Unit 3

Javascript Data Types

*JavaScript provides different **data types** to hold different types of values. There are two types of data types in JavaScript.*

1. *Primitive data type*
2. *Non-primitive (reference) data type*

*JavaScript is a **dynamic type language**, means you don't need to specify type of the variable because it is dynamically used by JavaScript engine. You need to use **var** here to specify the data type. It can hold any type of values such as numbers, strings etc.*

JavaScript Data Types

Primitive data type

Data Types	Description	Example
String	represents textual data	'hello', "hello world!" etc
Number	an integer or a floating-point number	3, 3.234, 3e-2 etc.
BigInt	an integer with arbitrary precision	900719925124740999n , 1n etc.
Boolean	Any of two values: true or false	true and false
undefined	a data type whose variable is not initialized	let a;
null	denotes a null value	let a = null;
Symbol	data type whose instances are unique and immutable	let value = Symbol('hello');

Example

```
let a=12345
let b=null
let c= true
```

```
let d=BigInt("12456677888888888888")
let e= "senam"
let f=Symbol("fsd")
console.log(a,b,c,d,e,f);
```

Difference between let,var and const

KEYWORD	SCOPE	REDECLARATION & REASSIGNMENT	HOISTING
var	Global, Local	yes & yes	yes, with default value
let	Global, Local, Block	no & yes	yes, without default value
const	Global, Local, Block	no & no	yes, without default value

Questions

1. create a var of some type and try to add other vaar to it. and use typeof operator to find datatype of all.
2. write js to create a word meaning to 5 words.

Non Primitive Data Type	**Description**
Object	represents instance through which we can access members
Array	represents group of similar values
RegExp	represents regular expression

operator

There are different types of JavaScript operators:

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- String Operators
- Logical Operators
- Bitwise Operators
- Ternary Operators
- Type Operators

Arithmetic Operators are used to perform arithmetic on numbers:

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
**	Exponentiation (ES2016)
/	Division
%	Modulus (Division Remainder)

++	Increment
--	Decrement

Example

```
let a=46
let b=4
console.log("a+b = ",a+b);
console.log("a-b = ",a-b);
console.log("a*b = ",a*b);
console.log("a/b = ",a/b);
console.log("a**b = ",a**b);
console.log("a%b = ",a%b);
a=4
console.log("++a -a++ = ",++a -a++);
console.log("--a -a-- = ",--a -a--);
```

Arrays

JavaScript array is an object that represents a collection of similar type of elements.

There are 3 ways to construct array in JavaScript

1. By array literal
2. By creating instance of Array directly (using new keyword)
3. By using an Array constructor (using new keyword)

Example

```
let arr=[1,2,3]
console.log(arr[2])
//another way
let arr1=new Array()
arr1[0]=45.9
arr1[1]="b"
arr1[2]=1
```

```
arr1[3]="d"
console.log(arr1[0])
//empty array
let arr2=new Array()
console.log(arr)
//2d array
let arr3=new Array(1,2,4)
console.log(arr3[0][0])
//push array
let arr4=[[1,2,3],[4,5,6]]
console.log(arr4[0][1])
arr3.push(299,300,400)
console.log(arr3)
arr3.unshift(1,2,3,4)
console.log(arr3)
//to add add element in between
arr3.splice(1,1,899)
console.log(arr3);

arr5=["A","c","d"]
let y=arr5.splice(1,1,"b")
console.log(arr3)
console.log(y)
```

Creating Objects in JavaScript

There are 3 ways to create objects.

1. By object literal
2. By creating instance of Object directly (using new keyword)
3. By using an object constructor (using new keyword)

example

```
let employee={}//simple object without value
console.log(typeof employee);//anything which is not have primitive type
is an object.
let comp1={
    ram:'4 gb',
    storage:'500',
```

```
    brand:"dell",
    'comp name':"lappy",
}
let input='brand';
console.log(comp1);//whole object
console.log(comp1.storage);//particular value
console.log(comp1['brand']);//another way of particular value
console.log(comp1['comp name']);//without_
console.log(comp1[input]);
```

We can add any number of element in object. like Array, objects, primitive data, function.

JavaScript Function Syntax

A JavaScript function is defined with the `function` keyword, followed by a name, followed by parentheses ().

Function names can contain letters, digits, underscores, and dollar signs (same rules as variables).

The parentheses may include parameter names separated by commas: (*parameter1, parameter2, ...*)

The code to be executed, by the function, is placed inside curly brackets: {}

```
function name(parameter1, parameter2, parameter3) {

  // code to be executed

}
```

Example

```
function greet()// A simple function is like this
{
    console.log("hello")
}
greet();//calling of func
function greet1(user)// return whatever entered by user
{
   return `hii ${user}`

}
console.log(greet1("senam"))//print directly by console with para
let name="abc";
console.log(greet1(name)); //print by var

//let str=greet();//print hello when get initialize
function add(x,y)
{
    return "hiii"
    return x+y
}// cant' return 2 values only one
let z=add(3,4)
console.log("add is " +z)

//we can alo use function with var

 let multiply=function (x,y)
{
    return x*y
}
console.log(multiply(3,2))
m=multiply
console.log(m(2,2))
```

Constructor in JavaScript

A constructor is a special function that creates and initializes an object instance of a class. In JavaScript, a constructor gets called when an object is created using the new keyword.

The purpose of a constructor is to create a new object and set values for any existing object properties.

When a constructor gets invoked in JavaScript, the following sequence of operations take place:

- A new empty object gets created.
- The this keyword begins to refer to the new object and it becomes the current instance object.
- The new object is then returned as the return value of the constructor.

Built-in Constructors

JavaScript has some built-in constructors, including the following:

var a = new Object();

var b = new String();

var c = new String('Bob')

var d = new Number();

var e = new Number(25);

var f = new Boolean();

var g = new Boolean(true);

Example

```
function cons()// simple constructor
{
  console.log("empty");
}
let c=new cons()//initialization
function cons1(name)//1 para constructor
{
  this.name=name
 return name
}
let c1=new cons1("senam")
console.log(c1);//whole c1 info
console.log(c1.name);//specific value detail print
```

constructor function

```
function comp(ram,cpu,brand)
{
    this.ram=ram;
    this.cpu=cpu;
    this.brand=brand;
    this.func=function(){
```

```
 console.log("constructor function");
}
}
let comp1=new comp(2,"dualcore","hp");
let comp2=new comp(4,"i7","dell");
console.log(comp1);
comp1.func();
```

JavaScript Loops

The **JavaScript loops** are used *to iterate the piece of code* using for, while, do while or for-in loops. It makes the code compact. It is mostly used in array.

There are four types of loops in JavaScript.

1. for loop
2. foreach
3. for-in loop
4. for-of-loop
5. While
6. Do while

```

let comp2={
    arr:[1,2,3,"hh"],
    ram:'4 gb',
    storage:'500',
    brand:"dell",
    'comp name':"lappy",
    owner:{
        name:"senam",
        profession:"AP",
        Qualification:"m.tech"
    }

}
//console.log(Object.keys(comp2));//print all keys
//console.log(Object.keys(comp2)[0]);//print first key
//console.log(Object.values(comp2)[0]);//print first value
```

```
/* for (let i = 0; i < Object.keys(comp2).length; i++) {
    console.log("the key "+ Object.keys(comp2)[i] +" has value "+
Object.values(comp2)[i]);
    console.log("the key "+ Object.keys(comp2)[i] +" has value "+
comp2[Object.keys(comp2)[i]]);

}//will printboth object and values */
/* for (const key in comp2) {
     console.log(key +" has value " + comp2[key]);

} */
/* for (const i of comp2.arr) {
    console.log(i+1);


} */
comp2.arr.forEach(i => {
    console.log(i*i);

});
```

Write program for comparison

```
let comp1={
    ram: 4,
    storage:'500',
    brand:"dell",
    'comp name':"lappy",
    compare:function(lap)
    {
        if(comp1.ram<lap.ram)//this.ram<lap.ram
        //we can also do this with the help of this keyword.
        {
            console.log(comp1.ram+ "gb is of comp1");
        }
        else
        {
            console.log(comp2.ram+ "gb is of comp2");
        }
    }
```

```
}
let comp2={
    ram:2,
    storage:'500',
    brand:"dell",
    'comp name':"lappy",
}
comp1.compare(comp2);//this is how two comp compare by themselves
```

pop up boxes

```
alert("hiii")

document.write("hiiiii")

let x=prompt("enter your age here")

document.write("your age is" + x)

confirm("you enter age "+x+" eligible for")

console.log(document);

//pause screen display untill window closes

//we cannot modify these

//and also not fix from brower where it come from
```

Window Object

The window object represents a window in browser. An object of window is created automatically by the browser.

Window is the object of browser, it is not the object of javascript. The javascript objects are string, array, date etc.

Note: if html document contains frame or iframe, browser creates additional window objects for each frame

The important methods of window object are as follows:

Method	*Description*
alert()	*displays the alert box containing message with ok button.*
confirm()	*displays the confirm dialog box containing message with ok and cancel button.*
prompt()	*displays a dialog box to get input from the user.*
open()	*opens the new window.*
close()	*closes the current window.*
setTimeout()	*performs action after specified time like calling function, evaluating expressions etc.*

Console

```
countReset: ƒ countReset()
createTask: ƒ createTask()
debug: ƒ debug()
dir: ƒ dir()
dirxml: ƒ dirxml()
error: ƒ error()
group: ƒ group()
groupCollapsed: ƒ groupCollapsed()
groupEnd: ƒ groupEnd()
info: ƒ info()
log: ƒ log()
memory: MemoryInfo {totalJSHeapSize: 10000000, usedJSHeapSize: 10000000,
jsHeapSizeLimit: 2190000000}
profile: ƒ profile()
profileEnd: ƒ profileEnd()
table: ƒ table()
time: ƒ time()
timeEnd: ƒ timeEnd()
timeLog: ƒ timeLog()
timeStamp: ƒ timeStamp()
trace: ƒ trace()
warn: ƒ warn()
Symbol(Symbol.toStringTag): "console"
```

```
    [[Prototype]]: Object
```

Example

```
alert("hiii")
document.write("hiiiii")
let x=prompt("enter your age here")
document.write("your age is" + x)
confirm("you enter age "+x+" eligible for")
console.log(document);
```

Javascript Date

The Javascript Date object in JavaScript is used to represent a moment in time. This time value is since 1 January 1970 UTC (Coordinated Universal Time). We can create a date using the Date object by calling the new Date() constructor as shown in the below syntax.

Syntax:

new Date();

new Date(value);

new Date(dateString);

new Date(year, month, day, hours, minutes, seconds, milliseconds);

Parameters: All of the parameters as shown in the above syntax are described below:

- value: This value is the number of milliseconds since January 1, 1970, 00:00:00 UTC.
- dateString: This represents a date format.
- year: This is represented by integer values ranging from the years 1900 to 1999.
- month: This is represented by integer values ranging from 0 for January to 11 for December.
- day: This is an optional parameter. This is represented by an integer value for the day of the month.
- hours: This is optional. This is represented by an integer value for the hour of the day.
- minutes: This is optional. This is represented by an integer value for the minute of a time.
- seconds: This is optional. This is represented by an integer value for the second of a time.
- milliseconds: This is optional. This is represented by an integer value for the millisecond of a time.

Return Values: It returns the present date and time if nothing as the parameter is given otherwise it returns the date format and time in which the parameter is given.

Array Method

Array length	Array join()
Array toString()	Array delete()
Array pop()	Array concat()
Array push()	Array flat()
Array shift()	Array splice()
Array unshift()	Array slice()

DOM (Document Object Model)

The document object represents the whole html document.

When html document is loaded in the browser, it becomes a document object. It is the root element that represents the html document. It has properties and methods. By the help of document object, we can add dynamic content to our web page.

Selecting elements

Note-check link in the underlined topic for further codes and details

getElementById() – select an element by id.

getElementsByName() – select elements by name.

getElementsByTagName() – select elements by a tag name.

getElementsByClassName() – select elements by one or more class names.

querySelector() – select elements by CSS selectors.

Traversing elements

Get the parent element – get the parent node of an element.

Get child elements – get children of an element.

Get siblings of an element – get siblings of an element.

Manipulating elements

createElement() – create a new element.

appendChild() – append a node to a list of child nodes of a specified parent node.
textContent – get and set the text content of a node.
innerHTML – get and set the HTML content of an element.
innerHTML vs. createElement – explain the differences between innerHTML and createElement when it comes to creating new elements.
DocumentFragment – learn how to compose DOM nodes and insert them into the active DOM tree.
after() – insert a node after an element.
append() – insert a node after the last child node of a parent node.
prepend() – insert a node before the first child node of a parent node.
insertAdjacentHTML() – parse a text as HTML and insert the resulting nodes into the document at a specified position.
replaceChild() – replace a child element by a new element.
cloneNode() – clone an element and all of its descendants.
removeChild() – remove child elements of a node.
insertBefore() – insert a new node before an existing node as a child node of a specified parent node.
insertAfter() helper function – insert a new node after an existing node as a child node of a specified parent node.

Working with Attributes

HTML Attributes & DOM Object's Properties – understand the relationship between HTML attributes & DOM object's properties.
setAttribute() – set the value of a specified attribute on a element.
getAttribute() – get the value of an attribute on an element.
removeAttribute() – remove an attribute from a specified element.
hasAttribute() – check if an element has a specified attribute or not.

Section 6. Manipulating Element's Styles

style property – get or set inline styles of an element.
getComputedStyle() – return the computed style of an element.
className property – return a list of space-separated CSS classes.
classList property – manipulate CSS classes of an element.
Element's width & height – get the width and height of an element.

Section 7. Working with Events

JavaScript events – introduce you to the JavaScript events, the event models, and how to handle events.
Handling events – show you three ways to handle events in JavaScript.
Page Load Events – learn about the page load and unload events.
load event – walk you through the steps of handling the load event originating from the document, image, and script elements.
DOMContentLoaded – learn how to use the `DOMContentLoaded` event correctly.
beforeunload event – guide you on how to show a confirmation dialog before users leave the page.
unload event – show you how to handle the unload event that fires when the page is completely unloaded.
Mouse events – how to handle mouse events.
Keyboard events – how to deal with keyboard events.
Scroll events – how to handle scroll events effectively.
scrollIntoView – learn how to scroll an element into view.
Focus Events – cover the focus events.
haschange event – learn how to handle the event when URL hash changes.
Event Delegation – is a technique of levering event bubbling to handle events at a higher level in the DOM than the element on which the event originated.
dispatchEvent – learn how to generate an event from code and trigger it.
Custom Events – define a custom JavaScript event and attach it to an element.
MutationObserver – monitor the DOM changes and invoke a callback when the changes occur.

Section 8. Scripting Web Forms

JavaScript Form – learn how to handle form `submit` event and perform a simple validation for a web form.
Radio Button – show you how to write the JavaScript for radio buttons.
Checkbox – guide you on how to manipulate checkbox in JavaScript.
Select box – learn how to handle the select box and its option in JavaScript.
Add / Remove Options – show you how to dynamically add options to and remove options from a select box.
Removing Items from <select> element conditionally – show you how to remove items from a `<select>` element conditionally.
Handling change event – learn how to handle the change event of the input text, radio button, checkbox, and select elements.
Handling input event – handle the input event when the value of the input element changes.

note -https://www.thatjsdude.com/interview/dom.html

XML

What is XML?

XML is a software- and hardware-independent tool for storing and transporting data. XML became a W3C Recommendation as early as in February 1998.

- XML stands for eXtensible Markup Language
- XML is a markup language much like HTML
- XML was designed to store and transport data
- XML was designed to be self-descriptive
- XML is a W3C Recommendation

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- XML was designed to carry data - with focus on what data is
- HTML was designed to display data - with focus on how data looks
- XML tags are not predefined like HTML tags are

This note is a note to Tove from Jani, stored as XML:

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

The XML above is quite self-descriptive:

- It has sender information
- It has receiver information
- It has a heading
- It has a message body

But still, the XML above does not DO anything. XML is just information wrapped in tags.

Someone must write a piece of software to send, receive, store, or display it:

XML is Extensible

Most XML applications will work as expected even if new data is added (or removed).

Imagine an application designed to display the original version of note.xml (<to> <from> <heading> <body>).

Then imagine a newer version of note.xml with added <date> and <hour> elements, and a removed <heading>.

The way XML is constructed, older version of the application can still work:

```
<note>
  <date>2015-09-01</date>
  <hour>08:30</hour>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

XML Separates Data from HTML

When displaying data in HTML, you should not have to edit the HTML file when the data changes.

With XML, the data can be stored in separate XML files.

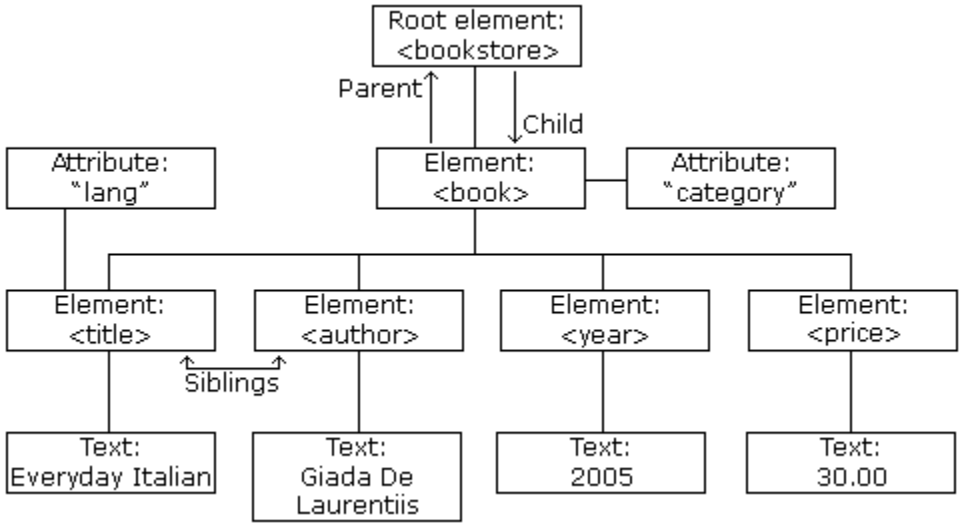
With a few lines of JavaScript code, you can read an XML file and update the data content of any HTML page.

```
<!DOCTYPE html>
<html lang="en">
<head>
</head>
<body id="D">
    <button onclick="readxml()">click </button>
    <script>
```

```
        function readxml()
{
var xmlfirst=new XMLHttpRequest();
xmlfirst.open('GET','xmlfirst.xml',false);
xmlfirst.send()
var xmldata = xmlfirst.responseXML
console.log(xmldata);
document.write("hh"+xmldata)
}
        </script>
</div>
</body>
</html>
```

XML documents form a tree structure that starts at "the root" and branches to "the leaves".

The XML Tree Structure



An Example XML Document

The image above represents books in this XML:

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

XML Parser

The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.

However, before an XML document can be accessed, it must be loaded into an XML DOM object.

All modern browsers have a built-in XML parser that can convert text into an XML DOM object.

Parsing a Text String

This example parses a text string into an XML DOM object, and extracts the info from it with JavaScript:

```
   <script>
        function readxml()
{
var xmlfirst=new XMLHttpRequest();
xmlfirst.open('GET','xmlfirst.xml',false);
xmlfirst.send()
var xmldata = xmlfirst.responseXML
//console.log(xmldata);
//document.write("hh"+xmldata)


var xmldata=(new
DOMParser()).parseFromString(xmlfirst.responseText,'text/xml');
var e=xmldata.getElementsByTagName('emp')[0]
let
age=e.getElementsByTagName("age")[0].innerHTML//.firstElementChild.data;
let age1=age.
console.log(age);
document.write(age1)
}

        </script>
```

Displaying XML with XSLT

XSLT (eXtensible Stylesheet Language Transformations) is the recommended style sheet language for XML.

XSLT is far more sophisticated than CSS. With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

XSLT uses XPath to find information in an XML document.

XSLT Example

We will use the following XML document:

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

<food>
<name>Belgian Waffles</name>
<price>$5.95</price>
<description>Two of our famous Belgian Waffles with plenty of real maple
syrup</description>
<calories>650</calories>
</food>

<food>
<name>Strawberry Belgian Waffles</name>
<price>$7.95</price>
<description>Light Belgian waffles covered with strawberries and whipped
cream</description>
<calories>900</calories>
</food>

<food>
<name>Berry-Berry Belgian Waffles</name>
<price>$8.95</price>
<description>Light Belgian waffles covered with an assortment of fresh
berries and whipped cream</description>
<calories>900</calories>
</food>

<food>
<name>French Toast</name>
<price>$4.50</price>
<description>Thick slices made from our homemade sourdough
bread</description>
<calories>600</calories>
</food>

<food>
<name>Homestyle Breakfast</name>
<price>$6.95</price>
<description>Two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>
<calories>950</calories>
</food>

</breakfast_menu>
```

Well Formed XML Documents

An XML document with correct syntax is called "Well Formed".

The syntax rules were described in the previous chapters:

- XML documents must have a root element
- XML elements must have a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML attribute values must be quoted

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

XML Errors Will Stop You

Errors in XML documents will stop your XML applications.

The W3C XML specification states that a program should stop processing an XML document if it finds an error. The reason is that XML software should be small, fast, and compatible.

HTML browsers are allowed to display HTML documents with errors (like missing end tags).

With XML, errors are not allowed.

XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

What is a DTD?

DTD stands for Document Type Definition.

A DTD defines the structure and the legal elements and attributes of an XML document.

Valid XML Documents

A "Valid" XML document is "Well Formed", as well as it conforms to the rules of a DTD:

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

The DOCTYPE declaration above contains a reference to a DTD file. The content of the DTD file is shown and explained below.

XML DTD

The purpose of a DTD is to define the structure and the legal elements and attributes of an XML document:

Note.dtd:

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

The DTD above is interpreted like this:

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

- !ELEMENT to - Defines the to element to be of type "#PCDATA"
- !ELEMENT from - Defines the from element to be of type "#PCDATA"
- !ELEMENT heading - Defines the heading element to be of type "#PCDATA"
- !ELEMENT body - Defines the body element to be of type "#PCDATA"

Tip: #PCDATA means parseable character data.

XML Schema

XML Schema is an XML-based alternative to DTD:

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

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

</xs:element>
```

The Schema above is interpreted like this:

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

---

XML Schemas are More Powerful than DTD

- XML Schemas are written in XML
- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

Why Use an XML Schema?

With XML Schema, your XML files can carry a description of its own format.

With XML Schema, independent groups of people can agree on a standard for interchanging data.

With XML Schema, you can verify data.

XML Schemas Support Data Types

One of the greatest strengths of XML Schemas is the support for data types:

- It is easier to describe document content
- It is easier to define restrictions on data
- It is easier to validate the correctness of data
- It is easier to convert data between different data types

XML Schemas use XML Syntax

Another great strength about XML Schemas is that they are written in XML:

- You don't have to learn a new language
- You can use your XML editor to edit your Schema files
- You can use your XML parser to parse your Schema files
- You can manipulate your Schemas with the XML DOM
- You can transform your Schemas with XSLT.