Parsing XML and JSON

- XML
- JSON

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Text Formats

- There are several popular textual formats that applications use to exchange data, especially in REST or SOAP Web services
- In this chapter, we will consider:
    - XML
    - JSON

While you could use just plain Readers and Writers and parse these formats by hand, it's more convenient to use an existing parsing API and/or library that will handle the parsing details for you so you can concentrate on business logic.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is XML?

- Like HTML, XML is a derivative of SGML and uses tags to markup text
- Unlike HTML, XML is designed for data, not presentation:
  - No predefined tags
  - No formatting elements
  - Stricter syntax rules
  - Optional validation against an application-defined vocabulary
- XML is administered by the World Wide Web Consortium (W3C) www.w3c.org

XML became an official W3C recommendation in 1998 and has since had a major impact on computing. In this chapter, we will discuss how XML fits into the J2EE picture.

The important difference between XML and HTML is that the latter was designed as a presentation language and is composed of tags that control display of data. On the other hand, XML is a data markup language and has no presentation tags at all. In fact, XML defines no tags! In other words, every XML application must define its own tags so that the tags describe the data.

Also note that XML syntax rules are stricter than those of HTML. For example, all tags must be explicitly closed. In addition, XML is case sensitive.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use XML?

- XML is self-describing textual data that can easily transformed into something else:
  - XML into HTML
  - XML into PDF
  - XML into another XML format
  - . . .
- Possible uses:
  - Business to business commerce
  - Common content for publishing, both hard-copy and on the Web
  - Configuration files
  - As part a remote method call infrastructure (SOAP and Web services)

One of XML's advantages is its chameleon-like ability to be "morphed" into something else. That makes it ideal for applications where different companies need to share data, for example, electronic data interchange, where trading partners may store things like purchase orders in different formats. Publishing companies, such as magazines can use XML to hold their content and then transform it into HTML for display on the Web or PDF for hard-copy.

Another use, which we've already seen in this course, is for configuration files that are easily parsed. Remember the .TLD files for custom actions?

Finally, a hot topic in computing these days is the Simple Object Access Protocol and Web services, which use XML as an infrastructure technology.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic XML Syntax Rules

- Single root element
- End tags required
- Case sensitivity
- Attribute values must be quoted
- Elements must nest properly

Content that follows these rules is said to be *well formed*

This page lists some of the fundamental syntax rules for well-formed XML content. It's not an exhaustive list, but covers the most common errors.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample XML Content

```
1    <?xml version="1.0"?>
2    <roster name="JSP Programming">
3      <student>
4        <name>Paul Westerberg</name>
5        <studentID>43</studentID>
6        <gpa>3.56</gpa>
7      </student>
8      <student>
9        <name>Bob Stinson</name>
10       <studentID>5678</studentID>
11       <gpa>2.55</gpa>
12     </student>
13   </roster>
```

This listing shows some simple XML content that follows all of the syntax rules. In line 1, we write the XML declaration, which is recommended by the specification. Then in line 2, we open the root element, "roster", which we close in line 13. Well formed XML content must define a single root element -- all other elements must nest within the root.

In lines 3 to 7, we define a "student" element that describes a student in a class. As child elements, we specify the data for that student -- note how the tag names describe the data itself. Then in lines 8 to 12, we define another student.

Note that overall, all of the tags are closed, nest properly and have matching case on start and end tags.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

More Sample XML Content

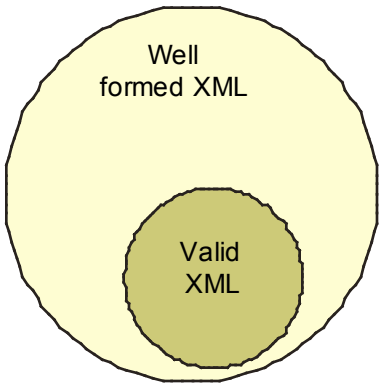
- Is this content well formed? Can you find any syntax errors?

```
1    <purchase>
2      <Amount currency=usd>
3         33.44
4      </amount>
5      <exempt/>
6      <date><time>10:30</date></time>
7    </purchase>
8
9    <purchase>
10     <exempt/>
11   </purchase>
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Well Formed Versus Valid

- XML that follows the basic syntax rules is *well formed*
- XML that adheres to an optional vocabulary is *valid*
- To define the vocabulary, you write either a *Document Type Definition (DTD)* or an *XML Schema*



2 - 8

Well formedness is a prerequisite for validity.

Validation requires that someone write the rules that govern the structure, order, type and multiplicity of elements and attributes in the content. You can write these rules, referred to as a vocabulary or grammar, using either the older DTD syntax or the newer, more powerful XML Schema syntax. We will cover only XML Schema in this chapter.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

A Sample Schema

```
1    <?xml version="1.0"?>
2    <xs:schema
3       xmlns:xs="http://www.w3.org/2001/XMLSchema">
4
5      <xs:element name="person">
6        <xs:complexType>
7          <xs:sequence>
8            <xs:element name="firstname"
9                        type="xs:string" />
10           <xs:element name="lastname"
11                       type="xs:string" />
12         </xs:sequence>
13       </xs:complexType>
14     </xs:element>
15
16   </xs:schema>
```


Here we show a simple schema that resides in a file named "simple.xsd" (xsd is the conventional extension for schema documents).

It's not so important at this point that you understand the syntax completely, but note that the schema itself is an XML document and uses a standard namespace. The conventional namespace prefix is "xs:", but you can use whatever you like.

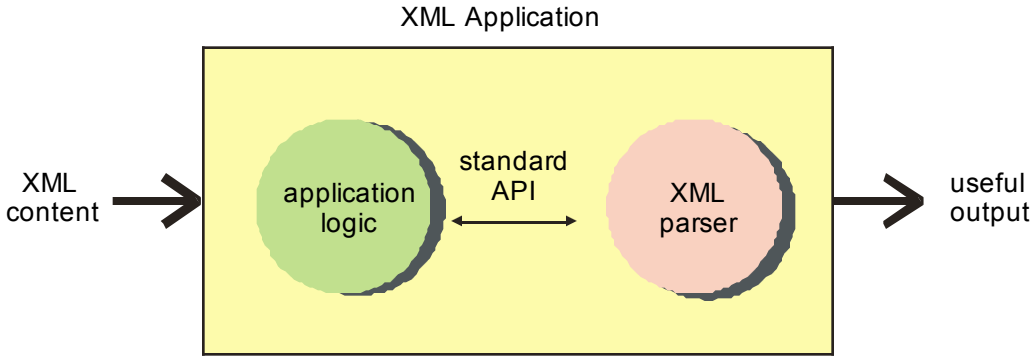
This schema defines a single element named "person" (line 5) that is composed of two child elements: "firstname" and "lastname". The two child elements contain only the string datatype, which means that the elements contain only text (similar to DTD PCDATA).

Exercise for the student: Write the DTD that corresponds to this schema.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Writing an XML Application

- The JVM includes XML parsers that do the "dirty work" of reading the XML content
- There are a few standard APIs such as the Document Object Model (DOM), the Simple API for XML (SAX) and the Streaming API for XML (StAX)



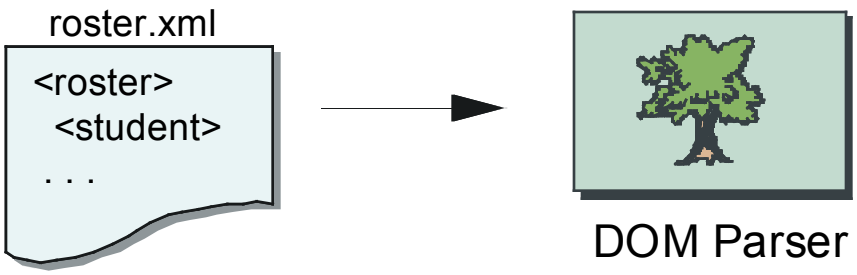
Here we show the structure of a typical XML program -- it accepts XML as input, parses the XML and produces some useful output.

Parsing is the act of determining where elements start, end and so forth -- it's a tedious job and it would be bad if every application had to reinvent that wheel. So instead, modern JVMs include support for XML parsing as a basic service.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to DOM

- The Document Object Model is an application programming interface that an XML parser can implement
- DOM is administered by the World Wide Web consortium (W3C)



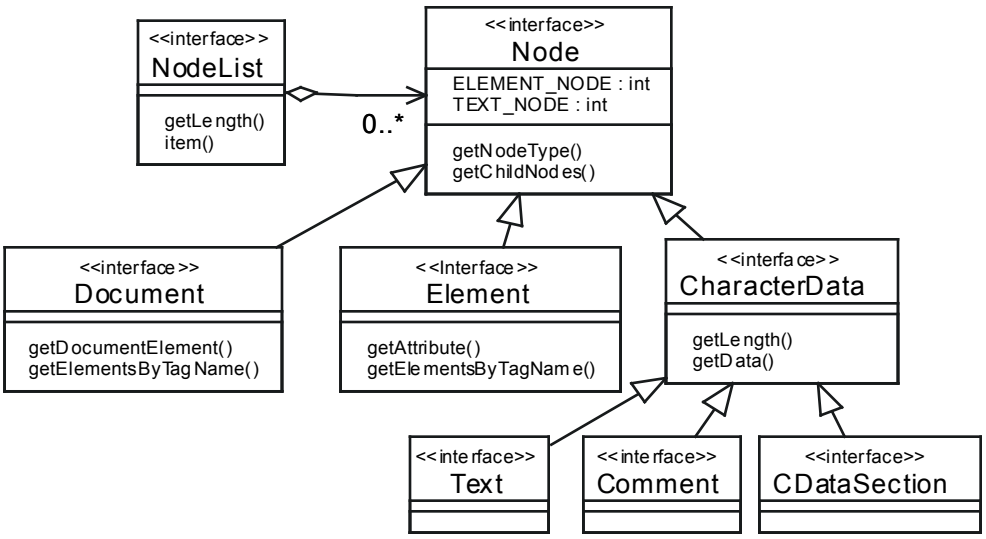
DOM is a rich specification for parsing and manipulating XML. The DOM API has been implemented in many programming languages including Java, C++, Perl, C# and others.

The primary feature of DOM is that the parser builds a tree in memory that represents the source XML. The tree contains DOM objects that map to different components of the source XML. For example, for each element in the source XML, the DOM tree contains an Element object. The DOM then provides methods to navigate the tree (e.g. enumerate child nodes) and modify the tree (e.g. add or delete nodes).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

DOM Interfaces

- DOM defines a set of interfaces that correspond to XML content
- This diagram shows only a few of the methods in the interfaces -- for example, the Node interface also defines methods to add to the tree



Basically, anything that lives in the DOM tree is a Node, but there are more specific types for Elements, AttributeNodes (not shown) and so forth. Because the DOM uses inheritance, that means that you can call any of the Node methods (e.g. getNodeType()) on any of the subtypes.

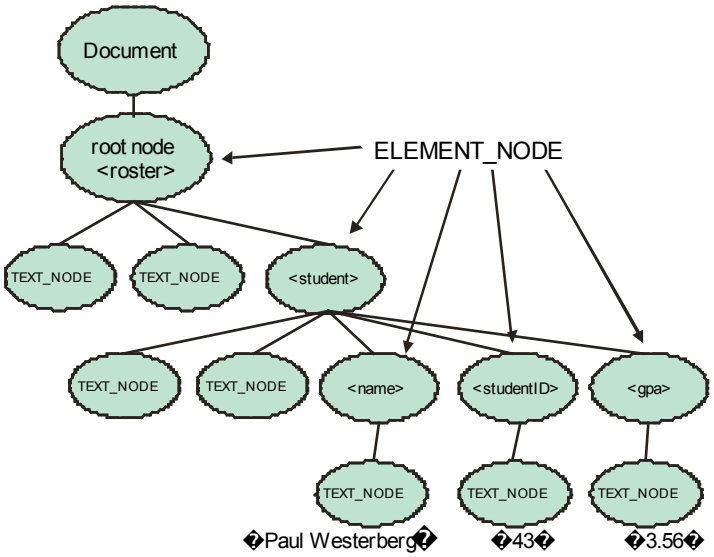
The Document type represents the entire document - note that it defines a method, getElementsByTagName(), that lets you search the DOM tree for elements with a given name. This same method also exists on the Element interface, which lets you perform a more focused search, since it only returns child elements of the Element upon which you issue the call.

A NodeList is a collection of Nodes -- you can use the item() method to index the collection. The getChildNodes and getElementsByTagName methods return NodeLists

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Objects in the DOM Tree for Sample

- This diagram shows only one of the "student" Element nodes
- Unlabeled TEXT_NODEs represent whitespace and new-line characters



This figure shows a graphical representation of a portion of the DOM tree for the sample XML content. Note that by default, the parser puts everything into the DOM tree, including whitespace nodes, even whitespace that's there just to make the XML easier to read (e.g. new-line and tab characters).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

DOM Sample Program

```
1    import javax.xml.parsers.*;
2    import org.w3c.dom.*;
3
4    public class ShowRoster
5    {
6      public static void main (String args[])
7          throws Exception
8      {
9        DocumentBuilderFactory dbf =
10           DocumentBuilderFactory.newInstance();
11
12       DocumentBuilder db = dbf.newDocumentBuilder();
13       Document doc = db.parse ( "roster.xml" );
14
15       Element root = doc.getDocumentElement();
16       String coursename = root.getAttribute ( "coursename" );
17       System.out.println ( "Coursename: " + coursename );
```


Here we start the listing of a sample DOM program that works with the sample XML content shown earlier. This program's goal is to print the average GPA of all students.

Lines 9 to 12 comprise the JAXP portion of the program - we first create a DOM-processor factory and then retrieve a DOM processor reference (the DocumentBuilder). Then in line 13, we parse the sample XML content and build the DOM tree. In this simple-minded program, if an error occurs, we simply throw the exception back to the virtual machine (see line 7).

Then in line 15, we retrieve the root element from the document and display the value of the root element's "coursename" attribute.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

DOM Sample Program. cont'd

```
18        NodeList gpas = doc.getElementsByTagName ( "gpa" );
19
20        double total = 0;
21
22        for ( int i = 0; i < gpas.getLength(); i++ )
23        {
24            Node gpaNode = gpas.item ( i );
25
26            Node txtNode = gpaNode.getFirstChild();
27            String s = txtNode.getNodeValue();
28
29            System.out.println ( "GPA: " + s );
30            total += Double.parseDouble ( s );
31        }
32
33        System.out.println ( "Average GPA: " +
34            total / gpas.getLength() );
35      }
36    }
```


Continuing the sample program, in line 18, we retrieve a NodeList of all "gpa" elements in the document. We then set up a conventional iteration loop to process the "gpa" elements.

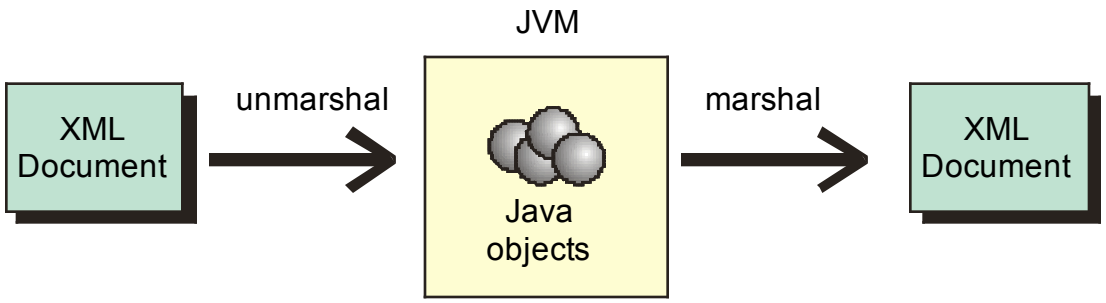
For each "gpa" element, we retrieve its first child and get the text from it. We convert the text to a double and then use it to accumulate the GPA total. When the loop is done, we print the average GPA.

Note that we are assuming that the first child under each GPA element is a text element containing a number. That's OK for this sample program, but if this were a real-world program, it would be a good idea to do a bit of checking first.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Marshaling XML to/from Java

- **Marshaling** refers to taking Java objects and storing their data as XML
- **Unmarshaling** refers to using data from XML to initialize Java objects
- You can marshal/unmarshal XML to/from Java yourself using a standard API such as DOM, or use the **JAX-B** binding API that's built into the JVM



In this context, we are talking about generically taking XML data and Java data -- since Java is object-oriented, that means using Java objects to represent the XML.

Marshaling and unmarshaling are also referred to as "encoding" and "decoding" or "serializing" and "deserializing".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

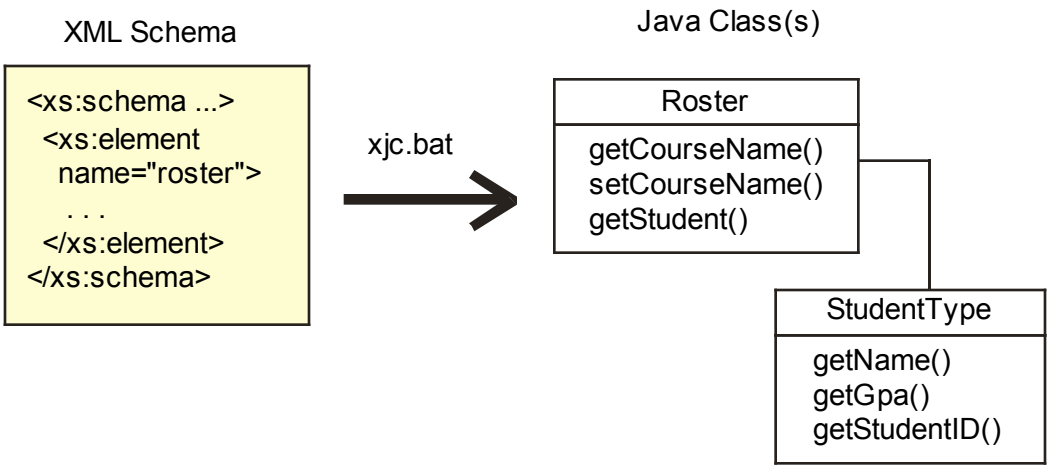
JAXB Development Steps

- Write or obtain an XML Schema
- Run the JAXB **Binding Compiler** to generate Java classes that correspond to elements in the schema
- Write a Java *main* program that uses JAXB utility classes to marshal or unmarshal to/from XML to/from Java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The JAXB Binding Compiler

- JAXB includes a **binding compiler** that reads an XML Schema and generates Java classes
- You then can use JAXB utility classes to marshal objects to XML or unmarshal XML into objects



The JDK defines the superclasses for generated Java classes, several utility classes and a binding compiler that you can run to bind Java classes to a schema.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Schema

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <xsd:schema
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
4      targetNamespace="http://www.example.org/student"
5      xmlns="http://www.example.org/student"
6      elementFormDefault="qualified">
7
8      <xsd:complexType name="StudentType">
9        <xsd:all>
10         <xsd:element name="name" type="xsd:string" />
11         <xsd:element name="gpa" type="xsd:decimal" />
12         <xsd:element name="studentID" type="xsd:integer" />
13       </xsd:all>
14     </xsd:complexType>
```


Here we start a simple schema. We first define a type for a "Student" that has child elements for a student's name, grade-point average and ID. Note how we use schema datatypes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Sample Schema, cont'd

```
15     <xsd:element name="roster">
16       <xsd:complexType>
17         <xsd:sequence>
18           <xsd:element name="course-name" type="xsd:string" />
19           <xsd:element name="student" type="StudentType"
20              minOccurs="1" maxOccurs="unbounded" />
21         </xsd:sequence>
22       </xsd:complexType>
23     </xsd:element>
24   </xsd:schema>
```


Finishing the schema, here we show the "roster" element that aggregates one or more "student" elements. The roster also has a child element that stores the course name.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Running the Binding Compiler

- You can run the compiler from the command line

```
C:\temp\jaxb>xjc -p university roster.xsd
parsing a schema...
compiling a schema...
university\ObjectFactory.java
university\Roster.java
university\StudentType.java
university\package-info.java
```


The compiler generated classes for the "roster" XML element (Roster.java) and for the "student" XML complex type (StudentType.java).

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Examining a Generated Class

```
1    package university;
2
3    import java.util.*;
4    import javax.xml.bind.annotation.*;
5
6    @XmlAccessorType(AccessType.FIELD)
7    @XmlType(name = "", propOrder =
8    { "courseName", "student" })
9    @XmlRootElement(name = "roster")
10
11   public class Roster
12   {
13       @XmlElement(name = "course-name",
14          namespace = "http://www.example.org/student")
15       protected String courseName;
16
17       @XmlElement(namespace = "http://www.example.org/student")
18       protected List<StudentType> student;
```


Here we show the start of the generated Roster class. Note how it uses annoations to define metadata used by the JAXB runtime.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Examining a Generated Class, cont'd

```
19        public String getCourseName()
20        {
21            return courseName;
22        }
23
24        public void setCourseName(String value)
25        {
26            this.courseName = value;
27        }
28
29        public List<StudentType> getStudent()
30        {
31            if (student == null)
32            {
33                student = new ArrayList<StudentType>();
34            }
35            return this.student;
36        }
37    }
```

2 - 23

Finishing the listing of the Roster class, note how it defines getters and setters for the data defined as child elements of the "roster" element in the schema. In addition, there's a getter method to traverse from the Roster to the list of Students.

The StudentType.java class (not shown here) follows a similar pattern.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unmarshaling: An Example

```
1    <?xml version="1.0" encoding="UTF-8"?>
2    <tns:roster
3       xmlns:tns="http://www.example.org/student">
4      <tns:course-name>
5        Introduction to XML
6      </tns:course-name>
7      <tns:student>
8        <tns:name>Bill Jones</tns:name>
9        <tns:gpa>3.55</tns:gpa>
10       <tns:studentID>12</tns:studentID>
11     </tns:student>
12     <tns:student>
13       <tns:name>Sue Smith</tns:name>
14       <tns:gpa>3.45</tns:gpa>
15       <tns:studentID>55</tns:studentID>
16     </tns:student>
17   </tns:roster>
```

Here we show an existing XML file containing data that's valid against the schema shown earlier. We will use JAXB to unmarshal this data into Java objects.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unmarshaling: An Example, cont'd

```
1    package client;
2
3    import java.io.*;
4    import java.util.*;
5    import javax.xml.bind.*;
6    import university.*;
7
8    public class TestUnMarshal
9    {
10       public static void main(String[] args)
11           throws Exception
12       {
13           // Create the JAXBContext, passing the package name
14           // of the JAX-B generated classes
15           JAXBContext jc = JAXBContext.newInstance(
16                   "university" );
17
18           // Create the unmarshaller that knows how to
19           // populate Java objects from XML
20           Unmarshaller u = jc.createUnmarshaller();
```

Here we start the listing of a Java program that uses the JAXB bindings and generated classes to create Java objects that correspond to the data the XML file shown on the last page. Proper exception handling is omitted for brevity.

We start by using the JAXBContext factory to create an "unmarshaller" that knows how to retrieve data from the XML. Note how in line 16 we specify the package name of the generated classes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unmarshaling: An Example, cont'd

```
21            // Unmarshal the XML into a Java object that
22            // corresponds to the root element in the XML
23            Roster roster =
24              (Roster)u.unmarshal(
25                new FileInputStream("roster.xml"));
26            System.out.println("Course name: "
27              + roster.getCourseName());
28
29            for(StudentType a : roster.getStudent())
30            {
31                System.out.println("Student name: "
32                   + a.getName());
33                System.out.println("GPA: " + a.getGpa());
34                System.out.println("StudentID: "
35                   + a.getStudentID());
36            }
37        }
38    }
```


Here we start the listing of a Java program that uses the JAXB bindings and generated classes to create Java objects that correspond to the data the XML file shown on the last page. Proper exception handling is omitted for brevity.

We start by using the JAXBContext factory to create an "unmarshaller" that knows how to retrieve data from the XML. Note how in line 16 we specify the package name of the generated classes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Unmarshaling Example Output

- Here is the output from the program -- note how the data matches the input XML

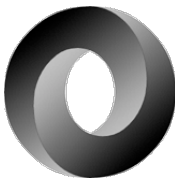
```
Course name: Introduction to XML
Student name: Bill Jones
GPA: 3.55
StudentID: 12
Student name: Sue Smith
GPA: 3.45
StudentID: 55
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is JSON?

- JSON is the **JavaScript Object Notation** and was originally syntax in JavaScript for initializing objects, but has now become a standard data format rivaling XML

```
{
  "id"  : 17,
  "name": "Sue Smith",
  "gpa":3.45
}
```



You can read more about JSON at http://www.json.org.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Uses of JSON

- Natively in JavaScript for creating and initializing objects
- As a general-purpose data exchange format, especially for AJAX-based Web applications
- As a data format in RESTful Web services

AJAX is a Web programming technique that makes Web applications more interactive (user doesn't need to submit a full page). Though AJAX applications originally used XML as the exchange format, JSON is becoming more popular.

REST is an architectural pattern in which services return formatted data. JSON is a popluar format for those type of services.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic JSON Syntax

- JSON has a simple, easy to read and understand syntax, both for humans and computers
- Basic syntax rules:
  - Denote **objects** using { } - objects have properties with names and values
  - Property names must be double-quoted and values can be double-quoted strings, numbers, *true*/*false* or *null*, arrays or nested objects
  - Denote **arrays** using [ ] - arrays contain simple values or objects or subarrays

Objects can contain sub-objects and arrays. Arrays can contain objects and sub-arrays. And so forth.

Note that you must use double quotes rather than single quotes for values.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic JSON Syntax, cont'd

```
1    -- sample object
2    {
3      "gpa"      : 3.33,
4      "name"     : "Sue Smith",
5      "address"  :
6      {
7         "street":      "123 Elm",
8         "city"  :      "Marion"
9      }
10   }
11
12   -- sample array
13   [ "red", "green", "blue" ]
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using JSON in Java

- Java programs can easily parse or generate JSON strings:
    - Using basic Java String, regex and StringBuilder methods
    - Using a library, such as www.json.org/java or Jackson

There are several other JSON libraries available for Java, including GSON and Eclipse Link MOXY.

Also note that JEE 7 has a JSON parser built in.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using the json.org Library

```
1    package json;
2    import org.json.JSONObject;
3
4    public class TestJSON
5    {
6      public static void main(String[] args)
7        throws Exception
8      {
9        JSONObject rect = new JSONObject();
10       JSONObject upperLeft = new JSONObject();
11       JSONObject lowerRight = new JSONObject();
12       upperLeft.put("x", 2);
13       upperLeft.put("y", 2);
14       lowerRight.put("x", 4);
15       lowerRight.put("y", 4);
16       rect.put("upperLeft", upperLeft);
17       rect.put("lowerRight", lowerRight);
18       System.out.println(rect.toString());
19     }
20   }
```

Here we use the json.org library to generate some JSON content. For this to work, you will need to put the org.json classes into the program's class path.

The output of this program is:

```
{"upperLeft":{"x":2,"y":2},"lowerRight":{"x":4,"y":4}}
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Jackson JSON Library

- **Jackson** is a popular library for processing JSON that aims to provide a good balance processing speed and ease of use
- Many other projects such as Jersey, RESTEasy and Spring use Jackson

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Jackson Example

student.json

Student.java

TestJSON.java

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    {
2      "gpa"      : 3.33,
3      "name"     : "Sue Smith",
4      "address"  :
5      {
6         "street":      "123 Elm",
7         "city"  :      "Marion"
8      }
9    }
```

```
1    package jacksonclient;
2
3    public class Student
4    {
5        public class Address
6        {
7            private String street;
8            private String city;
9
10           public String getStreet()
11           {
12               return street;
13           }
14
15           public void setStreet(String street)
16           {
17               this.street = street;
18           }
19
20           public String getCity()
21           {
22               return city;
23           }
24
25           public void setCity(String city)
26           {
27               this.city = city;
28           }
29
30           @Override
31           public String toString()
32           {
33               return "Address [street=" + street + ", city=" + city + "]";
34           }
35
36       }
37
38       private String name;
39       private double gpa;
40       private Address address;
41
42       public String getName()
43       {
44           return name;
45       }
```

```
46
47      public void setName(String name)
48      {
49          this.name = name;
50      }
51
52      public double getGpa()
53      {
54          return gpa;
55      }
56
57      public void setGpa(double gpa)
58      {
59          this.gpa = gpa;
60      }
61
62      public Address getAddress()
63      {
64          return address;
65      }
66
67      public void setAddress(Address address)
68      {
69          this.address = address;
70      }
71
72      @Override
73      public String toString()
74      {
75          return "Student [name=" + name + ", gpa=" + gpa +
76                  ", address=" + address
77                   + "]";
78      }
79  }
```

```
1    package jacksonclient;
2
3    import java.io.File;
4    import java.io.IOException;
5
6    import com.fasterxml.jackson.core.JsonParseException;
7    import com.fasterxml.jackson.databind.JsonMappingException;
8    import com.fasterxml.jackson.databind.ObjectMapper;
9
10   public class TestJSON
11   {
12       public static void main(String[] args)
13       {
14           ObjectMapper mapper = new ObjectMapper();
15
16           try
17           {
18               Student student = mapper.readValue(
19                  new File("student.json"), Student.class);
20
21               System.out.println(student);
22           }
23           catch (JsonParseException e)
24           {
25               e.printStackTrace();
26           }
27           catch (JsonMappingException e)
28           {
29               e.printStackTrace();
30           }
31           catch (IOException e)
32           {
33               e.printStackTrace();
34           }
35       }
36   }
```

Chapter Summary

In this chapter, you learned:

- About XML and how to process it in a Java program
- About JSON and how to process it in a Java program

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum