Sure! Let’s build a simple **XML parser in Java** from scratch.

We’ll:  
✅ Read an XML string or file.  
✅ Parse the elements, attributes, and text.  
✅ Store them in a tree-like structure (like a DOM).

**1️⃣ Create a data structure for XML Nodes**

import java.util.\*;

public class XmlNode {

private String name;

private String textContent;

private Map<String, String> attributes;

private List<XmlNode> children;

public XmlNode(String name) {

this.name = name;

this.attributes = new HashMap<>();

this.children = new ArrayList<>();

this.textContent = "";

}

public void addAttribute(String key, String value) {

attributes.put(key, value);

}

public void addChild(XmlNode child) {

children.add(child);

}

public void setTextContent(String text) {

this.textContent = text;

}

public String getName() {

return name;

}

public Map<String, String> getAttributes() {

return attributes;

}

public List<XmlNode> getChildren() {

return children;

}

public String getTextContent() {

return textContent;

}

@Override

public String toString() {

return toString(0);

}

private String toString(int indent) {

StringBuilder sb = new StringBuilder();

String indentStr = " ".repeat(indent);

sb.append(indentStr).append("<").append(name);

for (Map.Entry<String, String> attr : attributes.entrySet()) {

sb.append(" ").append(attr.getKey()).append("=\"").append(attr.getValue()).append("\"");

}

sb.append(">");

if (!textContent.isEmpty()) {

sb.append(textContent);

}

if (!children.isEmpty()) {

sb.append("\n");

for (XmlNode child : children) {

sb.append(child.toString(indent + 1)).append("\n");

}

sb.append(indentStr);

}

sb.append("</").append(name).append(">");

return sb.toString();

}

}

**2️⃣ Create a simple XML Parser**

This is a **basic implementation** (not as powerful as SAX or DOM but good for understanding):

public class SimpleXmlParser {

private String xml;

private int pos;

public XmlNode parse(String xmlContent) {

this.xml = xmlContent;

this.pos = 0;

skipWhitespace();

return parseNode();

}

private XmlNode parseNode() {

if (!xml.startsWith("<", pos)) {

throw new RuntimeException("Expected < at position " + pos);

}

pos++; // Skip '<'

String tagName = parseTagName();

XmlNode node = new XmlNode(tagName);

// Parse attributes

while (true) {

skipWhitespace();

if (xml.charAt(pos) == '>') {

pos++; // Skip '>'

break;

} else if (xml.charAt(pos) == '/') {

// Self-closing tag

pos += 2; // Skip "/>"

return node;

} else {

String attrName = parseTagName();

skipWhitespace();

if (xml.charAt(pos) != '=') {

throw new RuntimeException("Expected = after attribute at position " + pos);

}

pos++; // skip '='

skipWhitespace();

String attrValue = parseAttributeValue();

node.addAttribute(attrName, attrValue);

}

}

// Parse children or text content

StringBuilder textContent = new StringBuilder();

while (!xml.startsWith("</" + tagName, pos)) {

skipWhitespace();

if (xml.charAt(pos) == '<') {

if (xml.charAt(pos + 1) == '/') {

break; // End tag

}

XmlNode child = parseNode();

node.addChild(child);

} else {

textContent.append(parseText());

}

}

if (textContent.length() > 0) {

node.setTextContent(textContent.toString().trim());

}

// Skip end tag

if (!xml.startsWith("</" + tagName, pos)) {

throw new RuntimeException("Expected </" + tagName + "> at position " + pos);

}

pos += 2 + tagName.length(); // Skip "</tagName"

if (xml.charAt(pos) != '>') {

throw new RuntimeException("Expected > at position " + pos);

}

pos++; // Skip '>'

return node;

}

private String parseTagName() {

skipWhitespace();

int start = pos;

while (Character.isLetterOrDigit(xml.charAt(pos)) || xml.charAt(pos) == '-') {

pos++;

}

return xml.substring(start, pos);

}

private String parseAttributeValue() {

if (xml.charAt(pos) != '"' && xml.charAt(pos) != '\'') {

throw new RuntimeException("Expected \" or ' at position " + pos);

}

char quote = xml.charAt(pos);

pos++; // skip quote

int start = pos;

while (xml.charAt(pos) != quote) {

pos++;

}

String value = xml.substring(start, pos);

pos++; // skip quote

return value;

}

private String parseText() {

int start = pos;

while (pos < xml.length() && xml.charAt(pos) != '<') {

pos++;

}

return xml.substring(start, pos);

}

private void skipWhitespace() {

while (pos < xml.length() && Character.isWhitespace(xml.charAt(pos))) {

pos++;

}

}

}

**3️⃣ Test the Parser**

public class Main {

public static void main(String[] args) {

String xml = """

<note date="2025-07-17">

<to>Tin</to>

<from>ChatGPT</from>

<heading>Reminder</heading>

<body>Build your own XML parser!</body>

</note>

""";

SimpleXmlParser parser = new SimpleXmlParser();

XmlNode root = parser.parse(xml);

System.out.println("Parsed XML Tree:");

System.out.println(root);

}

}

✅ **Output:**

<note date="2025-07-17">

<to>Tin</to>

<from>ChatGPT</from>

<heading>Reminder</heading>

<body>Build your own XML parser!</body>

</note>

Do you want me to:

✅ **Make it support namespaces and comments?**  
✅ Or keep it **simple for learning purposes**?  
✅ Or should I show you **a version using javax.xml.parsers (DOM/SAX)** for production-level parsing?

Perfect! The **Composite Pattern** is *exactly* what fits for representing XML because:

✅ Each **XmlNode** can be a **composite** (with children) or a **leaf** (no children).  
✅ Both leaves and composites share the same interface (XmlNode).

Let’s refactor the parser to apply the **Composite Pattern**:

**🌿 1. Define XmlComponent (the common interface)**

import java.util.Map;

public interface XmlComponent {

String getName();

String getTextContent();

Map<String, String> getAttributes();

void print(String indent);

}

**🌿 2. Create XmlElement (Composite)**

This represents XML elements that may have children.

import java.util.\*;

public class XmlElement implements XmlComponent {

private String name;

private Map<String, String> attributes;

private List<XmlComponent> children;

public XmlElement(String name) {

this.name = name;

this.attributes = new HashMap<>();

this.children = new ArrayList<>();

}

public void addAttribute(String key, String value) {

attributes.put(key, value);

}

public void addChild(XmlComponent child) {

children.add(child);

}

@Override

public String getName() {

return name;

}

@Override

public String getTextContent() {

return "";

}

@Override

public Map<String, String> getAttributes() {

return attributes;

}

@Override

public void print(String indent) {

System.out.print(indent + "<" + name);

for (Map.Entry<String, String> attr : attributes.entrySet()) {

System.out.print(" " + attr.getKey() + "=\"" + attr.getValue() + "\"");

}

System.out.println(">");

for (XmlComponent child : children) {

child.print(indent + " ");

}

System.out.println(indent + "</" + name + ">");

}

}

**🌿 3. Create XmlTextNode (Leaf)**

This represents text content.

public class XmlTextNode implements XmlComponent {

private String text;

public XmlTextNode(String text) {

this.text = text;

}

@Override

public String getName() {

return "#text";

}

@Override

public String getTextContent() {

return text;

}

@Override

public Map<String, String> getAttributes() {

return Map.of(); // No attributes

}

@Override

public void print(String indent) {

System.out.println(indent + text);

}

}

**🛠️ 4. Update the parser**

Here we update SimpleXmlParser to return XmlComponent.

public class SimpleXmlParser {

private String xml;

private int pos;

public XmlComponent parse(String xmlContent) {

this.xml = xmlContent;

this.pos = 0;

skipWhitespace();

return parseNode();

}

private XmlComponent parseNode() {

if (!xml.startsWith("<", pos)) {

throw new RuntimeException("Expected < at position " + pos);

}

pos++; // Skip '<'

String tagName = parseTagName();

XmlElement element = new XmlElement(tagName);

// Parse attributes

while (true) {

skipWhitespace();

if (xml.charAt(pos) == '>') {

pos++; // Skip '>'

break;

} else if (xml.charAt(pos) == '/' && xml.charAt(pos + 1) == '>') {

pos += 2; // Skip "/>"

return element; // Self-closing tag

} else {

String attrName = parseTagName();

skipWhitespace();

if (xml.charAt(pos) != '=') {

throw new RuntimeException("Expected = after attribute at position " + pos);

}

pos++; // skip '='

skipWhitespace();

String attrValue = parseAttributeValue();

element.addAttribute(attrName, attrValue);

}

}

// Parse children or text content

while (!xml.startsWith("</" + tagName, pos)) {

skipWhitespace();

if (xml.charAt(pos) == '<') {

if (xml.charAt(pos + 1) == '/') {

break; // End tag

}

XmlComponent child = parseNode();

element.addChild(child);

} else {

String text = parseText().trim();

if (!text.isEmpty()) {

element.addChild(new XmlTextNode(text));

}

}

}

// Skip end tag

if (!xml.startsWith("</" + tagName, pos)) {

throw new RuntimeException("Expected </" + tagName + "> at position " + pos);

}

pos += 2 + tagName.length(); // Skip "</tagName"

if (xml.charAt(pos) != '>') {

throw new RuntimeException("Expected > at position " + pos);

}

pos++; // Skip '>'

return element;

}

private String parseTagName() {

skipWhitespace();

int start = pos;

while (Character.isLetterOrDigit(xml.charAt(pos)) || xml.charAt(pos) == '-') {

pos++;

}

return xml.substring(start, pos);

}

private String parseAttributeValue() {

if (xml.charAt(pos) != '"' && xml.charAt(pos) != '\'') {

throw new RuntimeException("Expected \" or ' at position " + pos);

}

char quote = xml.charAt(pos);

pos++; // skip quote

int start = pos;

while (xml.charAt(pos) != quote) {

pos++;

}

String value = xml.substring(start, pos);

pos++; // skip quote

return value;

}

private String parseText() {

int start = pos;

while (pos < xml.length() && xml.charAt(pos) != '<') {

pos++;

}

return xml.substring(start, pos);

}

private void skipWhitespace() {

while (pos < xml.length() && Character.isWhitespace(xml.charAt(pos))) {

pos++;

}

}

}

**✅ 5. Test**

public class Main {

public static void main(String[] args) {

String xml = """

<note date="2025-07-17">

<to>Tin</to>

<from>ChatGPT</from>

<heading>Reminder</heading>

<body>Use Composite Pattern!</body>

</note>

""";

SimpleXmlParser parser = new SimpleXmlParser();

XmlComponent root = parser.parse(xml);

System.out.println("Parsed XML Tree:");

root.print("");

}

}

**🖨 Output**

<note date="2025-07-17">

<to>

Tin

</to>

<from>

ChatGPT

</from>

<heading>

Reminder

</heading>

<body>

Use Composite Pattern!

</body>

</note>

Do you want me to:

✅ Make it **editable** (add/remove children dynamically like DOM)?  
✅ Extend it to support **comments, CDATA, and processing instructions**?  
✅ Or make a **pretty-print and XPath-like query API** over this composite tree?

Which one do you want me to prepare next?

Great question! Let’s break down **the two designs** (the **first version without Composite**, and the **second version using the Composite pattern**) to see their **pros and cons** side by side.

**✅ 1. First Design: Simple XmlNode class (no Composite pattern)**

This design:

* Has a single XmlNode class.
* Text content is just a String in the node itself.
* Children are stored in a List<XmlNode>.

**✅ Pros:**

✔ **Simplicity**:

* Easy to understand and implement.
* Minimal boilerplate: one class is enough for all nodes.

✔ **Faster for simple parsing**:

* Less object hierarchy to manage.
* Works well if you only need to parse and print XML.

✔ **Low memory usage for small XMLs**:

* No overhead of separate leaf/composite objects.

**❌ Cons:**

✖ **Text vs. Elements are mixed**:

* A XmlNode with textContent and children together feels ambiguous.
* If a node has both <tag> and text, it’s harder to distinguish.

✖ **Not flexible for tree operations**:

* You can’t treat text nodes ("Hello") and element nodes (<to>Hello</to>) uniformly.
* Harder to extend if you later want to support **comments**, **CDATA**, or **processing instructions**.

✖ **Violates Single Responsibility Principle**:

* One class (XmlNode) does too much: stores attributes, manages children, and holds text content.

**✅ 2. Second Design: Composite Pattern (XmlElement + XmlTextNode)**

This design:

* Uses an **interface** (XmlComponent).
* **Composite** (XmlElement) has children.
* **Leaf** (XmlTextNode) holds text only.

**✅ Pros:**

✔ **True tree structure**:

* Models XML as a **hierarchy** of nodes (elements) and leaves (text).

✔ **Uniform treatment of nodes and text**:

* You can call print(), getName(), or getTextContent() on any XmlComponent.

✔ **Extensible**:

* Easy to add **XmlComment**, **XmlProcessingInstruction**, or **XmlCDataSection** in the future.

✔ **Clean separation of concerns**:

* XmlElement manages children.
* XmlTextNode just holds text.
* Each class has a single responsibility.

✔ **Supports advanced tree operations**:

* Traversals (pre-order, post-order) and searches (like XPath) are easier to implement.

**❌ Cons:**

✖ **More boilerplate**:

* Multiple classes/interfaces to maintain.

✖ **Slightly higher memory usage**:

* Every text node becomes its own object.

✖ **Slower for very simple XML parsing**:

* More objects created for small XMLs.

✖ **Overkill for flat XML**:

* If your XML is trivial and you just want to parse attributes and text, this design feels heavy.

**🆚 Side-by-Side Summary**

| **Aspect** | **Simple XmlNode** | **Composite Pattern** |
| --- | --- | --- |
| **Simplicity** | ✅ Very simple | ❌ More complex hierarchy |
| **Extensibility** | ❌ Hard to extend for comments, etc | ✅ Easy to add new node types |
| **Tree Operations** | ❌ More effort to traverse | ✅ Supports recursive operations |
| **Memory usage** | ✅ Lower (single class) | ❌ Slightly higher (more objects) |
| **Code Maintenance** | ✅ Easy to maintain for small XMLs | ✅ Better for large/complex XMLs |
| **Best for** | Small, flat XML files | Nested, complex XML structures |

**🌟 When to choose which?**

✅ **Use Simple XmlNode if**:

* You only parse XML to display or extract basic data.
* You don’t need to modify the XML tree or support advanced node types.

✅ **Use Composite Pattern if**:

* You need to support **rich XML features** (comments, CDATA, etc.).
* You want **tree traversal, editing, or querying** (like XPath).
* You care about **SOLID principles** and **clean architecture**.

Do you want me to:

✅ **Refactor the Composite design** to also support **comments and CDATA** (*like a DOM*)?

✅ Or should I **show you how to traverse/search (like XPath)** in the Composite tree?

✅ Or **compare this with the standard Java DOM/SAX parsers** to help you decide if building your own makes sense?