaller` object. This object controls the process of unmarshalling. In particular, it contains methods that perform the actual unmarshalling operation.
3. Call the `unmarshal` method. This method does the actual unmarshalling of the XML document. In the example in the slide, the `Unmarshaller` instance (`u`) unmarshalls the XML data in `thesimple-read.xml` file.
4. Use the `get` methods in the schema-derived classes to access the XML data.



Marshalling

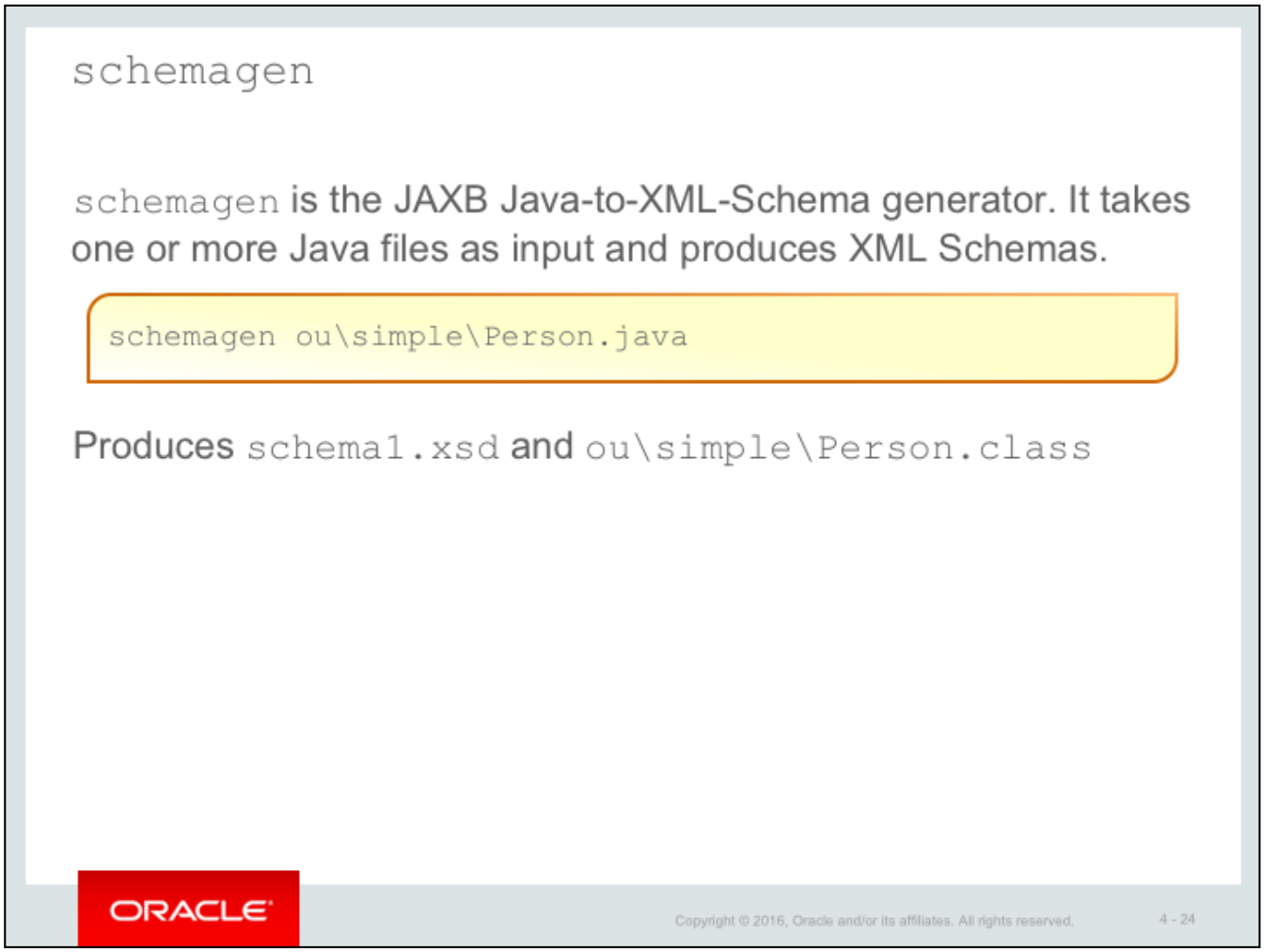
Marshalling is the opposite of unmarshalling. It creates an XML document from a content tree. Marshalling provides a client application the ability to convert a JAXB-derived Java object tree into XML data. To marshal a content tree, perform the following:

- Create a JAXBContext object and specify the appropriate context path—that is, the package that contains the classes and interfaces for the bound schema.
- Create a `Marshaller` object. This object controls the process of marshalling. In particular, it contains methods that perform the actual marshalling operation.
- Call the `marshal` method. This method does the actual marshalling of the content tree. When you call the method, you specify an object that contains the root of the content tree, and the output target.

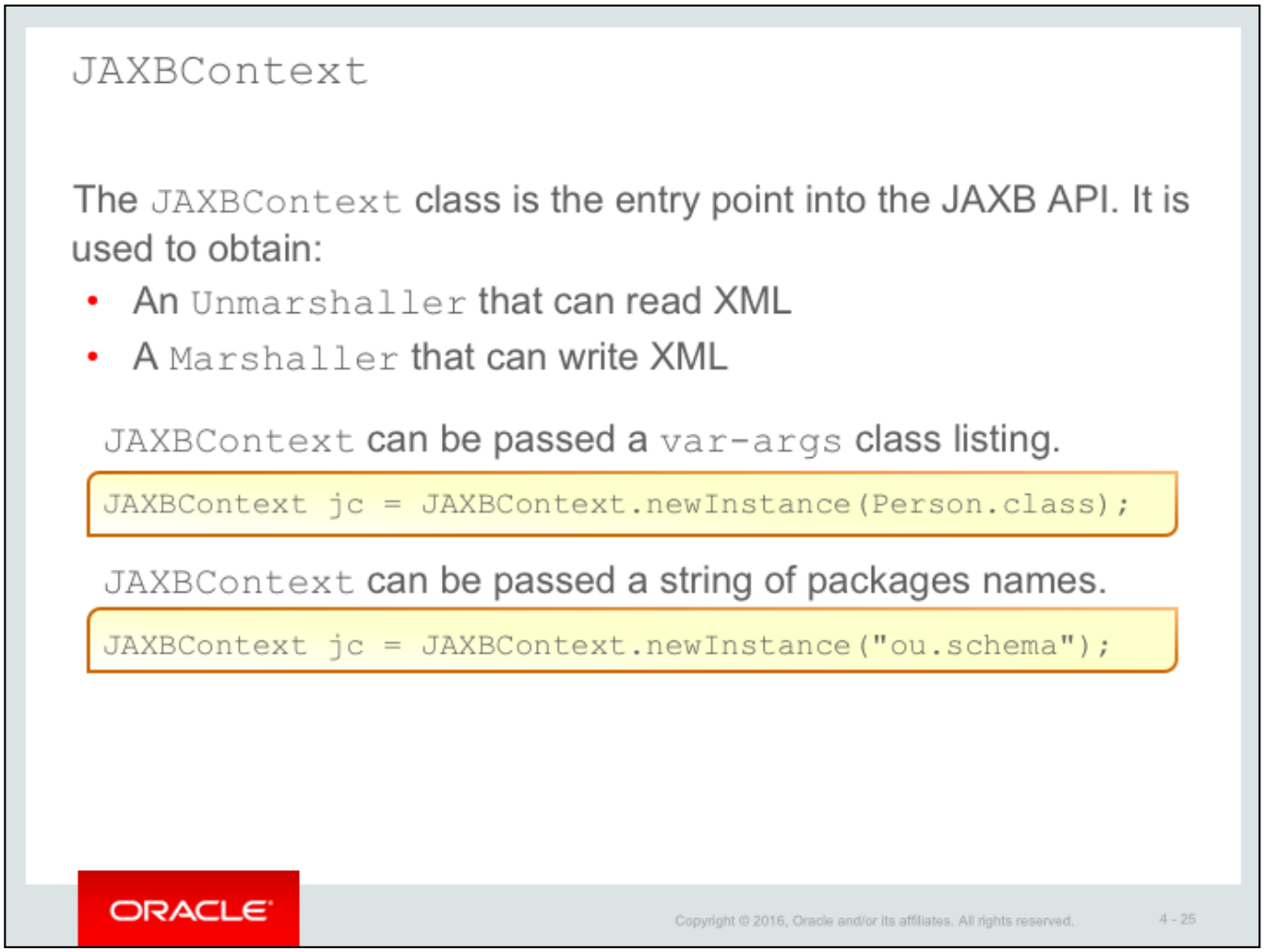


`person.xsd` XML Schema

```
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified" targetNamespace="urn:example">
    <xs:element name="person">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="name" type="xs:string"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```



JAXB does not require an XML Schema file for reading or writing unless you want to perform validation.



When supplying a string of package names, the following are applicable:

- Multiple package names can be supplied by separating the package names with colons
  `JAXBContext jc = JAXBContext.newInstance("ou.schema:ou.other");`
- Every package specified must contain either of the following:
  - An `ObjectFactory.java` that is annotated with `@XmlRegistry` and contains a factory method for each Java class that is bound to an XML type
  - A `jaxb.index` file. The `jaxb.index` file contains a list of JAXB-bound class names (short name without package), one class name per line.

Did you know that you can switch JAXB implementations? EclipseLink Moxy is one available JAXB implementation. Moxy has additional features such as support for external binding files and JSON reading and writing. When you create a new `JAXBContext` instance, several checks are made to see if you have configured an alternative JAXB implementation. For more information, see the section titled “Discovery of JAXB Implementation” at http://docs.oracle.com/javase/7/docs/api/javax/xml/bind/JAXBContext.html and http://www.eclipse.org/eclipselink/moxy.php.

JAXB Annotations

`@XmlRootElement`	**Indicates that a class is used as a global (root) XML element**
`@XmlType`	• Specifies the name of a `complexType` • Specifies the order of child elements
`@XmlAccessorType`	Controls which members are bound to XML
`@XmlElement`	Controls binding of class members to XML
`@XmlAttribute`	Maps a class member to an XML attribute
`@XmlValue`	Maps a class member to simple content within a complex type or a simple type
`@XmlEnum`	Enables Java enums to be mapped to XML enumerated values
`@XmlElements`	Is used to map a Java member to an XML Schema choice structure

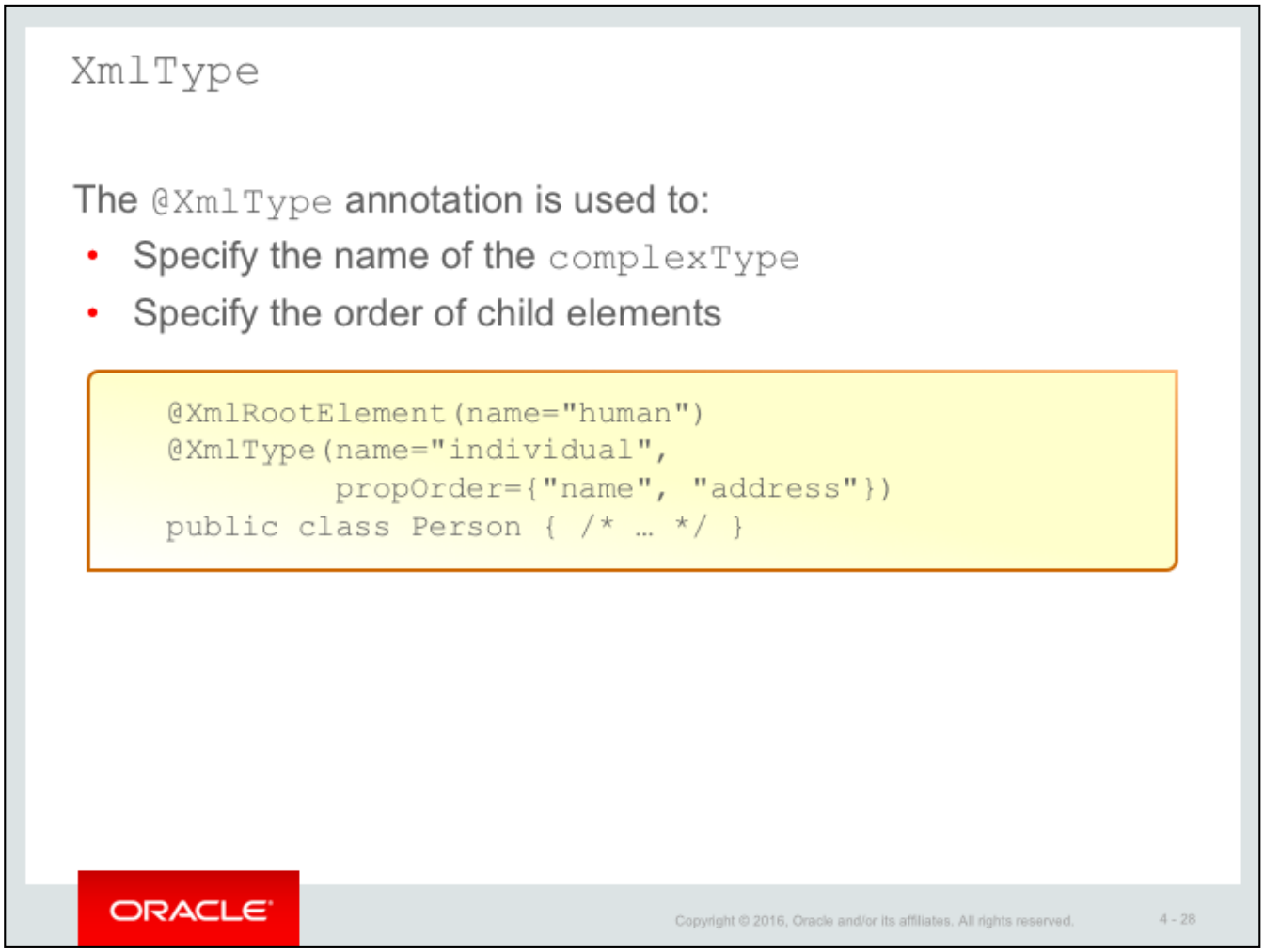
XmlRootElement

The `@XmlRootElement` annotation is used to indicate that a class is used as a global (root) XML element.

```
@XmlRootElement(name="human")
public class Person { /* … */
```

Corresponds to an XML Schema of:

```
<xs:element name="human" type="person"/>
<xs:complexType name="person">
  <!-- ... -->
</xs:complexType>
```



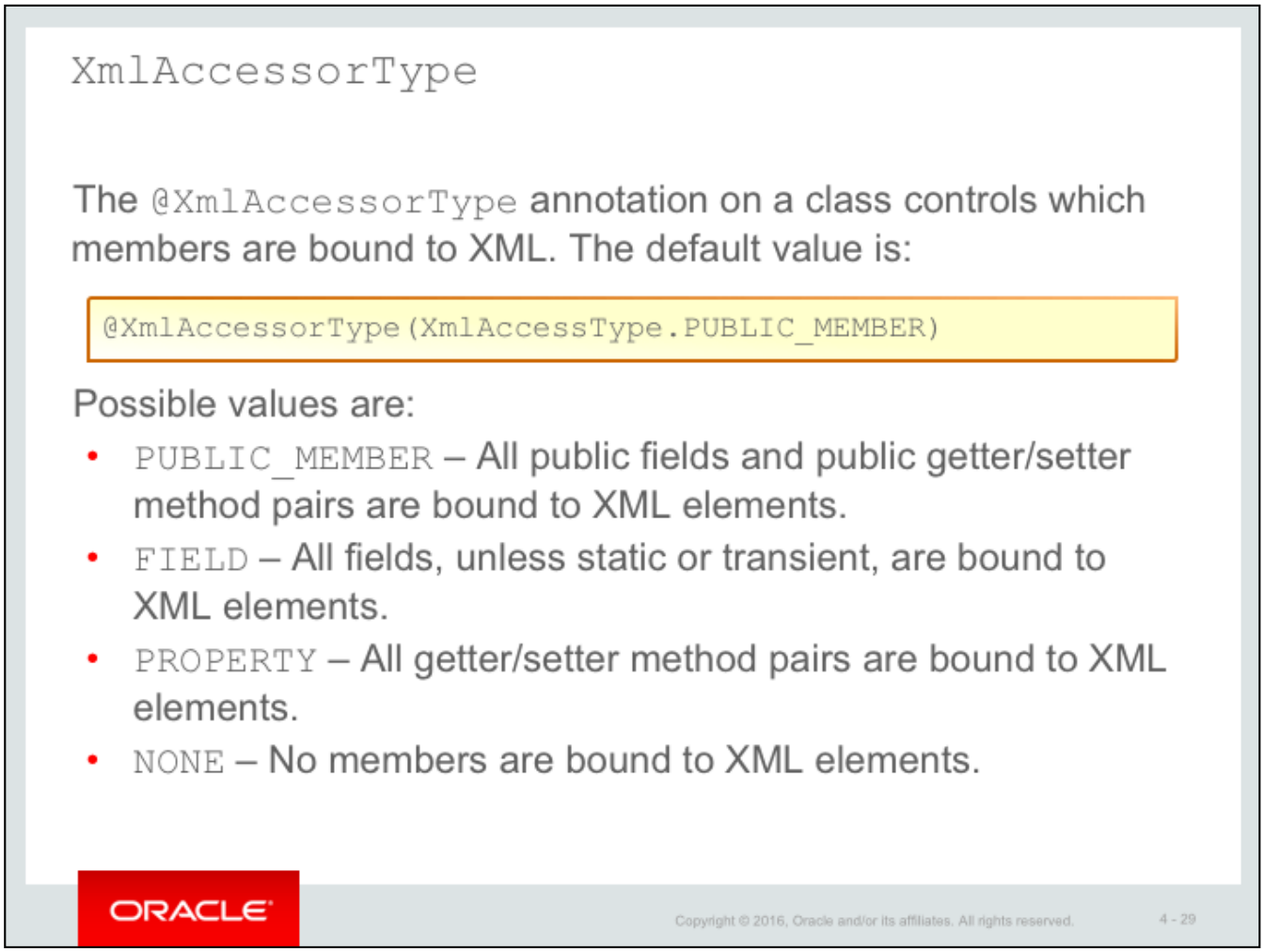
The `@XmlType` annotation can be used along with the `@XmlRootElement` or without it. When used without `@XmlRootElement`, the `@XmlType` annotation corresponds to a named global `complexType` but not a root element.

The code example in the slide corresponds to an XML Schema of

```
<xs:element name="human" type="individual"/>
<xs:complexType name="individual">
  <xs:sequence>
    <xs:element name="name" type="xs:string" minOccurs="0"/>
    <xs:element name="address" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

You can map a Java class to a nested or anonymous `complexType` by using a zero-length string for the name attribute value.

```
@XmlType(name="", propOrder={"name", "address"})
```



The XML accessor type defines default behavior. You may override the specified behavior on class members by applying JAXB annotations on fields or getter/setter method pairs. The `@XmlTransient` annotation on a field, or getter/setter, indicates that the member should not be mapped to XML. Using other annotations such as `@XmlValue`, `@XmlAttribute`, or `@XmlElement` instructs JAXB to map the member to XML even if the XML accessor type excludes the member from mapping.

XmlElement

The `@XmlElement` annotation is used to control binding of class members to XML.

```
@XmlRootElement(name="human")
@XmlAccessorType(XmlAccessType.NONE)
public class Person {
       @XmlElement(name="first-name", required=true)
       private String name;
```

- Map the field to XML even though the `XMLAccessorType` is `NONE`.
- The XML element name will be `first-name` instead of `name`.
- The `minOccurs` value is left at the default value of 1 instead of adding `minOccurs="0"`.

The XSD will contain:

```
<xs:element name="first-name" type="xs:string" minOccurs="1"/>
```

XmlAttribute

The `@XmlAttribute` annotation maps a class member to an XML attribute.

```
@XmlRootElement()
public class Person {
    @XmlAttribute
    public String name;
    public String address;
}
```

Corresponds to the following XML structure:

```
<person name="matt">
    <address>221B Baker Street</address>
</person>
```

XmlValue

The `@XmlValue` annotation maps a class member to simple content within a complex type or a simple type where possible.

```
@XmlRootElement()
public class Person {
    @XmlAttribute
    public String name;
    @XmlValue
    public String address;
}
```

Corresponds to the following XML structure:

```
<person name="matt">221B Baker Street</person>
```

Enumerations

Java enums can be mapped to XML enumerated values by using the `@XmlEnum` annotation.

```
@XmlType
@XmlEnum
public enum ProjectState {
    @XmlEnumValue("0")
    LATE,
    @XmlEnumValue("1")
    REALLY_LATE;
}
```

Java identifiers cannot start with a number but XML enumerated values can.

When running `xjc` on an XML Schema to generate Java classes, if you have enumerated values that start with numbers, they will be renamed. For example, "`1`" becomes "`value1`".

XmlElements

An `@XmlElements` annotation is used to map a Java member to an XML Schema choice structure.

```
@XmlElements(value = {
    @XmlElement(name = "pie",
    type = Pie.class,
    required=true),
    @XmlElement(name = "ice-cream",
    type = IceCream.class,
    required=true)
})
public Object obj;
```

Errors and Validation

JAXB does not perform strict validation checking by default.

- When reading XML, unexpected elements and attributes that are not mapped to Java elements are ignored.
- When reading XML, malformed XML will cause a `javax.xml.bind.UnmarshalException` when calling `unmarshal`.
- If you want to keep track of validation failures, you can attach a `ValidationEventHandler` to the `Unmarshaller`.

```
Unmarshaller u = jc.createUnmarshaller();
ValidationEventCollector vec =
    new ValidationEventCollector();
u.setEventHandler(vec);
```

NetBeans JAXB Support

- NetBeans supports the Ant `xjc` task through an XML Binding file type that is located in the XML new file category.
- However, there is no support for `schemagen`.
- You may run it on the command line, modify your `build.xml`, or generate the schema programmatically.

```
JAXBContext context =
JAXBContext.newInstance("mypackage");
context.generateSchema(new SchemaOutputResolver() {
  public Result createOutput(String namespaceUri,
                             String fileName) throws
IOException {
    return new StreamResult(new
File("src/myschema.xsd"));
  }
});
```

Quiz

The tags and attributes that are allowed in an XML document can be constrained by:

a. DTDs
b. JAXB
c. XML Namespaces
d. XML Schemas

Answer: a, d

Quiz

JAXB performs XML Schema validation by default.

a. True
b. False

Answer: b

Quiz

The default accessor type that is used by JAXB to obtain the state of an object is:

a. None. The class element must be annotated with a JAXB annotation to read class instance fields.
b. Fields. All fields, regardless of access level, are read.
c. Properties. Getter and setter methods are used.
d. Public members. Public fields and properties are used.

Answer: d

Summary

In this lesson, you should have learned how to:

- Describe the benefits of XML
- Describe the components of an XML document
- Define a simple XML Schema
- Define the benefits of JAXB
- Create XML data by using JAXB
- Unmarshal XML data
- Marshal XML data

Practice 4: Overview

This practice covers the following topics:

- Practice 4-1: Marshalling a Java Class to an XML File
- Practice 4-2: Marshalling a Java Collection to an XML File
- Practice 4-3: Generating a Schema from Java Source Files