FILE OPERATIONS IN PYTHON MODULE 4

Files

- Python views a text file as a sequence of characters and a binary file (for images, videos and more) as a sequence of bytes.
- As in lists and arrays, the first character in a text file and byte in a binary file is located at position 0, so in a file of n characters or bytes, the highest position number is n – 1

End of File

- Every operating system provides a mechanism to denote the end of a file.
- Some represent it with an end-of-file marker, while others might maintain a count of the total characters or bytes in the file.
- Programming languages generally hide these operating-system details from you.

Standard File Objects

- When a Python program begins execution, it creates three standard file objects:
  - `sys.stdin`
    - the standard input file object
  - `sys.stdout`
    - the standard output file object, and
  - `sys.stderr`
    - the standard error file object.
- They do not read from or write to files by default

Writing to a Text File

- Many applications ***acquire*** resources, such as files, network connections, database connections and more.
- You should ***release*** resources as soon as they're no longer needed.
- This practice ensures that other applications can use the resources.
- Python's with **`statement`**:
  - acquires a resource and assigns its corresponding object to a variable
  - allows the application to use the resource via that variable
  - calls the resource object's **`close`** method to release the resource when program control reaches the end of the **`with`** statement's suite

Writing to a Text File

```
with open('accounts.txt', mode='w') as accounts:
    accounts.write('100, Jack Bauer, 10000.0\n')
    accounts.write('200, Kate Austen, 20000.0\n')
    accounts.write('300, James Sawyer, 30000.0\n')
    accounts.write('400, Ben Linus, 40000.0\n')
    accounts.write('500, Sun Kwon, 50000.0\n')
```

- The built-in **`open`** function opens the file **`accounts.txt`** and associates it with a file object.
- The mode argument specifies the file-open mode, indicating whether to open a file for reading from the file, for writing to the file or both.
- The mode **`'w'`** opens the file for writing, creating the file if it does not exist.
- If you do not specify a path to the file, Python creates it in the current folder

Reading from a Text File

```
with open('accounts.txt', mode='r') as accounts:
    print(f'{"Account":<10}{"Full Name":<20}{"Balance":>10}')
    for record in accounts:
        account, name, balance = record.split(',')
        print(f'{account:<10}{name.strip():<20}{balance.strip():>10}')
```

```
Account   Full Name              Balance
100       Jack Bauer             10000.0
200       Kate Austen            20000.0
300       James Sawyer           30000.0
400       Ben Linus              40000.0
500       Sun Kwon               50000.0
```

File open modes

Mode	Description
`'r'`	Open a text file for reading. This is the default if you do not specify the file-open mode when you call open.
`'w'`	Open a text file for writing. Existing file contents are deleted.
`'a'`	Open a text file for appending at the end, creating the file if it does not exist. New data is written at the end of the file.
`'r+'`	Open a text file reading and writing.
`'w+'`	Open a text file reading and writing. Existing file contents are deleted.
`'a+'`	Open a text file reading and appending at the end. New data is written at the end of the file. If the file does not exist, it is created.

File Method **readlines**

> The file object's **readlines** method also can be used to read an entire text file.

> The method returns each line as a string in a list of strings.

> For small files, this works well, but iterating over the lines in a file object can be more efficient.

File Method **readlines**

```
accounts = open('accounts.txt', mode='r')
print(f'{"Account":<10}{"Full Name":<20}{"Balance":>10}')
for line in accounts.readlines():
    account, name, balance = line.split(',')
    print(f'{account:<10}{name.strip():<20}{balance.strip():>10}')
accounts.close()
```

```
Account   Full Name             Balance
100       Jack Bauer            10000.0
200       Kate Austen           20000.0
300       James Sawyer          30000.0
400       Ben Linus             40000.0
500       Sun Kwon              50000.0
```

Seeking to a Specific File Position

- While reading through a file, the system maintains a file-position pointer representing the location of the next character to read.
- Sometimes it's necessary to process a file sequentially from the beginning several times during a program's execution.
- Each time, you must reposition the file-position pointer to the beginning of the file, which you can do either by
  - closing and reopening the file
  - calling the file object's seek method, as in
    - `file_object.seek(0)`

Updating Text Files

- Formatted data written to a text file cannot be modified without the risk of destroying other data.
- Copy and update to a temporary file

```
accounts = open('accounts.txt', 'r')
```

```
temp_file = open('temp_file.txt', 'w')
```

```
with accounts, temp_file:
    for account in accounts:
        iban, fullname, balance = account.split(',')
        if  iban.strip() != "300":
            temp_file.write(account)
        else:
            new_account = f'{iban.strip()}, James Sawyer Ford, {balance.strip()}\n'
            temp_file.write(new_account)
```

os Module File-Processing Functions

> The **os** module provides functions for interacting with the operating system, including several that manipulate your system's files and directories.

```
import os
```

```
os.remove("accounts.txt")
```

```
os.rename("temp_file.txt", "accounts.txt")
```

READING AND WRITING TO JSON

Serialization with JSON

- JSON (JavaScript Object Notation) is a text-based, human-and-computer-readable, data-interchange format used to represent objects (such as dictionaries, lists and more) as collections of name–value pairs.
- JSON can even represent objects
- JSON has become the preferred data format for transmitting objects across platforms.
- This is especially true for invoking cloud-based web services, which are functions and methods that you call over the Internet.

JSON Data Format

- JSON objects are like Python dictionaries.
- Each JSON object contains a comma-separated list of property names and values, in curly braces

```
{
   "iban": "TR1",
   "fullname": "Jack Bauer",
   "balance": 10000.0
}
```

- JSON also supports arrays which, like Python lists, are comma-separated values in square brackets.

```
[100, 200, 300]
```

JSON Data Format

> Values in JSON objects and arrays can be:
  - strings in ***double quotes*** (like "Jack"),
  - numbers (like 100 or 24.98),
  - JSON Boolean values (represented as true or false in JSON),
  - **`null`** (to represent no value, like **None** in Python),
  - arrays (like [100, 200, 300]), and
  - other JSON objects.

Python Standard Library Module **`json`**

> The **`json`** module enables you to convert objects to JSON (JavaScript Object Notation) text format.

> This is known as ***serializing*** the data.

```
import json
```

```
accounts = { 'accounts': [
    { "iban": "TR1", "fullname": "kate austen", "balance" : 100000.},
    { "iban": "TR2", "fullname": "jack shephard", "balance" : 200000.},
    { "iban": "TR3", "fullname": "ben linus", "balance" : 300000.}
]}
```

```
with open("accounts.json", "w") as json_file:
    json.dump(accounts, json_file)
```

Deserializing the JSON Text

> The json module's load function reads the entire JSON contents of its file object argument and converts the JSON into a Python object.

> This is known as ***deserializing*** the data.

```
with open("accounts.json", "r") as json_file:
    accounts = json.load(json_file)
```

```
accounts
```

```
{'accounts': [{'iban': 'TR1', 'fullname': 'kate austen', 'balance': 100000.0},
  {'iban': 'TR2', 'fullname': 'jack shephard', 'balance': 200000.0},
  {'iban': 'TR3', 'fullname': 'ben linus', 'balance': 300000.0}]}
```

Displaying the JSON Text

> The json module's **dumps function** (dumps is short for "dump string") returns a Python string representation of an object in JSON format.

> Using dumps with load, you can read the JSON from the file and display it in a nicely indented format—sometimes called "pretty printing" the JSON.

> When the **`dumps`** function call includes the **`indent`** keyword argument, the string contains newline characters and indentation for pretty printing

```python
with open("accounts.json", "r") as json_file:
    print(json.dumps(json.load(json_file),indent=4))
```

```
{
    "accounts": [
        {
            "iban": "TR1",
            "fullname": "kate austen",
            "balance": 100000.0
        },
        {
            "iban": "TR2",
            "fullname": "jack shephard",
            "balance": 200000.0
        },
        {
            "iban": "TR3",
            "fullname": "ben linus",
            "balance": 300000.0
        }
    ]
}
```

READING AND WRITING TO

Serialization with CSV

> CSV is an important format for transferring, migrating and quickly visualizing data as all spreadsheets support viewing and editing CSV files directly whilst it's supported by most RDBMS support exporting and importing data.

> Compared with other serialization formats, it provides a compact and efficient way to transfer large datasets in an easy-to-read text format.

Writing to CSV File

```
import csv

countries = [
    ("tur", "turkey", "asia", 80000000),
    ("fra", "france", "europe", 67000000),
    ("ita", "italy", "europe", 60000000)
]

with open("world.csv", mode="wt", newline='') as file:
    writer = csv.writer(file)
    writer.writerows(countries)
```

Writing from CSV File

```
import pandas as pd

countries = [
    ("tur", "turkey", "asia", 80000000),
    ("fra", "france", "europe", 67000000),
    ("ita", "italy", "europe", 60000000)
]

df = pd.DataFrame(countries, columns=["code", "name",
"continent", "population"])
df.to_csv("world_pandas.csv")
```

Reading from CSV File

```
import csv

with open("world.csv", mode="rt") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Reading from CSV File

```
import pandas as pd

df = pd.read_csv("world_pandas.csv")
print(df.index)
print(df.columns)
print(df)
```

VALIDATING XML WITH DTD AND XML SCHEMA

Objectives

> After completing this lesson, you should be able to do the following:

- Describe XML
- Describe the structure of an XML document
- List the components of an XML document
  - Elements
  - Attributes
  - Entities
- Create a well-formed XML document

Extensible Markup Language

> Extensible Markup Language (XML):
  - Describes data objects called XML documents
  - Is composed of markup language for structuring data
  - Supports custom tags for definition, transmission, validation, and interpretation of data
  - Conforms to Standard Generalized Markup Language (SGML)
  - Has become a standard way to describe data on the Web
  - Is processed by XML processors

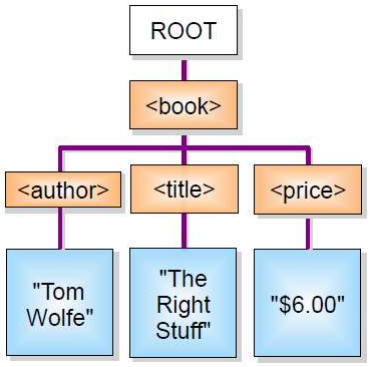
Example: A Simple XML Page

```
<?xml version="1.0"?>
<employees>
  <employee>
    <employee_id>120</employee_id>
    <last_name>Weiss</last_name>
    <salary>8000</salary>
  </employee>
  <employee>
    <employee_id>121</employee_id>
    <last_name>Fripp</last_name>
    <salary>8200</salary>
  </employee>
</employees>
```

Example Tree Representation of XML

```
<?xml version="1.0"?>
<book>
  <author>
    Tom Wolfe
  </author>
  <title>
    The Right Stuff
  </title>
  <price>
    $6.00
  </price>
</book>
```

=



A Simple XML Document - Basic Structure

`<?xml version="1.0"?>`	"Optional" first line; only required if encoding IS NOT UTF-8 or UTF-16*
`<book>`	Root element start tag

`<title>`
`  Alphabet from A to Z`
`</title>`
	First child element with data

`<isbn number="1112-23-4356" />`	Empty element (no data)
`<author>`	Begin element tag
`<firstName>Boreng</firstName>` `<lastName>Riter</lastName>`	Nested child elements
`</author>`	End element tag

`<chapter title="Letter A">`
`  The letter A is the first in`
`  the alphabet. It is also the`
`  first of five vowels.`
`</chapter>`
	Element containing an attribute and parsed character data (PCDATA) [TBD]

`<!-- The rest of the letter` `  chapters are missing -->`	Comment

`<chapter title="Letter Z">`
`  The letter Z is the last`
`  letter in the alphabet.`
`</chapter>`
	Last element in document

`</book>`	Root element end tag

XML Document Structure

> An XML document contains the following parts:

1. Prologue
2. Root element
3. Epilogue

```
<?xml version="1.0" encoding="WINDOWS-1252"?>          (1)
<!-- this is a comment -->
<employees>                                            (2)
 ...
</employees>
<?gifPlayer size="100,300" ?>                          (3)
```

The XML Declaration

> XML documents must start with an XML declaration.

> The XML declaration:

– Looks like a processing instruction with the `xml` name. For example:

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<document-root>
...
</document-root>
```

– Must contain the `version` attribute

– May (optionally) include:

- The `encoding` attribute
- The `standalone` attribute

– Is optional in XML 1.0, but mandatory in XML 1.1

Components of an XML Document

> XML documents comprise storage units containing:
  – Parsed data, including the:
    • Markup (elements, attributes, entities) used to describe the data they contain
    • Character data described by markup

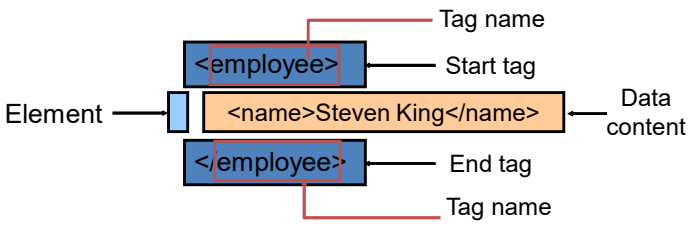
```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<employees>
  <employee id="100">
    <name>Rachael O&apos;Leary</name>
  </employee>
</employees>
```

  – Unparsed data, textual or binary information (graphic and sound data) taken as entered

```
<![CDATA[  ...unparsed data... ]]>
```

XML Elements

> An XML element:
  – Has a start tag, end tag, and optional data content
  – Tag names are case-sensitive (must be identical)



> Empty elements:
  – Do not contain any data
  – May appear as a single tag

```
<initials></initials>
<initials/>
```

Markup Rules for Elements

> There is one root element, sometimes called the top-level or document element.

> All elements:

- Must have matching start and end tags, or be a self-closing tag (an empty element)
- Can contain nested elements, such that their tags do not overlap
- Have case-sensitive tag names subject to naming conventions: Start with a letter, no spaces, and do not start with the letters `xml`
- May contain white space (spaces, tabs, new lines, and combinations of them) that is considered part of the element data content

XML Attributes

> An XML attribute is a name-value pair that:

– Is specified in the start tag, after the tag name

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<employees>
  <employee id="100" name='Rachael O&apos;Leary'>
    <salary>1000</salary>
  </employee>
</employees>
```

– Has a case-sensitive name

– Has a case-sensitive value that must be enclosed in matching single or double quotes

– Provides additional information about the XML document or XML elements

Using Elements Versus Attributes

1 Elements

```
<?xml version="1.0"?>
<employees>
  <employee>
    <id>100</id>
    <last_name>King</last_name>
    <salary>24000</salary>
  </employee>
</employees>
```

2 Attributes

```
<?xml version="1.0"?>
<employees>
  <employee id="100" last_name="King"
            salary="24000">
    <job>President</job>
  </employee>
</employees>
```

XML Entities

- An XML entity:
  - Is a unit of data storage
  - Is identified by a case-sensitive name
  - Is used as replacement text (substituted) when referencing its name between an ampersand (&), and a semicolon (;)

```
<comment>Salaries must not be &lt; 1000</comment>
```

  - Has predefined names for special XML characters:

Entity	Description	Character
<	"less than"	<
>	"greater than"	>
&	"ampersand"	&
'	"apostrophe"	'
"	"quote"	"

XML Comments

- XML comments:
  - Start with <!--
  - End with -->
  - May appear anywhere in the character data of a document, and before the root element
  - Are not elements, and can occupy multiple lines
  - May not appear inside a tag or another comment

```
<?xml version="1.0" encoding="WINDOWS-1252"?>
<!-- Comment: This document has information about
     employees in the company -->
<employees>
 <name>Steven King</name> <!-- Full name -->
</employees>
```

A Well-Formed XML Document

- Every XML document must be well-formed, such that:
  - An XML document must have one root element
  - An element must have matching start and end tag names, unless they are empty elements
  - Elements can be nested, but cannot overlap
  - All attribute values must be quoted
  - Attribute names must be unique in the start tag of an element
  - Comments and processing instructions do not appear inside tags
  - The < or & special characters cannot appear in the character data of an element or attribute value

Class Activity

> Identify errors in the following examples:

```
<?xml version="1.0"?>
<Question>Is this legal?</Question>
<Answer>No</Answer>
```
1

```
<!-- An XML document -->
<?xml version="1.0"?>
```
2

```
<Question 4You="Is this attribute name correct"/>
```
3

```
<EMAIL ID=Mark.Ant@oracle.com></EMAIL>
```
4

```
<Question>Is this legal</question>
```
5

White Space

> XML defines white space as any of these 4 characters
  - Horizontal tab
  - Line feed
  - Carriage return
  - Space
> An XML parser must pass all white space contained within content to the application
> An XML parser may remove white space in element tags and attribute values
> All end of line characters are converted to line feed characters by parsers

XML Names

- Element and other XML names may contain essentially any alphanumeric character.
- This includes the standard English letters A through Z and a through z as well as the digits 0 through 9.
- XML names may also include non-English letters, numbers, and ideograms such as ö, ç, Ω
- They may also include these three punctuation characters:
  - _ the underscore
  - - the hyphen
  - . the period

XML Names (Con't)

- XML names may only start with letters, ideograms, and the underscore character.
- They may not start with a number, hyphen, or period.
- There is no limit to the length of an element or other XML name.
- Thus these are all well-formed elements:
  - <Drivers_License_Number>98 NY 32 </Drivers_License_Number>
  - <month-day-year>7/23/2001</month-day-year>
  - <first_name>Alan</first_name>
  - <_4-lane>I-610</_4-lane>
  - <téléphone>011 33 91 55 27 55 27</téléphone>

XML Names (Con't)

```
<permitedNames>
    <name/>
    <xsl:copy-of>
     <A_long_element_name/>
    <A.name.separated.with.full.stops/>
    <a123323123-231-231/>
    <_12/>
</permitedNames>
```

```
<forbidenNames>
<A;name/>
<last@name>
<@#$%^()%+?=/>
<A*2/>
<1ex/>
</forbidenNames>
```

Entity References

- The character data inside an element may not contain a raw unescaped opening angle bracket (<).
- This character is always interpreted as beginning a tag
- If you need to use this character in your text, you can escape it using the **<**
- When a parser reads the document, it will replace the **<** entity reference with the actual **<** character
- <publisher>O'Reilly **&** Associates</publisher>

Entity References (Con't)

XML predefines exactly five entity references:

- **<**
  - The less-than sign; a.k.a. the opening angle bracket (<)
- **&**
  - The ampersand (&)
- **>**
  - The greater-than sign; a.k.a. the closing angle bracket (>)
- **"**
  - The straight, double quotation marks (")
- **'**
  - The apostrophe; a.k.a. the straight single quote (')

Character References

- Character references represent displayable characters that cannot otherwise be displayed
- Character references are either decimal or hexadecimal numbers
  - Decimals are preceded by "&#"
  - Hexadecimals are preceded by "&#x"
  - All character references end with a semicolon
- Example:
  - `&#169` or `&#xA9` will display as ©

CDATA - Character Data

Syntax:

<![CDATA[...Anything can go here...]]>

- Note: Anything except the literal string "**]]>**"; to embed "**]]>**" use "**]]>**"
- CDATA is not parsed and is treated as-is.
- Useful for embedding other languages within the XML.
  - HTML documents.
  - XML documents.
  - JavaScript source.
  - Or any other text with a lot of special characters.

CDATA Examples

- These script elements contain JavaScript:

```
<script><![CDATA[
function matchwo(a,b) {
  if (a < b && a < 0)
  then
    { return 1 }
  else
    { return 0 }
}
]]></script>
```

```
<script><![CDATA[
function matchwo(a,b) {
  if (a < b &#38;&#38; a < 0)
  then
     { return 1 }
  else
     { return 0 }
}
]]></script>
```

- This nameXML element stores actual XML to be treated as text:

```
<nameXML>
  <![CDATA[
    <name common="freddy" breed="springer-spaniel">
      Sir Frederick of Ledyard's End
    </name>
  ]]>
</nameXML>
```

Comparing XML and HTML

> XML
- Is a markup language for describing data
- Contains user-defined markup elements
- Is extensible
- Is displayed as a document tree in a Web browser
- Conforms to rules for a well-formed document

> HTML
- Is a markup language for formatting data in a Web browser
- Contains predefined markup tags
- Is not extensible
- Does not conform to well-formed document rules

XML Development

> Developing XML documents can be done using:
- A simple text editor, such as Notepad
- A specialized XML editor, such as XMLSpy
- Eclipse XML-related features that include:
  - Syntax checking for XML documents
  - XML Editor with code insight for XML schema-driven editing
  - Registering of external XML schemas
  - Validation of XML documents against registered XML schemas

QUIZ

XML Quiz 1

> Find errors:

```
<root>
<e1 a*b = "23432"/>
<e2 value = "12'/>
<e3 value="aa"aa"/>
<e4 value=bbbb/>
<e5 xml-ID = "xml2"/>
</root>
```

XML Quiz 1

> Solution:

```
<root>
<e1 a*b = "23432"/>
<e2 value = ''12'/>
<e3 value="aa"aa"/>
<e4 value='bbbb'/>
<e5 xml-ID = "xml2"/>
</root>
```

XML Quiz 2

> Find Errors:

```
<root>
<example>
  <![CDATA[ <P>Q&R]]>
</example>
<Name>
  Binnur Kurt
</Name>
<Address/>
</root>
```

XML Quiz 2

> Solution:

No error

XML Quiz 3

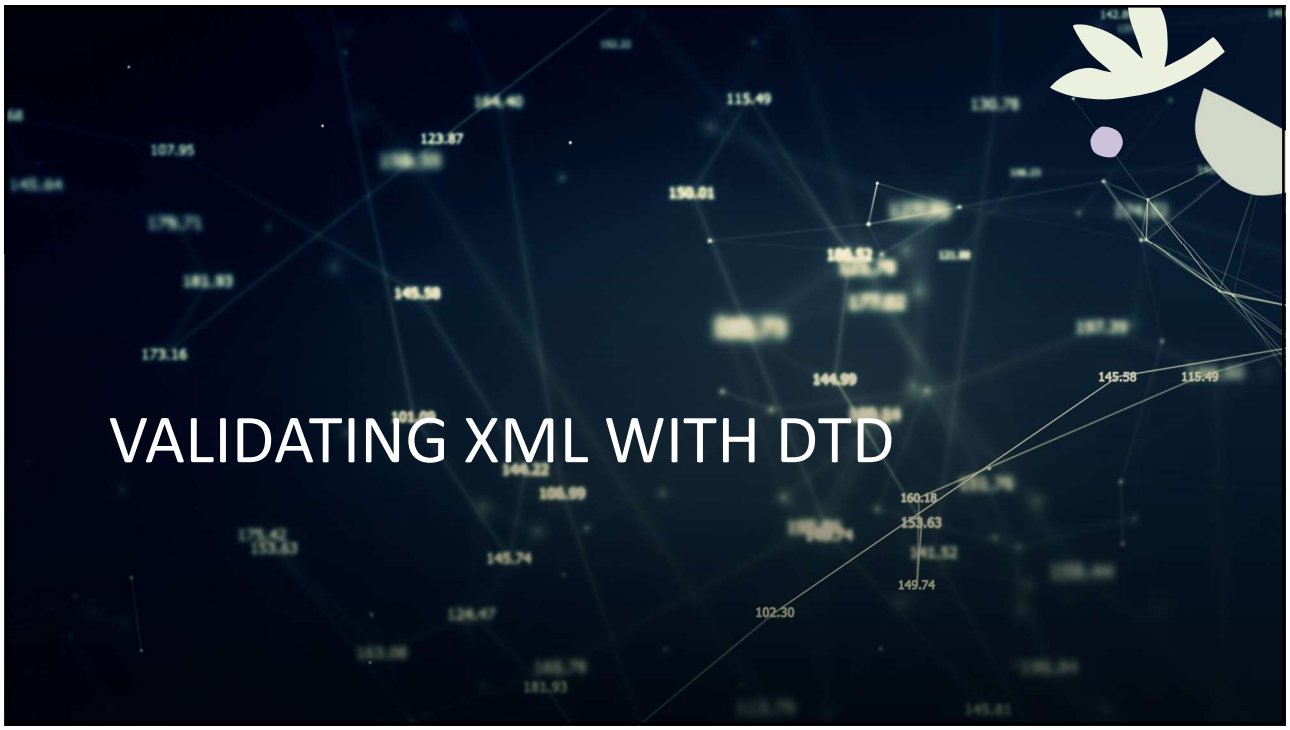
> Find Errors:

```
<root>
<isLower>
    23 < 46
</isLower>
<Name>
    Willey & Sons
</name>
</root>
```

XML Quiz 3

> Solution:

```
<root>
<isLower>
    23 < 46
</isLower>
<Name>
    Willey & Sons
</name>
</root>
```



Objectives

> After completing this lesson, you should be able to do the following:
- Describe a DTD
- Compare an internal DTD with an external DTD
- Read and write a DTD
- Validate XML documents by using an internal DTD, an external DTD, or their combination

What Is a Document Type Definition?

> A document type definition (DTD):
- Is the grammar for an XML document
- Contains the definitions of
  - Elements
  - Attributes
  - Entities
  - Notations
- Contains specific instructions that the XML Parser interprets to check the document validity
- May be stored in a separate file (external)
- May be included within the document (internal)

Why Validate an XML Document?

- Well-formed documents satisfy XML syntax rules, and not the business requirements about the content and structure.
- Business rules often require validation of the content and structure of a document.
- XML documents must satisfy structural requirements imposed by the business model.
- A valid XML document can be reliably processed by XML applications.
- Validations can be done by using a DTD or using an XML schema.

General DTD Rules

- A DTD:
  - Must provide a declaration for items used in an XML document, such as:
    - Elements
    - Attributes
    - Entities
  - Is case sensitive, but spacing and indentation are not significant
  - May use XML comment syntax for documentation, but comments cannot appear inside declarations
  - Forbids using anything that is not explicitly permitted

The Contents of a DTD

> A DTD contains declarations (using the syntax shown) for:

– Elements:

```
<!ELEMENT element-name content-model>
```

– Attributes:

```
<!ATTLIST element-name attrib-name type default>
```

– Entities:

```
<!ENTITY entity-name "replacement text">
```

– Notations:

```
<!NOTATION notation_name SYSTEM "text">
```

Example of a Simple DTD Declaration

> Example of a simple DTD with element declarations:

```
<!ELEMENT employees (employee)>
<!ELEMENT employee (name)>
<!ELEMENT name (#PCDATA)>
```

> A valid XML document based on the DTD is:

```
<?xml version="1.0"?>
<employees>
  <employee>
    <name>Steven King</name>
  </employee>
</employees>
```

> Note: All child elements must be defined.

Referencing the DTD

> The XML document references the DTD:
  – After the XML declaration and before the root by using:

```
<!DOCTYPE employees [ ... ]>
```

  – Externally with the `SYSTEM` or `PUBLIC` keywords:

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

```
<!DOCTYPE employees PUBLIC "-//formal-public-ID">
```

  – Internally in the `<!DOCTYPE root [...]>` entry:

```
<?xml version="1.0"?>
<!DOCTYPE employees [
 <!ELEMENT employees (#PCDATA)>
]>
<employees>Employee Data</employees>
```

Note: Use the root element name after `<!DOCTYPE`.

Element Declarations

> Element declaration syntax:

```
<!ELEMENT element-name content-model>
```

> Four kinds of content models:

```
<!ELEMENT job EMPTY>   <!-- Empty -->
```
1

```
<!-- Elements: single, ordered list, or choice -->
<!ELEMENT employees (employee)>
<!ELEMENT employee (employee_id,last_name,job_id)>
<!ELEMENT job_id (manager | worker)>
```
2

```
<!-- Mixed -->
<!ELEMENT last_name (#PCDATA)>
<!ELEMENT hire_date (date| (day,month,year))>
```
3

```
<!ELEMENT employee_id ANY>   <!-- Any -->
```
4

Specifying Cardinality of Elements

> Cardinality symbols:
  – Indicate the number of children permitted
  – Appear as suffixes:

No symbol (default)	Mandatory (one and only one)
? (question mark)	Optional (zero or one)
* (asterisk)	Zero or more (optional)
+ (plus sign)	One or more (mandatory)

  – Are placed after:
    • An element or group in the content model
    • The content model
> Note: A group is formed using parentheses.

Attribute Declarations

> The syntax for declaring an attribute is:

```
<!ATTLIST element-name attrib-name type default>
```

> Attribute declaration requires:
  – An element name
  – An attribute name
  – An attribute type, specified as:
    CDATA, enumerated, ENTITY, ENTITIES, ID, IDREF, IDREFS, NMTOKEN, NMTOKENS, and NOTATION
  – An attribute default, specified as:
    #IMPLIED, #REQUIRED, #FIXED, or a literal value
> Example:

```
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee manager_id CDATA #IMPLIED>
```

CDATA and Enumerated Attribute Types

> CDATA: For character data values

```
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee manager_id CDATA #IMPLIED>
```

```
<employee manager_id="102">          <!-- XML -->
  <employee_id>104</employee_id>
  <last_name>Ernst</last_name>
</employee>
```

> Enumerated: For a choice from a list of values

```
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee gender (male|female) #IMPLIED>
```

```
<employee gender="male">             <!-- XML -->
  <employee_id>104</employee_id>
  <last_name>Ernst</last_name>
</employee>
```

NOTATION Declaration and Attribute Type

> Declaring a NOTATION:

```
<!NOTATION notation_name SYSTEM "text">
```

> The NOTATION attribute type represents a name of a NOTATION declared in the DTD:

```
<?xml version="1.0"?>
<!DOCTYPE photos [
<!ELEMENT photos (image+)>
<!ELEMENT image EMPTY>
<!NOTATION gif SYSTEM "image/gif">
<!NOTATION jpeg SYSTEM "image/jpeg">
<!ATTLIST image
   source CDATA #REQUIRED
   type NOTATION (gif | jpeg) #REQUIRED>
]>
<photos>
  <image source="myphoto.gif" type="gif"/>
  <image source="mypet.jpg" type="jpeg"/>
</photos>
```

Specifying Default Attribute Values

- A quoted default attribute value:
  - Can be specified in the DTD after the attribute type:

```
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee department_id (10|60|90) '90'>
```

  - Is not specified when using the `#IMPLIED` or `#REQUIRED` keywords
  - Is required in the DTD if using the `#FIXED` keyword

```
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee manager_id CDATA #IMPLIED>
<!ATTLIST employee min_salary CDATA #FIXED '4000'>
```

- An attribute value is mandatory in the XML document when using the `#REQUIRED` keyword.

Entities in XML

- Entity types predefined by XML standards:
  - Built-in entities
  - Character entities
- Entity types that can be declared in the DTD:
  - General entities
  - Parameter entities
- Entity references for:
  - Built-in, character, and general entities can be made in an XML document or the DTD using:

```
&entity-name;
```

  - Parameter entities can be made in the DTD using:

```
%entity-name;
```

General Entity Declarations

> Internal entity declaration (parsed)

```
<!ENTITY entity-name "replacement text">
```

```
<!ENTITY company "Oracle Corporation">
<!ENTITY Emperor "Alexander &quot;The Great&quot;">
<!ENTITY president "<employee>
                    <last_name>King</last_name>
                    </employee>">
```

> External entity declarations:

– Parsed (text stored in external location)

```
<!ENTITY entity-name SYSTEM "file.ext | URL">
```

– Unparsed (requires a notation declaration)

```
<!ENTITY entity SYSTEM "file.ext" NDATA notation>
<!NOTATION notation SYSTEM "text">
```

Parameter Entities

> Declared as:

```
<!ENTITY % entity-name "replacement text">
```

– Requires a space after the percent (%) symbol

– Must be declared before referenced in the DTD

– Must be defined in an external DTD

– Replacement text can be in external location using:

```
<!ENTITY % entity-name SYSTEM "file.dtd | URL ">
```

> Examples:

```
<!ENTITY % employee_elements "last_name, salary">
<!ENTITY % employee_elements SYSTEM "empelm.txt">
```

> Referenced in the DTD using:

```
%entity-name;
```

```
<!ELEMENT employee (%employee_elements;)>
```

Complete DTD: Example

File: employees.dtd

```
<!ELEMENT employees (employee+)>
<!ELEMENT employee (employee_id, last_name)>
<!ATTLIST employee manager_id CDATA #IMPLIED
                   department_id (10|60|90) '90'>
<!ELEMENT employee_id (#PCDATA )>
<!ELEMENT last_name (#PCDATA )>
```

```
<?xml version="1.0"?>
<!DOCTYPE employees SYSTEM "employees.dtd" [
<!ENTITY title "Mr">
]>
<employees>
  <employee manager_id="100" department_id="10">
    <employee_id>100</employee_id>
    <last_name>&title; king</last_name>
  </employee>
</employees>
```

Validating XML Against a DTD

- The `oraxml` command-line utility or XML Parser:
  - Requires `xmlparserv2.jar` in the `CLASSPATH`
  - Uses `-dtd` option for full validation with a DTD:

```
java oracle.xml.parser.v2.oraxml -dtd emp.xml
```

  - Produces the following message when the XML document is valid:

```
The input XML file is parsed without errors
using DTD validation mode.
```

  - Can be invoked, in this course, as an External Tool in Oracle JDeveloper 10*g*
    - Right-click the XML document
    - Select `OraXML-dtd` from the menu

Objectives

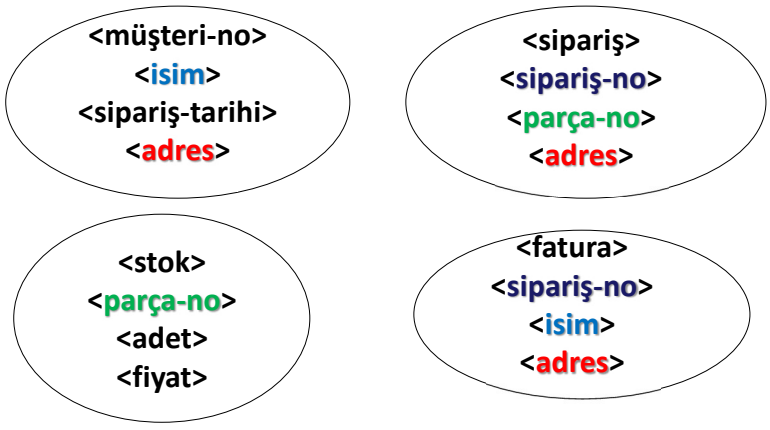
> After completing this lesson, you should be able to do the following:

- Describe XML Schema
- Create an XML Schema document
- Reference an XML Schema in an XML document
- Create an XML Schema document by using:
  - Built-in data types
  - Simple types
  - Complex types with sequences and choices
- Validate XML documents with XML Schema

Geçerli XML

- XML dokümanın belirli bir sözdizimine uyması istenebilir
- Söz dizimi iki farklı biçimde tanımlanabilir
  - DTD (Document Type Definition)
  - XML Schema

XML İsim Uzayı (=namespace)

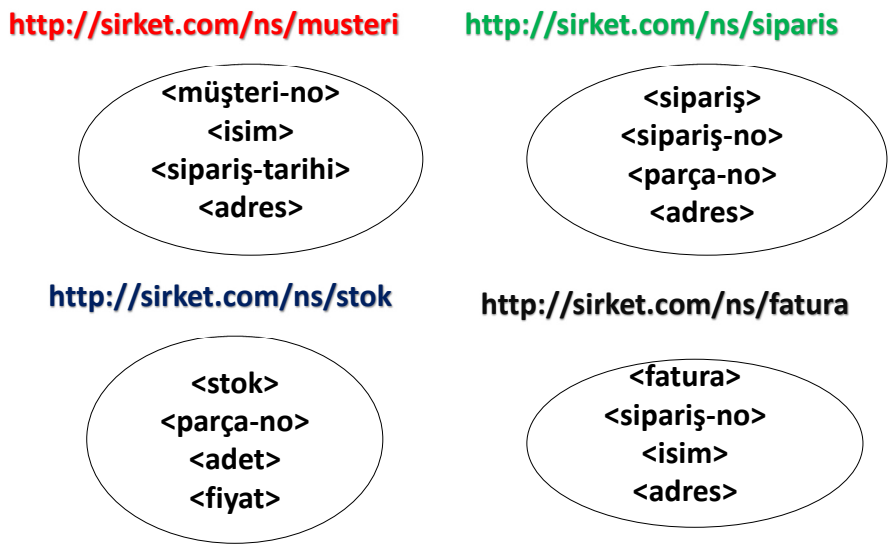


XML İsim Uzayı (=namespace)

> XML isim uzayı takı ve öznitelik isimlerinin oluşturduğu bir kümeyi çevreler.

> Bu küme bir URI (globally unique identifier) referansı ile ilişkilendirilir.

XML İsim Uzayı (=namespace)

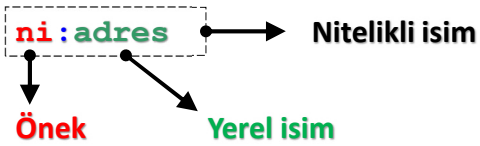


XML İsim Uzayı (=namespace)

> Artık XML dokümanında takı isimlerine isim uzayı kimliği eşlik edecektir
> Bu şekilde, XML ayrıştırıcı bir XML dokümanı herhangi bir belirsizliğe yol açmadan işleyebilecektir.

Nitelikli İsim (QName)

> Bir XML dokümanında yer alan isim uzayında tanımlanmış her isim nitelikli isim olarak adlandırılır
> Nitelikli isim, **:** sembolü ile ayrılmış önek ve yerel bileşenden oluşur



Önek ile isim uzayını bağlamak

> İsim uzayı özel amaçlı bir sözcük olan **xmlns** kullanılarak tanımlanır

<adres **xmlns**:ni="http://sirket.com/ns/musteri">

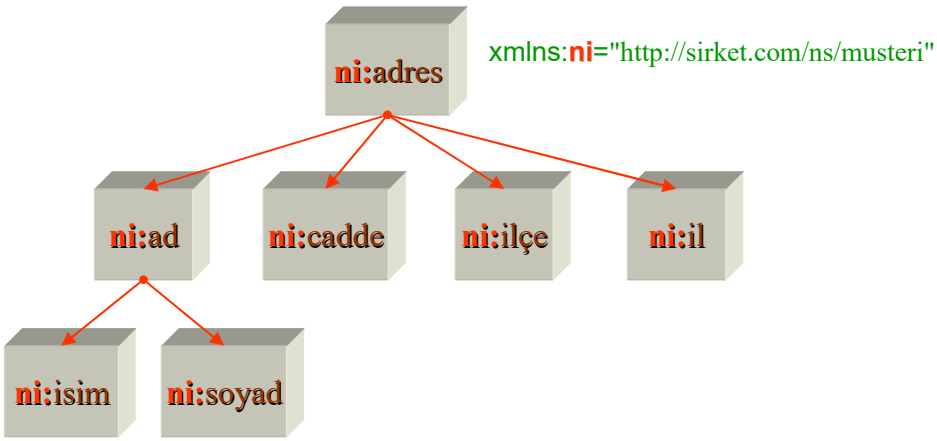
> Eğer bir eleman için herhangi bir önek, dolayısı ile bir isim uzayı tanımlanmamış ise varsayılan isim uzayı geçerlidir

<adres **xmlns**="http://sirket.com/ns/musteri">

Önek ile isim uzayını bağlamak

```
<ni:adres xmlns:ni="http://sirket.com/ns/musteri">
   <ni:ad>
     <ni:isim>hakan </ni:isim>
     <ni:soyad>sevinç</ni:soyad>
   </ni:ad>
   <ni:cadde>pak</ni:cadde>
   <ni:ilçe>üsküdar</ni:ilçe>
   <ni:il>istanbul</ni:il>
</ni:adres>
```

Önek ile isim uzayını bağlamak



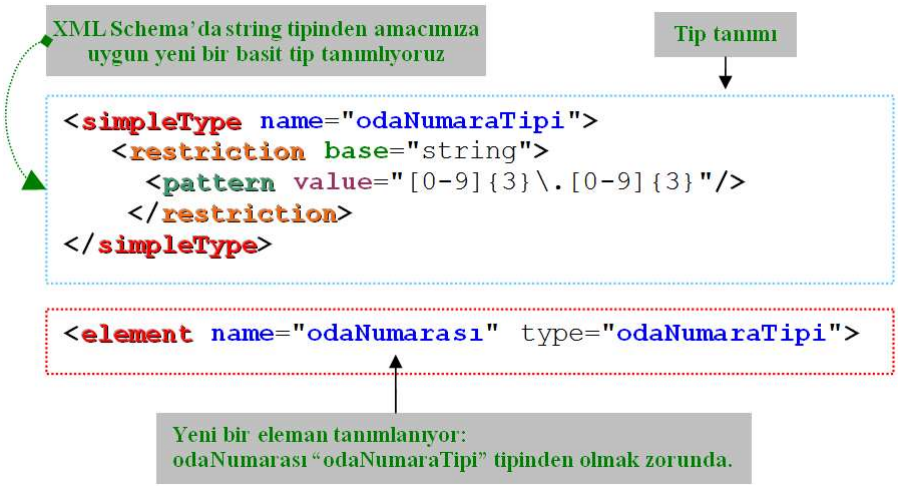
Önek ile isim uzayını bağlamak

```
<adres xmlns="http://sirket.com/ns/musteri">
<ad>
  <isim>hakan</isim>
  <soyad>sevinç</soyad>
<ad>
<cadde>pak</cadde>
<ilçe>üsküdar</ilçe>
<il>istanbul</il>
<adres>
```

Önek ile isim uzayını bağlamak

```
<ni:adres xmlns:ni="http://sirket.com/ns/musteri"
          xmlns:tl="http://sirket.com/ns/fatura">
<ni:ad>
  <ni:isim>hakan</ni:isim>
  <ni:soyad>sevinç</ni:soyad>
</ni:ad>
<ni:cadde>pak</ni:cadde>
<ni:ilçe>üsküdar</ni:ilçe>
<ni:il>istanbul</ni:il>
<ni:adres>
<tl:telefon>
  <tl:ülke>90</tl:ülke>
  <tl:alan>212</tl:alan>
  <tl:no>5554721</tl:no>
</tl:telefon>
</ni:adres>
```

XML Schema Örneği



XML Schema Tanımlaması

- XML Schema tanımının kendisi bir XML dokümanıdır.
- Böylelikle XML'in getirdiği tüm kazanımlara sahiptir.
- XML ayrıştırıcısı XML Schema tanımını işleyebilir
- Schema tanımında kök eleman <schema>'dır
- XML Schema için isim uzayı

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    …
</xsd:schema>
```

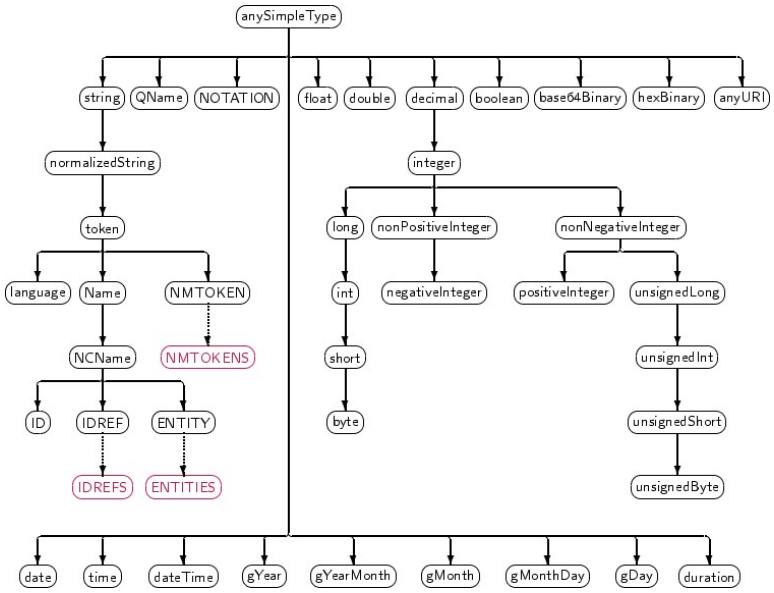
element

- XML dokümanında kullanılabilecek yeni bir takı tanımlamak için kullanılır
- Her eleman bir isme ve tipe sahip olmalıdır.

Örnek: <element>

> <xsd:element name="soyad" type="xsd:string"/>
> <xsd:element name="yaş" type="xsd:integer"/>
> <xsd:element name="doğum-tarihi"
type="xsd:date"/>
> <xsd:element name="renk" type="xsd:string"
default="kırmızı"/>

XML Schema'da Temel Tipler



attribute

- öznitelik tanımlamak için kullanılır
- <element> gibi her öznitelik bir isme ve tipe sahip olmalıdır.

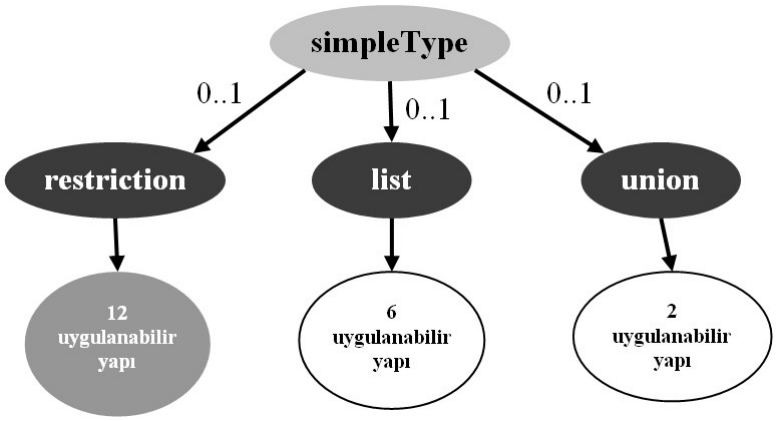
Örnek: <attribute>

```
<xsd:attribute name="dil" type="xsd:string"/>
<xsd:attribute name="dil" type="xsd:string"
                                use="optional"/>
<xsd:attribute name="dil" type="xsd:string"
                                 use="required"/>
```

XML Schema'da Basit Tip Tanımlamak

> <restriction>
> <list>
> <union>

<simpleType>



<restriction>

▶ Örnek #1

[0,100] aralığından değer almak üzere bir yaş tipi

```
<xsd:simpleType name="yasTipi">
   <xsd:restriction base="xsd:integer">
         <xsd:minInclusive value="0"/>
         <xsd:maxInclusive value="100"/>
    </xsd:restriction>
</xsd:simpleType>
```

<restriction>

▶ Örnek #2

"*Audi*", "*Golf*", "*BMW*" markalarından sadece birini alabilecek bir araç tipi

```
<xsd:simpleType name="aracTipi">
     <xsd:restriction base="xsd:string">
          <xsd:enumeration value="Audi"/>
          <xsd:enumeration value="Golf"/>
          <xsd:enumeration value="BMW"/>
      </xsd:restriction>
</xsd:simpleType>
```

<restriction>

- Örnek #3

  En fazla 8 karakterden oluşan parola tipi

```
<xsd:simpleType name=“parola">
    <xsd:restriction base="xsd:string">
        <xsd:maxLength value=“8"/>
    </xsd:restriction>
</xsd:simpleType>
```

<restriction> ile kullanılabilecek örüntüler

enumeration	Kabul edilebilir değerlerin listesi
fractionDigits	**Maksimum basamak sayısı**
length	Karakter sayısı ya da listedeki eleman sayısı
maxExclusive	**Sayısal değerler için üst sınır (sınır dahil değil)**
maxInclusive	Sayısal değerler için üst sınır (sınır dahil)
maxLength	**Maksimum karakter sayısı ya da listedeki maksimum eleman sayısı**
minExclusive	Sayısal değerler için alt sınır (sınır dahil değil)
minInclusive	**Sayısal değerler için alt sınır (sınır dahil)**
minLength	Minimum karakter sayısı ya da listedeki minimum eleman sayısı
totalDigits	**Kesin basamak sayısı**
whiteSpace	Boşluk karakterlerinin nasıl ele alınması gerektiğini tanımlar
pattern	**Düzenli ifade tanımı içerir**

<list>

- itemType özniteliği listedeki elemanların tipini tanımlar.
- Bu tipin bölünemez atomik bir tip olması gerekir.

Örnek

```
<xsd:simpleType name="sayisalTamsayi">
   <xsd:restriction base="xsd:integer">
         <xsd:minInclusive value="1"/>
         <xsd:maxInclusive value="49"/>
    </xsd:restriction>
</xsd:simpleType>
<xs:element name="sayısal" type="sayısalLoto">
   <xs:simpleType name="sayısalLoto ">
      <xs:list itemType=" sayisalTamsayi "/>
   </xs:simpleType>
</xs:element>
```

```
<sayısal>4 8 15 16 23 42 </sayısal>
```

<union>

> **memberTypes** özniteliği elemanın alabileceği değerlerin tiplerini tanımlar.

Örnek

```
<xs:element name="jeansSize">
    <xs:simpleType>
      <xs:union
        memberTypes="sizeByNo sizeByString“
       />
 </xs:simpleType>
</xs:element>
<jeansSize>36</jeansSize>
<jeansSize>small</jeansSize>
```

Örneğin devamı

```
<xs:simpleType name="sizeByNo">
    <xs:restriction base="xs:positiveInteger">
      <xs:maxInclusive value="42"/>
    </xs:restriction>
</xs:simpleType>
```

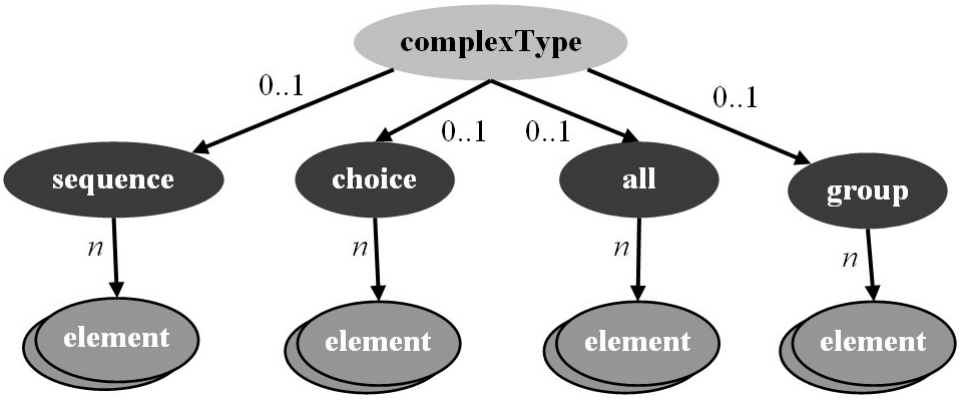
Örneğin devamı

```
<xsd:simpleType name="sizeByString">
    <xsd:restriction base="xsd:string">
          <xsd:enumeration value="small"/>
          <xsd:enumeration value="medium"/>
          <xsd:enumeration value="large"/>
     </xsd:restriction>
</xsd:simpleType>
```

XML Schema'da Karmaşık Tip Tanımlamak

- <sequence>
- <choice>
- <all>
- <group>

<complexType>



<sequence>

- Eğer elemanlar belirli bir sırada gözükmesi gerekiyor ise bu elemanlar <sequence> ile tanımlanır
- Tanımlama sırası elemanların sırasını da belirler

Örnek

```
<xsd:complexType name="müşteriTipi">
    <xsd:sequence>
        <xsd:element name="ad" type="xsd:string"/>
        <xsd:element name="soyad" type="xsd:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:element name="müşteri" type="müşteriTipi"/>
<müşteri>
    <ad></ad>
    <soyad></soyad>
</müşteri>
```

<choice>

> <choice> ile tanımlanmış eleman sıralanan elemanlardan sadece birini öğe olarak içerebilir

Örnek

```
<xsd:element name="personel">
    <xsd:complexType>
        <xsd:choice>
            <xsd:element name="müdür" type="çalışan"/>
            <xsd:element name="öğretmen" type="çalışan"/>
            <xsd:element name="memur" type="çalışan"/>
        </xsd:choice>
    </xsd:complexType>
</xsd:element>
```

<all>

> <all> ile tanımlanmış eleman verilen elemanları herhangi bir kural tanımlamaksızın değişik sıralarda öğe olarak içerebilir

<group>

> <sequence>, <choice>, <all> ile tanımlanan elemanları gruplamak ve daha sonra başka bir tanımda referans vermek için kullanılır

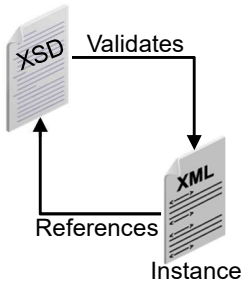
Örnek

```
<xsd:group name="personelGrubu">
     <xs:sequence>
          <xsd:element name="ad" type="xsd:string"/>
          <xsd:element name="soyad" type="xsd:string"/>
          <xsd:element name="doğumTarihi" type="xsd:date"/>
     </xs:sequence>
</xsd:group>
 <xsd:complexType name="personelBilgisi">
    <xsd:sequence>
        <xsd:group ref="personelGrubu"/>
        <xsd:element name="ülke" type="xs:string"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:element name="personel" type="personelBilgisi"/>
```

What Is an XML Schema?

> XML Schema

- Is an XML language that defines and validates the structure of XML documents
- Is stored in an XML Schema Document (XSD)
- Defines components, such as:
  - Simple types definitions
  - Complex type definitions
  - Element declarations
  - Attribute declarations



- Supports XML Namespaces, and built-in, simple, and complex data types

The Benefits of XML Schemas

> XML Schemas:
  - Unify both document and data modeling
  - Validate XML documents
  - Are created using XML
  - Support the Namespace Recommendation
  - Allow validation of text elements content based on built-in or user-defined data types
  - Allow modeling of object inheritance and type substitution
  - Allow easy creation of complex and reusable content models

An Example XML Schema Document

> The simple XML Schema uses:
  - A required XML Namespace string, with an `xs` prefix, `http://www.w3.org/2001/XMLSchema`
  - The `<schema>` element as its document root
  - The `<element>` element to declare an element

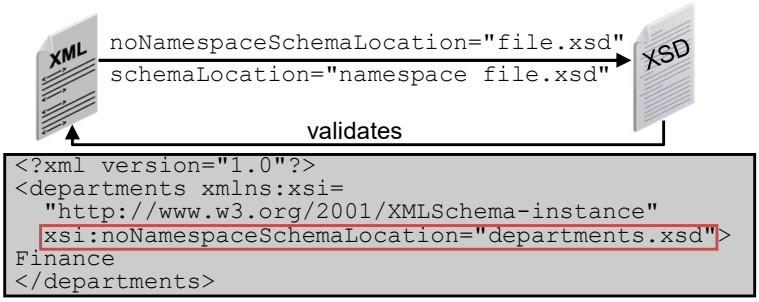
```
<?xml version="1.0"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="departments" type="xs:string"/>
</xs:schema>
```

> A valid XML instance document:

```
<?xml version="1.0"?>
<!-- The element cannot contain child elements -->
<departments>
Finance
</departments>
```

Validating an XML Document with an XML Schema Document

> In the XML instance document, use the `http://www.w3.org/2001/XMLSchema-instance` XML namespace and reference the XML Schema using:
  - The `noNamespaceSchemaLocation` attribute, or
  - The `schemaLocation` attribute




```
<?xml version="1.0"?>
<departments xmlns:xsi=
  "http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="departments.xsd">
Finance
</departments>
```

Referencing an XML Schema with the `schemaLocation` Attribute

> The XML Schema defines the `targetNamespace` value.

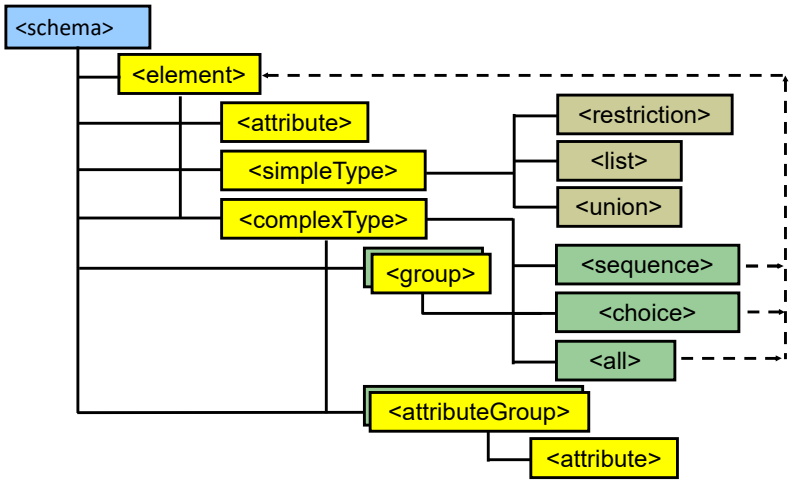
```
<?xml version="1.0"?> <!-- departments.xsd -->
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.hr.com/departments">
 <xs:element name="departments" type="xs:string"/>
</xs:schema>
```

> The XML document references the `targetNamespace` in the `schemaLocation` and the default namespace.

```
<?xml version="1.0"?>  <!-- XML document -->
<departments xmlns="http://www.hr.com/departments"
 xmlns:xsi=
  "http://www.w3.org/2001/XMLschema-instance"
 xsi:schemaLocation=
  "http://www.hr.com/departments departments.xsd">
Finance
</departments>
```

Components of an XML Schema

> Partial component hierarchy



Example of XML Schema Components

```
<?xml version="1.0"?>                                    1
<xsd:schema                                              2
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="region">                            3
    <xsd:complexType>                                    4
      <xsd:sequence>                                     5
        <xsd:element name="region_id"                    6
                     type="xsd:int"/>
        <xsd:element name="region_name"
                     type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

> XML document

```
<?xml version="1.0"?>
<region xmlns:xsi="http://...">
  <region_id>1</region_id>
  <region_name>Europe</region_name>
</region>
```

The <schema> Declaration

- Is the root element in an XML Schema document
- Contains:
  - Namespace information
  - Defaults
  - Version

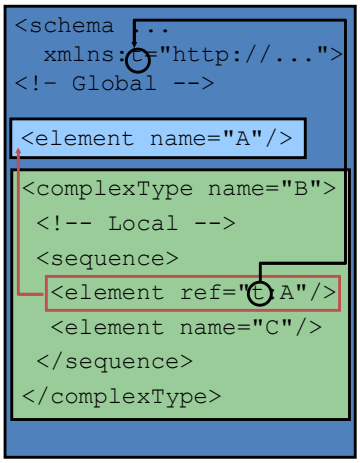
```
<schema targetNamespace="URI"
  attributeFormDefault="qualified"|"unqualified"
  elementFormDefault="qualified"|"unqualified"
  version="version number">
```

- Example:

```
<?xml version="1.0"?>
<xsd:schema
   xmlns:xsd="http://www.w3.org/2001/XMLschema">
...
```

Global and Local Declarations

- Global declarations:
  - Appear as direct children of the <schema> element
  - Can be reused within the XML Schema document
- Local declarations:
  - Are valid in the context they are defined
  - Are not children of the <schema> element
  - Can reference global type declarations by using a namespace prefix



Declaring an Element

> Declare a simple `<element>` with:
  - A `name` attribute to specify the tag name
  - A `type` attribute to specify the content allowed

```
<xsd:element name="first_name" type="xsd:string"/>
<first_name>Steven</first_name>      <!-- XML -->
```

> Full syntax:

```
<element
  name="name-of-element"
  type="global-type | built-in-type"
  ref="global-element-name"
  form="qualified | unqualified"
  minOccurs="non-negative number"
  maxOccurs="non-negative number | unbounded"
  default="default-value"
  fixed="fixed-value">
```

Built-in XML Schema Data Types

> The XML Schema Language provides built-in data types, such as:
  - The `string` type
  - The `int` type
  - The `decimal` type
  - Many others (`boolean`, `float`, and so on)

> Examples:

```
<xsd:element name="employee_id" type="xsd:int"/>
<employee_id>100</employee_id>      <!-- XML -->
```

```
<xsd:element name="salary" type="xsd:decimal"/>
<salary>1000.50</salary>            <!-- XML -->
```

Declaring a <simpleType> Component

- The <simpleType>:
  - Is a derived type extending built-in or other types
  - Provides three primary derived types:
    - A <restriction>
    - A <list>
    - A <union>
  - Has facets (properties), such as maxInclusive

```
<xsd:simpleType name="empid">
  <xsd:restriction base="xsd:positiveInteger">
    <!- for positiveInteger the minimum is 1 -->
    <xsd:maxInclusive value="1000"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="employee_id" type="empid"/>
```

Using <list> and <union> SimpleTypes

- Declaring a <list>:

```
<xsd:element name="email">
  <xsd:simpleType>
    <xsd:list itemType="xsd:string"/>
  </xsd:simpleType>
</xsd:element>
```

```
<email>SKING sking@hr.com</email>
```

- Declaring a <union>:

```
<xsd:simpleType name="mgrTypes">…</xsd:simpleType>
<xsd:simpleType name="repTypes">…</xsd:simpleType>
...
<xsd:element name="job_id">
  <xsd:simpleType>
    <xsd:union memberTypes="mgrTypes repTypes">
  </xsd:simpleType>
</xsd:element>
```

Declaring <complexType> Components

> The <complexType> declaration:

```
<xsd:complexType name="..." mixed="true | false">
  ...
</xsd:complexType>
```

– Must be identified by a name attribute if it is global; otherwise, it is an anonymous complex type

– Provides a content model that can contain:

- Simple content
- A <sequence> declaration
- A <choice> declaration
- A reference to a global <group>
- An <all> declaration

– Can allow mixed or empty content

Declaring a <sequence>

> Defines an ordered sequence of elements

> Must be contained within a <complexType>

```
<xsd:element name="department">
  <xsd:complexType>
    <xsd:sequence>
       <xsd:element name="department_id"
                    type="xsd:int"/>
       <xsd:element name="department_name"
                    type="xsd:string"/>
       <xsd:element name="manager_id"
                    type="xsd:int" minOccurs="0"/>
       <xsd:element name="location_id"
                    type="xsd:int" minOccurs="0"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Declaring a <choice>

- Defines a choice of alternative elements
- Must also be contained within a <complexType>

```
<xsd:complexType name="employeeType">
  <xsd:choice>
     <xsd:element name="full_time"
                  type="xsd:string"/>
     <xsd:element name="part_time"
                  type="xsd:string"/>
  </xsd:choice>
</xsd:complexType>
<xsd:element name="employee">
 <xsd:complexType>
   <xsd:sequence>
     <xsd:element name="full_name"
                  type="xsd:string"/>
     <xsd:element name="contract"
                  type="employeeType"/>
   </xsd:sequence>
 </xsd:complexType>
</xsd:element>
```

Declaring an Empty Element

- Using a <complexType> declaration without any elements or a content model
- Typically contains attributes

```
<xsd:element name="departments">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="department"
                   maxOccurs="unbounded"/>
        <xsd:complexType>
          <xsd:attribute name="department_id"
                         type="xsd:int"/>
          <xsd:attribute name="department_name"
                         type="xsd:string"/>
        <xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Using Element Wildcards

> Element wildcard declarations:
  – Allow including elements without much control
  – Provide a way to include elements from:
    • Within the XML Schema and its namespace
    • Another namespace
  – Are declared using `<any>`:

```
<xsd:complexType name="commentType">
  <xsd:sequence>
    <xsd:element name="author"/>
    <xsd:any namespace="##any"
             processContents="lax"
             minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:element name="comments" type="commentType"/>
```

Declaring Attributes

> Declare an `<attribute>`:
  – Identified by the `name` attribute
  – With the `type` attribute restricted to built-in or user-defined simple types

```
<xsd:attribute name="department_id"
               type="xsd:string"/>
```

> Full syntax:

```
<attribute
  name="name-of-attribute"
  type="global-type | built-in-type"
  ref="global-attribute-declaration"
  form="qualified | unqualified"
  used="optional | prohibited | required"
  default="default-value"
  fixed="fixed-value">
```

Attribute Declarations: Example

1. Based on a <simpleType>:

```
<xsd:attribute name="department_id">
  <xsd:simpleType>
    <xsd:restriction base="xsd:positiveInteger">
      <xsd:maxInclusive value="100"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

2. With a default value:

```
<xsd:attribute name="department_id"
               type="xsd:int" default="10"/>
```

3. Referencing a global type:

```
<xsd:attribute ref="department_id"/>
```

Declaring and Referencing an <attributeGroup>

> Declare an <attributeGroup>:
  – Identified by the name attribute
  – Containing one or more <attribute>s, or references to <attributeGroup>s

```
<xsd:attributeGroup name="departmentGroup">
  <xsd:attribute name="name" type="xsd:string"/>
  <xsd:attribute name="id" type="xsd:int"/>
</xsd:attributeGroup>
```

> Reference in a <complexType> or <attributeGroup>:

```
<xsd:element name="department">
  <xsd:complexType>
    <xsd:attributeGroup ref="departmentGroup"/>
  </xsd:complexType>
</xsd:element>
```

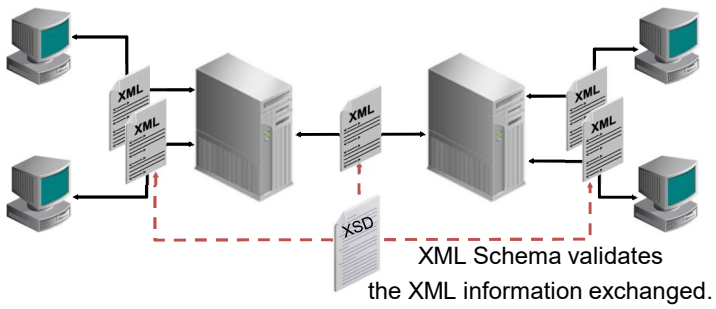
Documenting the XML Schema

- Document the XML Schema by using:
  - XML comments
  - Attributes from other namespaces
  - An `<annotation>` declaration:
    - Provides an `<appinfo>` element
    - Contains a `<documentation>` element
    - Can be included within all other component types

```
<xsd:annotation>
  <xsd:appinfo source="file:///e:/labs/anno.xml"/>
  <xsd:documentation>
  A <b>big</b> comment about this Schema
  </xsd:documentation>
</xsd:annotation>
```

Applications for XML Schema

- E-commerce
- Web publications and syndication
- Enterprise Application Integration (EAI)
- Process control and data acquisition



XML Schema validates the XML information exchanged.

XML Schema Versus DTD

- XML Schema:
  - Is more powerful and flexible than a DTD
  - Provides better namespace support than a DTD
  - Is written in XML syntax
  - Is extensible
  - Provides data type support
- DTD:
  - Provides `ENTITY` functionality that is not supported by XML Schema
  - Can be embedded in an XML document
  - Is written in SGML

Converting a DTD to an XML Schema

> XML document:

```
<employee>
   <employee_id>120</employee_id>
   <last_name>Weiss</last_name>
</employee>
```

> DTD:

```
<!ELEMENT employee (employee_id, last_name)>
<!ELEMENT employee_id (#PCDATA)>
<!ELEMENT last_name (#PCDATA)>
```

> XML Schema:

```
<xsd:complexType name="employeeType">
 <xsd:sequence>
 <xsd:element name="employee_id" type="xsd:int"/>
 <xsd:element name="last_name" type="xsd:string"/>
 </xsd:sequence>
</xsd:complexType>
<xsd:element name="employee" type="employeeType"/>
```

PARSING XML IN PYTHON

XML Parser Architectures and APIs

> The Python provides a minimal but useful set of interfaces to work with XML.

> The two most basic and broadly used APIs to XML data are the SAX and DOM interfaces.

– Simple API for XML (SAX)

- You register callbacks for events of interest and then let the parser proceed through the document.
- This is useful when your documents are large or you have memory limitations, it parses the file as it reads it from disk and the entire file is never stored in memory.

– Document Object Model (DOM) API

- This is a World Wide Web Consortium recommendation wherein the entire file is read into memory and stored in a hierarchical (tree-based) form to represent all the features of an XML document.

SAX Overview

- SAX parsers parse XML documents in a serial fashion, based on event handling:
  - As various types of information are encountered during parsing (such as start tags, end tags, and character data), specific methods are called.
  - All such methods in SAX are part of some handler.

When to use SAX

- You need to scan large XML documents.
  - Scan huge files with little overhead
- You need more control over the parsing process.
  - SAX does not actually do much.
  - Your program decides how anything is used, if at all.
- You have limited system resources.
  - SAX does not create many objects.
- You need high performance.
  - SAX quickly locates elements within any size XML document.

History of SAX

- SAX is considered a de facto standard and not endorsed by any standards body.
- SAX was created by David Megginson
- SAX1
  - Released in May 1998
- SAX2
  - Released on May 5, 2000
  - Provides additional support for:
    - Namespaces
    - Querying and setting parser features
    - Filter chains
  - Deprecates several SAX1 APIs

SAX Language Options

- Originally written for Java
- No official bindings for languages other than Java
- Other languages available for SAX:
  - Perl
  - Python
  - C++
  - COM (Microsoft Visual Basic, Delphi, and so on)
- Fully supported in Xerces
  - Xerces supports both SAX 1 and SAX 2.

Parsing XML with SAX APIs

```
<?xml version="1.0" encoding="UTF-8"?>
<countries>
    <country>
        <Code>NLD</Code>
        <Name>Netherlands</Name>
        <Continent>Europe</Continent>
        <Region>Western Europe</Region>
        <SurfaceArea>41526.00</SurfaceArea>
        <IndepYear>1581</IndepYear>
        <Population>15864000</Population>
        <LifeExpectancy>78.3</LifeExpectancy>
        <GNP>371362.00</GNP>
        <GNPOld>360478.00</GNPOld>
        <LocalName>Nederland</LocalName>
        <GovernmentForm>Constitutional Monarchy</GovernmentForm>
        <HeadOfState>Beatrix</HeadOfState>
        <Capital>5</Capital>
        <Code2>NL</Code2>
    </country>
    . . .
</countries>
```

Parsing XML with SAX APIs

```
class Country:
    def __init__(self):
        self.code = ""
        self.name = ""
        self.population = ""
        self.continent = ""
        self.gnp = ""
        self.surface_area = ""
        self.independence_year = ""

    def __str__(self):
        return f"Country[code={self.code}, name={self.name}, population={self.population}," \
               f" continent= {self.continent}," \
               f" surfaceArea={self.surface_area}, gnp= {self.gnp}]"
```

```
class CountryHandler(xml.sax.ContentHandler):
    def __init__(self):
        self.countries = []
        self.country = Country()
        self.opening_tag = ""
        self.property_tag_names = {
            "GNP": "gnp",
            "Population": "population",
            "SurfaceArea": "surface_area",
            "Name": "name",
            "Code": "code",
            "Continent": "continent"
        }

    def startElement(self, tag, attributes):
        self.opening_tag = tag
```

Parsing XML with SAX APIs

```
    def characters(self, content):
        if self.opening_tag in self.property_tag_names.keys():
            prop = self.property_tag_names[self.opening_tag]
            setattr(self.country, prop, content)

    def endElement(self, tag):
        self.opening_tag = ""
        if tag == "country":
            self.countries.append(self.country)
            self.country = Country()
```

Parsing XML with SAX APIs

```
if (__name__ == "__main__"):
    # create an XMLReader
    parser = xml.sax.make_parser()
    # turn off namepsaces
    parser.setFeature(xml.sax.handler.feature_namespaces, 0)

    # override the default ContextHandler
    handler = CountryHandler()
    parser.setContentHandler(handler)

    parser.parse("resources/countries.xml")
    for country in handler.countries:
        print(country)
```

DOM Overview

- **Document Object Model** (DOM) is a W3C recommendation.
- DOM represents the tag structure in XML documents with an ***in-memory tree structure***.
- DOM enables **random access** of XML contents.
- DOM lets you add, remove, and modify nodes in the tree structure.
- DOMLevel 2 ***does not specify*** how reading and writing of XML documents is performed:
  - JAXP provides support for input.
  - The **`XmlDocument.write`** method of the reference implementation provides output.

History of DOM

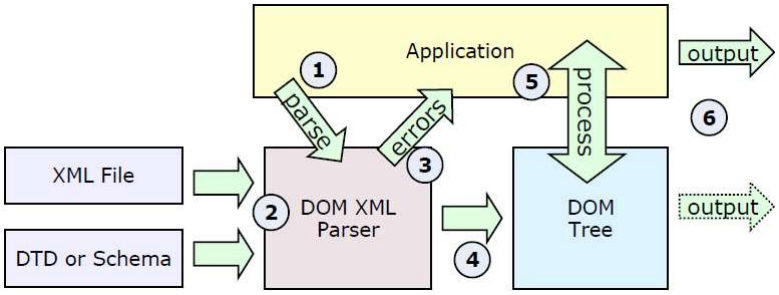
- DOM originally grew out of JavaScript to represent HTML
  - Dynamic HTML is its direct ancestor.
  - Later formalized for use in XML
- DOM Level 1
  - W3C recommendation - October, 1998
  - Defines core interface support for XML and HTML.
- DOM Level 2
  - W3C recommendation - November, 2000
  - Added support for namespaces, CSS and events
- DOM Level 3–W3C recommendation - April, 2004
  - Enhanced support for namespaces, XPath addressing, abstract DTD and schema support, and new support for serialization and validation

When to use DOM

- You need to make multiple processing passes over an XML document
- No need to rescan the document on each pass
- You need to merge several XML documents together
  - Structures can be the same or different
- You need to do a lot with document content
  - Sharing content with other processes
  - Frequent access of content
- You need to modify document…
  - Content
  - StructureOrder

DOM Processing Overview

1. Application creates a DOM parser.
2. Parser reads in XML and any vocabulary.
3. Parser returns any errors to the application.
4. Parser generates a DOM tree.
5. Application can read, write, or update the DOM tree.
6. Application writes final XML to output.



Parsing XML with DOM APIs

> Document Object Model ("DOM") is a cross-language API from the World Wide Web Consortium (W3C) for accessing and modifying XML documents.

> The DOM is extremely useful for random-access applications.

– SAX only allows you a view of one bit of the document at a time.

– If you are looking at one SAX element, you have no access to another.

> Here is the easiest way to quickly load an XML document and to create a **`minidom`** object using the **`xml.dom`** module.

> The **`minidom`** object provides a simple parser method that quickly creates a DOM tree from the XML file.

Parsing XML with DOM APIs

```
from xml.dom import minidom

# Open XML document using minidom parser
DOMTree = minidom.parse("resources/countries.xml")
collection = DOMTree.documentElement
# Get all the countries in the collection
countries = collection.getElementsByTagName("country")
for country in countries:
  name = country.getElementsByTagName("Name")[0].childNodes[0].data
  continent = country.getElementsByTagName("Continent")[0].childNodes[0].data
  code = country.getElementsByTagName("Code")[0].childNodes[0].data
  population = int(country.getElementsByTagName("Population")[0].childNodes[0].data)
  surfaceArea = float(country.getElementsByTagName("SurfaceArea")[0]
                      .childNodes[0].data)
   print("%3s %-48s %-12s %12d %12.1f" %
                      (code,name,continent,population,surfaceArea) )
```