XML
DTD and XML Schema

Discussion Sessions 1A and 1B

Session 2

Terminology

- Data Model: general conceptual way of structuring data
  - e.g. XML
- Schema: structure of a particular database under a certain data model
  - e.g. DTD, XML Schema
- Instance: actual data conforming to a schema
  - e.g. an actual XML document

HTML and XML

- HTML
  - Simple, Text-based
- HTML is mainly for human consumption
  - HTML Tags are for formatting, not for meaning
  - XML: data representation standard with "semantic" tag

XML

eXtensible Markup Language

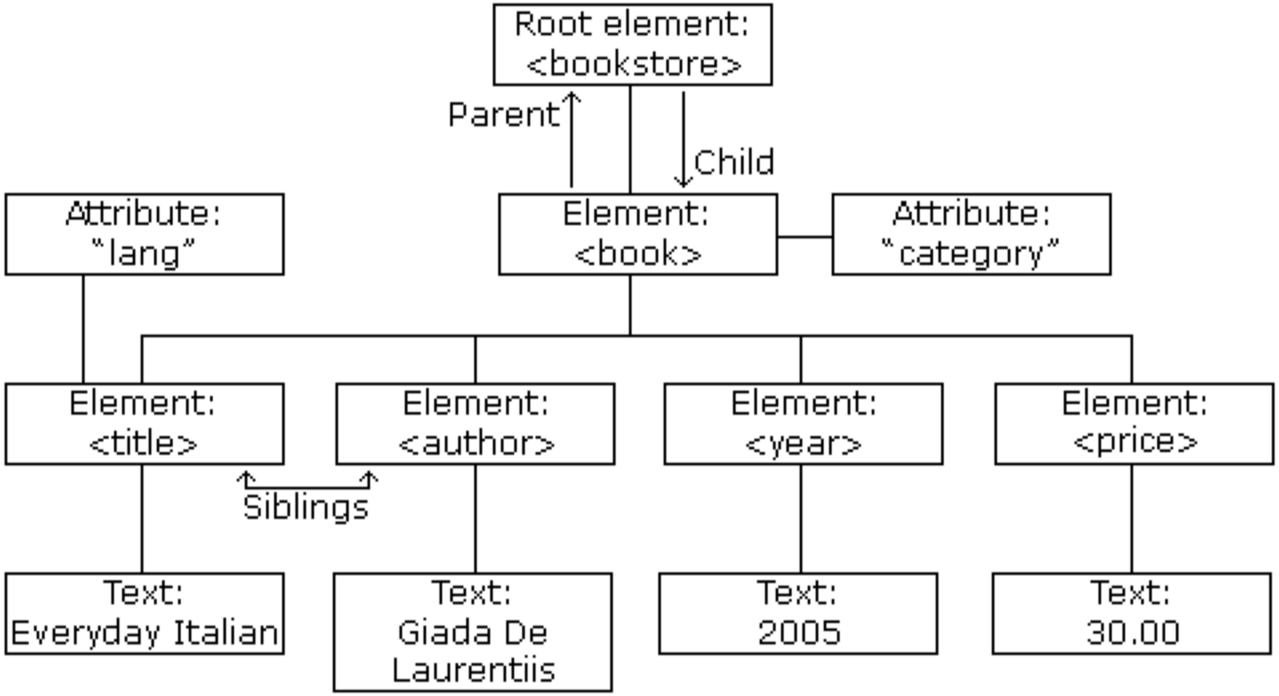
```xml
<?xml version="1.0" encoding="UTF-8"?>
<wonders><!-- Wonders of the ancient world -->
    <wonder>
        <name>Colossus of Rhodes</name>
        <location>Greece</location>
        <height units="feet">107</height>
    </wonder>
    <wonder>
        <name>Great Pyramid of Giza</name>
        <location>Egypt</location>
        <height units="meters">147</height>
    </wonder>
</wonders>
```

XML Components

- Tagged elements, which may be nested within one another
- Attributes on elements
- Text

XML DOM

- XML DOM (Document Object Model): Tree-based model of XML data

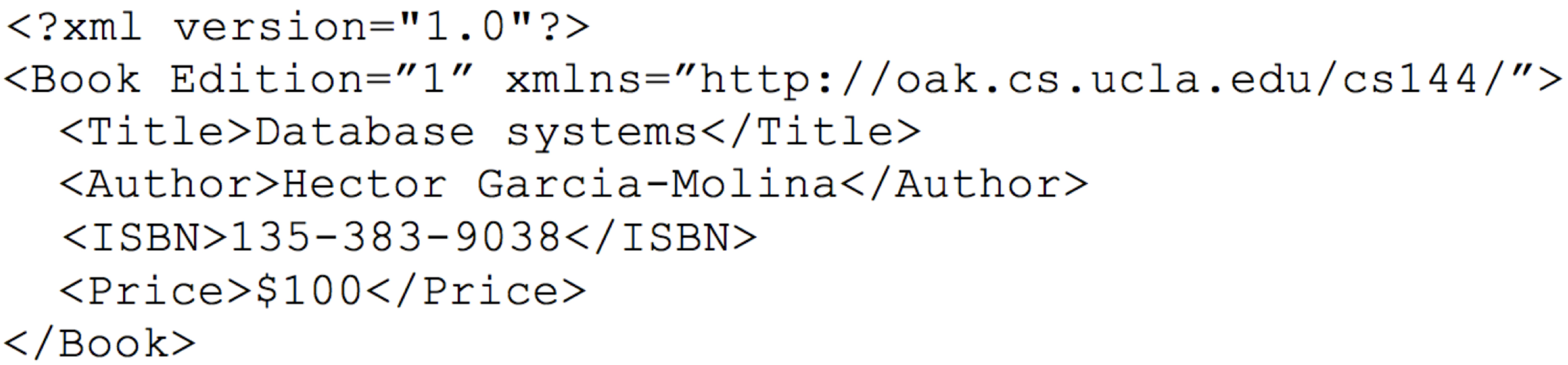


XML Namespace

- A way to avoid name conflict
- XML Namespace allows specifying what we truly mean by a tag

```
<?xml version="1.0"?>
<Book Edition=”1” xmlns=”http://oak.cs.ucla.edu/cs144/”>
  <Title>Database systems</Title>
  <Author>Hector Garcia-Molina</Author>
  <ISBN>135-383-9038</ISBN>
  <Price>$100</Price>
</Book>
```

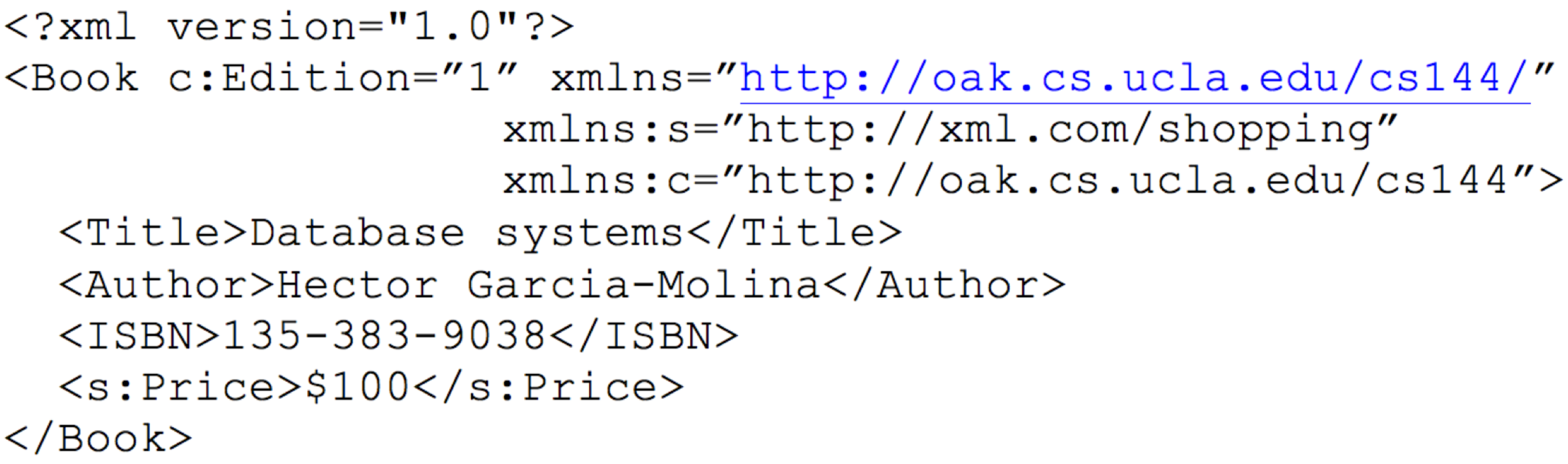
Example



```
<?xml version="1.0"?>
<Book Edition=”1” xmlns=”http://oak.cs.ucla.edu/cs144/”>
  <Title>Database systems</Title>
  <Author>Hector Garcia-Molina</Author>
  <ISBN>135-383-9038</ISBN>
  <Price>$100</Price>
</Book>
```

- xmlns=“uri” defines a **default namespace**
- What is the namespaces of **Title** and attribute **Edition**?
- Note: The **default namespace** does not apply to attributes. Unprefixed **attributes** belong to no namespace.
- Check wiki page for more details: https://en.wikipedia.org/wiki/XML_namespace

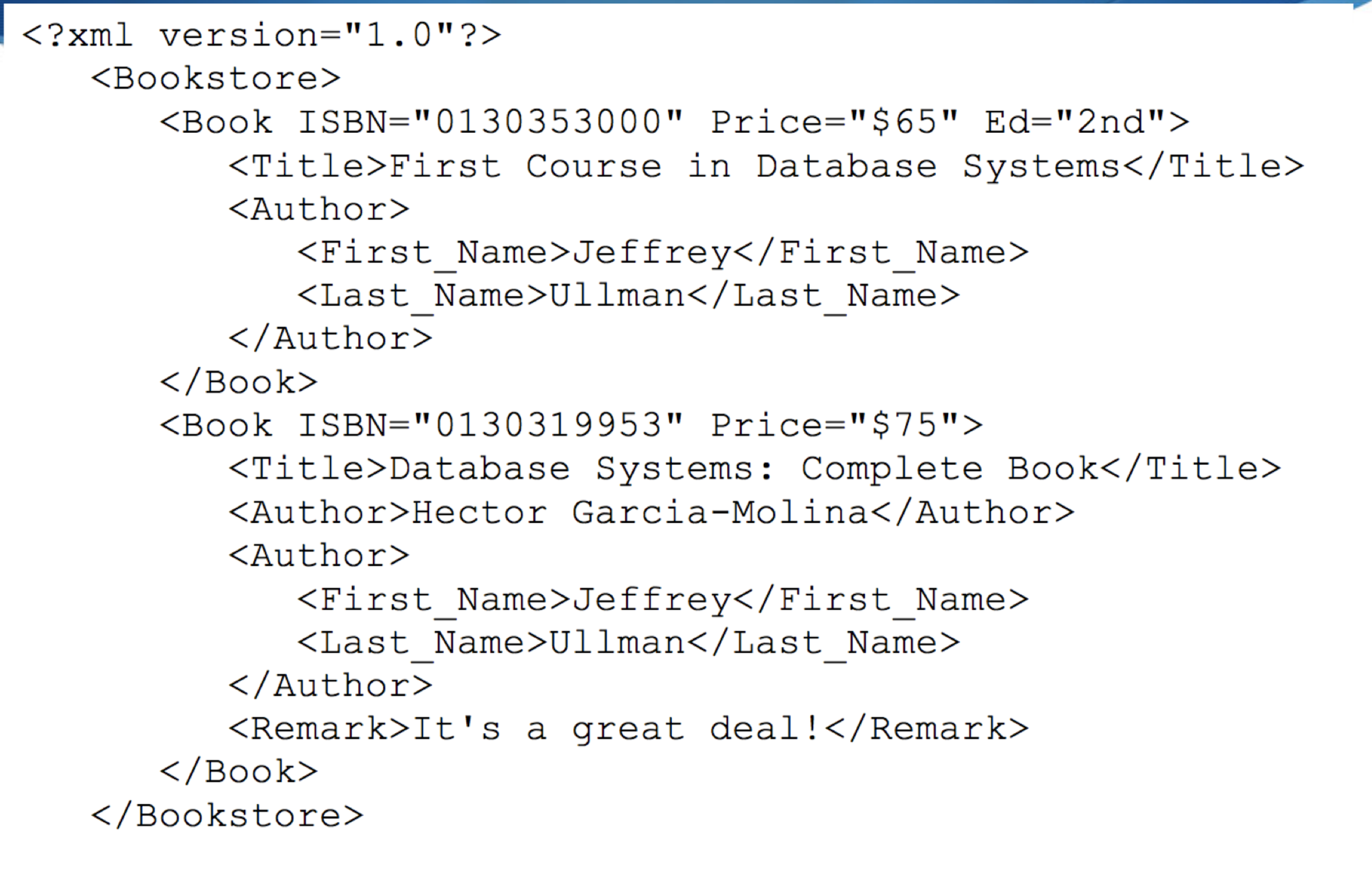
Different Namespaces



```
<?xml version="1.0"?>
<Book c:Edition=”1” xmlns=”http://oak.cs.ucla.edu/cs144/”
                  xmlns:s=”http://xml.com/shopping”
                  xmlns:c=”http://oak.cs.ucla.edu/cs144”>
  <Title>Database systems</Title>
  <Author>Hector Garcia-Molina</Author>
  <ISBN>135-383-9038</ISBN>
  <s:Price>$100</s:Price>
</Book>
```

- **Default and non-default namespaces**

What is the structure of the data?



```
<?xml version="1.0"?>
   <Bookstore>
      <Book ISBN="0130353000" Price="$65" Ed="2nd">
         <Title>First Course in Database Systems</Title>
         <Author>
            <First_Name>Jeffrey</First_Name>
            <Last_Name>Ullman</Last_Name>
         </Author>
      </Book>
      <Book ISBN="0130319953" Price="$75">
         <Title>Database Systems: Complete Book</Title>
         <Author>Hector Garcia-Molina</Author>
         <Author>
            <First_Name>Jeffrey</First_Name>
            <Last_Name>Ullman</Last_Name>
         </Author>
         <Remark>It's a great deal!</Remark>
      </Book>
   </Bookstore>
```

Same-origin policy

- XMLHttpRequest can send a request only to the same host of the page
  - Due to this policy, a third-party site cannot be contacted through XMLHttpRequest
  - Run a “proxy” on the same host, which takes a request and forwards it to the third-party Web site
  - Cross-Origin Resource Sharing (CORS) and JSONP have been developed to get around this restriction

Cross-Origin Resource Sharing (CORS)

- The browser can inquire server-approved cross-request domains through Origin: header
- The server replies the list of allowed domains with Access-Control-AllowOrigin: header

In request to server

Origin : http :// oak .cs. ucla .edu

In response from the server

Access - Control -Allow - Origin : http :// www. google .com

JSONP (JSON with Padding)

- A “hack” to get around same-origin policy restriction
- Using JavaScript, set src to the URL to which a request should be sent –
  - Same origin policy is not applied to src in <script src=’url’>!
- The response is considered as a JavaScript by the browser and gets executed
  - If the response is in JSON, a JavaScript object is created!

Web Storage

- HTML5 provides localStorage: a persistent “storage” to store data locally

```
// store and retrieve data local
Storage [" username "] = " John ";
localStorage [" object "] = JSON . stringify (obj );
let name = localStorage [" username "];
 // iterate over all stored keys
for(let key in localStorage ) {
let value = localStorage [key ];
}
localStorage . removeItem (" username ");
localStorage . clear () ; // delete everything
```

Web Storage

- LocalStorage and sessionStorage
  - Associative key-value store
  - HTML5 standard allows storing any object, but most browsers support only string
  - localStorage persists over multiple browser sessions
    - * Separate storage is allocated per each server
  - sessionStorage persists only within the current browser tab
    - * Data disappears once the browser tab is closed
    - * If two tabs from the same server is opened, they get separate storage

TypeScript

- Superset of JavaScript (a.k.a. JavaScript++) to make it easier to program for largescale JavaScript projects
- Transpilation: TypeScript code is “compiled” to a JavaScript code using TypeScript compiler

```
// --- hello .ts ---
 function hello ( name : string ): string
{
return " Hello " + name ;
}
console . log( hello (" world !"));
$tsc hello .ts
```

TypeScript

- The previous command runs the TypeScript compiler tsc on hello.ts and produces the hello.js file, which contains a standard JavaScript code.

```
$ node hello .js
Hello world !
```

Types

- Types can be added to functions and variables as an intended “contract”

```
function hello ( name : string ): string
{
return " Hello " + name ;
}
let user = [0 , 1, 2];
hello ( user );
```

Types

- Compiler produces an error for the above code due to type mismatch

```
$ tsc hello .ts hello .ts (6 ,33) :
error TS2345 : Argument of type 'number [] ' is not assignable to
parameter of type 'string '.
```

- Use any type to specifically indicate that any type is possible
- Use void as the return type of a function with no return value

Interfaces

- Like Java, TypeScript supports interfaces
- Two types are compatible if their internal structure is compatible
  - We can implement an interface simply by having the needed structure of the interface, without an explicit implements clause

Interfaces

```
interface Person {
             firstName : string ;
            lastName : string ;
}
function hello ( person : Person ) {
             return "Hello , " + person . firstName + " " + person .
lastName ;
}
let user = { firstName : " Jane ", lastName : " User " };
hello ( user );
```

- – No error in the above example because user is compatible with Person

Generics

- • Like Java generics, TypeScript allows creating generic functions/classes using parameterized types

```
class Pair {
        x: T;
        y: T;
        constructor (x: T, y: T) {
                this .x = x;
                this .y = y; }
}
let p = new Pair < number >(1 , 2) ;
function log ( arg: T) : void
{
        console . log(arg);
}
log<number> (1) ;
```

Decorators

- We can “decorate” classes, methods, properties, and parameters using a decorator
  - Syntax: @decorator

```
@sealed  // <- decorator
class Greeter {
        greeting : string ;
        constructor ( greeting : string ) {
                this . greeting = greeting ;
        }
        greet () {
                return "Hello , " + this . greeting ;
        }
}
```

Extension to Class

- TypeScript allows public, private, protected modifiers to class property/methods declaration
  - If one of the three keywords are added to a constructor parameter, the parameter becomes such a property
  - constructor(private id: number): id becomes a private property of the class
- TypeScript allows abstract class declaration