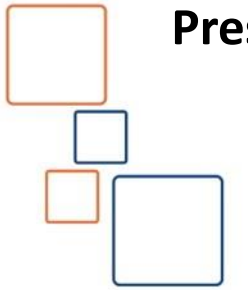
XML and JSON Essentials with Java



Presented By:

Mona Mahrous, MSc

Course Outlines

- **Chapter (1):** XML

  What is XML? What XML can do? How to write XML?

- **Chapter (2):** JSON

  What is JSON? What JSON can do? How to write JSON?

- **Chapter (3):** JSON APIs

  Used to parse JSON using Java API

Java™ Education and Technology Services

Chapter 1

XML

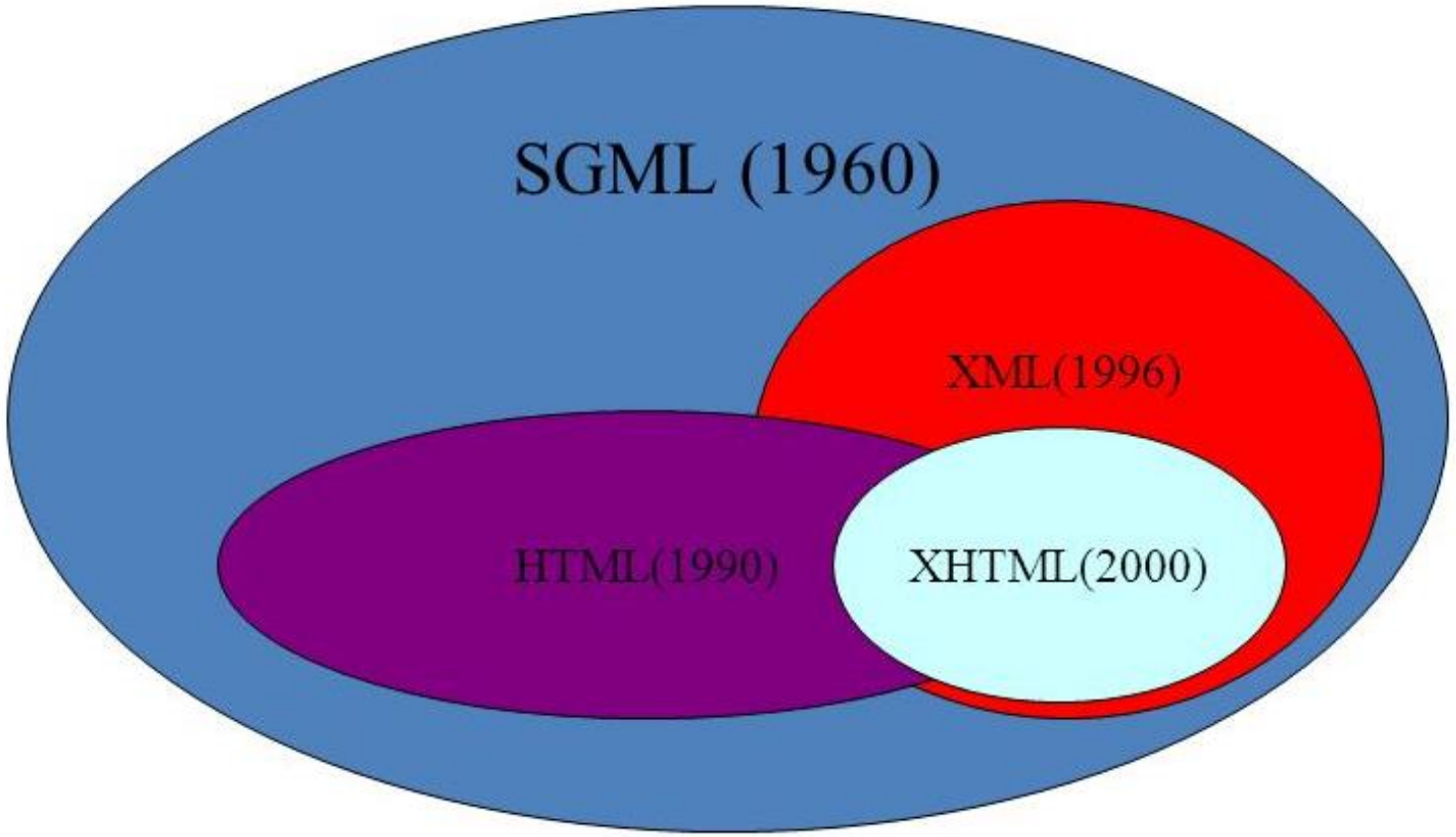
What, Why & How?

Ch1 : What is XML?

Java™ Education and Technology Services

- Stands for e**X**tensible **M**arkup **L**anguage.
- XML is a markup language much like HTML used to describe data.
- W3C Recommendation, since February 1998.
- XML was designed to be both human- and machine-readable.

History: SGML vs. HTML vs. XML



http://www.w3.org/TR/2006/REC-xml-20060816/

Ch1 : XML Features

Java™ Education and Technology Services

- XML is easy to understand.
- It is platform independent.
- XML was designed to ***store*** and ***transport*** data.
  - **Store Data**
    - XML Databases
      - XML-Enabled (Oracle, MS SQL Server, DB2, …)
      - Native-XML (BaseX, eXist, …)
    - User Interface Design (Qt, JavaFX, Android, …)
    - Configuration Files (Frameworks, Libraries, Mapping, …)
    - Settings Files(IDEs, Programs, Games, …)
  - **Transport Data**
    - Web Services (SOAP, REST, …)

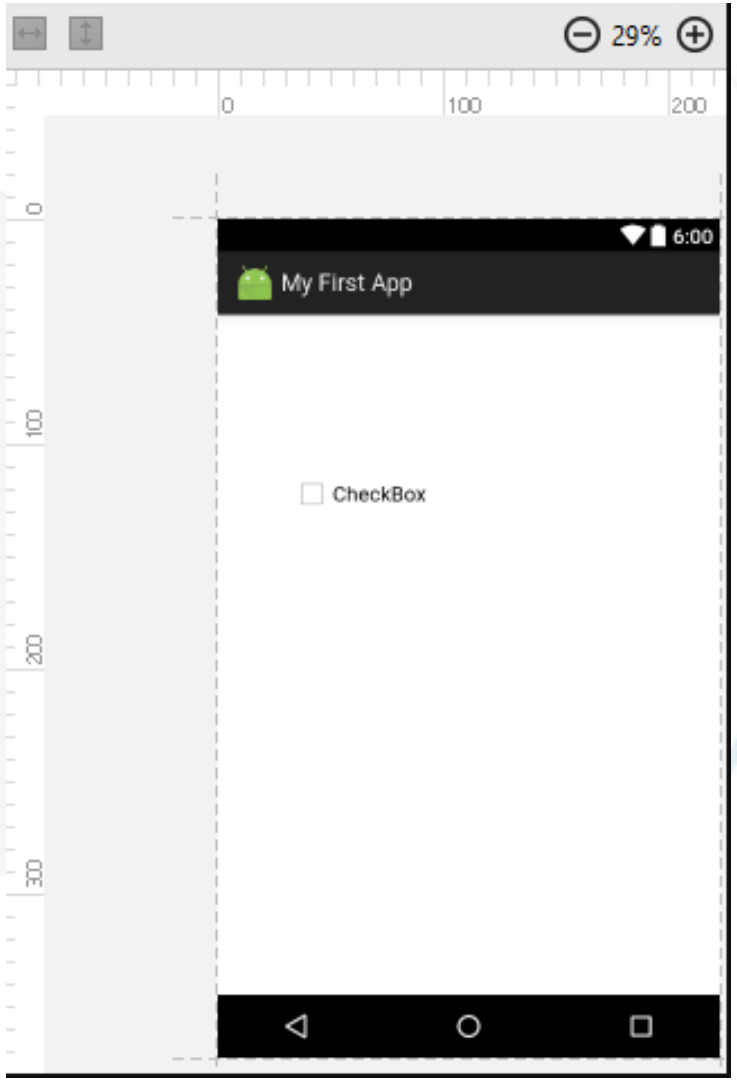
Ch1 : Why XML?

- **Android Views:**

`view_customs.xml` to be:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/custom_view"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    >

    <TextView android:id="@+id/tv"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/app_name"
        />
</LinearLayout>
```



Ch1 : Why XML?

Java™ Education and Technology Services

- **iOS Storyboard:**

```
<document type="com.apple.InterfaceBuilder3.CocoaTouch.Storyboard.XIB" version="3.0" toolsVersion="14490.70" targetRuntime="iOS.CocoaTouch" propertyAccessControl="none" useAutolayout="YES" useTraitCollections="YES" useSafeAreas="YES" colorMatched="YES">
    <device id="retina6_1" orientation="portrait">
        <adaptation id="fullscreen"/>
    </device>
    <dependencies>
        <deployment identifier="iOS"/>
        <plugIn identifier="com.apple.InterfaceBuilder.IBCocoaTouchPlugin" version="14490.49"/>
        <capability name="Safe area layout guides" minToolsVersion="9.0"/>
        <capability name="documents saved in the Xcode 8 format" minToolsVersion="8.0"/>
    </dependencies>
    <scenes>
        <!--View Controller-->
        <scene sceneID="SAQ-91-eGP">
            <objects>
                <viewController id="sRn-T1-EGi" sceneMemberID="viewController">
                    <view key="view" contentMode="scaleToFill" id="xQm-XW-XAG">
                        <rect key="frame" x="0.0" y="0.0" width="414" height="896"/>
                        <autoresizingMask key="autoresizingMask" widthSizable="YES" heightSizable="YES"/>
                        <subviews>
                            <button opaque="NO" contentMode="scaleToFill" contentHorizontalAlignment="center" contentVerticalAlignment="center" buttonType="roundedRect" lineBreakMode="middleTruncation" translatesAutoresizingMaskIntoConstraints="NO" id="etx-p4-ZLn">
                                <rect key="frame" x="80" y="674" width="254" height="44"/>
                                <color key="backgroundColor" red="0.0" green="0.5" blue="0.91494277400000001" alpha="1" colorSpace="custom" customColorSpace="sRGB"/>
                                <constraints>
                                    <constraint firstAttribute="height" constant="44" id="8sH-Ou-My2"/>
                                </constraints>
                                <state key="normal" title="Login">
                                    <color key="titleColor" white="1" alpha="1" colorSpace="custom" customColorSpace="genericGamma22GrayColorSpace"/>
                                </state>
                                <connections>
                                    <segue destination="jG5-aa-SSP" kind="presentation" id="fJK-lO-pcU"/>
                                </connections>
                            </button>
                        </subviews>
```

Ch1 : XML Example

Java™ Education and Technology Services

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

<note>
    <to>Aya</to>
    <from>Ahmed</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
</note>
```

Ch1 : XML Does Not Do Anything

Java™ Education and Technology Services

- Maybe it is a little hard to understand, but XML does not DO anything.

- The XML above is quite self-descriptive:
  - It has sender information.
  - It has receiver information
  - It has a heading
  - It has a message body.
  - But still, the XML above does not DO anything.
  - XML is just information wrapped in tags.

- Someone must write a piece of software to ***send***, ***receive***, ***store***, or ***display*** it.

Ch1 : Self-Describing Syntax

Java™ Education and Technology Services

- A prolog defines the XML version and the character encoding

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

- The next line is the root element of the document

```
<bookstore>
```

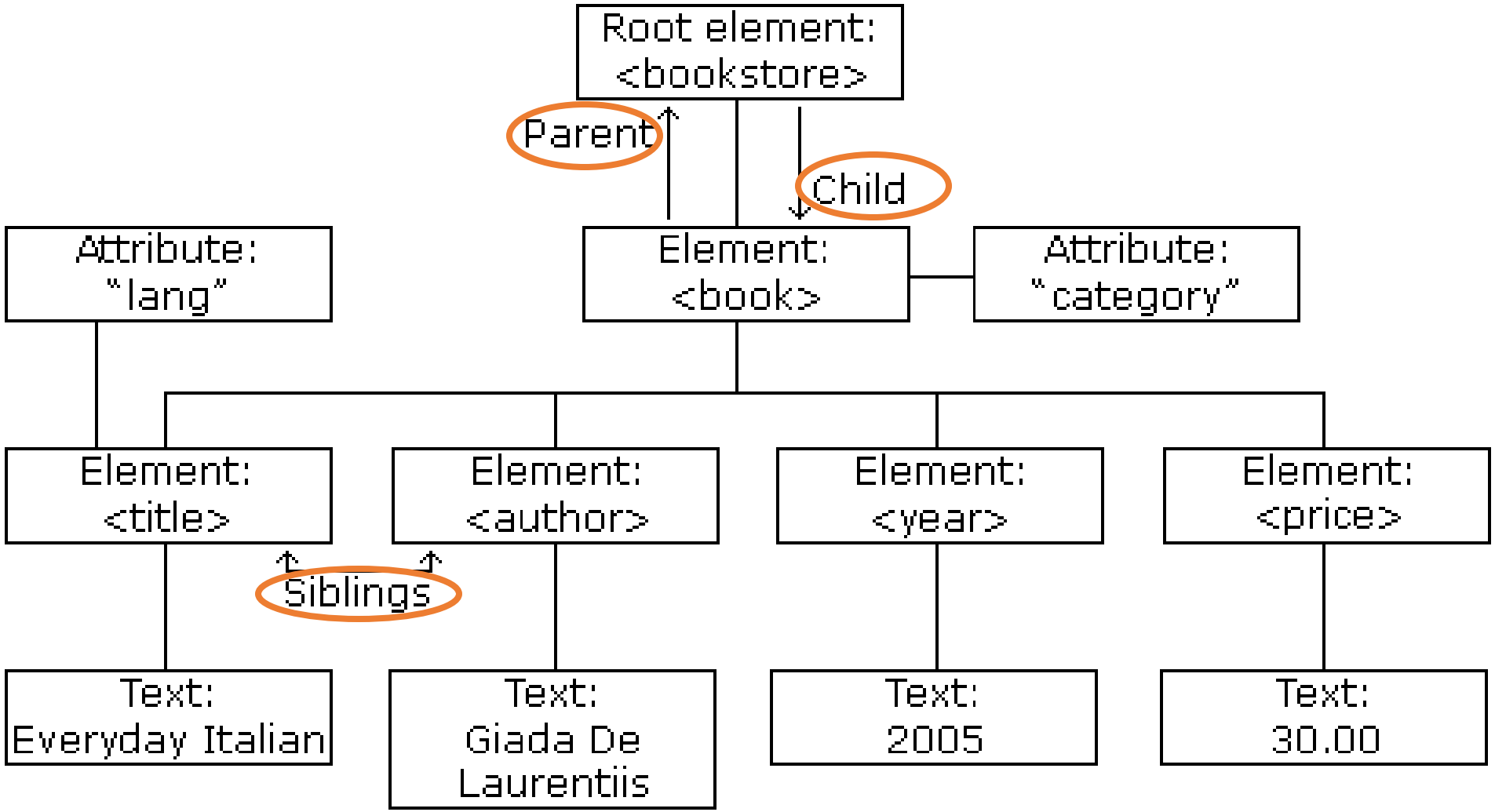
- The next line starts a <book> element
  - The <book> elements have 4 child elements:
    <title>, <author>, <year>, <price>
- The next line ends the book element

```
<book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
</book>
```

- The next line ends the root element of the document

```
</bookstore>
```

Ch1 : XML Tree



Ch1 : XML Tree

Java™ Education and Technology Services

- XML documents form a tree structure that starts at "the root" and branches to "the leaves".
- 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 (child elements):

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

- The terms **parent**, **child**, and **sibling** are 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 and attributes.

Ch1 : XML Syntax Rules

Java™ Education and Technology Services

- XML Documents Must Have a Root Element
- The XML Prolog
  `<?xml version="1.0" encoding="UTF-8"?>`
- All XML Elements Must Have a Closing Tag
- XML Tags are Case Sensitive
- XML Elements Must be Properly Nested
- XML Attribute Values Must Always be Quoted

Ch1 : XML Syntax Rules

Java™ Education and Technology Services

- Entity References
- CDATA Sections
- Comments in XML
- White-space is Preserved in XML
- Well Formed XML

Ch1 : XML Documents Must Have a Root Element

- XML documents must contain one root element that is the parent of all other elements:

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

- In this example <note> is the root element:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Aya</to>
  <from>Amin</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Ch1 : The XML Declaration (Prolog)

Java™ Education and Technology Services

- This line is called the XML prolog:

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

- The XML prolog is optional. If it exists, it must come first in the document.
- XML documents can contain international characters, like Norwegian øæå or French êèé.
- To avoid errors, you should specify the encoding used, or save your XML files as UTF-8.
- UTF-8 is the default character encoding for XML documents.

- UTF-8 is also the default encoding for HTML5, CSS, JavaScript, PHP, and SQL.

Ch1 : Syntax Rules for XML Declaration (Prolog)

Java™ Education and Technology Services

- If the XML declaration is included, it must contain version number attribute.
- The Parameter names and values are case-sensitive.
- The names are always in lower case.
- The order of placing the parameters is important.
  The correct order is: version, encoding and standalone.
- Either single or double quotes may be used.
- The XML declaration has no closing tag i.e. </?xml>

Ch1 : XML declaration

Java™ Education and Technology Services

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no" ?>
```

Parameter	Parameter_value	Parameter_description
Version	**1.0** or **1.1**	Specifies the version of the XML standard used. ***(The Only Mandatory Part)***
Encoding	**UTF-8**, UTF-16, ISO-8859-1, Windows-1251, …	It defines the character encoding used in the document. ***UTF-8 is the default encoding used.***
Standalone	**yes** or **no**	It informs the parser whether the document relies on the information from an external source, such as external document type definition (DTD), for its content. ***The default value is set to no.*** Setting it to yes tells the processor there are no external declarations required for parsing the document.

Ch1 : All XML Elements Must Have a Closing Tag

- In XML, it is illegal to omit the closing tag. All elements must have a closing tag:

```
<paragraph>This is a paragraph.</paragraph>

<line-break />
```

- Note: The XML prolog does not have a closing tag! This is not an error.
- The prolog is not a part of the XML document.

Ch1 : XML Tags are Case Sensitive

Java™ Education and Technology Services

- 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:

```
<message>This is correct</message>

<Message>This is NOT correct</message>
```

- "**Opening** and **Closing** tags" are often referred to as "**Start** and **End** tags".
- Use whatever you prefer.
- It is exactly the same thing.

Ch1 : XML Elements Must be Properly Nested

- In HTML, you might see ***improperly nested*** elements:

```
<b> <i> This text is bold and italic </b> </i>
```

- In XML, all elements ***must be properly nested*** within each other:

```
<b> <i> This text is bold and italic </i> </b>
```

- In the example above, "Properly nested" simply means that since the <i> element is opened inside the <b> element, it must be closed inside the <b> element.

Ch1 : XML Attribute Values Must Always be Quoted

- XML elements can have attributes in name/value pairs just like in HTML.
- In XML, the attribute values must always be quoted:

```
<note date="12/11/2020">
  <to>Aya</to>
  <from>Amin</from>
</note>

<note date='12/11/2020'>
  <to>Aya</to>
  <from>Amin</from>
</note>
```

Ch1 : Entity References

Java™ Education and Technology Services

- 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 &lt; 1000</message>
```

- Entity References: Begin with ampersand (&) and end with semicolon (;)

Ch1 : Entity References

Java™ Education and Technology Services

- There are 5 pre-defined entity references in XML:

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

- Only < and & are strictly illegal in XML, but it is a good habit to replace > with > as well.

Ch1 : CDATA Sections

Java™ Education and Technology Services

- May contain text, reserved characters and white space
  - Reserved characters need not be replaced by entity references

- Not processed by XML parser

- Commonly used for scripting code (e.g., JavaScript)

- Begin with : `<![CDATA[`

- Terminate with : `]]>`

Ch1 : CDATA Example

```
1 <?xml version = "1.0"?>
2
3 <!-- Fig. 5.7 : cdata.xml                    -->
4 <!-- CDATA section containing C++ code      -->
5
6 <book title = "C++ How to Program" edition = "3">
7
8    <sample>
9          // C++ comment
10         if ( this-&gt;getX() &lt; 5 &amp;&amp; value[ 0 ] != 3
11            cerr &lt;&lt; this-&gt;displayError();
12   </sample>
13
14   <sample>
15      <![CDATA[
16
17         // C++ comment
18         if ( this->getX() < 5 && value[ 0 ] != 3 )
19            cerr << this->displayError();
20      ]]>
21   </sample>
22
23   C++ How to Program by Deitel &amp; Deitel
24</book>
```

Entity references required if not in **CDATA** section

XML does not process **CDATA** section

Note the simplicity offered by **CDATA** section

Ch1 : Comments in XML

- The syntax for writing comments in XML is similar to that of HTML:

```
<!-- This is a comment -->
```

- Two dashes in the middle of a comment are not allowed:

```
<!-- This is an invalid -- comment -->
```

Ch1 : White-space is Preserved in XML

Java™ Education and Technology Services

- XML does not truncate multiple white-spaces
  (HTML truncates multiple white-spaces to one single white-space):

XML:	Hello Aya
HTML:	Hello Aya

Ch1 : Well-formed XML

XML document Considered ***well formed*** if it has:

1. Single root element.
2. Each element has start tag and end tag.
   - Empty element is defined as: <element/>
3. Tags well nested.
   - Incorrect: `<x><y>hello</x></y>`
   - Correct: `<x><y>hello</y></x>`
4. Attribute values in quotes(single or double).
5. Tag & Attributes names written as variable names:
   - Start with character,
   - One word “must not contain spaces”,
   - Case sensitive.
6. An element may not have two attributes with the same name.

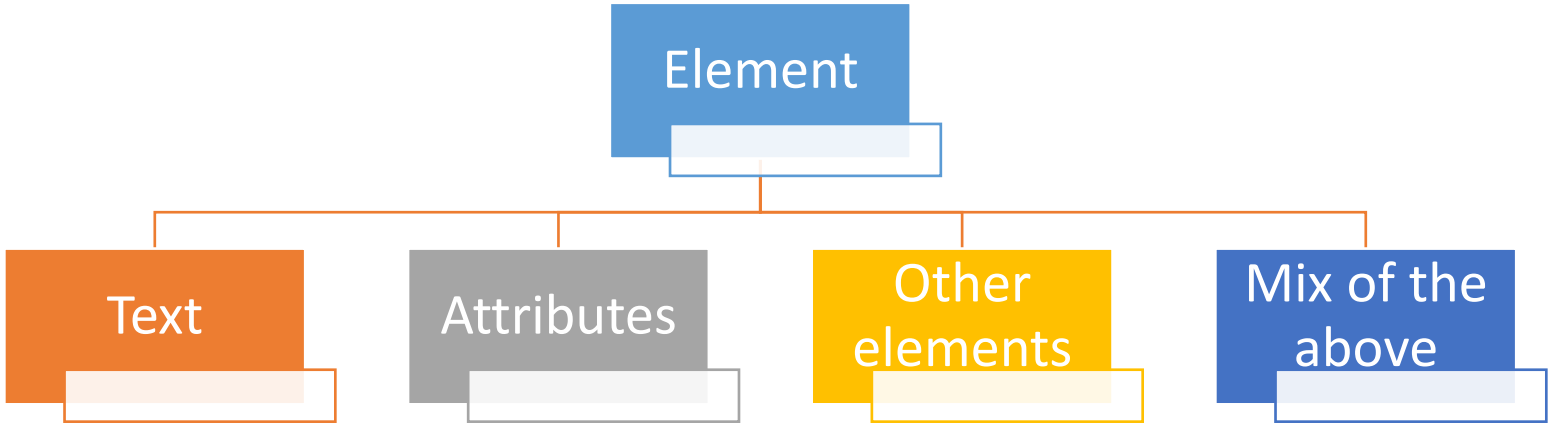
Ch1 : XML Elements

- What is an XML Element?
- Empty XML Elements
- XML Naming Rules
- Best Naming Practices
- Naming Styles

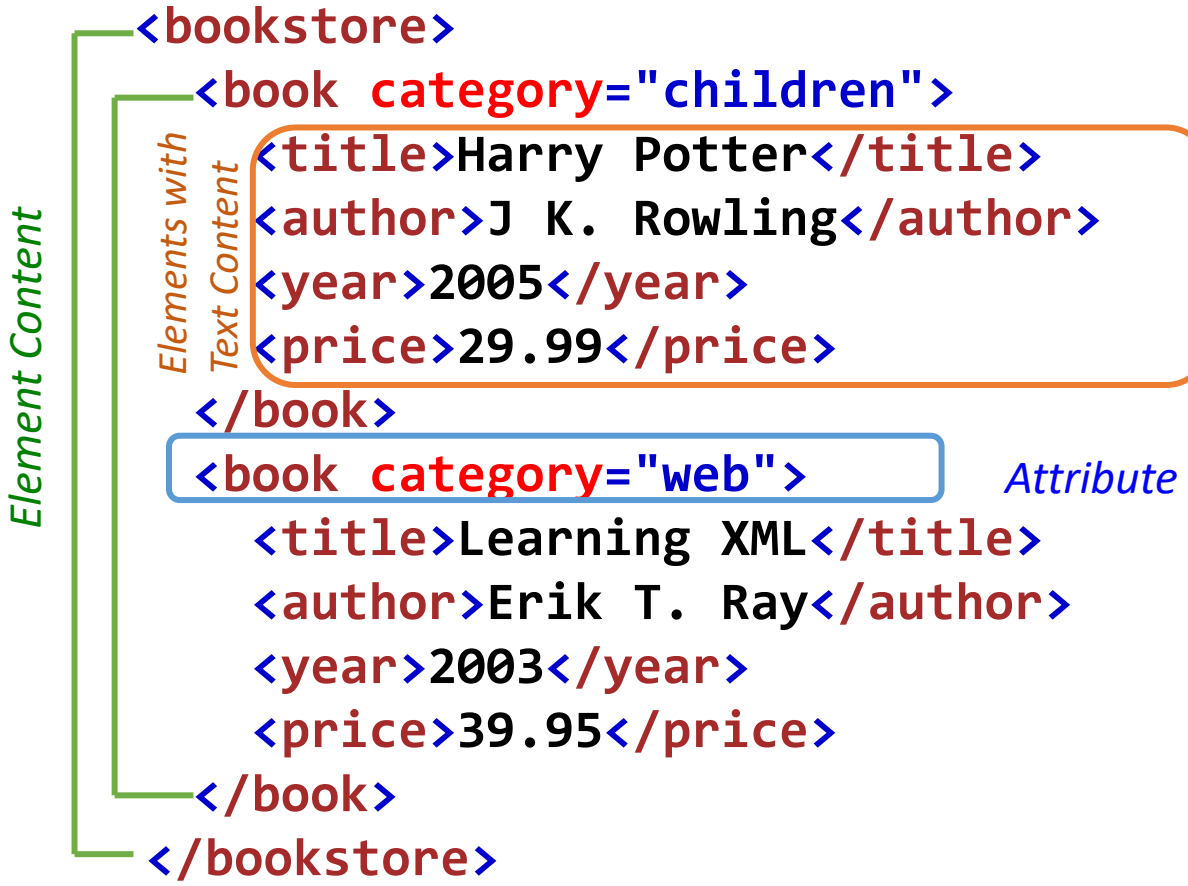
Ch1 : What is an XML Element?

- An XML document contains XML Elements.
- An XML element is everything from (including) the element's start tag to (including) the element's end tag.

```
<price>29.99</price>
```



Ch1 : What is an XML Element?



Ch1 : What is an XML Element?

- In this example:

- <title>, <author>, <year>, and <price> have **text content** because they contain text (like 29.99).

- <bookstore> and <book> have **element contents**, because they contain elements.

- <book> has an **attribute** (category="children").

Ch1 : Empty XML Elements

Java™ Education and Technology Services

- An element with no content is said to be empty.
- Empty elements can have attributes.

```
<element></element>
```

```
<element />
(Self-Closing)
```

- The two forms produce identical results in XML software (Readers, Parsers, Browsers).

Ch1 : XML Naming Rules

Java™ Education and Technology Services

- XML elements must follow these naming rules:
  - **Element names**
    - are case-sensitive
    - must start with a letter or underscore
    - can contain letters, digits, hyphens, underscores, and periods
    - cannot start with the letters xml (or XML, or Xml, etc)
    - cannot contain spaces

Ch1 : Best Naming Practices

- Create descriptive names, like this: <person>, <firstname>, <lastname>.

- Create short and simple names,
  like this: <book_title> not like this: <the_title_of_the_book>.

- Avoid "-" If you name something "first-name",
  some software may think you want to subtract "name" from "first".

- Avoid "." If you name something "first.name",
  some software may think that "name" is a property of the object "first".

- Avoid ":" Colons are reserved for namespaces (more later).

- Non-English letters like éòá are perfectly legal in XML,
  but watch out for problems if your software doesn't support them.

Ch1 : Naming Styles

Java™ Education and Technology Services

- There are no naming styles defined for XML elements. But here are some commonly used:

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Underscore	<first_name>	Underscore separates words
Pascal case	<FirstName>	Uppercase first letter in each word
Camel case	<firstName>	Uppercase first letter in each word except the first

- If you choose a naming style, it is good to be consistent!
- XML documents often have a corresponding database.
- A common practice is to use the naming rules of the database for the XML elements.
- Camel case is a common naming rule in Java, JavaScript.

Ch1 : XML Attributes

- XML Attributes Must be Quoted
- XML Elements vs. Attributes
- My Favorite Way
- Avoid XML Attributes?
- XML Attributes for Metadata

Ch1 : XML Attributes Must be Quoted

- XML elements can have attributes, just like HTML.
- Attributes are designed to contain data related to a specific element.
- Attribute values must always be quoted. Either single or double quotes can be used.

Ch1 : XML Attributes Must be Quoted

Java™ Education and Technology Services

- For a person's gender, the <person> element can be written like this:

- or like this:

```
<person gender="female">
```

- If the attribute value itself contains double quotes you can use single quotes, like this:

```
<person gender='female'>
```

- or you can use character entities:

```
<gangster name='George "Shotgun" Ziegler'>
```

```
<gangster name="George &quot;Shotgun&quot;
Ziegler">
```

Ch1 : XML Elements vs. Attributes

Java™ Education and Technology Services

- Take a look at these examples:

```
<person gender="female">
  <firstname>Aya</firstname>
  <lastname>Amin</lastname>
</person>
```

```
<person>
  <gender>female</gender>
  <firstname>Aya</firstname>
  <lastname>Amin</lastname>
</person>
```

- In the first example gender is an attribute.
- In the last, gender is an element. Both examples provide the same information.

- There are no rules about when to use attributes or when to use elements in XML.

Ch1 : How we do it:

Java™ Education and Technology Services

- The following three XML documents contain exactly the same information:

- A date attribute is used in the first example:
- A <date> element is used in the second example:
- **An expanded <date> element is used in the third example (*Preferred*):**

```
<note date="2020-12-30">
  <to>Aya</to>
  <from>Amin</from>
</note>
```

```
<note>
  <date>2020-12-30</date>
  <to>Aya</to>
  <from>Amin</from>
</note>
```

```
<note>
  <date>
    <year>2020</year>
    <month>12</month>
    <day>30</day>
  </date>
  <to>Aya</to>
  <from>Amin</from>
</note>
```

Ch1 : Avoid XML Attributes?

Java™ Education and Technology Services

- Some things to consider when using attributes are:
  - ✓attributes cannot contain multiple values (elements can)
  - ✓attributes cannot contain tree structures (elements can)
  - ✓attributes are not easily expandable (for future changes)

- **Don't** end up like this:

```
<note day="30" month="12" year="2020"
to="Aya" from="Amin" heading="Reminder"
body="Don't forget me this weekend!">
</note>
```

Ch1 : XML Attributes for Metadata

Java™ Education and Technology Services

- 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.
- This example demonstrates this:

```
<messages>

  <note id="501">
    <to>Aya</to>
    <from>Amin</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>

  <note id="502">
    <to>Aya</to>
    <from>Amin</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>

</messages>
```

- The id attributes above are for identifying the different notes. It is not a part of the note itself.
- **What I'm trying to say here is that metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.**

Ch1 : XML Parser

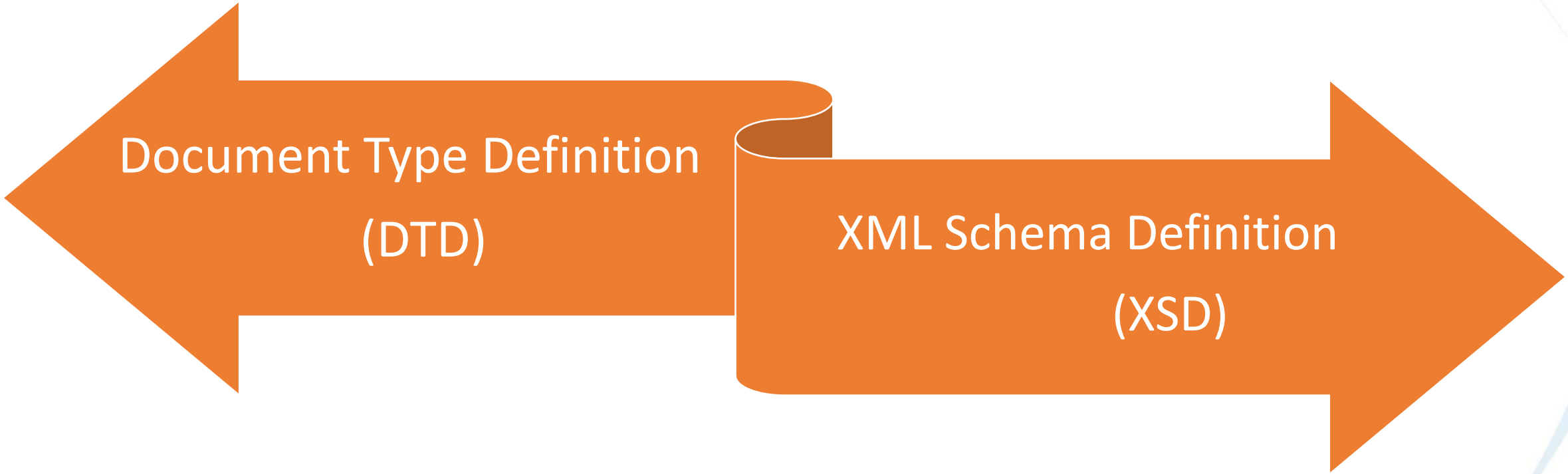
Java™ Education and Technology Services

- XML parser is a software, library or a package that provides interface for client applications to work with XML documents.
- It checks for proper format of the XML document and may also validate the XML documents.

- XML Parser:
  - Processes XML document
  - Reads XML document
  - Checks syntax
  - Reports errors (if any)

- Example:
  - Internet browsers (Chrome, Firefox, Edge, Internet Explorer, …)
  - XML Editors (XmlSpy, Microsoft XML Notepad, XMLQuire, OxygenXML, …)
  - Built-in components in the Java JDK and several 3rd Party Libraries

Ch1 : Validating XML Document



Ch1 : DTD

Java™ Education and Technology Services

- What is a DTD?
  - A DTD is a **D**ocument **T**ype **D**efinition.
  - A DTD defines the structure and the legal elements and attributes of an XML document.
- Why Use a DTD?
  - With a DTD, independent groups of people can agree on a standard DTD for interchanging data.
  - An application can use a DTD to verify that XML data is valid.

Ch1 : An Internal DTD Declaration (Internal subsets)

- If the DTD is declared inside the XML file, it must be wrapped inside the <!DOCTYPE> definition:

```
<?xml version="1.0"?>

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

<note>
        <to>Aya</to>
        <from>Ahmed</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend</body>
</note>
```

Ch1 : An Internal DTD Declaration (Internal subsets)

- **The DTD above is interpreted like this:**
  - !DOCTYPE note defines that the root element of this document is note
  - !ELEMENT note defines that the note element must contain four 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"

Ch1 : An External DTD Declaration (External subsets)

- If the DTD is declared in an external file,
  the <!DOCTYPE> definition must contain a reference to the DTD file:

File "note.xml"

```
<?xml version="1.0"?>

<!DOCTYPE note SYSTEM "note.dtd">

<note>
        <to>Aya</to>
        <from>Amin</from>
        <heading>Reminder</heading>
        <body>Hello</body>
</note>
```

File "note.dtd"

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

Ch1 : DTD limitations

1. Not written in XML syntax, DTD has its own syntax.
   So, it is hard to learn.
2. Precise number of element repetitions can’t be achieved.
3. XML document can reference only 1 DTD.
4. Do not support namespaces
5. No constraints on character data
6. Too simple attribute value models
7. Attributes in DTD can be duplicated to the same element.

Ch1 : What is an XSD?

Java™ Education and Technology Services

•XSD: XML Schema Definition

- An XML Schema describes the structure of an XML document.
- XML Schema is an XML-based (and more powerful) alternative to DTD.
- XML document that conforms to an XML schema is said to be " ***Schema Valid*** "

- The purpose of an XML Schema is to define the legal building blocks of an XML document:
  - the elements and attributes that can appear in a document
  - the number of (and order of) child elements
  - data types for elements and attributes
  - default and fixed values for elements and attributes

Ch1 : XSD Example

Java™ Education and Technology Services

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="https://jets.iti.gov.eg/xml/note"
    xmlns="https://jets.iti.gov.eg/xml/note"
    elementFormDefault="qualified">

      <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>

</xs:schema>
```

Ch1 : A Reference to an XML Schema

Java™ Education and Technology Services

```
<note
    xmlns="https://jets.iti.gov.eg/xml/note"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="https://jets.iti.gov.eg/xml/note note.xsd">

        <to>Aya</to>
        <from>Amin</from>
        <heading>Reminder</heading>
        <body>Don't forget me this weekend!</body>

</note>
```

Lab Exercise

Java™ Education and Technology Services

- **1st Assignment: A Configuration File**
  - Design a configuration file for a library.
    - Info. of library consists of a location, a description of the library, a librarian and a lot of books.
    - Each book has title, ISBN, and Author.
    - The book contains also a preface and many of parts.
    - Each part has title and contains many of chapters.
    - Each chapter has title and contains a summary and many of sections.
    - Sections contain the content of the book as paragraphs.

XML must have elements (usual and empty), attributes, and CDATA section

Chapter 2

Java™ Education and Technology Services

JSON

Ch2 : Introduction to JSON

- Stands for **J**ava **S**cript **O**bject **N**otation

- What is JSON?
  - JSON is a syntax for storing and exchanging text information.
  - JSON is lightweight text-data interchange format
  - JSON is language independent.

- JSON is smaller than XML, and faster and easier to parse.

Ch2 : Introduction to JSON

Java™ Education and Technology Services

- JSON Files:
  - The filename extension is **".json"**
  - JSON Internet Media type is **"application/json"**

- All modern programming languages support JSON.

Ch2 : JSON vs XML

- Much Like XML:
  - JSON is plain text
  - JSON is "self-describing" (human readable)
  - JSON is hierarchical (values within values)
  - JSON can be parsed by JavaScript
  - JSON data can be transported using AJAX

- Much Unlike XML:
  - No end tag
  - Shorter
  - Quicker to read and write
  - Can be parsed using built-in JavaScript eval()
  - Uses arrays
  - No reserved words

Ch2 : Why JSON?

- For AJAX applications, JSON is faster & easier than XML:

- Using XML
  - Fetch an XML document
  - Use the XML DOM to loop through the document
  - Extract values and store in variables

- Using JSON
  - Fetch a JSON string
  - eval() the JSON string

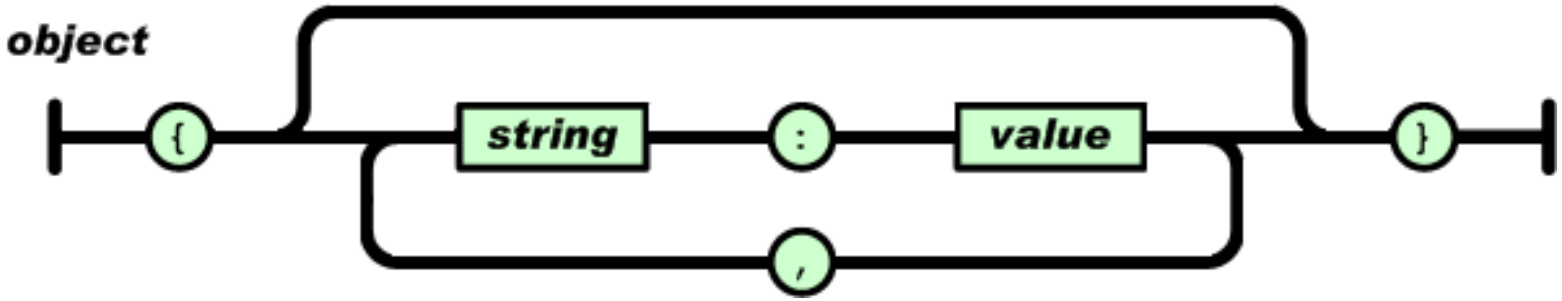
Ch2 : Example


```
{
    "book": [
    {
        "id":100,
        "language":"Java",
        "edition":"third",
        "author":"Herbert Schildt"
    },
    {
        "id":200,
        "language":"C++",
        "edition":"second",
        "author":"E.Balagurusamy"
    }]
}
```

Ch2 : JSON Structure

- **JSON is built on two structures:**
  - ***A collection of name/value pairs.***
    - Like an *object*, record, struct, dictionary, hash table.
  - ***An ordered list of values.***
    - Like an *array*, vector, list, or sequence.

Ch2 : JSON Object



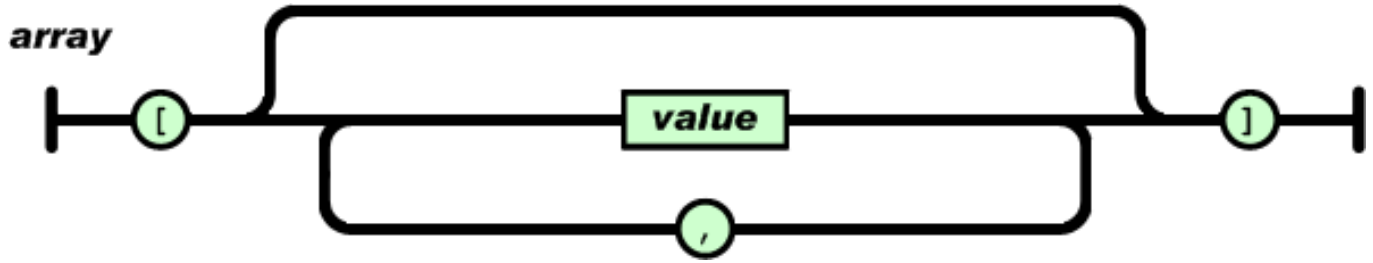
- An ***object***
  - is an unordered set of name/value pairs.
  - An object begins with **{** and ends with **}**
  - Each name is followed by **:** and the name/value pairs are separated by **,**

- Example:

```
{ "firstName":"Ahmed" , "lastName":"Omar" }
```

Ch2 : JSON Array

- **An *array***
  - is an ordered collection of values.
  - An array begins with [and ends with]
  - Values are separated by ,



- **Example:**

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

Ch2 : JSON Value

- A ***value*** can be
  - a *string* in double quotes,
  - a *number*,
  - true or false
  - null,
  - an *object*
  - an *array*

```
name:"Amin"
```

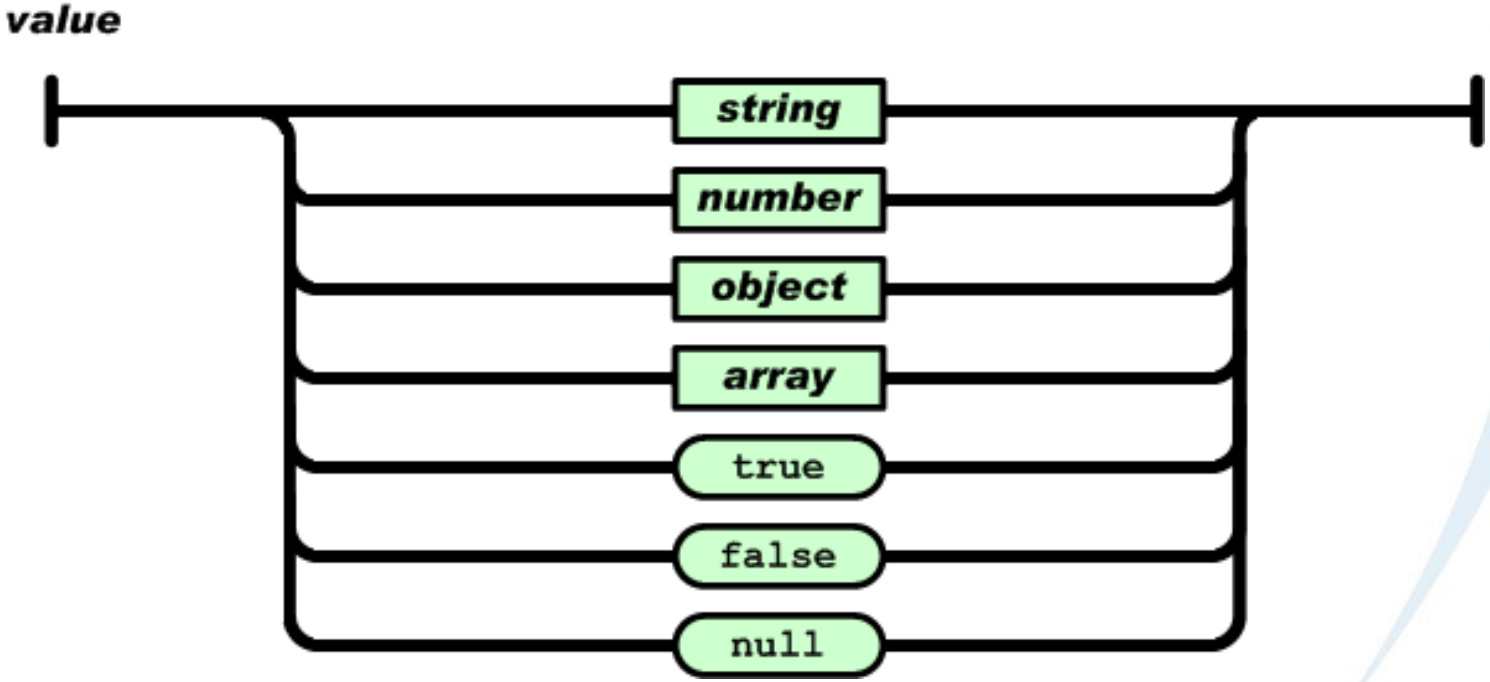
```
id:100
```

```
flag:true
```

```
MyVar:null
```

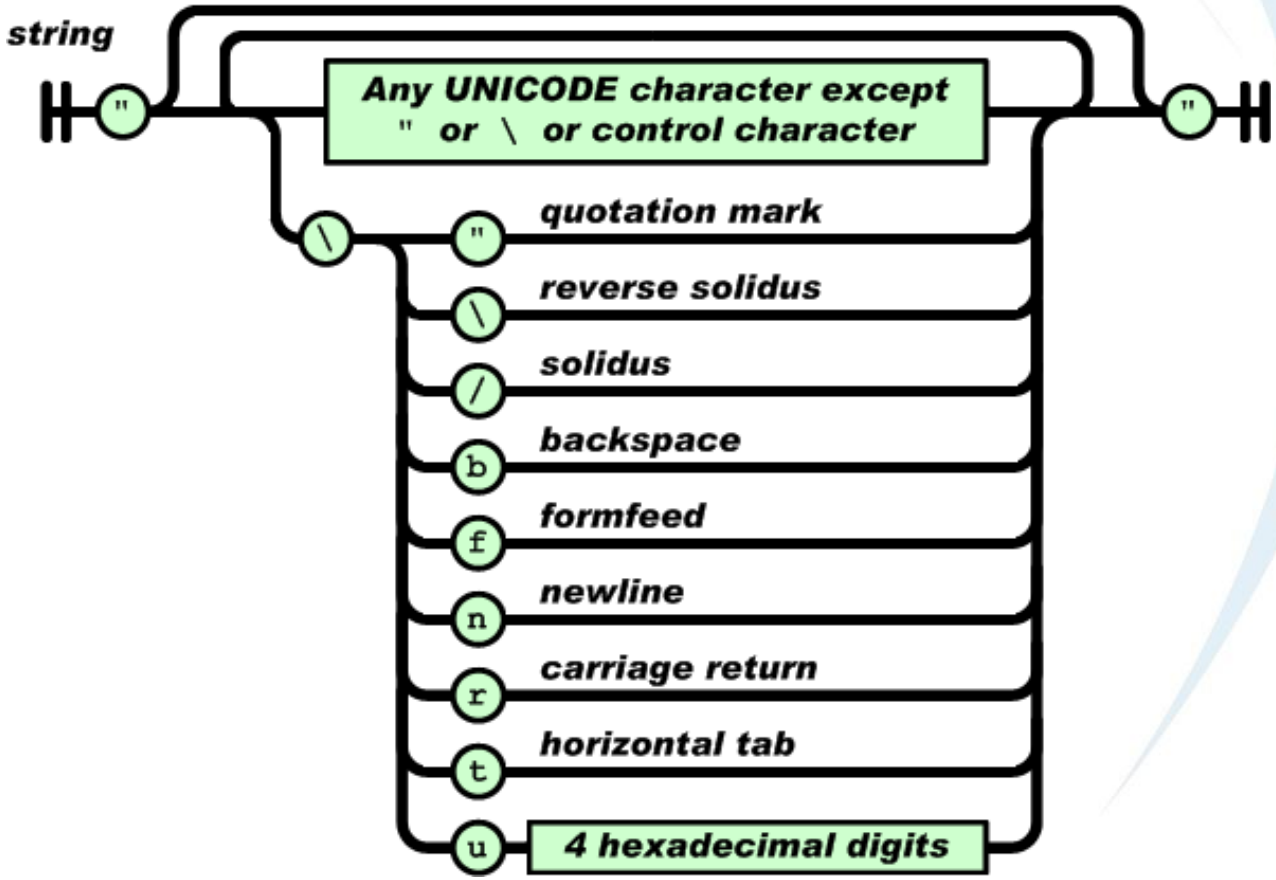
```
{"id":1,"x":"y"}
```

```
"Students":[...,...]
```



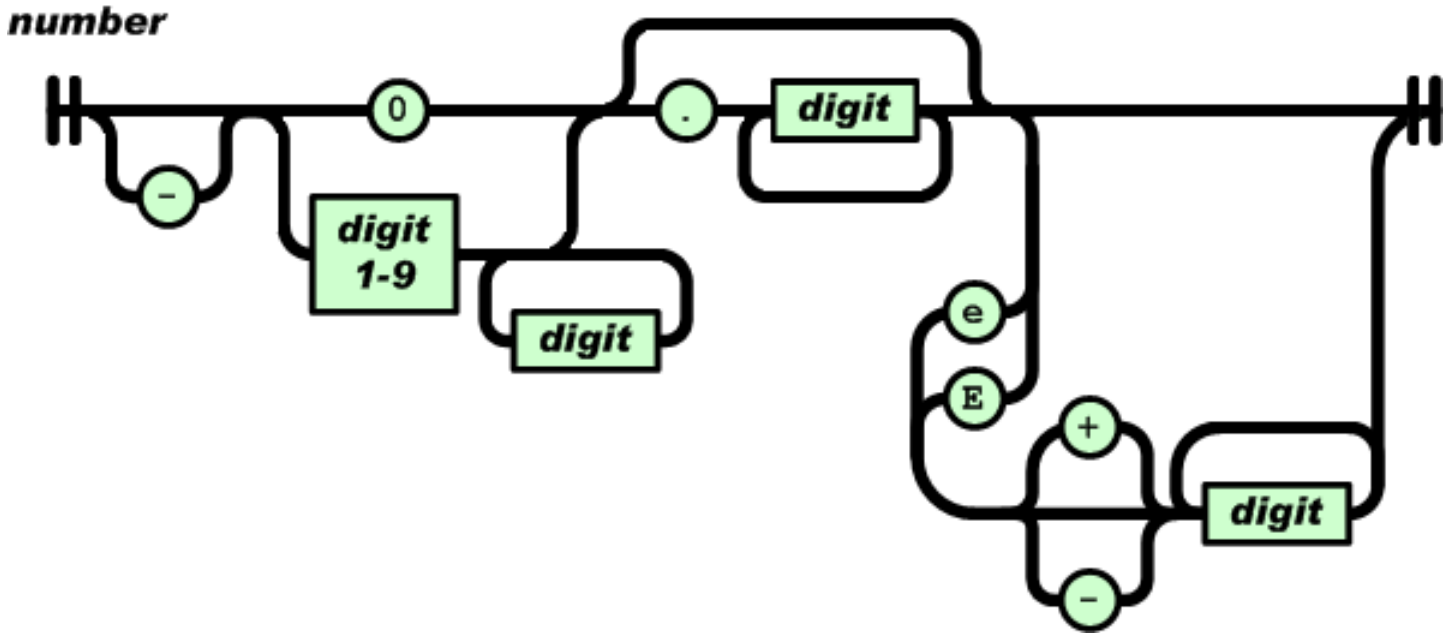
Ch2 : String

- A ***string*** is
  - Very much like a C or Java string.
  - A sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes.
  - A character is represented as a single character string.



Ch2 : Number

- A ***number*** is
  - Very much like a C or Java number.
  - Except that the octal and hexadecimal formats are not used.



Ch2 : JSON Technologies

Java™ Education and Technology Services

- JSON started unlike XML without any Validation Rules

- After the widespread usage of JSON,
  new Technologies emerged related to JSON similar to XML Technologies.

- **JSON Schema**: to validate JSON Documents
- **JSON Pointer**: to navigate and fetch data from JSON Documents
- And new JSON Technologies are still emerging

- Both XML and JSON are widely Adopted Data Exchange Formats

Lab Exercise

- **2nd Assignment: A JSON File**

- Write a JSON file for a library.
  - Info. of library consists of a location, a description of the library, a librarian and an array of books.
  - Each book has title, ISBN, and Author.
  - The book contains also a preface and many of parts.
  - Each part has title and contains many of chapters.
  - Each chapter has title and contains a summary and many of sections.
  - Sections contain the content of the book as paragraphs.

Chapter 3

Java™ Education and Technology Services

JSON-P

JSON Usage

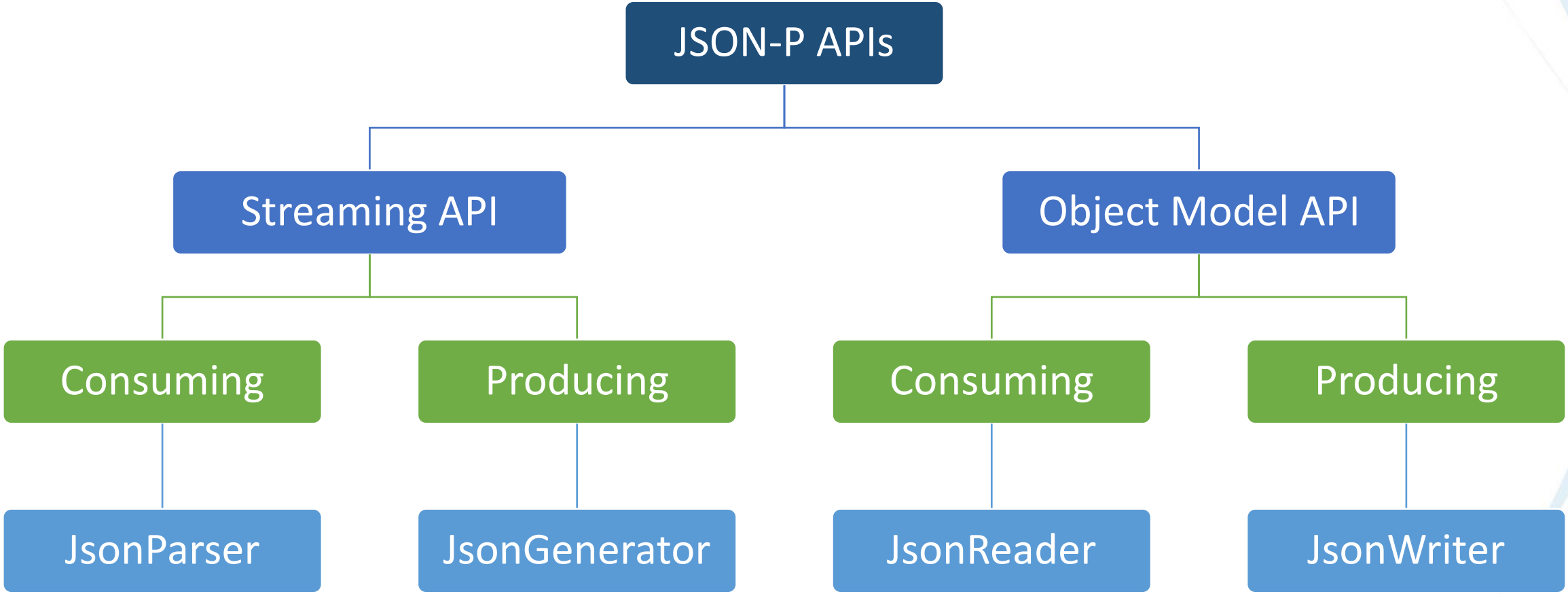
Java™ Education and Technology Services

- JSON is often used as a common format to serialize and deserialize data in applications that communicate with each other over the Internet.
- These applications are created using different programming languages and run in very different environments.
- JSON is suited to this scenario because it is an open standard, it is easy to read and write, and it is more compact than other representations.
- RESTful web services use JSON extensively as the format for the data inside requests and responses.

JSON Processing

- Java API introduced two ways for JSON Processing:
  - Java EE 7 introduced JSON-P, the Java API for **JSON Processing**
  - Java EE 8 introduced an additional JSON API, namely, the Java API for **JSON Binding** (**JSON-B**)

Generating and Parsing JSON Data Using JSON-P



Generating and Parsing JSON Data Using JSON-P

- For generating and parsing JSON data, there are two programming models, which are similar to those used for XML documents.
  - The **streaming model** uses an event-based parser that reads JSON data one element at a time.
  - The parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value.
  - Each element can be processed or discarded by the application code, and then the parser proceeds to the next event.
  - This approach is adequate for local processing, in which the processing of an element does not require information from the rest of the data.

  - The **object model** creates a tree that represents the JSON data in memory.
  - The tree can then be navigated, analyzed, or modified.
  - This approach is the most flexible and allows for processing that requires access to the complete contents of the tree.
  - However, it is often slower than the streaming model and requires more memory.

JSON APIs

- The Java API for JSON Processing contains the following packages:
  - The **javax.json** package contains
    - a reader interface,
    - a writer interface, and
    - a model builder interface for the object model.

    This package also contains other utility classes and Java types for JSON elements.
  - The **javax.json.stream** package contains
    - a parser interface and
    - a generator interface for the streaming model.

Streaming API

Streaming API

- The Streaming API provides a way to parse and generate JSON text in a streaming fashion (a low-level API).
- The API provides an event-based parser and allows an application developer to ask for the next event (i.e., pull the event), rather than handling the event in a callback.
- The streaming model is adequate for local processing where only specific parts of the JSON structure need to be accessed, and random access to other parts of the data is not required.
- Parser events can be processed or discarded, or the next event may be generated.
- Other JSON frameworks (such as JSON binding) is implemented with this API.

Consuming JSON Using the Streaming API

- **JsonParser** contains methods to parse JSON data using the streaming model.

- Json Parser provides ***forward***, ***read-only*** access to JSON data using the pull parsing programming model.

- In this model, the application code controls the thread and calls methods in the parser interface to move the parser forward or to obtain JSON data from the current state of the parser.

```
JsonParser parser = Json.createParser( InputStream OR Reader );
```

- https://docs.oracle.com/javaee/7/api/javax/json/stream/JsonParser.html

Producing JSON Using the Streaming API

Java™ Education and Technology Services

- The Streaming API provides a way to generate well-formed JSON to a stream by writing one event at a time.

- **JsonGenerator** contains **writeXXX** methods to write name/value pairs in JSON objects and values in JSON arrays.

```
JsonGenerator gen = Json.createGenerator( OutputStream OR Writer );
```

- https://docs.oracle.com/javaee/7/api/javax/json/stream/JsonGenerator.html

Object Model API

Object Model API

Java™ Education and Technology Services

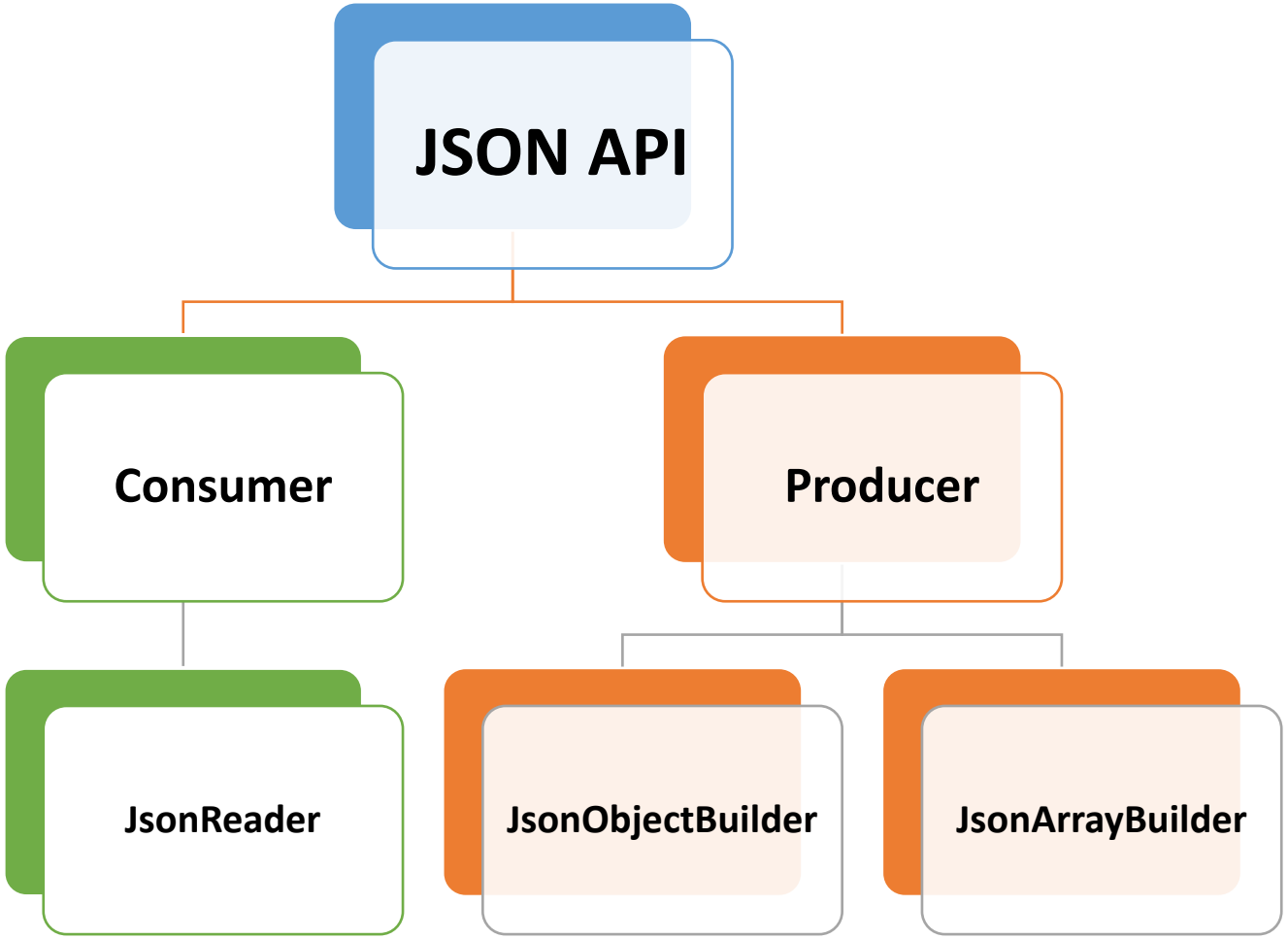
- The Object Model API is a high-level API that provides immutable object models for JSON object and array structures.

- These JSON structures are represented as object models via the Java types **JsonObject** and **JsonArray**.

- **JsonObject** provides a Map view to access the unordered collection of zero or more name/value pairs from the model.

- Similarly, **JsonArray** provides a List view to access the ordered sequence of zero or more values from the model.

Object Model API



Consuming JSON Using the Object Model API

- Interfaces you will deal with are:
  - JsonReader
  - JsonStructure
    - JsonObject
    - JsonArray

Consuming JSON Using the Object Model API

Java™ Education and Technology Services

- **JsonReader** contains methods to read JSON data using the object model from an input source.

- **JsonReader** can be created from InputStream:
- This code shows how to create a new reader from an InputStream obtained from a new FileInputStream.

```
JsonReader reader = Json.createReader(new FileInputStream(...));
```

- **JsonReader** can also be created from Reader:
- This code shows how to create a reader from a FileReader.

```
JsonReader reader = Json.createReader(new FileReader(...));
```

Example Using StringReader

- An object with two name/value pairs can be read as:

```
jsonReader = Json.createReader(new StringReader("{"
                                + " \"apple\":\"red\","
                                + " \"banana\":\"yellow\""
                                + "}"));

JsonObject json = jsonReader.readObject();
json.getString("apple");
json.getString("banana");
```

```
{
apple:”red”,
“banana”:”yellow”
}
```

- In this code,
  the getString method returns the string value for the specific key in the object.

- Other ***getXXX()*** methods can be used to access the value based upon the data type.

Example

- An array with two objects with each object with a name/value pair can be read as:

```
jsonReader = Json.createReader(new StringReader("["
                                + " { \"apple\":\"red\" },"
                                + " { \"banana\":\"yellow\" }"
                                + "]"));

JsonArray jsonArray = jsonReader.readArray();
```

```
[
{apple:”red”},
{“banana”:”yellow”}
]
```

- In this code, calling the readArray method returns an instance of the **JsonArray** interface. This interface has convenience methods to get boolean, integer, and String values at a specific index.

- This interface extends from **java.util.List**,
  so usually the list operations are available as well.

Example

- A nested structure can be read as:

```
jsonReader = Json.createReader(new FileReader("film.json"));

JsonObject movieJsonObject =(JsonObject) jsonReader.read();
JsonArray cast = movieJsonObject.getJsonArray("cast");
for(int i=0;i<cast.size();i++){
        System.out.println(cast.getString(i));
}
```

film.json

```
{
  “title”:”The Matrix”,
  “year”:1999,
  “cast”:[
      “Keanu Reeves“,
     ” Laurence Fishburne”,
     ” Carrie-Anne Moss”]
}
```

Producing JSON Using the Object Model API

- Interfaces you will deal with are:
  - JsonObjectBuilder
  - JsonArrayBuilder

Producing JSON Using the Object Model API

Java™ Education and Technology Services

- **JsonObjectBuilder** can be used to create models that represent JSON objects.
- The resulting model is of type JsonObject.

```
JsonObjectBuilder objectBuilder = Json.createObjectBuilder();
```

- Similarly, **JsonArrayBuilder** can be used to create models that represent JSON arrays where the resulting model is of type **JsonArray**:

```
JsonArrayBuilder objectBuilder = Json.createArraytBuilder();
```

Example

```
JsonObject model = Json.createObjectBuilder()
 .add("firstName","Duke")
 .add("lastName","Java")
 .add("age", 18)
 .add("city", "JavaTown")
 .add("phoneNumbers", Json.createArrayBuilder()
        .add(Json.createObjectBuilder()
                .add("type", "mobile")
                .add("number", "111-111-1111"))
        .add(Json.createObjectBuilder()
                .add("type", "home")
                .add("number", "222-222-2222")))
 .build();
```

```
{
    “firstName”: ” Duke”,
    “lastName”: ” Java”,
    “age”:18,
    "city“: "JavaTown“,
   "phoneNumbers“: [
        {"type“: "mobile“,
        "number“: "111-111-1111"},
        {"type“: "home“,
         "number“: "222-222-2222"}
     ]
}
```

Creating a JsonWriter

- The generated JsonObject can be written to an output stream via **JsonWriter**:

```
JsonWriter writer = Json.createWriter(System.out);
writer.writeObject(jsonObject);
writer.writeArray(jsonArray);
```

- In this code, a new **JsonWriter** instance is created and configured to write to **System.out**.
- The previously created **jsonObject and jsonArray** is then written when the **writeObject() and writeArray()** methods are called.
- **JsonWriter** may be configured to write to a Writer as well.

DEMO

- Consuming JSON (Using Object Model API - ***JsonReader***)
  - Write a Simple JSON Document
  - Create a JSON Reader, to read JSON object/array from that Document
  - Use *getXXX()* methods to retrieve data from those objects

- Producing JSON (Using Object Model API – ***JsonWriter***)
  - Create a Json Writer
  - Create a JSON Structure (Object/Array) using Builders
  - Write out those JSON Structures
  - Write the JSON Contents to a File by flushing/closing the generator

Chapter 4

Java™ Education and Technology Services

JSON-B

Introduction

What is JSON-B?

JSON-B is a standard binding layer for converting Java objects to/from JSON messages.

It defines a default mapping algorithm for converting existing Java classes to JSON, while enabling developers to customize the mapping process through the use of Java annotations.

Java API for JSON Binding

JEE8: JSR 367 Specification

Mapping an Object

- The main JSON-B entry point is ***Jsonb*** class.
- It provides all necessary methods to serialize and deserialize Java objects.
- Jsonb instances are thread safe. They can be cached and reused.

```
public class Dog { public String name; public int age; public boolean bitable; }

// Create a dog instance
Dog dog = new Dog();
dog.name = "Falco";
dog.age = 4;
dog.bitable = false;

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dog);

// Deserialize back
dog = jsonb.fromJson("{\"age\":4,\"bitable\":false,\"name\":\"Falco\"}", Dog.class);
```

Mapping a raw Collection

- JSON-B supports native arrays serialization.

```
// List of dogs
Dog[] dogs = new Dog[] {
                new Dog( ... ),
                new Dog( ... )
};

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dogs);

// We can also deserialize back into a raw collection,
dogs = jsonb.fromJson(result, Dog[].class);
```

Mapping a raw Collection

- JSON-B supports raw collections serialization.

```
// List of dogs
List dogs = new ArrayList();
dogs.add(falco);
dogs.add(cassidy);

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dogs);

// We can also deserialize back into a raw collection,
// but since there is no way to infer a type here,
// the result will be a list of java.util.Map instances with string keys.
dogs = jsonb.fromJson(result, new ArrayList().getClass());
```

Mapping a Generic Collection

Java™ Education and Technology Services

- JSON-B supports collections and generic collections handling.
- For proper deserialization,
  the runtime type of resulting object needs to be passed to JSON-B during deserialization.

```
// List of dogs
List<Dog> dogs = new ArrayList<>();
dogs.add(falco);
dogs.add(cassidy);

// Create Jsonb and serialize
Jsonb jsonb = JsonbBuilder.create();
String result = jsonb.toJson(dogs);

// Deserialize back
dogs = jsonb.fromJson(result,
            new ArrayList<Dog>(){}.getClass().getGenericSuperclass());
```

Gson

Google Gson

- Google Gson is an open source,
  Java-based library developed by Google.

- It facilitates serialization of Java objects to JSON and vice versa.

- This tutorial adopts a simple and intuitive way to describe the
  basic-to-advanced concepts of Google Gson and how to use its APIs.

- Available at : github.com/google/gson

Example

- Before going into the details of the Google Gson library, let's see an application in action.

- In this example, we've created a **Student class**.

- We'll create a JSON string with student details and deserialize it to student object and then serialize it to an JSON String.

Student Class


```
public class Student {

   private String name;
   private int age;

   public Student(){ }

   public String getName() { return name; }

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

   public int getAge() { return age; }

   public void setAge(int age) { this.age = age; }

   public String toString() {
      return "Student [ name: "+name+", age: "+ age+ " ]";
   }

}
```

Steps to Remember

- **Step 1 –** *Create Gson object using GsonBuilder*

```
GsonBuilder builder = new GsonBuilder();
builder.setPrettyPrinting();
Gson gson = builder.create();
```

- **Step 2 –** *Deserialize JSON to Object*

```
//Object to JSON Conversion
Student student = gson.fromJson(jsonString, Student.class);
```

- **Step 3 –** *Serialize Object to JSON*

```
//Object to JSON Conversion
jsonString = gson.toJson(student);
```

Tester Class

```
public class GsonTester {

   public static void main(String[] args) {
      String jsonString = "{\"name\":\"Ahmed Ali\", \"age\":25}";

      GsonBuilder builder = new GsonBuilder();
      builder.setPrettyPrinting();

      Gson gson = builder.create();
      Student student = gson.fromJson(jsonString, Student.class);
      System.out.println(student);

      jsonString = gson.toJson(student);
      System.out.println(jsonString);
   }

}
```

```
Student [ name: Ahmed Ali, age: 25 ]

{
  "name": "Ahmed Ali",
  "age": 25
}
```

Gson Class

- Gson is the main actor class of Google Gson library.
- It provides functionalities to convert Java objects to matching JSON constructs and vice versa.
- Gson is first constructed using GsonBuilder and then, toJson(Object) or fromJson(String, Class) methods are used to read/write JSON constructs.

- Java Doc API for Gson
  - https://www.javadoc.io/doc/com.google.code.gson/gson/latest/com.google.gson/com/google/gson/Gson.html

Gson - Object Serialization

- Let's serialize a Java object to a Json file
  and then read that Json file to get the object back.

- In this example, we've created a Student class.

- We'll generate a **student.json** file which will have
  a json representation of Student object.

GsonTester Class

```
public static void main(String[] args) {
   GsonTester tester = new GsonTester();
   try {
      Student student = new Student();
      student.setAge(25);
      student.setName("Ahmed Ali");
      tester.writeJSON(student);
      Student student1 = tester.readJSON();
      System.out.println(student1);
   } catch (IOException e) {
      e.printStackTrace();
   }
}
```

GsonTester Class

Java™ Education and Technology Services

```
private void writeJSON(Student student) throws IOException {
   GsonBuilder builder = new GsonBuilder();
   builder.setPrettyPrinting();
   Gson gson = builder.create();
   FileWriter writer = new FileWriter("student.json");
   writer.write(gson.toJson(student));
   writer.close();
}
```

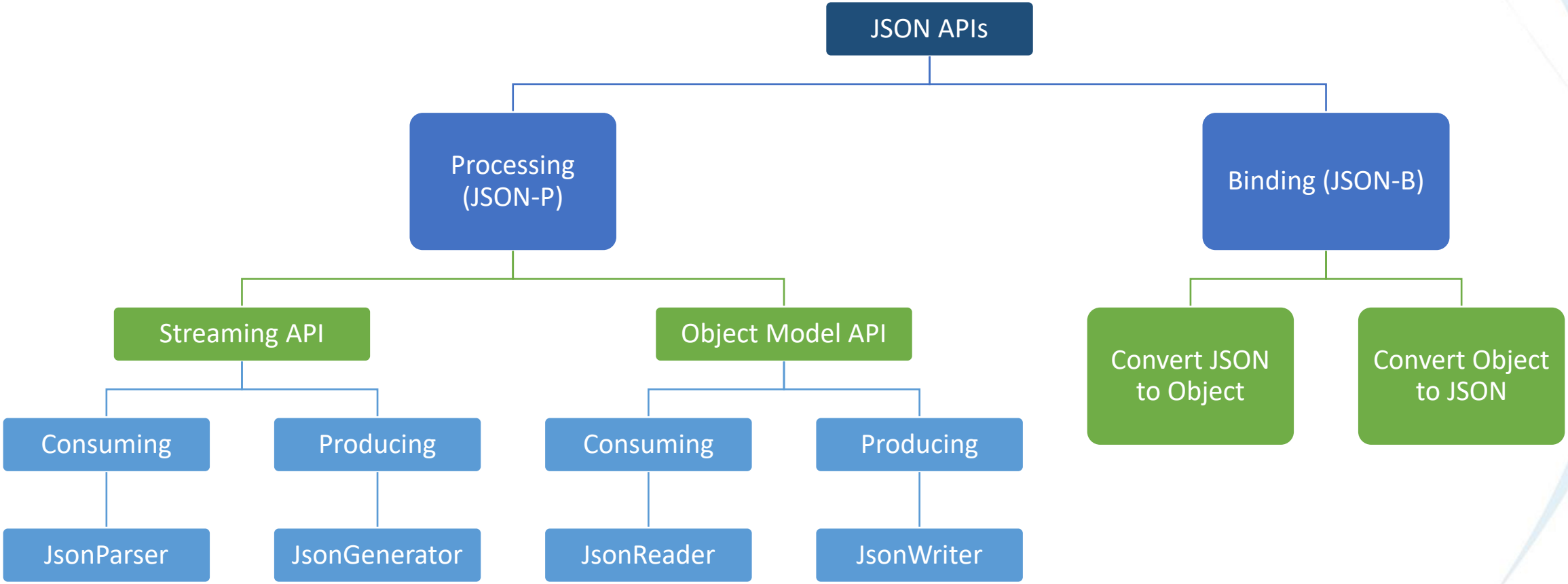
GsonTester Class

Java™ Education and Technology Services

```
private Student readJSON() throws FileNotFoundException {
   GsonBuilder builder = new GsonBuilder();
   Gson gson = builder.create();
   BufferedReader bufferedReader = new BufferedReader(new FileReader("student.json"));

   Student student = gson.fromJson(bufferedReader, Student.class);
   return student;
}
```

Summary



Lab Exercise

- **Assignments:**

- This JSON Data represents company Data.
- Create classes that correspond to this JSON data.
- Create application that uses JSON Object Model to read and write company object.
- Create application that uses GSON to serialize and deserialize company object.
- Create application that uses Jackson to serialize and deserialize company object.

```
{
 “name": "TechCorp",
 "location": "New York",
 "employees": [
  {
   "id": 2,
   "name": "Bob Smith",
   "age": 45,
   "position": "Project Manager",
   "salary": 120000,
   "contact": {
    "email": "bob.smith@techcorp.com",
    "phone": "555-5678"
   }
  },
  {
   "id": 3,
   "name": "Charlie Brown",
   "age": 28,
   "position": "UI/UX Designer",
   "salary": 85000,
   "contact": {
    "email": "charlie.brown@techcorp.com",
    "phone": "555-8765"
   }
  }
 ]
}
```

References & Recommended Reading

- Book: Java XML and JSON
- Book: JavaEE 7 Essentials

- Java SE APIs for XML
  - https://docs.oracle.com/javase/tutorial/jaxp/index.html
  - https://docs.oracle.com/javase/tutorial/jaxb/index.html

- Java EE APIs for JSON
  - https://javaee.github.io/jsonp/
  - https://javaee.github.io/jsonb-spec/

References & Recommended Reading

- Learning XML and JSON:
  - https://www.w3schools.com/xml
  - https://www.w3schools.com/js/js_json_intro.asp
- JSON Binding API User Guide
  - http://json-b.net/docs/user-guide.html
- Guides to XML, JSON in Java
  - https://www.baeldung.com/java-xml
  - https://www.baeldung.com/java-json