Republic of the Philippines

BATANGAS STATE UNIVERSITY

The National Engineering University

Lipa Campus

Marawoy, Lipa City

IT332

INTEGRATIVE PROGRAMMING & TECHNOLOGIES

LEARNING MODULE

Prepared by:

Zymon Andrew M. Maquinto, JD

Lecturer II

CONTENTS IN BRIEF

Course Code: IT 332
Course Title: Integrative Programming and Technologies
Prerequisite: IT 314

Module 1
Integrative Programming

ILO: Understand the concepts of integrative programming and design patterns as applied to distributed systems.

No.	Unit	Lesson Content
1	Integrative Coding and Design Patterns	• Overview of Integrative Programming and Technologies • Integration Models, Technologies and Methods • Introduction to Design Patterns • Java Inheritance and Interfaces • JavaFX Toolkit with CSS • Java Event-driven Programming
2	Inter-Systems Communication and Integration	• Introduction to Middleware and Web Services • Low-Level Data Communication • Network Programming and Messaging • Java Remote Method Invocation • Java TCP Socket Programming • Java UDP-based Messaging
	Module Reviewer	

Module 2
Full-Stack Systems Development

ILO: Apply programming and data integration to full-stack systems development.

No.	Unit	Lesson Content
3	Full-Stack Systems Integration	• Aspects of Frontend-Backend Integration • Modern Libraries and Framework for Web Frontend • Programming Paradigms for Enterprise Backend • Overview of Full-Stack Systems Development
4	Data Integration Pipeline	• Metadata and Data Representation • Java XML Data Handling • Java JSON Data Handling • Parsing XML Documents
	Module Reviewer	

Module 3
Integrative Technologies
(Via RedHat Academy)

ILO: Understand the concepts of integrative programming and design patterns as applied to distributed systems.

No.	Unit	Lesson Content
5	Middleware and Web Services	• Acrhitectures for Integrating Systems • Java API for RESTful Web Services • Implementing Context and Dependency Injection • Java Messaging Service
6	API and Object-Related Mapping	• Java Persistence API • CRUD Operations with Entity Manager • Java Authentication and Authorization Service • Basic JBoss Enterprise Application Platform
	Module Reviewer	

MODULE 2

UNIT 4
DATA INTEGRATION PIPELINE

In this module, you will learn about the following topics:

- Metadata and Data Representation
- Java XML Data Handling
- Java JSON Data Handling
- Parsing XML Documents

Metadata and Data Representation

A **data integration pipeline** is a sequence of processes that automatically collects, transforms, and moves data from various sources to a destination system, such as a data warehouse, data lake, or analytics dashboard. It ensures data is available in a consistent, clean, and usable form for analysis or application use. The following are some of the key components of data integration pipeline:

Data Sources	Systems that generate or store raw data like relational databases, APIs, IoT devices or flat files.
Extraction (E)	Pulls data from source systems.
Tranformation (T)	Cleans, enriches, formats, or restructures data to meet requirements.
Loading (L)	Moves the processed data into a destination system like cloud storage or DBMS.
Orchestration	Coordinates the flow and timing of data across stages. Often managed by tools like Apache Airflow or Azure Data Factory.
Monitoring and Logging	Tracks pipeline performance and errors for maintenance and reliability.

Central to the data integration pipeline is the ETL framework. The **ETL model**, which stands for Extract, Transform, Load, is a widely used data integration approach in data warehousing and analytics systems. It involves three key stages that enable the movement and processing of data from multiple sources into a centralized repository.

In the **Extract** phase, data is collected from various sources such as relational databases, APIs, flat files, or cloud services. This raw data is often heterogeneous and unstructured. Next, in the **Transform** phase, the extracted data is cleaned, formatted, and converted into a consistent structure. This step may involve operations like filtering, sorting, joining tables, removing duplicates, standardizing

formats, and applying business rules to ensure data quality and usability. Finally, in the **Load** phase, the transformed data is inserted into a target system, usually a data warehouse or data lake, where it becomes available for querying, analysis, and report generation.

The subject of the foregoing processes and phases are data and information. **Data** refers to raw facts, figures, or information collected from various sources, which can be processed and analyzed to gain insights or make decisions. It can come in different forms, such as numbers, text, images, or audio, and may be structured like data in relational databases, semi-structured like XML or JSON files, or unstructured like emails or social media posts. In any system, data represents the actual content being stored or transmitted, such as customer names, transaction records, or product details.

Here is a comparison matrix of the three classifications:

	Structured	Semi-Structured	Unstructured
Definition	Organized into fixed fields and formats	Contains tags or markers to separate elements	No predefined format or data model
Data Format	Tables	XML, JSON, CSV	Text, images, videos
Storage	RDBMS	NoSQL dB	File systems
Schema	Fixed and predefined	Self-describing	No schema
Processing	SQL Engines	NoSQL and Parsers	Artificial Intelligence (ML, NLP, OCR)
Use Cases	Reporting, Time-Series Analysis	Data Exchange, Configurations	Sentiment Analysis, Multimedia Archiving

On the other hand, **metadata** is "data about data." It describes and gives context to the actual data, helping users and systems understand what the data means, how it is structured, and how it should be used. For example, if a file contains a list of customer names, the metadata might include information such as the file format, creation date, data source, field definitions, or also even access permissions. In databases, metadata defines table schemas, data types, and relationships between tables. Metadata is crucial for organizing, managing, and retrieving data efficiently, it enables effective data governance, improves data quality, and supports interoperability across systems. Together, data and metadata form the foundation of any data management or integration system. In essence, metadata acts as a guide that makes data easier to locate, interpret, and use correctly, playing a crucial role in data governance, integration, and analysis.

Java XML Data Handling

XML or Extensible Markup Language is a markup language that is designed to store and transport data, and be readable to both humans and machines.

XML does not do anything. This XML is quite self-descriptive. It contains information about the sender (Ji), the receiver (Senpai), the heading (Greeting), and the message body (Hello there!).

Simple Code:

```
<note>
	<to>Senpai</to>
	<from>Ji</from>
	<heading>Greeting</heading>
	<body>Hello there!</body>
</note>
```

But still, the XML does not do anything. We can say that XML is just information wrapped in tags. A software must be written by someone to send, receive, display, or store it.

Note
To: Senpai
From: Ji
Greeting
Hello there!

Difference between XML and HTML**.** XML and HTML were designed with different goals. XML was designed to carry data, focusing on what data is, and HTML, on the other hand was designed to display data, focusing on how data looks. XML tags are not predefined like HTML tags are.

XML is Extensible. Most XML applications will work as expected even if new data is added (or removed). For instance, we have an application designed to display the original version of `greeting.xml` (`<to>` `<from>` `<heading>` `<body>`). Then, we will create a new version, where we will add more elements in the XML, such as `<date>` and `<hour>`, and we will remove the `<heading>`.

```
<note>
	<to>Senpai</to>
	<from>Ji</from>
	<date>2022-06-22</date>
	<hour>08:30</hour>
	<body>Hello there!</body>
</note>
```

Old Version	New Version
Note To: Senpai From: Ji Greeting Hello there!	Note To: Senpai From: Ji Date: 2022-06-22 08:30 Hello there!

XML simplifies things. Many computer systems contain data in incompatible formats. Web developers find it time consuming exchanging data between incompatible systems (or upgraded systems). Large amounts of data must be converted, and incompatible data is often lost.

Data is stored in plain text format in XML, which provides a software-independent and hardware-independent way of storing, transporting, and sharing data. XML also makes it easier in expanding or upgrading to new operating systems, new applications, or new browsers, without losing data. With XML, data can be available to all kinds of reading machines like people, computers, voice machines, newsfeeds, et cetera.

XML is used in many aspects of web development, and is often used to separate data from presentation:

XML Separates Data from Presentation	XML does not carry any information about how to be displayed. The same XML data can be used in many different presentation scenarios. With XML, there is a full separation between data and presentation.
XML Separates Data from HTML	When displaying data in HTML, you should not have to edit the HTML file when the data changes. With XML, the data can be stored in separate XML files. You can read an XML file and update the content of any HTML page with a few lines of JavaScript code.

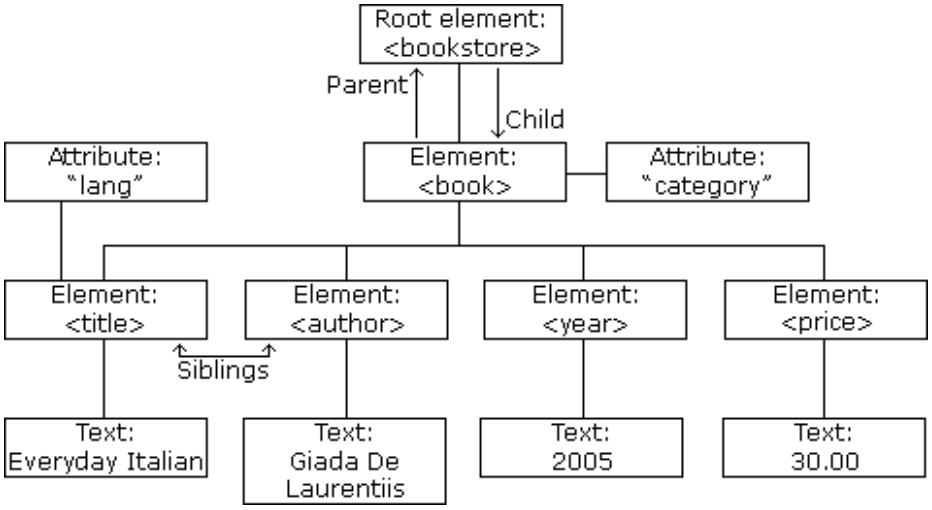
XML Tree Structure. XML documents are formed as element trees. An XML tree starts at a root element and branches from the root to child elements. All elements can have sub elements, or child elements. Parent, child, and sibling are terms used to describe the relationships between elements.

> Parents have children. Children have parents. Siblings are children on the same level (brothers and sisters). All elements can have text content (The Lost Hero) and attributes (category="fiction").

Example Code:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

Corresponding Tree Structure:



The tree structure follows the same hierarchy of elements, to wit:

```
<root>
  <child>
    <subchild>.... </subchild>
  </child>
</root>
```

Basically, An XML tree structure represents data in a hierarchical format where elements are nested within parent elements, starting from one single root element of the hierarchy.

Self-describing syntax. XML uses a much self-describing syntax. A prolog defines the XML version and the character encoding.

A. Prologue (aka Prolog)

```
<?xml version="1.0" encoding="UTF-8"?>
```

B. Root Element

```
<bookstore>
```

C. Element with Attribute

```
<book category="cooking">
```

D. Child Elements

```
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
```

Syntax Rule. Aside from the Tree Structure, the XML adheres to syntax rule which establishes the correct form and valid code of the XM. The syntax rules of XML are very simple and logical. They are easy to learn, and easy to use.

Fundamental Rules:

1. **XML Documents must have a Root Element.** XML documents must contain one root element that is the parent of all other elements.
2. **XML Prolog.** The XML prolog is optional. If it exists, it must come first in the document. XML documents can contain international characters, like Norwegian ø, æ, å or French ê, è, é. To avoid errors, you should specify the encoding used, or save your XML files as UTF-8. It is the default character encoding for XML documents. UTF-8 is also the default encoding for HTML5, CSS, JavaScript, PHP, and SQL.
3. **All XML Tags should have a Closing Tag.** In XML, it is illegal to omit the closing tag. All elements must have a closing tag. The XML prolog does not have a closing tag. This is not an error. The prolog is not a part of the XML document.
4. **XML Tags are Case-Sensitive.** XML tags are case sensitive. The tag `<Letter>` is different from the tag `<letter>`. Opening and closing tags must be written with the same case.
5. **XML Elements must be properly nested.** In XML, all elements must be properly nested within each other.

6. **XML Attribute Values must be quoted.** XML elements can have attributes in name/value pairs just like in HTML. In XML, the attribute values must always be quoted.
7. **Comments in XML.** Comments in XML The syntax for writing comments in XML is similar to that of HTML: `<! This is a comment>`
8. **White Space is Preserved in XML.** XML does not truncate multiple white-spaces (HTML truncates multiple white-spaces to one single white-space):

   XML `Hello  World`

   HTML `Hello World`
9. **XML Naming Element.** An XML element is everything from the element's start tag to the element's end tag. An element can contain text, attributes, or combination of both.
   - Any name can be used, no words are reserved, except xml.
   - Element names are case-sensitive
   - Element names must start with a letter or underscore
   - Element names cannot start with the letters xml (or XML, Xml, etc)
   - Can contain letters, digits, hyphens, underscores, and periods
   - Cannot contain spaces.

In naming your elements and attributes, create descriptive names like: `<person>`, `<firstname>`, and `<lastname>`. Make use of short and simple names, like `<book_title>`, not like <the_title_of_the_book>.

Avoid using dashes (-). If you name something, first-name, some software may think you are subtracting name from first. Using dot(.) in naming, for instance, first.name, some software may think that "name" is a property of the object "first".

Entity References. Some characters have a special meaning in XML. If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element. This will generate an XML error: <message>salary < 1000</message>. To avoid this error, replace the "<" character with an entity reference: <message>salary < 1000</message>.

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

XML Attributes. XML elements can have attributes, just like HTML. They are designed to contain data related to a specific element. Attributes allow developers to extract certain elements in the file, which has this kind of attributes.

Sometimes ID References are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. An ID will make for each element.

Data Exchange and XML Schema. There are two main checks that XML processors make:

1. Checking that your document is well-formed (Syntax rule)

2. Checking that it's valid (syntax-check your XML either in XML DTD or XSD)
    - DTD - Document Type Definition
    - XSD - XML Schema Definition

Document Type Definition. An XML document with correct syntax is called "Well Formed", and an XML document validated against a DTD is "Well Formed" and "Valid". The purpose of a DTD is to define the structure of an XML document and a list of legal elements.

A DTD can be added in XML by separate documents, or they can be built into an XML document using a special element named <!DOCTYPE>. Here is an example of XML Document with a DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

DTD may be externally declared in a file with name extension as *.dtd. The form is as follows:

```
<!DOCTYPE note
[
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
```

Meanwhile, the DTD above is interpreted as follows:

`!DOCTYPE note` - Defines that the root element of the document is note
`!ELEMENT note` - Defines that the note element must contain the elements: "to, from, heading, body"
`!ELEMENT to` - Defines the to element to be of type "#PCDATA"
`!ELEMENT from` - Defines the from element to be of type "#PCDATA"
`!ELEMENT heading` - Defines the heading element to be of type "#PCDATA"
`!ELEMENT body` - Defines the body element to be of type "#PCDATA"

Note: PCDATA means Parseable Character Data

When to use DTD:

With a DTD, independent groups of people can agree to use a standard DTD for interchanging data. With a DTD, you can verify that the data you receive from the outside world is valid. You can also use a DTD to verify your own data.

When NOT to use a DTD:

XML does not require a DTD. When you are experimenting with XML, or when you are working with small XML files, creating DTDs may be a waste of time. If you develop applications, wait until the specification is stable before you add a DTD. Otherwise, your software might stop working because of validation errors.

XML Schema Definition. Another way of validating XML documents is by using XML schemas. The XML Schema language is also referred to as XML Schema Definition (XSD), that describes the structure of an XML document.

Also, it defines the legal building blocks such as elements and attributes of an XML document like DTD, which elements are child elements, the number and order of child elements. It also defines whether an element is empty or can include text. Data types for elements and attributes and its default values are also being defined by the schema.

The XML Schemas will be used in most Web applications as a replacement for DTDs, for such reasons as XML Schemas are extensible to future additions, are richer and more powerful than DTDs, XML Schemas are written in XML and support data types and namespaces.

Simple Element: A simple element is an XML element that can contain only text, that cannot contain any other elements or attributes. The XML Schema has a lot of built-in data types. The most common types are:

- `xs:string`
- `xs:decimal`
- `xs:integer`
- `xs:boolean`
- `xs:date`

The syntax for defining a simple element is:

```
<xs:element name="xxx" type="yyy"/>
```

Example:

```
<xs:element name="note">

<xs:complexType>
  <xs:sequence>
    <xs:element name="to" type="xs:string"/>
    <xs:element name="from" type="xs:string"/>
    <xs:element name="heading" type="xs:string"/>
    <xs:element name="body" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

</xs:element>
```

The Schema above is interpreted like this:

`<xs:element name="note">` defines the element called "note"
`<xs:complexType>` the "note" element is a complex type
`<xs:sequence>` the complex type is a sequence of elements
`<xs:element name="to" type="xs:string">` the element "to" is of type string (text)
`<xs:element name="from" type="xs:string">` the element "from" is of type string
`<xs:element name="heading" type="xs:string">` the element "heading" is of type string
`<xs:element name="body" type="xs:string">` the element "body" is of type string

Complex Element: A complex element is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

Empty	`<product pid="1345"/>`

Contains only other elements	`<employee>`
	`<firstname>Charlie</firstname>`
	`<lastname>Puth</lastname>`
	`</employee>`

Contains only text	`<food type="dessert">Ice cream</food>`

Contains both elements and text	`<description>`
	`Date: <date lang="EN">03.03.99</date>`
	`</description>`

A Complex XML element, "employee", which contains only other elements:

```
<xs:element name="employee">
      <xs:complexType>
            <xs:sequence>
                  <xs:element name="firstname" type="xs:string"/>
                  <xs:element name="lastname" type="xs:string"/>
            </xs:sequence>
      </xs:complexType>
</xs:element>
```

An empty complex element cannot have contents, only attributes.

```
<product prodid="1345" />
```

It is possible to declare the "product" element more compactly:

```
<xs:element name="product">
	<xs:complexType>
		<xs:attribute name="prodid" type="xs:positiveInteger"/>
	</xs:complexType>
</xs:element>
```

XSD indicators may be used to define the order of the elements. They are all, choice, and sequence.

The `<all>` indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

```
<xs:element name="person">
	<xs:complexType>
	<xs:all>
		<xs:element name="firstname" type="xs:string"/>
		<xs:element name="lastname" type="xs:string"/>
	</xs:all>
	</xs:complexType>
</xs:element>
```

The `<choice>` indicator specifies that either one child element or another can occur:

```
<xs:element name="person">
	<xs:complexType>
	<xs:choice>
		<xs:element name="employee" type="employee"/>
		<xs:element name="member" type="member"/>
	</xs:choice>
	</xs:complexType>
</xs:element>
```

The `<sequence>` indicator specifies that child elements must appear in specific order:

```
<xs:element name="person">
	<xs:complexType>
	<xs:sequence>
		<xs:element name="firstname" type="xs:string"/>
		<xs:element name="lastname" type="xs:string"/>
	</xs:sequence>
	</xs:complexType>
</xs:element>
```

Using Java, **Java XML handling using the DOM (Document Object Model)** is a technique where the entire XML document is loaded into memory and represented as a tree-like structure of nodes. Each element, attribute, and piece of text in the XML becomes a part of this tree, allowing developers to navigate and manipulate the document using standard methods. This approach is useful when you need to access and possibly modify different parts of the XML multiple times, since everything is stored in memory and easily searchable.

Java provides this functionality through the `javax.xml.parsers` package, specifically using classes like `DocumentBuilderFactory`, `DocumentBuilder`, and the `org.w3c.dom` interfaces such as `Document`, `Element`, and `NodeList`. While DOM is relatively easy to use and powerful for reading and editing structured XML content, it is best suited for small to moderately sized XML files because it loads the entire document into memory, which can become inefficient with very large files. Here is the list of the most important code components:

Code Fragment	Purpose
`DocumentBuilderFactory`	Used to configure and create a parser for building a DOM Document
`DocumentBuilder`	Parses XML files and returns a DOM object representing the XML structure
`Document`	Main entry point for accessing XML content as a node tree
`getElementsByTagName(String tag)`	Used to retrieve specific elements like all <title> tags
`NodeList`	Allows iteration over multiple elements in the XML
`getTextContent()` `getNodeValue()`	Used to read the actual data contained in an XML tag
`normalize()`	Prepares the XML tree for consistent processing

Steps in Java XML Data Handling:

1. Load the XML file – Locate and open the XML file using the `File` class to prepare it for parsing.
2. Create a Document Object – Use a `DocumentBuilder` to parse the file and create a `Document` object that represents the XML structure in memory.
3. Normalize the Document – Clean and standardize the document structure by merging adjacent text nodes and removing irregularities, making it easier to traverse.

4. Get the data you want – Retrieve specific XML elements (like `<TITLE>` or `<YEAR>`) using methods such as `getElementsByTagName()`.
5. Loop through the data – Iterate over the retrieved elements, apply logic (e.g., filtering), and extract needed values.
6. Output the results – Display the final processed data (such as titles of CDs before 1990) to the user or console.

Sample XML File:

```
<CATALOG>
  <CD>
    <TITLE>Empire Burlesque</TITLE>
    <ARTIST>Bob Dylan</ARTIST>
    <COUNTRY>USA</COUNTRY>
    <COMPANY>Columbia</COMPANY>
    <PRICE>10.90</PRICE>
    <YEAR>1985</YEAR>
  </CD>
</CATALOG>
```

Full Code for XML Handling in Java using DOM:

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.*;

import java.io.File;

public class ReadXMLExample {
    public static void main(String[] args) {
        try {
            // Load and parse the XML file
            File xmlFile = new File("cd_catalog.xml");
            DocumentBuilderFactory dbFactory =
            DocumentBuilderFactory.newInstance();
            DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
            Document doc = dBuilder.parse(xmlFile);

            doc.getDocumentElement().normalize();

            // Get all TITLE and YEAR elements
            NodeList titleList = doc.getElementsByTagName("TITLE");
            NodeList yearList = doc.getElementsByTagName("YEAR");

            System.out.println("CDs released before 1990:\n");

            for (int i = 0; i < yearList.getLength(); i++) {
                int year =
                Integer.parseInt(yearList.item(i).getTextContent());
                if (year < 1990) {
                    String title = titleList.item(i).getTextContent();
```

```
                    System.out.println(title);
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Java JSON Data Handling

JSON, which stands for JavaScript Object Notation, is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is commonly used to transfer data between a server and a web application, store configuration settings, and exchange information between systems or programming environments. JSON represents data using key-value pairs and supports various data types such as strings, numbers, booleans, arrays, objects, and null values. Its syntax is strict: keys must be in double quotes, and values must follow specific formatting rules. For example, a JSON object might include a person's name, age, and a list of their skills, all structured in a clean, readable format. JSON syntax is derived from JavaScript object notation syntax:

- **Data is in name/value pairs** - In JSON, each piece of data is expressed as a name (key) followed by a value, separated by a colon.

```
"name": "Zymon"
```

- **Data is separated by commas** - When you have multiple name/value pairs in an object, they are separated by commas.

```
{
  "name": "Zymon",
  "age": 25,
  "isStudent": false
}
```

- **Curly braces hold objects** - Curly braces are used to group key-value pairs into a JSON object.

```
{
  "city": "Batangas",
  "country": "Philippines"
}
```

- **Square brackets hold arrays** - square brackets `[]` are used to store arrays, which are ordered lists of values.

```
{
  "skills": ["Java", "Python", "SQL"]
}
```

Both JSON and XML are widely used formats for receiving data from a web server, especially in web applications and APIs. They serve the same fundamental purpose: to structure and transmit data in a readable and organized way between clients and servers. JSON (JavaScript Object Notation) is lightweight, easier to read and write, and more compatible with JavaScript, making it the preferred choice in modern web development. XML (eXtensible Markup Language), on the other hand, is more verbose and supports complex data structures, attributes, and custom tags, which can be useful in certain enterprise or legacy systems. While XML was more common in the early days of the web, JSON has become the dominant format due to its simplicity, smaller file size, and faster parsing.

XML Example:

```
<employees>
  <employee>
    <firstName>John</firstName> <lastName>Doe</lastName>
  </employee>
  <employee>
    <firstName>Anna</firstName> <lastName>Smith</lastName>
  </employee>
  <employee>
    <firstName>Peter</firstName> <lastName>Jones</lastName>
  </employee>
</employees>
```

JSON Example:

```
{"employees":[
  { "firstName":"John", "lastName":"Doe" },
  { "firstName":"Anna", "lastName":"Smith" },
  { "firstName":"Peter", "lastName":"Jones" }
]}
```

JSON is like XML because:

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages
- Both JSON and XML can be fetched with an XMLHttpRequest

Note: XML has to be parsed with an XML parser. JSON can be parsed by a standard JavaScript function.

This parsing process can be complex and requires more code or libraries, especially if the XML includes nested elements or validation rules (like DTD or XSD). On the other hand, JSON can be parsed directly by a standard JavaScript function called `JSON.parse()`, which quickly converts a JSON string into a JavaScript object. This makes JSON easier and faster to work with in modern web development, especially in browsers and JavaScript-based environments, because no additional tools are needed to understand or use the data.

	XML	JSON
Format	Markup language	Data format
Data Structure	Tree structure with nested tags	Object and array-based structure
Readability	More verbose, harder to read	Simpler, more human-readable
Parsing	Requires an XML parser	Can be parsed using `JSON.parse()` in JavaScript
Data Size	Larger due to opening and closing tags	Smaller and more compact
Syntax Rules	Uses tags (`<tag>`) and attributes	Uses key/value pairs with `{}` and `[]`
Comments	Yes	No
Schema	Supports DTD and XSD for validation	Supports JSON Schema (less formal than XML schemas)

In Java, JSON data handling involves reading, writing, parsing, and generating JSON data. This is essential when working with APIs, configurations, or data exchange between systems. Java does not have built-in JSON support in the core libraries, but several popular libraries make JSON handling easy and efficient.

A. Jackson

Jackson simplifies working with JSON by handling the conversion automatically based on class structure, making it a powerful tool for data exchange in modern Java applications.

```
import com.fasterxml.jackson.databind.ObjectMapper;

// Define a simple Java class
class Person {
    public String name;
    public int age;
    public boolean isStudent;

    // Required no-argument constructor
    public Person() {}

    public Person(String name, int age, boolean isStudent) {
        this.name = name;
        this.age = age;
        this.isStudent = isStudent;
    }
}
```

```
public class JsonExample {
    public static void main(String[] args) throws Exception {
        // Create a Person object
        Person person = new Person("Zymon", 25, true);

        // Create ObjectMapper instance
        ObjectMapper mapper = new ObjectMapper();

        // Convert Person object to JSON string (serialization)
        String jsonString = mapper.writeValueAsString(person);
        System.out.println("JSON Output:\n" + jsonString);

        // Convert JSON string back to Person object (deserialization)
        Person newPerson = mapper.readValue(jsonString, Person.class);
        System.out.println("\nDeserialized Person:");
        System.out.println("Name: " + newPerson.name);
        System.out.println("Age: " + newPerson.age);
        System.out.println("Is Student: " + newPerson.isStudent);
    }
}
```

The Java JSON handling example using Jackson demonstrates how to serialize and deserialize JSON data using a simple Java class. In the code, a `Person` object is first created and then converted into a JSON string using Jackson's `ObjectMapper.writeValueAsString()` method, showing how Java objects can be easily represented as JSON. The same JSON string is then converted back into a `Person` object using `ObjectMapper.readValue()`, proving how JSON data received from external sources, such as APIs or files, can be parsed into usable Java objects.

B. Gson

Gson handles the conversion automatically by mapping JSON keys to Java fields based on the class structure, making it simple and efficient for developers to work with JSON data in Java applications, especially for tasks like API communication and data storage.

```
import com.google.gson.Gson;

// Define a simple Java class
class Person {
    public String name;
    public int age;
    public boolean isStudent;

    public Person(String name, int age, boolean isStudent) {
        this.name = name;
        this.age = age;
        this.isStudent = isStudent;
```

```
    }
}

public class GsonExample {
    public static void main(String[] args) {
        // Create a Gson instance
        Gson gson = new Gson();

        // Create a Person object
        Person person = new Person("Zymon", 25, true);

        // Convert Person object to JSON string
        String jsonString = gson.toJson(person);
        System.out.println("JSON Output:\n" + jsonString);

        // Convert JSON string back to Person object
        Person newPerson = gson.fromJson(jsonString, Person.class);
        System.out.println("\nDeserialized Person:");
        System.out.println("Name: " + newPerson.name);
        System.out.println("Age: " + newPerson.age);
        System.out.println("Is Student: " + newPerson.isStudent);
    }
}
```

The Java JSON handling example using Gson shows how to easily convert Java objects to JSON strings and back again with minimal code. In the example, a `Person` object is created and then serialized into a JSON string using Gson's `toJson()` method, which transforms the object's data into a readable JSON format. Afterwards, the JSON string is deserialized back into a new `Person` object using `fromJson()`, allowing the program to reconstruct the original data from the JSON.

Parsing XML Documents

Parsing XML documents means reading and processing the XML data so that a program can understand and use its content. In Java and many other programming languages, this is done using specialized XML parsers that read the XML file or string and convert it into a structured format like a tree of objects. There are two main types of XML parsers:

DOM	Document Object Model, a parser that loads the entire XML document into memory as a tree structure, allowing easy navigation and modification of elements.
SAX	Simple API for XML, an event-driven parser that reads the XML document sequentially and triggers events for elements without loading the whole document into memory, making it efficient for large files.

The XMLHttpRequest object can be used to request data from a web server. It is considered as a developers dream, because you can:

- Update a web page without reloading the page
- Request data from a server - after the page has loaded
- Receive data from a server - after the page has loaded
- Send data to a server - in the background

Sending an XMLHttpRequest. A common syntax for using the XMLHttpRequest object looks much like this:

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       // Typical action to be performed when the document is ready:
       document.getElementById("demo").innerHTML = xhttp.responseText;
    }
};
xhttp.open("GET", "filename", true);
xhttp.send();
```

Steps:

1. Create an XMLHttp Request object:

```
var xhttp = new XMLHttpRequest();
```

2. Specify the function using `onreadystatechange` property:

```
xhttp.onreadystatechange = function()
```

3. Create a conditional statement predicated on the same property.

```
if (this.readyState == 4 && this.status == 200)
```

4. Capture the response:

```
document.getElementById("demo").innerHTML = xhttp.responseText;
```

All modern browsers have a built-in XML parser. An XML parser converts an XML document into an XML DOM (Document Object Mode) object - which can then be manipulated with JavaScript.

This DOM represents the XML data as a tree of nodes, where each node corresponds to an element, attribute, or piece of text from the XML. Because the XML is now in this structured object form, JavaScript running in the browser can easily access, navigate, and manipulate the XML content dynamically such as reading values, changing elements, or adding new nodes, making XML data interactive and usable within web applications.

The following code fragment parses an XML document into an XML DOM object:

```
if (window.XMLHttpRequest) {
       // code for IE7+, Firefox, Chrome, Opera, Safari
       xmlhttp=new XMLHttpRequest();
       }
else {
       // code for IE6, IE5
       xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
       }

xmlhttp.open("GET","books.xml",false);
xmlhttp.send();
xmlDoc=xmlhttp.responseXML; </xs:element>
```

The XML DOM defines a standard way for accessing and manipulating XML documents. It views an XML document as a tree-structure. All elements can be accessed through the DOM tree. Their content (text and attributes) can be modified or deleted, and new elements can be created. The elements, their text, and their attributes are all known as nodes.

Meanwhile, The HTML DOM defines a standard way for accessing and manipulating HTML documents. All HTML elements can be accessed through the HTML DOM.

Sample Script:

```
<script>
  // Sample XML string
  const xmlString = `
    <person>
      <name>Zymon</name>
      <age>25</age>
      <city>Batangas</city>
    </person>`;

  // Parse the XML string into an XML DOM object
  const parser = new DOMParser();
    const xmlDoc = parser.parseFromString(xmlString, "application/
xml");

  // Access elements using DOM methods
  const name = xmlDoc.getElementsByTagName("name")[0].textContent;
  const age = xmlDoc.getElementsByTagName("age")[0].textContent;
  const city = xmlDoc.getElementsByTagName("city")[0].textContent;

  // Display the extracted values
  console.log("Name:", name);
  console.log("Age:", age);
  console.log("City:", city);
</script>
```

BATANGAS STATE UNIVERSITY
The National Engineering University
College of Informatics and Computing Sciences

IT332 - INTEGRATIVE PROGRAMMING & TECHNOLOGIES

Laboratory Assessment Form

Name: _____ Section: _____ Score: _____

Activity: XML HTTP Request

Objectives: Create an HTML web page that manages a Plant Catalog which must be able to integrate the following items:

Rubric: XML & HTML Project (Total: 50 points)

Criteria	Description	Points
1. HTML & CSS Design	Use of semantic HTML, organized structure, and styling with good visual layout	**10 pts**
2. XML Parsing & Integration	Correctly parses XML data using JavaScript and integrates it into the webpage	**10 pts**
3. JavaScript Functionality	DOM manipulation, loop through XML, error handling, and dynamic rendering	**10 pts**
4. Output Accuracy	Displays correct book details: title, author, year, price, category with styles	**10 pts**
5. Code Quality & Organization	Clean code, proper indentation, clear naming, and readable structure	**10 pts**