

# Lektion 6

Datateknik GR(B), Java III, 7,5 högskolepoäng

**Syfte:** Att lära sig att hantera XML i Java. Att känna till skillnader och likheter mellan SAX och DOM, samt att kunna använda DOM för att läsa, förändra och skapa nya XML-dokument. Känna till och kunna använda JAXB för att mappa Java-objekt till XML och tvärtom.

**Att läsa:** Kursbok, finns inget kapitel om XML

Java API for XML Processing (JAXP) Tutorial

<http://docs.oracle.com/javase/tutorial/jaxp/>

Introduction to JAXB

<http://docs.oracle.com/javase/tutorial/jaxb/intro/>



## Java III, 5 poäng



### Lektion 6 - XML-parsers och DOM-interface

## XML



*Extensible Markup Language, XML, är ett universellt och utbyggbart märkspråk och en förenklad efterträdare till SGML. XML blev en W3C-rekommendation 10 februari 1998. XML-rekommendationen beskriver både strukturen på XML och vad som krävs av en XML-tolk. Bland annat XHTML, XSL och SMIL, är baserade på XML.*

*XML-koden kan inte definiera vilka element eller attribut som kan användas. Denna definition görs av en dokumentmall som antingen är intern eller länkas in i dokumentet. Syftet med XML är att kunna utväxla data mellan olika informationssystem. Detta görs genom att skicka data som ren text; text som även kan förstås av människor.*

*Den 4 februari 2004 blev den senaste specifikationen av XML, XML 1.1, en W3C-rekommendation.*

<http://sv.wikipedia.org/wiki/XML>



## XML-deklaration

- Skrivs överst i ett XML-dokument
- Berättar bl.a. vilken version och teckenuppsättning som används
- Inleds med `<?xml` och avslutas med `?>`
- Är ej obligatorisk men bör anges
- Exempel:

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

```
<?xml version="1.0" encoding="iso-8859-1">
```

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes">
```



## XML-element

- Är huvudbeståndsdelen i ett XML-dokument och används för att beskriva data i dokumentet
- Består av tre delar:
  - **En start-tag:** namnet på elementet omslutet av `<` och `>` (ex: `<namn>`)
  - **Innehåll:** data, andra element eller både och
  - **En slut-tag:** ett snedstreck följt av namnet på elementet omslutet av `<` och `>` (ex: `</namn>`)



## XML-element

### ■ Exempel:

```
<namn>Robert Jonsson</namn>  
<telefon>063-165938</telefon>  
<alder />  
<email>robert.jonsson@miun.se</email>
```

start-tag                      innehåll                      slut-tag

- Alla XML-dokument innehåller ett ytterst och unikt element som kallas för rotelementet
- Beskriver vilken typ av data/objekt som XML-dokumentet representerar



## XML-element

- Element kan fungera som behållare för andra element - nästlade element
- Barn-element är element som ryms inom ett annat element
- Förälder-element är element som rymmer andra element
- Syskon-element är element som delar samma förälder-element



## XML-element

### ■ Exempel:

```
<familj>
  <föräldrar>
    <pappa>Kalle</pappa>
    <mamma>Sara</mamma>
  </föräldrar>
  <barn>
    <flicka>Stina</flicka>
    <pojke>Pelle</pojke>
  </barn>
</familj>
```

Diagram labels:

- rot-element (points to <familj>)
- syskon-element (points to <pappa>Kalle</pappa>)
- förälder-element (points to <flicka>Stina</flicka>)

### ■ Element får inte överlappa varandra

```
<barn>
  <flicka>Stina<pojke>Pelle</flicka></pojke>
</barn>
```



## XML-attribut

- Skrivs alltid i ett elements start-tag
- Är case sensitive
- Måste ha ett värde
- Kan ersätta element och tvärtom

```
<person>
  <namn>Robert</namn>
  <telefon>063-165938</telefon>
</person>
```

```
<person namn="Robert" telefon="063-165938"></person>
```

- Ofta en smaksak hur man väljer att göra (finns dock vissa riktlinjer)



## XML-attribut

### ■ Exempel:

```
<familj bostadsadress="Storgatan 1">
  <föräldrar>
    <pappa ålder="42">Kalle</pappa>
    <mamma ålder="40">Sara</mamma>
  </föräldrar>
  <barn>
    <flicka ålder="14">Stina</flicka>
    <pojke ålder="10">Pelle</pojke>
  </barn>
</familj>
```



## Välformad XML

- Ett XML-dokument som följer syntaxen (grundreglerna) sägs vara välformad
  - XML-dokumentet måste innehålla ett och endast ett rotelement
  - Alla element måste ha en start- och sluttag
  - Element måste nästlas på ett korrekt sätt
  - Alla attribut måste ha ett värde och omslutas av tecknet ' eller "
  - Attribut måste placeras i starttaggen
  - Vissa tecken får inte förekomma i ett elements innehåll (t.ex. < > &)
  - Namn på ett element måste börja med en bokstav eller \_ (kan sen innehålla siffror)



## XML-tolk

- En XML-tolk (parser) är en mjukvarumodul (en eller flera klasser i Java) som kontrollerar XML-dokumentet och ger tillgång till dess innehåll och struktur
- Genererar fel om XML-dokumentet inte följer syntaxen
- Fungerar ungefär som en kompilator (t.ex. till Java)



## XML-tolk

- Finns två typer av tolkar
  - Icke validerande tolk  
Kontrollerar så att XML-dokumentet innehåller välformad XML
  - Validerande tolk  
Kontrollerar även om syntax stämmer överens med den DTD man angett
- Finns inbyggd XML-tolk i Internet Explorer (heter MSXML och är en icke validerande tolk)



## Document Type Definition

- Med en DTD kan man definiera:
  - Vilka element ett XML-dokument kan innehålla
  - I vilken ordning elementen måste förekomma i XML-dokumentet
  - Hur många gånger ett visst element får förekomma i XML-dokumentet
  - Vilken typ av data ett element kan innehålla
  - Vilka barnelement en förälder kan ha
  - Vilka attribut ett element kan ha



## Document Type Definition

- En DTD kan definieras direkt i XML-dokumentet eller i en separat fil
- I senare fallet måste en DOCTYPE-deklaration användas enligt:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no"?>
<!DOCTYPE familj SYSTEM "familj.dtd">
<familj bostadsadress="storgatan 1">
  <pappa alder="42">Kalle</pappa>
```

```
<!DOCTYPE familj SYSTEM "familj.dtd">
```

↑  
Namnet på rot-elementet

↑  
Sökväg till lokal fil på hårddisk eller  
en URL till fil på Internet





# Document Type Definition

## ■ Exempel på innehåll:

```
<!-- rot-elementet familj -->
<!ELEMENT familj (foraldrar, barn?)>
<!ATTLIST familj bostadsadress CDATA #REQUIRED>

<!-- elementet foraldrar -->
<!ELEMENT foraldrar (pappa, mamma)>
<!ELEMENT pappa (#PCDATA)>
<!ATTLIST pappa alder CDATA #IMPLIED>
<!ELEMENT mamma (#PCDATA)>
<!ATTLIST mamma alder CDATA #IMPLIED>

<!-- elementet barn -->
<!ELEMENT barn (pojke*, flicka*)>
<!ELEMENT pojke (#PCDATA)>
<!ATTLIST pojke alder CDATA #IMPLIED>
<!ELEMENT flicka (#PCDATA)>
<!ATTLIST flicka alder CDATA #IMPLIED>
```

Vi börjar beskriva rot-elementet familj. Inom parenteserna anger vi att elementet familj innehåller elementen foraldrar och barn (i den ordningen). Frågetecknet efter barn anger att elementet barn kan förekomma ingen eller en gång. Dvs XML-dokumentet är giltigt även om barn-elementet helt utesluts. Däremot måste det alltid förekomma ett och endast ett foraldrar-element.

Elementet familj har ett attribut som vi beskriver med en ATTLIST. En ATTLIST har följande format:  
<!ATTLIST namn\_på\_element namn\_på\_attribut data\_typ typ\_av\_attribut>

Datatyp sätter vi till CDATA vilket är en vanlig textsträng (character data). Vi bestämmer att en bostadsadress alltid måste anges genom att sätta typ av attribut till #REQUIRED. Andra typer av attribut är #IMPLIED (valbar) och #FIXED (värdet bestäms av DTD).

Elementet foraldrar bestämmer vi måste innehålla ett pappa-element och ett mamma-element (i den ordningen). Elementet pappa bestämmer vi enbart ska innehålla textsträngar (#PCDATA – parsed character data). Attributet alder i elementet pappa sätter vi till datatypen CDATA och typen till #IMPLIED, d.v.s. åldern behöver inte anges. Samma gäller för elementet mamma.

Elementet barn bestämmer vi måste innehålla elementen pojke och flicka (i den ordningen). Tecknet \* efter namnen innebär att det kan förekomma ingen, en eller flera element av den typen. Elementen pojke och flicka samt dess attribut är uppbyggd på samma sätt som elementen pappa och mamma.

Använder vi nu en validerande XML-tolk måste XML-dokumentet vara uppbyggd enligt detta sätt. Dvs rot-elementet måste innehålla ett och endast ett foraldrar-element, som i sin tur måste innehålla ett och endast ett pappa-element följt av ett mamma-element. Det behöver inte finnas några barn-element, men finns det ett barn-element får det endast finnas ett barn-element och det måste komma efter foraldrar-elementet. Barn-elementet kan innehålla ingen, en eller flera pojke-element följt av ingen, en eller flera flicka-element.



## Slutord XML

- Detta var en kort och snabb genomgång av det mest grundläggande vad gäller XML
- Räcker för att vi ska kunna hantera XML i Java
- Finns betydligt mycket mer att lära sig om XML, men det tas upp i en separat kurs



## Java och XML

- Det finns två huvudsakliga APIer för att hantera XML i Java
  - Simpel API for XML (SAX)
  - Document Object Model (DOM)
- Utöver dessa finns bl.a. även:
  - JDOM, dom4j, ElectricXML och XMLPULL
- Går att ladda ner och använda andra



## Simple API for XML

- Är ett händelsestyrt API
- XML-dokumentet läses sekventiellt och när t.ex. ett element påträffas anropas metod i applikationen (får info om taggen som namn, attribut)
- Fungerar ungefär som händelsehantering i ett GUI (javax.swing)
- Enda alternativet vid stora XML-filer
- Kan inte skriva till XML-dokument



## Läsa XML med SAX

- Börja importera nödvändiga klasser

```
// IOException eftersom filer läses/skrives
import java.io.IOException;

// För att kunna skapa och konfigurera SAX-baserad tolk
import javax.xml.parsers.SAXParserFactory;

// Ovan kastar detta undantag om något fel uppstår
import javax.xml.parsers.ParserConfigurationException;

// Används för att tolka XML-dokumentet
import javax.xml.parsers.SAXParser;

// Om något fel uppstår vid tolkningen
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;

// Hantera elementens attribut
import org.xml.sax.Attributes;

// Klass som implementerar gränssnitt för att hantera SAX
import org.xml.sax.helpers.DefaultHandler;
```



## DocumentHandler

- Implementerar följande gränssnitt:
  - `EntityResolver`, `DTDHandler`,  
`ContentHandler` och `ErrorHandler`
- Gränssnitten innehåller metoder som XML-tolken kommer att anropa allt eftersom XML-dokumentet läses
- `DefaultHandler` innehåller endast tomma metoder (jmf med adapter-klasserna i `javax.swing`)



# Läsa XML med SAX

## ■ Låt klassen ärva `DefaultHandler`:

```
public class TestSAXParser extends DefaultHandler {  
    // Konstruktor  
    public TestSAXParser() {  
        parseDocument();  
    }  
  
    public static void main(String[] args) {  
        TestSAXParser parser = new TestSAXParser();  
    }  
}
```

En klass som vi vill ska använda en SAX-parser sätter vi till att ärva från `DefaultHandler`. På så sätt ärver vi alla nödvändiga metoder som tolken kommer att anropa allteftersom den stöter på t.ex. nya element. Observera att alla metoder i `DefaultHandler` är tomma så varje applikation måste överskugga de metoder som behövs för applikationens behov.

I `main`-metoden skapar vi ett nytt objekt av klassen. I konstruktorn gör vi ett anrop till vår egna metod `parseDocument` som kommer att skapa själva tolken. Observera att `parseDocument` inte är någon metod som ingår i `DefaultHandler` utan är en helt egen metod.



## Läsa XML med SAX

- Skapa en `SAXParserFactory` och konfigurera den
- Skapa en `SAXParser` och anropa `parse` för att läsa XML-dokumentet

```
private void parseDocument() {  
    SAXParserFactory factory = SAXParserFactory.newInstance();  
    factory.setValidating(true);  
    try {  
        SAXParser parser = factory.newSAXParser();  
        parser.parse("sommarkurser.xml", this);  
    }  
    catch(SAXException se) {se.printStackTrace();}  
    catch(ParserConfigurationException pce) {pce.printStackTrace();}  
    catch(IOException ie) {ie.printStackTrace();}  
}
```

I `parseDocument` börjar vi med att skapa en ny `SAXParserFactory` genom att anropa den statiska metoden `newInstance`. Vi anropar därefter `setValidating` och anger som argument `true`. Detta gör att tolken kontrollerar så att XML-dokumentet, förutom att vara välformad, även är giltigt (om t.ex. en DTD används).

I en try-catch skapas nu en `SAXParser` genom att anropa `newSAXParser` på `SAXParserFactory`-objektet. Därefter anropar vi `parse` och anger som första argument vilket XML-dokument som ska tolkas (i det här fallet är det `sommarkurser.xml`). Som andra argument registrerar vi oss själva (klassen) hanterar av händelser som tolken genererar när den tolkar XML-dokumentet.

Tre olika typer av `Exception` fångar vi. Ett `SAXException` kastas om något går fel vid tolkningen. Ett `ParserConfigurationException` kastas av metoden `newSAXParser` om SAX-tolken inte kan skapas. Till sist fångar vi eventuella `IOException` som kastas om t.ex. XML-dokumentet inte existerar.



## Läsa XML med SAX

- Överlagra de metoder XML-tolken anropar och bestäm vad som ska ske:

```
public void startDocument() throws SAXException {  
    System.out.println("XML-dokument start");  
}  
  
public void endDocument() throws SAXException {  
    System.out.println("XML-dokument slut");  
}  
  
public void error(SAXParseException se) {  
    System.out.println("ERROR: " + se.getMessage());  
}  
  
public void warning(SAXParseException se) {  
    System.out.println("WARNING: " + se.getMessage());  
}
```

När vi väl har anropat `parse` i `SAXParser` kommer tolken att läsa innehållet i XML-dokumentet uppifrån och ner (sekventiellt). När tolken stöter på början av dokumentet kommer den att anropa metoden `startDocument`. I vårt exempel gör vi inget annat än att enbart skriva ut texten XML-dokument start. När XML-tolken kommer till XML-dokumentets slut kommer den att anropa `endDocument`. Precis som tidigare gör vi endast en utskrift om att slutet är nått.

När tolken läser XML-dokumentet och stöter på något fel anropar den någon av metoderna `error` eller `warning` (beroende på typ av fel). I dessa metoder kan vi ta emot felet, undersöka det och vidta åtgärder. I detta exempel skriver vi dock bara ut felmeddelandet.



## Läsa XML med SAX

- Överlagra de metoder XML-tolken anropar och bestäm vad som ska ske:

```
public void startElement(String uri, String localName,
    String qName, Attributes attributes) throws SAXException
{
    System.out.print("<" + qName);

    // Hantera eventuella attribut
    for (int i = 0; i < attributes.getLength(); i++) {
        System.out.print(" " + attributes.getQName(i) + "=\"" +
            attributes.getValue(i) + "\"");
    }

    System.out.println(">");
}
```

En av de mer intressanta metoderna att överlagra är `startElement`. Denna metod anropas varje gång XML-tolken stöter på ett nytt element (start-tag). Som parametrar till metoden skickas elementets namn och attribut. De två första parametrarna är endast intressanta om Namespace används. Om inte är dessa tomma strängar (längden noll). Den sista parameteren innehåller eventuella attribut (ATTLIST) elementet har. Via klassen `Attributes` kan vi ta reda på hur många attribut som finns genom att anropa `getLength`. Via metoderna `getQName` och `getValue` kan vi ta reda på attributens namn och värde.

I metoden börjar vi med att skriva ut tecknet `<` följt av namnet på elementet. Därefter loopar vi igenom de attribut som elementet eventuellt har. `attributes.getLength` returnerar 0 (noll) om elementet inte har några attribut. För varje attribut tar vi reda på namnet genom att anropa `getQName` och anger elementets index som parameter. Vi skriver även ut värdet genom att anropa `getValue`. Sist av allt skriver vi ut `>` för att avsluta start-taggen.

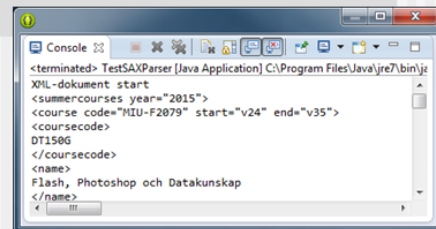




## Läsa XML med SAX

- Överlagra de metoder XML-tolken anropar och bestäm vad som ska ske:

```
public void endElement(String uri, String localName,  
    String qName) throws SAXException {  
    System.out.println("</" + qName + ">");  
}  
  
public void characters(char[] ch, int start, int length)  
    throws SAXException {  
    System.out.println(new String(ch, start, length));  
}
```



När XML-tolken kommer till slut-taggen för ett element anropar den metoden `endElement`. Till skillnad från `startElement` skickas här inga attribut utan endast namnet på elementet. I vårt exempel skriver vi ut `</` följt av namnet på elementet följt av `>`.

När XML-tolken stöter på innehåll i ett element kommer den att anropa metoden `characters`. Denna tar en `char`-array som parameter följt av vilken startposition som ska användas i arrayen samt hur många tecken i arrayen som ska användas. Vi skapar en ny sträng av `char`-arrayen och skriver ut den på skärmen.

Vi har nu en applikation som kan läsa ett XML-dokument och skriva ut elementen, dess attribut och innehåll på skärmen. Vi skulle givetvis kunna snygga till utskriften en aning genom att indentera varje ny start-taggt ett steg.

Ta en titt på exemplet **TestSAXParser.java** som följer med lektionen. Prova gärna att ändra i XML-dokumentet så att den dels inte anses vara välformad och sen även så att den inte är giltig enligt DTDn. Prova även om det blir någon skillnad om `setValidating` sätts till `false`.

Ta även en titt på **Course.java** och **CourseSAXParser.java** för ett exempel på hur vi kan skapa `Course`-objekt från informationen i XML-dokumentet.



## Document Object Model

- Är ett trädbaserat API
- XML-dokumentet avbildas i en trädstruktur i minnet
- Varje nod i trädet motsvarar t.ex. ett element i XML-dokumentet
- Kan skapa och förändra noder och skriva dessa till XML-dokumentet
- Tar upp mycket minne vid stora XML-dokument



## Läsa XML med DOM

- Börja importera nödvändiga klasser

```
// IOException eftersom filer läses/skrivs
import java.io.IOException;

// För att kunna skapa och konfigurera DOM-baserad tolk
import javax.xml.parsers.DocumentBuilderFactory;

// Ovan kastar detta undantag om något fel uppstår
import javax.xml.parsers.ParserConfigurationException;

// Används för att tolka XML-dokumentet eller skapa ett nytt
import javax.xml.parsers.DocumentBuilder;

// Om något fel uppstår vid tolkningen
import org.xml.sax.SAXException;

// Representerar hela XML-dokumentet och elementen i detta.
// NodeList innehåller en lista av noder
import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
```



# Läsa XML med DOM

## ■ Skapa klassen:

```
public class TestDOMParser {  
    // Dokumentet  
    private Document dom;  
  
    public TestDOMParser() {  
        // Anropar egen metod som skapar XML-tolken  
        createParser();  
  
        // Anropar egen metod som tolkar dokumentet  
        parseDocument();  
    }  
  
    public static void main(String[] args) {  
        TestDOMParser tdp = new TestDOMParser();  
    }  
}
```

Tillskillnad från när vi använde SAX behöver vi inte ärva någon speciell klass när vi använder DOM. Vill vi ta hand om eventuella fel eller varningar som kan uppstå under tolkningen av XML-dokumentet måste vi dock implementera gränssnittet `ErrorHandler` och därefter implementera metoderna `fatalError`, `error` och `warning` (som i exemplet med SAX). Det brukar vara vanligt att man skriver en ny klass som tar hand om felhanteringen (välformad och giltig).

Som enda instansvariabel i klassen har vi ett objekt av `Document`. Det är i detta objekt som trädstrukturen lagras när vi läser ett XML-dokument med DOM. I `main`-metoden skapar vi ett nytt objekt av klassen. I konstruktorn gör vi ett anrop till `createParser` och `parseDocument`. Båda dessa metoder är egna metoder och inget som krävs för att läsa XML-dokument med DOM.



## Läsa XML med DOM

- Skapa en `DocumentBuilderFactory` och konfigurera den
- Skapa en `DocumentBuilder` och anropa `parse` för att läsa XML-dokumentet

```
private void createParser() {  
    DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();  
    dbf.setValidating(true);  
    dbf.setIgnoringElementContentWhitespace(true);  
  
    try {  
        DocumentBuilder db = dbf.newDocumentBuilder();  
        dom = db.parse("sommarkurser.xml");  
    }  
    catch (SAXException se) {se.printStackTrace();}  
    catch (ParserConfigurationException pce) {pce.printStackTrace();}  
    catch (IOException ie) {ie.printStackTrace();}  
}
```

I `createParser` börjar vi med att skapa en ny `DocumentBuilderFactory` genom att anropa den statiska metoden `newInstance`. Vi anropar därefter `setValidating` och anger som argument `true`. Detta gör att tolken kontrollerar så att XML-dokumentet, förutom att vara välformad, är giltigt (om t.ex. en DTD används). Vi anropar även `setIgnoringElementContentWhitespace` och anger som argument `true`. Detta gör att element som endast innehåller s.k. white spaces (som radbrytningar och tabbar) ignoreras. Default för båda dessa metoder är `false`.

I en try-catch skapas nu en `DocumentBuilder` genom att anropa `newDocumentBuilder` på `DocumentBuilderFactory`-objektet. Därefter anropar vi `parse` och anger som argument vilket XML-dokument som ska tolkas (i det här fallet är det `sommarkurser.xml`). Om allt går bra kommer detta anrop att returnera ett `Document`-objekt som innehåller alla noder i trädet.

Tre olika typer av `Exception` fångar vi. Ett `SAXException` kastas om något går fel vid tolkningen. Ett `ParserConfigurationException` kastas av metoden `newDocumentBuilder` om DOM-tolken inte kan skapas. Till sist fångar vi eventuella `IOException` som kastas om t.ex. XML-dokumentet inte existerar.



# Läsa XML med DOM

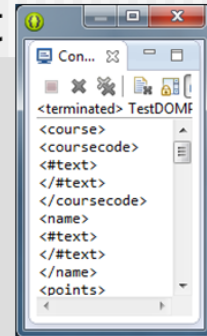
## ■ Lista noder m.m. från trädet

```
private void parseDocument() {
    Element root = dom.getDocumentElement();
    NodeList nl = root.getChildNodes();
    listNodes(nl);
}

private void listNodes(NodeList nl) {
    if (nl != null && nl.getLength() > 0) {
        for (int i = 0 ; i < nl.getLength(); i++) {
            Node currentNode = nl.item(i);
            System.out.println("<" + currentNode.getNodeName() + ">");

            if (currentNode.hasChildNodes())
                listNodes(currentNode.getChildNodes());

            System.out.println("</" + currentNode.getNodeName() + ">");
        }
    }
}
```



När vi väl har läst in hela XML-dokumentet till Document-objektet är det dags att hämta information från trädet. Beroende på vad vi är ute efter finns det olika angreppssätt. Vanligt är att man först tar reda på dokumentets root-element. Detta gör vi genom att anropa metoden `getDocumentElement` på Document-objektet. Därefter tar vi reda på root-elementets alla barn-element genom att anropa metoden `getChildNodes`. Observera att denna returnerar alla barn-element, d.v.s en nivå djupare i trädet. Metoden returnerar inte eventuella barn-element till root-elementets barn. Dessa finns dock ”inbakade” i de barn-element som `getChildNode` just returnerade.

Vi skriver en separat metod som tar ett `NodeList`-objekt som argument och som listar namnet på alla noder i listan. Vi börjar med att kontrollera att listan verkligen innehåller några noder och därefter loopar vi igenom alla noder i en `for`-loop. Från listan plockar vi ut nästa nod genom att anropa `item` och anger vilket index noden finns som vi vill ha. Vi tar reda på noden namn genom att anropa `getNodeName`. Namnet skriver vi ut på skärmen mellan tecknen `<` och `>`.

Genom att anropa `hasChildNodes` kan vi ta reda på om aktuell nod har några barn-noder. Om så är fallet anropar vi `listNodes` och skickar aktuell nods barn-noder (`NodeList`) som argument. Vi gör med andra ord ett rekursivt metoanrop.

I detta exempel har vi inte skrivit ut elementens innehåll eller eventuella attribut som elementen har. Vi listar enbart namnen på alla noder. Ta en titt på exemplet **TestDOMParser.java** som följer med lektionen. Som du ser hanteras elementens innehåll som egna noder med namnet `#text`.

När tolken läser XML-dokumentet och stöter på något fel anropar den någon av metoderna `fatalError`, `error` eller `warning` (beroende på typ av fel). I dessa metoder kan vi ta emot felet, undersöka det och vidta åtgärder. I detta exempel skriver vi dock bara ut felmeddelandet.

Ta även en titt på exemplet **CourseDOMParser.java** som fungerar på samma sätt som **KursSAXParser** men som använder DOM.



## Gränssnittet Document

- Representerar XML-dokumentet
- Ger tillgång till dokumentets noder samt möjlighet att skapa nya
- Ärver från klassen Node

Element createElement(String name)	Skapar ett nytt element med namnet name.
Attr createAttribute(String name)	Skapar ett nytt attribut med namnet name.
Text createTextNode(String data)	Skapar en text-nod av innehållet i data.
Comment createComment(String data)	Skapar en kommentar.
CDataSection createCDataSection(String)	Skapar en CDATA-sektion.
Element getDocumentElement()	Returnerar rot-elementet.
Node appendChild(Node newNode)	Lägger till en ny nod sist i listan av barn-noder.
NodeList getChildNodes()	Returnerar denna nods alla barn-noder.



## Gränssnittet Node

- Representerar en nod i dokument-trädet

Node appendChild(Node newNode)	Lägger till en ny nod sist i listan av barnnoder.
Node cloneNode(boolean deep)	Skapar en kopia av aktuell node. Om deep är satt till true kopieras även barnnoder.
NamedNodeMap getAttributes()	Returnerar nodens attribut om typen är element.
NodeList getChildNodes()	Returnerar denna nods alla barnnoder.
Node getNextSibling()	Returnerar nästa syskon eller null om ingen finns.
String getNodeName()	Returnerar nodens namn (beror på nodens typ).
short getNodeType()	Returnerar denna nods typ (element, text etc)
String getNodeValue()	Returnerar denna nods värde (beroende på typ).
Node getParentNode()	Returnerar denna nods förälder (null om ej finns).
boolean hasChildNodes()	Returnerar true om denna nod har barn-noder.
boolean hasAttributes()	Returnerar true om denna nod har attribut.
void setNodeValue(String value)	Sätter denna nods värde till value.
Node removeChild(Node oldChild)	Tar bort barnnoden childNode och returnerar den.



## Gränssnittet `NodeList`

- Representerar en read-only lista av `Node`-objekt

<code>int getLength()</code>	Returnerar antalet <code>Node</code> -objekt i listan.
<code>Node item(int index)</code>	Returnerar det <code>Node</code> -objekt som befinner sig på position <code>index</code> i listan eller <code>null</code> om felaktigt <code>index</code> anges.



## Gränssnittet `Element`

- Representerar ett element i XML-dokumentet
- Ärver från klassen `Node`

<code>String getAttribute(String name)</code>	Returnerar värdet på attributet som ges av <code>name</code> .
<code>String getTagName()</code>	Returnerar detta elements namn.
<code>void removeAttribute(String name)</code>	Tar bort attributet i elementet som ges av <code>name</code> .
<code>void setAttribute(String name, String data)</code>	Skapar ett nytt attribut med namnet <code>name</code> och värdet <code>value</code> .
<code>boolean hasAttribute(String name)</code>	Ger <code>true</code> om attributet med namnet <code>name</code> finns.
<code>NodeList getElementsByTagName(String name)</code>	Returnerar en <code>NodeList</code> med alla underordnade <code>Element</code> som ges av namnet <code>name</code> . Namnet "*" motsvarar alla underordnande element.



## Ändra data med DOM

- Exempel som ändrar innehållet för elementet description. Visar även hur man enkelt kan skriva ut XML-dokumentet till en OutputStream.

```
public class DOMChangeElement {  
    private Document dom;  
    public DOMChangeElement() {  
        createParser();  
        parseDocument();  
        DOMWriter.writeDocument(dom, System.out);  
    }  
  
    public static void main(String[] args) {  
        DOMChangeElement dce = new DOMChangeElement();  
    }  
}
```

Vi ska nu ge exempel på hur vi kan förändra data i ett element. Vi ska leta efter kursen Java I och ändra dess beskrivning. Mycket av koden är densamma från TestDOMParser. Metoden `createParser` är samma som tidigare. Den stora skillnaden ligger i metoden `parseDocument` som vi nu ändrar i för att söka efter kursen och ändra beskrivningen.

Vi kommer även att skriva en ny klass med namnet `DOMWriter` vilken innehåller en statisk publik metod. Med denna metod kan vi skriva ut ett XML-dokument (`Document`) till en `OutputStream` (`System.out` eller `FileOutputStream` etc).





# Ändra data med DOM

```
private void parseDocument() {
    Element root = dom.getDocumentElement();
    NodeList allNames = root.getElementsByTagName("name");

    if (allNames != null && allNames.getLength() > 0) {
        Node java1 = null;

        for (int i = 0; i < allNames.getLength(); i++) {
            Node currentName = allNames.item(i);
            Text text = (Text)allNames.getFirstChild();
            if (text.getNodeValue().equalsIgnoreCase("Java I")) {
                java1 = currentName.getParentNode();
                break;
            }
        }

        if (java1 != null) {
            Node description = java1.getChildNodes().item(3);
            Text oldDescription = (Text)description.getFirstChild();
            Text newDescription = dom.createTextNode("Ny beskrivning för Java I");
            description.replaceChild(newDescription, oldDescription);
        }
    }
}
```

I metoden `parseDocument` börjar vi som vanligt med att hämta root-elementet. Genom att anropa `getElementsByTagName` och ange `name` som argument får vi en `NodeList`, som vi kallar `allNames`, vilken innehåller alla element med namnet kursnamn. Vi kontrollerar om listan innehåller några element genom att kontrollera listans längd.

Fanns det element i listan deklarerar vi ett `Node`-objekt och ger det namnet `java1`. Detta objekt kommer att innehålla `course`-noden (elementet) för kursen Java I. Vi loopar nu igenom alla kursnamn i en `for`-loop. I loopen börjar vi med att hämta nästa nod från listan och lagrar i `currentName`. Detta ska nu motsvara ett `name`-element i XML-dokumentet och texten mellan taggarna ska vara namnet på kursen. Eftersom ett elements innehåll (text) lagras som en egen barn-nod i trädstrukturen kan vi anropa `getFirstChild` för att komma åt innehållet. Metoden `getFirstChild` returnerar ett `Node`-objekt som vi typkonverterar till ett `Text`-objekt.

Genom att anropa `getNodeValue` på `Text`-objektet hämtar vi innehållet som vi kan jämföra mot strängen "Java I". Om kursnamnet stämmer vet vi att vi är inne på rätt `course`-nod i trädstrukturen. Eftersom elementet `name` är ett barn-element till elementet `course` kan vi anropa `getParentNode` på noden `currentName` för att komma åt hela `course`-noden för Java I. Vi avbryter nu loopen med `break` eftersom vi har hittat rätt kurs.

Efter `for`-loopen kontrollerar vi om `Node`-objektet `java1` refererar till något annat än `null`. Om så är fallet hämtar vi noden innehållandes beskrivningen genom att på

`java1` anropa `getChildNodes`. Detta returnerar alla barn-noder till `course-`noden (dvs `coursecode`, `name`, `point`, `description`). Eftersom vi i vår DTD har specificerat vilken ordning dessa element måste anges kan vi direkt ange att barn-nod 3 ska returneras (`item(3)`). Skulle vi inte ha använt någon DTD eller om vi vid skapandet av tolken inte satt `setValidating(true)` hade vi behövt kontrollera namnen på noderna för att vara säkra på att vi hämtar rätt nod. Ett annat alternativ vore att i `for`-loopen ovan spara `Node`-objektet `currentName` för att användas här.

Hursomhelst hämtar vi texten genom att anropa `getFirstChild` på `Node`-objektet `beskrivning`. Vi skapar nu en helt ny `Text`-nod genom att på `Document`-objektet `dom` anropa metoden `createTextNode` och som argument ange vilken text noden ska ha. På `Node`-objektet `beskrivning` anropas nu `replaceChild` för att byta ut den gamla beskrivningen mot den nya.

Som du säkert märkt kan det vara ganska bökigt att söka efter specifika noder och veta vilka metoder man ska anropa när för att komma åt det man är ute efter. Det kan ta ett tag att lära sig hur DOM fungerar, men när man väl gjort det är det ganska lätt att stega sig igenom trädstrukturen.



# Ändra data med DOM

- Börja med att importera de klasser som behövs i DOMWriter

```
import java.io.OutputStream;
import java.io.File;

import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.TransformerException;
import javax.xml.transform.TransformerConfigurationException;

import org.w3c.dom.Document;
```



# Ändra data med DOM

## ■ Klass för att skriva XML till en OutputStream

```
public class DOMWriter {
    public static void writeDocument(Document document, OutputStream out) {
        Source xmlSource = new DOMSource(document);
        Result result = new StreamResult(out);
        String systemValue = (new File(
            document.getDoctype().getSystemId()).getName());

        TransformerFactory tf = TransformerFactory.newInstance();
        Transformer transformer = tf.newTransformer();

        transformer.setOutputProperty(OutputKeys.DOCTYPE_SYSTEM, systemValue);
        transformer.setOutputProperty(OutputKeys.VERSION, document.getXmlVersion());
        transformer.setOutputProperty(OutputKeys.STANDALONE,
            document.getXmlStandalone() ? "yes" : "no");
        transformer.setOutputProperty(OutputKeys.INDENT, "yes");

        transformer.transform(xmlSource, result);
    }
}
```

Den statiska metoden `writeDocument` tar som argument det XML-dokument (`Document`) som ska skrivas till utströmmen, samt den utström som ska användas (av typen `OutputStream`). Vi börjar med att skapa en `DOMSource` av XML-dokumentet samt en `StreamResult` av utströmmen. Dessa kan man enkelt uttryckt ses som källan och till vilken destination källan ska skrivas. Vi tar reda på vilken `DOCTYPE` XML-dokumentet använder och sparar det i en sträng för att användas senare.

Precis som med `SAX` och `DOM` skapar vi en `TransformerFactory` genom att anropa den statiska metoden `newInstance`. Fabriken använder vi för att skapa en nytt `Transformer`-objekt. På `transformer`-objektet sätter vi nu en del inställningar för att ange vilken `DOCTYPE` som användes. Vi tar även reda på vilken version, encoding och vilket värde `standalone` har så att detta är korrekt när vi skriver till utströmmen.

Sist anropar vi metoden `transform` på `transformer`-objektet för att transformera XML-dokumentet.

I exemplet har jag inte importerat nödvändiga paket eller fångat eventuella `Exception` som kan kastas. Ta en titt på exemplen **`DOMWriter.java`** och **`DomChangeElement.java`** som följer med lektionen.



# Nytt dokument med DOM

## ■ Exempel som visar hur ett nytt tomt XML-dokument skapas

```
public class XMLCourse {
    private Document document;

    public XMLCourse() {
        createDocument();
    }

    private void createDocument() {
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setValidating(true);
        DocumentBuilder db = dbf.newDocumentBuilder();
        DOMImplementation impl = db.getDOMImplementation();
        DocumentType doctype = impl.createDocumentType("summercourses",
            null, "summercourses");
        document = impl.createDocument(null, "summercourses", doctype);
        Element root = document.getDocumentElement();
        root.setAttribute("year", "2015");
    }
}
```

Som ett sista exempel i lektionen ska vi titta på hur man skapar ett helt nytt och tomt Document-objekt och hur man sen lägger till nya element/noder i dokumentet. Vi skriver en klass med namnet XMLCourse som är tänkt att användas för att skapa XML-dokument med kursinformation i (som sommarkurser.xml). Som enda instansvariabel används ett Document-objekt.

I metoden createDocument, som anropas från konstruktorn, skapar vi en DocumentBuilderFactory och sätter setValidating till true. Därefter skapar vi en ny DocumentBuilder. Så långt är allt precis som vanligt. Om vi hade tänkt att skapa ett XML-dokument utan DOCTYPE skulle vi kunna skriva document = db.newDocument() för att få ett nytt tomt dokument att arbeta med.

När vi nu däremot ska använda en DOCTYPE (<!DOCTYPE summercourses SYSTEM "summercourses.dtd">) måste vi gå tillväga lite annorlunda. På DocumentBuilder-objektet anropar vi metoden getDOMImplementation för att erhålla ett DOMImplementation-objekt. Via detta objekt kan vi skapa DocumentType-objekt (DOCTYPE) som sen kan användas för att skapa dokument där DOCTYPE är satt.

Som första argument till createDocumentType anger vi namnet på root-elementet. Som andra argument anger vi null eftersom det inte är en publik DTD som ska användas. Som tredje argument anger vi vilken (system) DTD som ska användas, som i det här fallet är summercourses.dtd.

För att skapa ett `Document`-objekt av en `DOMImplementation` anropas metoden `createDocument`. Som första argument anger vi `null` eftersom vi inte använder oss av namespace. Som andra argument anger vi namnet på root-elementet och som sista argument anges den `DOCTYPE` (`DocumentType`) som ska användas.

Vi har nu ett nytt tomt XML-dokument (i minnet som en trädstruktur) och där root-elementets namn är `summercourses`. Som du kanske kommer ihåg från DTDn måste root-elementet ha ett attribut med namnet `year`. Vi anropar därför metoden `getDocumentElement` för att erhålla dokumentets root-element. Därefter anropar vi metoden `setAttribute` och anger som första argument namnet på attributet som ska skapas. Som andra argument anger vi det värde attributet ska ha. Så där ja. Vi har nu skapat ett nytt tomt dokument som är redo på att fyllas med `course`-element.



# Nytt dokument med DOM

## ■ Lägga till element i dokumentet

```
public void addCourse(Course c) {  
    Element course = document.createElement("course");  
    course.setAttribute("code", k.getApplicationCode());  
    course.setAttribute("start", k.getStarts());  
    course.setAttribute("end", k.getEnds());  
  
    Element coursecode = document.createElement("coursecode");  
    coursecode.setTextContent(c.getCourseCode());  
    course.appendChild(coursecode);  
  
    Element name = document.createElement("name");  
    name.setTextContent(c.getName());  
    course.appendChild(name);  
  
    // Samma upplägg för att lägga till elementen points och description  
  
    Element root = document.getDocumentElement();  
    root.appendChild(course);  
}
```

För att lägga till nya `course`-element skriver vi en metod med namnet `addCourse`. Denna metod tar ett `Course`-objekt som argument som vi använder för att plocka ut den kursinformation som ska finnas i `course`-elementet.

Vi börjar med att skapa ett nytt `Element`-objekt genom att anropa metoden `createElement` och där vi som argument till metoden anger namnet på det element vi vill skapa. Eftersom `course`-elementet (enligt DTDn) måste ha attributet `code`, `start` och `end` skapar vi dessa genom att anropa `setAttribute` på samma sätt som vi gjorde för att sätta attributet `year` för `root`-elementet tidigare.

Som första barn-element i `course`-elementet har vi `coursecode`. Vi skapar därför ett nytt `Element`-objekt. Detta element har inga attribut, men väl ett innehåll (text). På ett `Element`-objekt kan vi anropa metoden `setTextContent` och ange som argument den text elementet ska ha. Vi använder `Course`-objektet och anropar metoden `getCourseCode` för att ange detta. `coursecode`-elementet ska som sagt vara ett barn-element till `course`-elementet. För att lägga till `coursecode` som ett barn-element till `course` anropar vi metoden `appendChild`.

På samma sätt som beskrivits ovan gör vi nu för att skapa och lägga till elementen `name`, `point` och `description` som barn-element till `course`. Sist av allt i metoden är det dags att lägga till hela det skapade `course`-elementet (med alla dess barn-element) till `root`-elementet. Vi anropar därför metoden `getDocumentElement` och följt av ett anrop till `appendChild`.



# Nytt dokument med DOM

## ■ Spara till XML-fil

```
public void saveToFile(String fileName) {  
    try {  
        FileOutputStream out = new FileOutputStream(fileName);  
        DOMWriter.writeDocument(document, out);  
    }  
    catch (FileNotFoundException e) {  
        System.out.println("Hittar inte filen " + fileName);  
        System.out.println(e.getMessage());  
    }  
}
```

För att kunna spara XML-dokumentet till fil skriver vi en metod med namnet `saveToFile`. Som argument tar metoden en sträng innehållandes namn på filen till vilken XML-dokumentet ska sparas. I metoden skapar vi en ny `FileOutputStream` och använder därefter klassen `DOMWriter` (som vi skrev i ett tidigare exempel) och anropar den statiska metoden `writeDocument`. Den tog, som du kanske kommer ihåg, det `Document`-objekt som ska skrivas ut och vilken utström (`OutputStream`) som ska användas för att skriva ut dokumentet.

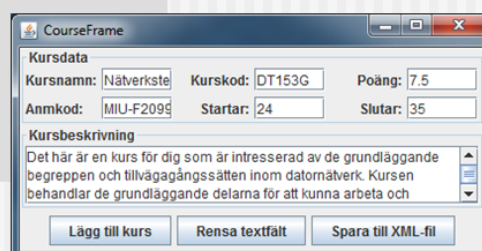




## Nytt dokument med DOM

- Grafisk applikation för att mata in ny kursinformation och spara dessa

```
public void actionPerformed(ActionEvent e)
{
    if (e.getSource() == addButton) {
        Course c = new Course();
        c.setCourseCode(codeText.getText());
        c.setName(nameText.getText());
        c.setPoints(pointsText.getText());
        // etc
        xml.addCourse(tmp);
    }
    else if (e.getSource() == saveButton) {
        xml.saveToFile("newsummercourses.xml");
    }
}
```



För att använda klassen `XMLCourse` skapar vi en applikation med ett grafiskt användargränssnitt i vilken man får skriva in kursinformation som kursnamn, kurskod etc. Hur detta gränssnitt är uppbyggt är inte det viktiga. Det viktiga är hur ett `Course`-objekt skapas för att sen sparas till fil.

När användaren trycker på knappen `Lägg till kurs` skapar vi ett nytt temporärt `Course`-objekt (`c`). Vi anropar därefter `Course`-objektets olika `set`-metoder för att sätta kursinformationen som användaren fyllt i. Som en instansvariabel i klassen har vi ett objekt av `XMLCourse` (`xml`). På detta objekt anropar vi metoden `addCourse` som kommer att skapa ett `course`-element som läggs in i XML-dokumentet.

När användaren trycker på knappen `Spara till XML-fil` anropar vi metoden `saveToFile` och anger som argument vilken fil XML-dokumentet ska sparas i (`newsummercourses.xml`).

Ta en titt på exemplen **`XMLCourse.java`** och **`CourseFrame.java`**.

Det finns väldigt mycket att lära om XML och Java. I denna lektion har vi enbart koncentrerat oss på SAX och DOM, men det finns som nämnts tidigare fler APIer. Förhoppningsvis har du lärt dig hur vi kan använda XML i dina applikationer genom att utnyttja SAX och/eller DOM.



## Java-objekt <--> XML

- Det vi sett exempel på är hur vi:
  - Mapper (läser) XML-dokument till Java-objekt
  - Mapper (skriver) Java-objekt till XML-dokument
- Veldig omständigt med Java API for XML Processing (JAXP)
- Underlättas som tur är med Java Architecture for XML Binding (JAXB)



## JAXB

- Är ett API som ingår i standard Java (från version 1.6)
- Kan användas för att:
  - Läsa in (unmarshalling) XML-dokument till Java-objekt
  - Skriva (marshalling) Java-objekt till XML-dokument
  - Skapa XML-scheman (XSD) (utökad variant av DTD) utifrån Java-objekt
  - Skapa Java-objekt utifrån en XSD



## JAXB

- Använder annotationer för att mappa
  - Rotelement till klasser
  - Övriga element till instansvariabler
  - Attribut till instansvariabler

```
@XmlRootElement
```

```
@XmlElement
```

```
@XmlAttribute
```

- Finns många fler



## JAXBContext

- Klassen är ingångspunkten för applikationer som vill använda sig av JAXB-apiet
- För att skapa en instans av klassen använd till exempel:

```
JAXBContext context =  
JAXBContext.newInstance(Class... classesToBeBound)
```

- Ange en kommaseparerad lista över klasser som den behöver känna till



## Marshaller

- Klassen ansvarar för att styra processen att serialisera Java-objekt till XML-dokument

- För att skapa en instans:

```
Marshaller marshaller = context.createMarshaller();
```

- För att serialisera till en "källa" (File, OutputStream eller Writer m.fl.):

```
marshaller.marshal(object, new File("object.xml"));  
marshaller.marshal(object, System.out);
```



## Unmarshaller

- Klassen ansvarar för att styra processen att deserialisera XML-data till Java-objekt

- För att skapa en instans:

```
Unmarshaller unmarshaller = context.createUnmarshaller();
```

- För att deserialisera från en "källa" (File, Reader eller URL m.fl.):

```
Object o1 = unmarshaller.unmarshal(new File("object.xml"));  
URL url = new URL("http://min.hemsida.se/object.xml");  
Object o2 = unmarshaller.unmarshal(url);
```



## Java-objekt till XML

- Skapa en klass i Java för varje förälder-element i XML

förälder-element

```
<summercourses year="2015">  
  <course code="MIU-F2079" start="v24" end="v35">  
    <coursecode>DT150G</coursecode>  
    <name>Flash, Photoshop och Datakunskap</name>  
    <points>7.5</points>  
    <description>  
      Lär du dig skapa flashapplikationer.  
    </description>  
  </course>  
</summercourses>
```

- Börja med innersta elementet Course
- Därefter de yttre (SummerCourses)



# Java-objekt till XML

## ■ Klassen Course

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement
public class Course {
    @XmlElement(name="coursecode")
    private String courseCode = "";
    @XmlElement
    private String name = "";
    @XmlElement
    private double points = 0;
    @XmlAttribute(name="code")
    private String applicationCode = "";
    @XmlElement
    private String description = "";
    @XmlAttribute
    private int start = 0;
    @XmlAttribute
    private int end = 0;
```

Klassen `Course` innehåller en del annotationer från JAXB som tillåter oss att indikera vilka XML-noder (element och attribut med mera) som vi vill generera. På klassnivå börjar vi med att använda `@XmlAccessorType` med värdet `XmlAccessType.FIELD` för att ange att det är instansvariablerna vi vill annotera. Annars är det som default set- eller get-metoderna som annoteras. Personligen tycker jag dock att man får en bättre överblick om vi annoterar instansvariablerna. På klassnivå fortsätter vi sen med `@XmlRootElement` för att ange rotelementet i det genererade XML-dokumentet. Namnet på rotelementet fås från namnet på klassen. Vill vi specificera ett annat namn kan vi göra det genom att lägga till attributet `name` med namnet på rotelementet som värde. Ex: `@XmlRootElement(name="kurs")`.

För att ange övriga XML-element, enbart element som i sig inte är förälder-element, använder vi `@XmlElement`. Dessa deklarerar vi ovanför den instansvariabel vars värde vi vill ska mappas till ett XML-element. Skriver vi bara `@XmlElement` hämtas namnet på XML-elementet från namnet på instansvariabeln (namnet blir exakt som instansvariabeln med små och stora bokstäver). Vill vi att instansvariabeln ska mappas mot ett annat namn använder vi även här attributet `name` och som värde det namn vi vill elementet ska ha. I vårt exempel vill vi att instansvariabeln `courseCode` ska mappas mot XML-elementet `coursecode` (bara små bokstäver).

För att ange att en instansvariabel ska mappas mot ett attribut till ett XML-element använder vi `@XmlAttribute`. I fallet med instansvariabeln `applicationCode` vill vi att den ska mappas mot attributet som har namnet `code`. Därför använder vi attributet `name` till `@XmlAttribute` och anger vilket namn XML-attributet har.

Så här fortsätter vi sen att annotera alla instansvariabler så att de mappas mot sina motsvarigheter i XML-dokumentet. Något mer behöver inte göras i klassen förutom att tillhandahålla publika set- och get-metoder för alla instansvariabler.



# Java-objekt till XML

## ■ Testklass

```
// Skapa Course-objekt
Course java3 = new Course();
java3.setName("Java III");
...

// Skapa en JAXB marshaller (skrivare) för vår klass Course
JAXBContext context = JAXBContext.newInstance(Course.class);
Marshaller marshaller = context.createMarshaller();

// Ange att utmatningen ska formateras (indenteras)
marshaller.setProperty(
    Marshaller.JAXB_FORMATTED_OUTPUT, true);

// Skriver objektet till xml (både till fil och standard ut)
marshaller.marshal(java3, new File("JavaIII_course.xml"));
marshaller.marshal(java3, System.out);
```

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<course code="MIU-C4036" start="24" end="36">
  <coursecode>DT066G</coursecode>
  <name>Java III</name>
  <points>7.5</points>
  <description>Sista kursen i vår Java-
stege.</description>
</course>
```

För att testa att skriva (serialisera) ett objekt av klassen `Course` till XML skriver vi en testklass. Vi börjar med att skapa ett nytt `Course`-objekt och sätter alla värden den ska ha. Därefter skapar vi en instans av klassen `JAXBContext` och anger som argument till metoden `newInstance` att den måste känna till klassen `Course`. Därefter skapar vi ett `Marshaller`-objekt genom att anropa metoden `createMarshaller` på `JAXBContext`-instansen.

På detta `Marshaller`-objekt sätter vi en inställning att utmatningen ska formateras (indenteras) genom att anropa metoden `setProperty` och som argument ange rätt värde som `true`. Nu är vi redo att skriva objektet till en källa med hjälp av `Marshaller`-objektet och en variant av dess överlagrade `marshal`-metoder. I exemplet skriver vi objektet både till en fil och till kommandofönstret (`System.out`).

Resultatet blir:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<kurs code="MIU-C4036" start="24" end="36">
  <coursecode>DT066G</coursecode>
  <name>Java III</name>
  <points>7.5</points>
  <description>Sista kursen i vår Java-stege.</description>
</kurs>
```

Se exempel 6, `Course.java` och `TestCourseToXML.java`.



# XML till Java-objekt

## ■ Testklass

```
// Skapa en JAXB unmarshaller (läsare) för vår klass Course
JAXBContext context = JAXBContext.newInstance(Course.class);
Unmarshaller unmarshaller = context.createUnmarshaller();

// Läs data från fil och gör en cast till rätt typ
Course c = (Course) unmarshaller.unmarshal(
    new File("JavaIII_course.xml"));

System.out.println(c);
```

DT066G, Java III, 7.5 points  
Anmälningkod: MIU-C4036 Pågår: 24 - 36  
Beskrivning: Sista kursen i vår Java-stege.

För att testa att läsa (deserialisera) ett objekt av klassen `Course` från XML skriver vi ytterligare en testklass. Vi skapar återigen en instans av klassen `JAXBContext` och anger som argument till metoden `newInstance` att den måste känna till klassen `Course`. Därefter skapar vi ett `Unmarshaller`-objekt genom att anropa metoden `createUnmarshaller` på `JAXBContext`-instansen.

På detta `Unmarshaller`-objekt anropar vi metoden `unmarshal` och anger som argument från vilken källa den ska läsa. I det här fallet är det från filen `JavaIII_course.xml`. Metoden `unmarshal` returnerar ett objekt av typen `Object` varför vi måste göra en typomvandling till `Course`. För att se om något lästes in skriver vi ut objektet `java3` till `System.out`. Resultatet blir:

DT066G, Java III, 7.5 points  
Anmälningkod: MIU-C4036 Pågår: 24 - 36  
Beskrivning: Sista kursen i vår Java-stege.

Se exempel 6, **TestXMLToCourse.java**.

Det vi nu sett exempel på är att skriva och läsa ett Java-objekt till/från fil. Vanligtvis innehåller ett XML-dokument flera nästlade element. För att kunna läsa och skriva denna typ av dokument måste vi i Java skapa motsvarande nästlade klasser. Detta ska vi se exempel på härnäst.





# Java-objekt till XML

## ■ Klassen SummerCourses innehåller flera Course-objekt

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlRootElement(name="summercourse")
public class SummerCourses {
    @XmlAttribute
    private int year;

    @XmlElement(name="course")
    private ArrayList<Course> courses =
        new ArrayList<Course>();
    ...
}
```

Klassen `SummerCourses` innehåller samma annotationer från JAXB som användes i klassen `Course`. På klassnivå börjar vi med att använda `@XmlAccessorType` med värdet `XmlAccessType.FIELD` för att ange att det är instansvariablerna vi vill annotera. På klassnivå fortsätter vi sen även med `@XmlRootElement` för att ange rotelementet i det genererade XML-dokumentet. Namnet på rotelementet fås normalt från namnet på klassen. Default sätts första ordet med liten bokstav och därefter varje nytt ord stor bokstav. Vi vill att namnet på elementet i XML endast ska ha små bokstäver varför vi lägger till attributet `name` med värdet `summercourses`.

Instansvariabeln `year` vill vi ska mappas mot attributet `year` i rotelementet `summercourses`. Vi anoterar därför instansvariabeln med `@XmlAttribute`.

Instansvariabeln `courses`, som kommer att innehålla alla kurser som ges under aktuell sommar, vill vi ska mappas mot elementet `course` i `summercourses`. Vi anoterar därför instansvariabeln med `@XmlElement` och ger den ett nytt namn, `course`.

Förutom dessa två instansvariabler innehåller klassen även ett par lämpliga konstruktörer, set- och get-metoder för de båda instansvariablerna, samt en metod `add` med vilken vi kan lägga till ett nytt `Course`-objekt till listan `courses`.



# Java-objekt till XML

## ■ Testklass

```
// Skapa SummerCourses-objekt
SummerCourses courses =
    new SummerCourses(2015);
courses.add(java1);
...

// Skapa en marshaller för klasserna SummerCourses & Course
JAXBContext context =
    JAXBContext.newInstance(SummerCourses.class, Course.class);
Marshaller marshaller = context.createMarshaller();

// Ange att utmatningen ska formateras (indenteras)
marshaller.setProperty(
    Marshaller.JAXB_FORMATTED_OUTPUT, true);

// Skriver objektet till xml
marshaller.marshal(courses, new File("summercourses.xml"));
```

```
<?xml version="1.0" encoding="UTF-8"
standalone="yes"?>
<course code="MIU-C4036" start="24" end="36">
  <coursecode>DT066G</coursecode>
  <name>Java III</name>
  <points>7.5</points>
  <description>Sista kursen i vår Java-
stege.</description>
</course>
```

För att testa att skriva ett objekt av klassen SummerCourse till XML skriver vi en testklass. Vi börjar med att skapa ett nytt SummerCourses-objekt och som argument till konstruktorn skickar vi 2015 (sommarkurser för år 2015). Därefter skapar vi tre Course-objekt och lägger till i SummerCourses genom att anropa vår egna metod add.

Därefter skapar vi en instans av klassen JAXBContext och anger som argument till metoden newInstance att den måste känna till klassen SummerCourses och Course (egentligen räcker det att enbart ange SummerCourses här). Därefter skapar vi ett Marshaller-objekt genom att anropa metoden createMarshaller på JAXBContext-instansen.

På detta Marshaller-objekt sätter vi en inställning att utmatningen ska formateras (indenteras). Nu är vi reda att skriva objektet till en källa med hjälp av Marshaller-objektet och en variant av dess överlagrade marshal-metoder. I exemplet skriver vi objektet till en fil.

Öppnar vi filen ser resultatet (en del av det) ut så här:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<summercourse year="2015">
  <course code="MIU-C4034" start="24" end="36">
    <coursecode>DT006G</coursecode>
    <name>Java I</name>
    <points>7.5</points>
    <description>Första kursen i vår Java-stege.</description>
  </course>
</summercourses>
```

Se exempel 7, **SummerCourses.java** och **TestSummerCoursesToXML.java**.



# XML till Java-objekt

## ■ Testklass

```
// Skapa en JAXB unmarshaller för klassen SummerCourses
JAXBContext context =
    JAXBContext.newInstance(SummerCourses.class);
Unmarshaller unmarshaller = context.createUnmarshaller();

// Läs data från fil och gör en cast till rätt typ
SummerCourses courses = (SummerCourses)
    unmarshaller.unmarshal(new File("summercourses.xml"));

System.out.println(courses);
```

```
Sommarkurser år 2015
Java I
Java II
Java III
```

För att testa att läsa in ett objekt av klassen `SummerCourses` från XML går vi tillväga på samma sätt som när vi läste in ett objekt av `Course`. Vi skapar en instans av klassen `JAXBContext` och anger som argument till metoden `newInstance` att den måste känna till klassen `SummerCourses` (som i sin tur använder `Courses`). Därefter skapar vi ett `Unmarshaller`-objekt genom att anropa metoden `createUnmarshaller` på `JAXBContext`-instansen.

På detta `Unmarshaller`-objekt anropar vi metoden `unmarshal` och anger som argument från vilken källa den ska läsa. I det här fallet är det från filen `summercourses.xml`. Metoden `unmarshal` returnerar ett objekt av typen `Object` varför vi måste göra en typomvandling till `SummerCourses`. För att se om något lästes in skriver vi ut objektet till `System.out`. Resultatet blir:

```
Sommarkurser år 2015
Java I
Java II
Java III
```

Se exempel 7, **TestXMLToSummerCourses.java**.

Det vi nu sett exempel på är att skriva och läsa ett Java-objekt, som i sin tur innehåller andra Java-objekt, till/från fil.