WEB 2.0
(CSE 2056)

Department of Computer Science and Engineering

School of Engineering,

PRESIDENCY UNIVERSITY

MODULE 2

- Data interchange formats: XML, XML basics; XML Schema; Types, Sample program for XML, Overview of JQuery, JQuery example, Overview Angular JS

eXtensible Markup Language (XML)

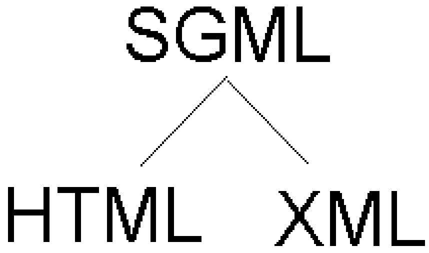
- **Outline of Presentation**
- Introduction
- Comparison between XML and HTML
- XML Syntax
- XML Queries and Mediators
- Challenges
- Summary

What is XML?

- *e***X***tensible* **M***arkup* **L***anguage*
- Markup language for documents containing structured information
- Defined by four specifications:
  - XML, the Extensible Markup Language
  - XLL, the Extensible Linking Language
  - XSL, the Extensible Style Language
  - XUA, the XML User Agent

XML….

- Based on Standard Generalized Markup Language (SGML)
- Version 1.0 introduced by World Wide Web Consortium (W3C) in 1998
- Bridge for data exchange on the Web



SGML

HTML XML

Comparisons

- XML
- Extensible set of tags
- Content orientated
- Standard Data infrastructure
- Allows multiple output forms
- HTML
- Fixed set of tags
- Presentation oriented
- No data validation capabilities
- Single presentation

Authoring XML Elements

- An XML element is made up of a start tag, an end tag, and data in between.
- Example:

  <director> Matthew Dunn </director>
- Example of another element with the same value:

  <actor> Matthew Dunn </actor>
- XML tags are case-sensitive:

  <CITY> <City> <city>
- XML can abbreviate empty elements, for example:

  <married> </married> can be abbreviated to

  <married/>

Authoring XML Elements (cont’d)

- An attribute is a name-value pair separated by an equal sign (=).
- Example:

  <City ZIP=“94608”> Emeryville </City>
- Attributes are used to attach additional, secondary information to an element.

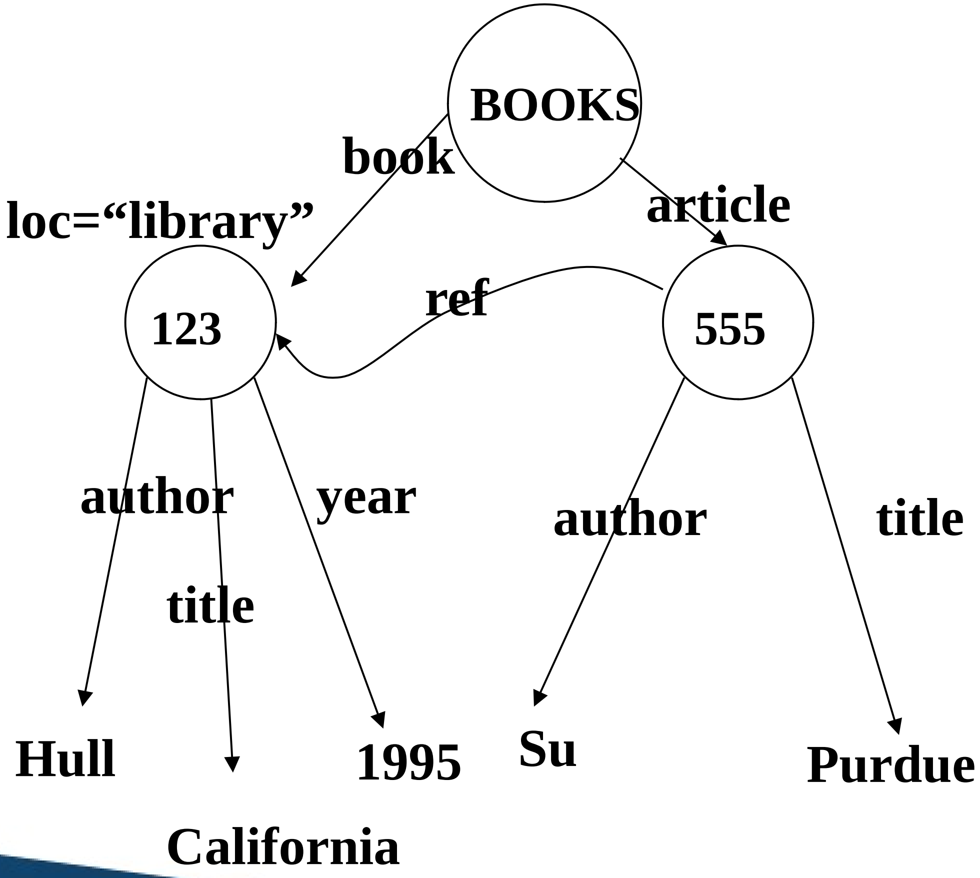
Authoring XML Documents

- A basic XML document is an XML element that can, but might not, include nested XML elements.
- Example:

```
<books>
    <book isbn=“123”>
        <title> Second Chance </title>
        <author> Matthew Dunn </author>
    </book>
</books>
```

XML Data Model: Example

```
<BOOKS>
<book id="123"
  loc="library">
   <author>Hull</author>
   <title>California</title>
   <year> 1995 </year>
</book>
<article id="555"
  ref="123">
   <author>Su</author>
   <title> Purdue</title>
</article>
</BOOKS>
```



Authoring XML Documents (cont’d)

- Authoring guidelines:
  - All elements must have an end tag.
  - All elements must be cleanly nested (overlapping elements are not allowed).
  - All attribute values must be enclosed in quotation marks.
  - Each document must have a unique first element, the root node.

Authoring XML Data Islands

- A data island is an XML document that exists within an HTML page.
- The <XML> element marks the beginning of the data island, and its ID attribute provides a name that you can use to reference the data island.

Authoring XML Data Islands (cont’d)

- Example:

```
<XML ID=“XMLID”>
    <customer>
        <name> Mark Hanson </name>
        <custID> 29085 </custID>
    </customer>
</XML>
```

Document Type Definitions (DTD)

- An XML document may have an optional DTD.
- DTD serves as grammar for the underlying XML document, and it is part of XML language.
- DTDs are somewhat unsatisfactory, but no consensus exists so far beyond the basic DTDs.
- DTD has the form:

  <!DOCTYPE name [markupdeclaration]>

DTD (cont’d)

- Consider an XML document:

```
 <db><person><name>Alan</name>
                  <age>42</age>
                  <email>agb@usa.net </email>
       </person>
       <person>………</person>
       ……….
 </db>
```

DTD (cont’d)

- DTD for it might be:

```
<!DOCTYPE db [
   <!ELEMENT db (person*)>
   <!ELEMENT person (name, age, email)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT age (#PCDATA)>
   <!ELEMENT email (#PCDATA)>
 ]>
```

DTD (cont’d)

Occurrence Indicator:

Indicator	Occurrence	
(no indicator)	Required	One and only one
?	Optional	None or one
*	Optional, repeatable	None, one, or more
+	Required, repeatable	One or more

XML Query Languages

- The first XML query languages
  - LOREL (Stanford)
  - XQL
- Several other query languages have been developed (e.g. UNQL, XPath)
- XML-QL considered by W3C for standardization
- Currently W3C is considering and working on a new query language: XQuery

A Query Language for XML: XML-QL

- Developed at AT&T labs
- To extract data from the input XML data
- Has variables to which data is bound and templates which show how the output XML data is to be constructed
- Uses the XML syntax
- Based on a where/construct syntax
  - ***Where*** combines ***from*** and ***where*** parts of SQL
  - ***Construct*** corresponds to SQL’s ***select***

XML-QL Query: Example 1

- Retrieve all authors of books published by Morgan Kaufmann:

```
where <book>
            <publisher><name>
             Morgan Kaufmann
            </name>  </publisher>
          <title>    $T  </title>
          <author> $A  </author>
     </book> in “www.a.b.c/bib.xml”
construct <result> $A </result>
```

XML-QL Query: Example 2

- XML-QL query asking for all bookstores that sell *The Java Programming Language* for under $25:

```
where <store>
          <name> $N </name>
          <book>
          <title> The Java Programming Language  </title>
          <price> $P </price>
          </book>
      </store> in “www.store/bib.xml”
       $P < 25
construct <result> $N </result>
```

Semistructured Data and Mediators

- Semistructured data is often encountered in data exchange and integration
- At the sources the data may be structured (e.g. from relational databases)
- We model the data as semistructured to facilitate exchange and integration
- Users see an integrated semistructured view that they can query
- Queries are eventually reformulated into queries over the structured resources (e.g. SQL)
- Only results need to be materialized

What is a *mediator* ?

- A complex software component that integrates and transforms data from one or several sources using a declarative specification
- Two main contexts:
  - Data conversion: converts data between two different models
    - e.g. by translating data from a relational database into XML
  - Data integration: integrates data from different sources into a common view

Converting Relational Database to XML

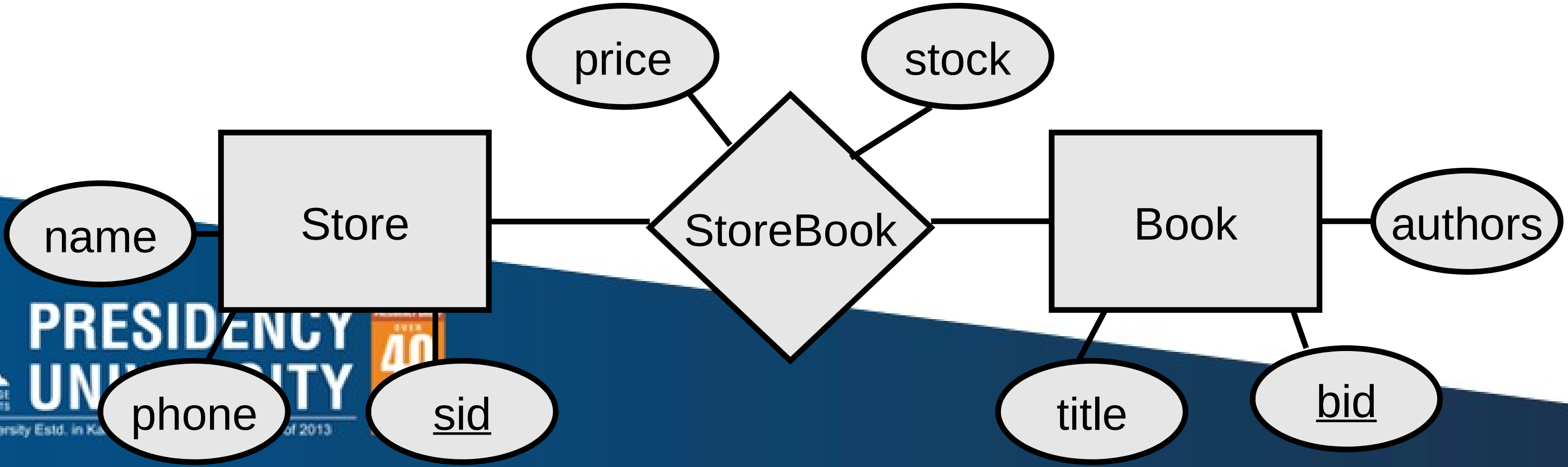
Example: Export the following data into XML and group books by store

- Relational Database:

  Store (sid, name, phone)

  Book (bid, title, authors)

  StoreBook (sid , bid, price, stock)



Converting Relational Database to XML (Cont’d)

- XML:

```
<store> <name> … </name>
          <phone> … </phone>
          <book> <title>… </title>
                 <authors> … </authors>
                 <price> … </price>
          </book>
          <book>…</book>
          …
<store>
```

Challenges facing XML

- Integration of data sharing

- Security

RSA PRESS
XML Security
Develop and deploy a solid XML security strategy
Learn about the structure and syntax of XML signatures
Counteract XML-specific security breaches effectively
Blake Dournaee

XML

- Structure of XML Data
- XML Document Schema
- Querying and Transformation
- Application Program Interfaces to XML
- Storage of XML Data
- XML Applications

Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
- Documents have tags giving extra information about sections of the document
  - E.g. <title> XML </title> <slide> Introduction …</slide>
- **Extensible**, unlike HTML
  - Users can add new tags, and *separately* specify how the tag should be handled for display

XML Introduction (Cont.)

- The ability to specify new tags, and to create nested tag structures make XML a great way to exchange **data**, not just documents.
  - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
  - E.g.

```
<university>
      <department>
        <dept_name> Comp. Sci. </dept_name>
        <building> Taylor </building>
        <budget> 100000 </budget>
     </department>
     <course>
         <course_id> CS-101 </course_id>
         <title> Intro. to Computer Science </title>
         <dept_name> Comp. Sci </dept_name>
         <credits> 4 </credits>
     </course>
  </university>
```

XML: Motivation

- Data interchange is critical in today’s networked world
  - Examples:
    - Banking: funds transfer
    - Order processing (especially inter-company orders)
    - Scientific data
      - Chemistry: ChemML, …
      - Genetics: BSML (Bio-Sequence Markup Language), …
  - Paper flow of information between organizations is being replaced by electronic flow of information
- Each application area has its own set of standards for representing information
- XML has become the basis for all new generation data interchange formats

XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
  - Similar in concept to email headers
  - Does not allow for nested structures, no standard “type” language
  - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
  - XML type specification languages to specify the syntax
    - DTD (Document Type Descriptors)
    - XML Schema
  - Plus textual descriptions of the semantics
- XML allows new tags to be defined as required
  - However, this may be constrained by DTDs
- A wide variety of tools is available for parsing, browsing and querying XML documents/data

Comparison with Relational Data

- Inefficient: tags, which in effect represent schema information, are repeated
- Better than relational tuples as a data-exchange format
  - Unlike relational tuples, XML data is self-documenting due to presence of tags
  - Non-rigid format: tags can be added
  - Allows nested structures
  - Wide acceptance, not only in database systems, but also in browsers, tools, and applications

Structure of XML Data

- **Tag**: label for a section of data
- **Element**: section of data beginning with <*tagname*> and ending with matching </*tagname*>
- Elements must be properly nested
  - Proper nesting
    - <course> … <title> …. </title> </course>
  - Improper nesting
    - <course> … <title> …. </course> </title>
  - Formally: every start tag must have a unique matching end tag, that is in the context of the same parent element.
- Every document must have a single top-level element

Example of Nested Elements

```
<purchase_order>
<identifier> P-101 </identifier>
<purchaser>  …. </purchaser>
<itemlist>
   <item>
      <identifier> RS1 </identifier>
      <description> Atom powered rocket sled </description>
      <quantity> 2 </quantity>
      <price> 199.95 </price>
   </item>
   <item>
      <identifier> SG2 </identifier>
      <description> Superb glue </description>
      <quantity> 1 </quantity>
      <unit-of-measure> liter </unit-of-measure>
      <price> 29.95 </price>
   </item>
 </itemlist>
</purchase_order>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
  - Example: elements representing *item* nested within an *itemlist* element
- Nesting is not supported, or discouraged, in relational databases
  - With multiple orders, customer name and address are stored redundantly
  - normalization replaces nested structures in each order by foreign key into table storing customer name and address information
  - Nesting is supported in object-relational databases
- But nesting is appropriate when transferring data
  - External application does not have direct access to data referenced by a foreign key

Structure of XML Data (Cont.)

- Mixture of text with sub-elements is legal in XML.
  - Example:

```
 <course>
   This course is being offered for the first time in 2009.
   <course id> BIO-399 </course id>
   <title> Computational Biology </title>
   <dept name> Biology </dept name>
   <credits> 3 </credits>
</course>
```

  - Useful for document markup, but discouraged for data representation

Attributes

- Elements can have **attributes**

```
<course course_id= “CS-101”>
    <title> Intro. to Computer Science</title>
    <dept name> Comp. Sci. </dept name>
    <credits> 4 </credits>
</course>
```

- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

```
<course  course_id = “CS-101”  credits=“4”>
```

Attributes vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <course course_id= “CS-101”> … </course>
      - <course>
        <course_id>CS-101</course_id> …
        </course>
  - Suggestion: use attributes for identifiers of elements, and use subelements for contents

Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use unique-name:element-name
- Avoid using long unique names all over document by using XML Namespaces

```
<university xmlns:yale="http://www.yale.edu">
    …
    <yale:course>
         <yale:course_id> CS-101 </yale:course_id>
         <yale:title> Intro. to Computer Science</yale:title>
         <yale:dept_name> Comp. Sci. </yale:dept_name>
         <yale:credits> 4 </yale:credits>
    </yale:course>
    …
</university>
```

More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a /> and deleting the end tag
  - <course course_id="CS-101" Title="Intro. To Computer Science"
    dept_name = "Comp. Sci." credits="4" />
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - <![CDATA[<course> … </course>]]>

  Here, <course> and </course> are treated as just strings

  CDATA stands for "character data"

XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema
  - **Document Type Definition (DTD)**
    - Widely used
  - **XML Schema**
    - Newer, increasing use

Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax
  - <!ELEMENT element (subelements-specification) >
  - <!ATTLIST element (attributes) >

Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

```
<! ELEMENT department (dept_name  building, budget)>
<! ELEMENT dept_name (#PCDATA)>
<! ELEMENT budget (#PCDATA)>
```

- Subelement specification may have regular expressions

```
<!ELEMENT university ( ( department | course | instructor | teaches )+)>
```

  - Notation:
    - “|” - alternatives
    - “+” - 1 or more occurrences
    - “*” - 0 or more occurrences

University DTD

```
<!DOCTYPE  university [
  <!ELEMENT university ( (department|course|instructor|
  teaches)+)>
  <!ELEMENT department ( dept name, building, budget)>
  <!ELEMENT course ( course id, title, dept name, credits)>
  <!ELEMENT instructor (IID, name, dept name, salary)>
  <!ELEMENT teaches (IID, course id)>
  <!ELEMENT dept name( #PCDATA )>
  <!ELEMENT building( #PCDATA )>
  <!ELEMENT budget( #PCDATA )>
  <!ELEMENT course id ( #PCDATA )>
  <!ELEMENT title ( #PCDATA )>
  <!ELEMENT credits( #PCDATA )>
  <!ELEMENT IID( #PCDATA )>
  <!ELEMENT name( #PCDATA )>
  <!ELEMENT salary( #PCDATA )>
]>
```

Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)

- Examples
  - <!ATTLIST course course_id CDATA #REQUIRED>, or
  - <!ATTLIST course

    ```
    course_id     ID          #REQUIRED
      dept_name  IDREF   #REQUIRED
    instructors    IDREFS #IMPLIED   >
    ```

IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID value must contain the ID value of an element in the same document

University DTD with Attributes

- University DTD with ID and IDREF attribute types.

```
<!DOCTYPE university-3 [
    <!ELEMENT university ( (department|course|instructor)+)>
    <!ELEMENT department ( building, budget )>
    <!ATTLIST department
          dept_name ID #REQUIRED >
    <!ELEMENT course (title, credits )>
    <!ATTLIST course
          course_id ID #REQUIRED
          dept_name IDREF #REQUIRED
          instructors IDREFS #IMPLIED >
    <!ELEMENT instructor ( name, salary )>
    <!ATTLIST instructor
          IID ID #REQUIRED
          dept_name IDREF #REQUIRED >
    · · · declarations for title, credits, building,
          budget, name and salary · · ·
]>
```

XML data with ID and IDREF attributes

```
<university-3>
    <department dept name=“Comp. Sci.”>
        <building> Taylor </building>
        <budget> 100000 </budget>
    </department>
    <department dept name=“Biology”>
        <building> Watson </building>
        <budget> 90000 </budget>
    </department>
    <course course id=“CS-101” dept name=“Comp. Sci”
              instructors=“10101 83821”>
         <title> Intro. to Computer Science </title>
         <credits> 4 </credits>
    </course>
    ….
    <instructor IID=“10101” dept name=“Comp. Sci.”>
         <name> Srinivasan </name>
         <salary> 65000 </salary>
    </instructor>
    ….
</university-3>
```

Limitations of DTDs

- No typing of text elements and attributes
    - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
    - Order is usually irrelevant in databases (unlike in the document-layout environment from which XML evolved)
    - (A | B)* allows specification of an unordered set, but
        - Cannot ensure that each of A and B occurs only once
- IDs and IDREFs are untyped
    - The *instructors* attribute of an course may contain a reference to another course, which is meaningless
        - *instructors* attribute should ideally be constrained to refer to instructor elements

XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User-defined, comlex types
  - Many more features, including
    - uniqueness and foreign key constraints, inheritance
- XML Schema is itself specified in XML syntax, unlike DTDs
  - More-standard representation, but verbose
- XML Scheme is integrated with namespaces
- BUT: XML Schema is significantly more complicated than DTDs.

XML Schema Version of Univ. DTD

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="university" type="universityType" />
<xs:element name="department">
     <xs:complexType>
          <xs:sequence>
              <xs:element name="dept name" type="xs:string"/>
              <xs:element name="building" type="xs:string"/>
              <xs:element name="budget" type="xs:decimal"/>
          </xs:sequence>
     </xs:complexType>
</xs:element>
….
<xs:element name="instructor">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="IID" type="xs:string"/>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="dept name" type="xs:string"/>
            <xs:element name="salary" type="xs:decimal"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
… Contd.
```

XML Schema Version of Univ. DTD (Cont.)

```
….
<xs:complexType name=“UniversityType”>
    <xs:sequence>
        <xs:element ref=“department” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“course” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“instructor” minOccurs=“0” maxOccurs=“unbounded”/>
        <xs:element ref=“teaches” minOccurs=“0” maxOccurs=“unbounded”/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
```

- Choice of “xs:” was ours -- any other namespace prefix could be chosen
- Element “university” has type “universityType”, which is defined separately
  - xs:complexType is used later to create the named complex type “UniversityType”

More features of XML Schema

- Attributes specified by xs:attribute tag:
  - <xs:attribute name = “dept_name”/>
  - adding the attribute use = “required” means value must be specified
- Key constraint: “department names form a key for department elements under the root university element:

```
<xs:key name = “deptKey”>
        <xs:selector xpath = “/university/department”/>
        <xs:field xpath = “dept_name”/>
<\xs:key>
```

- Foreign key constraint from course to department:

```
<xs:keyref name = “courseDeptFKey” refer=“deptKey”>
        <xs:selector xpath = “/university/course”/>
        <xs:field xpath = “dept_name”/>
<\xs:keyref>
```

XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a January 2005 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a

  for … let … where … order by …result …

  syntax

  for □ SQL **from**
  where □ SQL **where**
  order by □ SQL **order by**

  result □ SQL **select**
  let allows temporary variables, and has no equivalent in SQL

XSLT

- A **stylesheet** stores formatting options for a document, usually separately from document
  - E.g. an HTML style sheet may specify font colors and sizes for headings, etc.
- The **XML Stylesheet Language (XSL)** was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML, and XML to HTML
- XSLT transformations are expressed using rules called **templates**
  - Templates combine selection using XPath with construction of results

Application Program Interface

- There are two standard application program interfaces to XML data:
  - **SAX** (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
  - **DOM** (Document Object Model)
    - **XML** data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      getParentNode( ), getFirstChild( ), getNextSibling( )
      getAttribute( ), getData( ) (for text node)
      getElementsByTagName( ), …
    - Also provides functions for updating DOM tree

Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems discussed in Chapter 1 (no concurrency, no recovery, …)
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems
  - Relational databases
    - Data must be translated into relational form
    - Advantage: mature database systems
    - Disadvantages: overhead of translating data and queries

Storage of XML in Relational Databases

- Alternatives:
  - String Representation
  - Tree Representation
  - Map to relations

String Representation

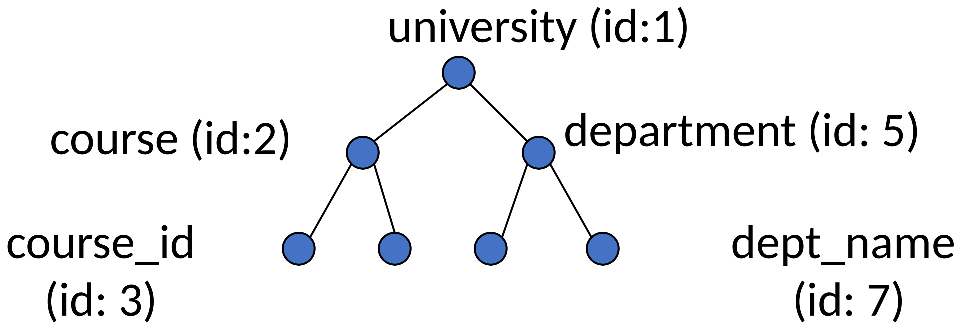
- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g. account, customer, depositor relations
      - Each with a string-valued attribute to store the element
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation, and build indices on these fields
    - E.g. customer_name or account_number
  - Some database systems support **function indices,** which use the result of a function as the key value.
    - The function should return the value of the required subelement/attribute

String Representation (Cont.)

- Benefits:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - Allows fast access to individual elements.
- Drawback: Need to parse strings to access values inside the elements
  - Parsing is slow.

Tree Representation

- **Tree representation:** model XML data as tree and store using relations
  nodes(id, parent_id, type, label, value)



- Each element/attribute is given a unique identifier
- Type indicates element/attribute
- Label specifies the tag name of the element/name of attribute
- Value is the text value of the element/attribute
- Can add an extra attribute *position* to record ordering of children

Tree Representation (Cont.)

- Benefit: Can store any XML data, even without DTD
- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

Mapping XML Data to Relations

- Relation created for each element type whose schema is known:
  - An id attribute to store a unique id for each element
  - A relation attribute corresponding to each element attribute
  - A parent_id attribute to keep track of parent element
    - As in the tree representation
    - Position information (ith child) can be store too
- All subelements that occur only once can become relation attributes
  - For text-valued subelements, store the text as attribute value
  - For complex subelements, can store the id of the subelement
- Subelements that can occur multiple times represented in a separate table
  - Similar to handling of multivalued attributes when converting ER diagrams to tables

Storing XML Data in Relational Systems

- Applying above ideas to department elements in university-1 schema, with nested course elements, we get

  department(*id*, *dept_name*, *building*, *budget*)

  course(*parent id*, *course_id*, *dept_name*, *title*, *credits*)
- **Publishing**: process of converting relational data to an XML format
- **Shredding**: process of converting an XML document into a set of tuples to be inserted into one or more relations
- XML-enabled database systems support automated publishing and shredding
- Many systems offer *native storage* of XML data using the **xml** data type. Special internal data structures and indices are used for efficiency

SQL/XML

- New standard SQL extension that allows creation of nested XML output
  - Each output tuple is mapped to an XML element *row*

```
<university>
 <department>
    <row>
       <dept name> Comp. Sci. </dept name>
       <building> Taylor </building>
       <budget> 100000 </budget>
    </row>
   …. more rows if there are more output tuples …
 </department>
 … other relations ..
</university>
```

SQL Extensions

- **xmlelement** creates XML elements
- **xmlattributes** creates attributes

```
    select xmlelement (name “course”,
       xmlattributes (course id as course id, dept name as dept name),
       xmlelement (name “title”, title),
       xmlelement (name “credits”, credits))
  from course
```

- Xmlagg creates a forest of XML elements

```
       select xmlelement (name “department”,
                  dept_name,
                  xmlagg (xmlforest(course_id)
                              order by course_id))
     from course
     group by dept_name
```

XML Applications

- Storing and exchanging data with complex structures
  - E.g. Open Document Format (ODF) format standard for storing Open Office and Office Open XML (OOXML) format standard for storing Microsoft Office documents
  - Numerous other standards for a variety of applications
    - ChemML, MathML
- Standard for data exchange for Web services
  - remote method invocation over HTTP protocol
  - More in next slide
- Data mediation
  - Common data representation format to bridge different systems

Web Services

- The Simple Object Access Protocol (SOAP) standard:
  - Invocation of procedures across applications with distinct databases
  - XML used to represent procedure input and output
- A *Web service* is a site providing a collection of SOAP procedures
  - Described using the Web Services Description Language (WSDL)
  - Directories of Web services are described using the Universal Description, Discovery, and Integration (UDDI) standard

What is jQuery?

- jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development. (jQuery.com)

Why learn jQuery?

- Write less, do more:
    - *`$("p.neat").addClass("ohmy").show("slow");`*
- Performance
- Plugins
- It’s standard
- … and fun!

Example: Show/Hide Button

window.onload

- We cannot use the DOM before the page has been constructed. jQuery gives us a more compatibile way to do this.
  - The DOM way

```
window.onload = function() { // do stuff with the
```

  - The direct jQuery translation

```
$(document).ready(function() { // do stuff with th
```

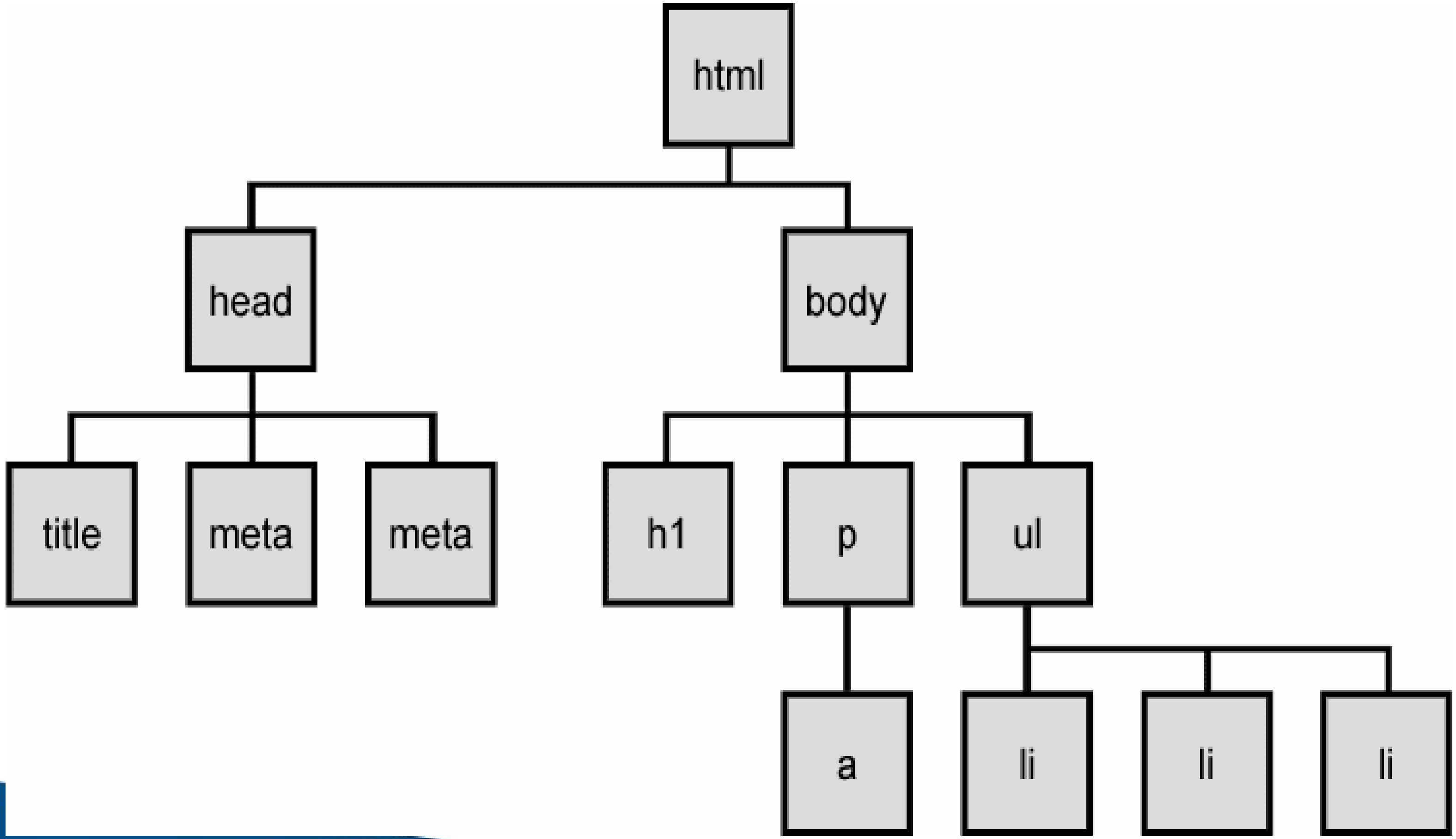
  - The jQuery way

```
$(function() { // do stuff with the DOM })
```

Aspects of the DOM and jQuery

- **Identification:** how do I obtain a reference to the node that I want.
- **Traversal:** how do I move around the DOM tree.
- **Node Manipulation:** how do I get or set aspects of a DOM node.
- **Tree Manipulation:** how do I change the structure of the page.

The DOM tree



Selecting groups of DOM objects

name	description
getElementById	returns array of descendents with the given tag, such as "div"
getElementsByTagName	returns array of descendents with the given tag, such as "div"
getElementsByName	returns array of descendents with the given name attribute (mostly useful for accessing form controls)
querySelector *	returns the first element that would be matched by the given CSS selector string
querySelectorAll *	returns an array of all elements that would be matched by the given CSS selector string

jQuery node identification

```
// id selector
var elem = $("#myid");

// group selector
var elems = $("#myid, p");

// context selector
var elems = $("#myid < div p");

// complex selector
var elems = $("#myid < h1.special:not(.classy)");
```

jQuery Selectors

- http://api.jquery.com/category/selectors/

jQuery / DOM comparison

DOM method	jQuery equivalent
getElementById("id")	$("#id")
getElementsByTagName("tag")	$("tag")
getElementsByName("somename")	$("[name='somename']")
querySelector("selector")	$("selector")
querySelectorAll("selector")	$("selector")

Exercise

- Use jQuery selectors to identify elements with these properties in a hypothetical page:
  - All p tags that have no children, but only if they don't have a class of ignore
  - Any element with the text "REPLACE_ME" in it.
  - All div tags with a child that has a class of special
  - All heading elements (h1, h2, h3, h4, h5, h6)
  - Every other visible li.
- Use the DOM API to target the #square and periodically change it's position in a random direction.
- Use jQuery selectors instead of the DOM API.

jQuery terminology

- the jQuery function

  refers to the global jQuery object or the $ function depending on the context

- a jQuery object

  the object returned by the jQuery function that often represents a group of elements

- selected elements

  the DOM elements that you have selected for, most likely by some CSS selector passed to the jQuery function and possibly later filtered further

The jQuery object

- The $ function always (even for ID selectors) returns an array-like object called a jQuery object.
- The jQuery object wraps the originally selected DOM objects.
- You can access the actual DOM object by accessing the elements of the jQuery object.

```
// false
document.getElementById("id") == $("#myid");
document.querySelectorAll("p") == $("p");
// true
document.getElementById("id") == $("#myid")[0];
document.getElementById("id") == $("#myid").get(0);
document.querySelectorAll("p")[0] == $("p")[0];
```

Using $ as a wrapper

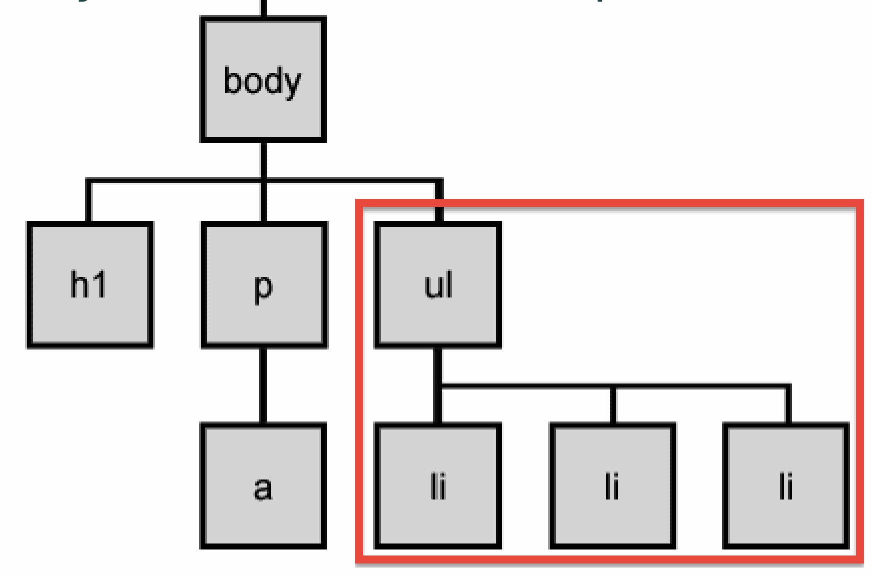
- $ adds extra functionality to DOM elements
- passing an existing DOM object to $ will give it the jQuery upgrade

```
// convert regular DOM objects to a jQuery object
var elem = document.getElementById("myelem");
elem = $(elem);
var elems = document.querySelectorAll(".special");
elems = $(elems);
```

DOM context identification

- You can use querySelectorAll() and querySelector() on any DOM object.
- When you do this, it simply searches from that part of the DOM tree downward.
- Programmatic equivalent of a CSS context selector

```
var list = document.getElementsByTagName("ul")[0];
var specials = list.querySelectorAll('li.special');
```



find / context parameter

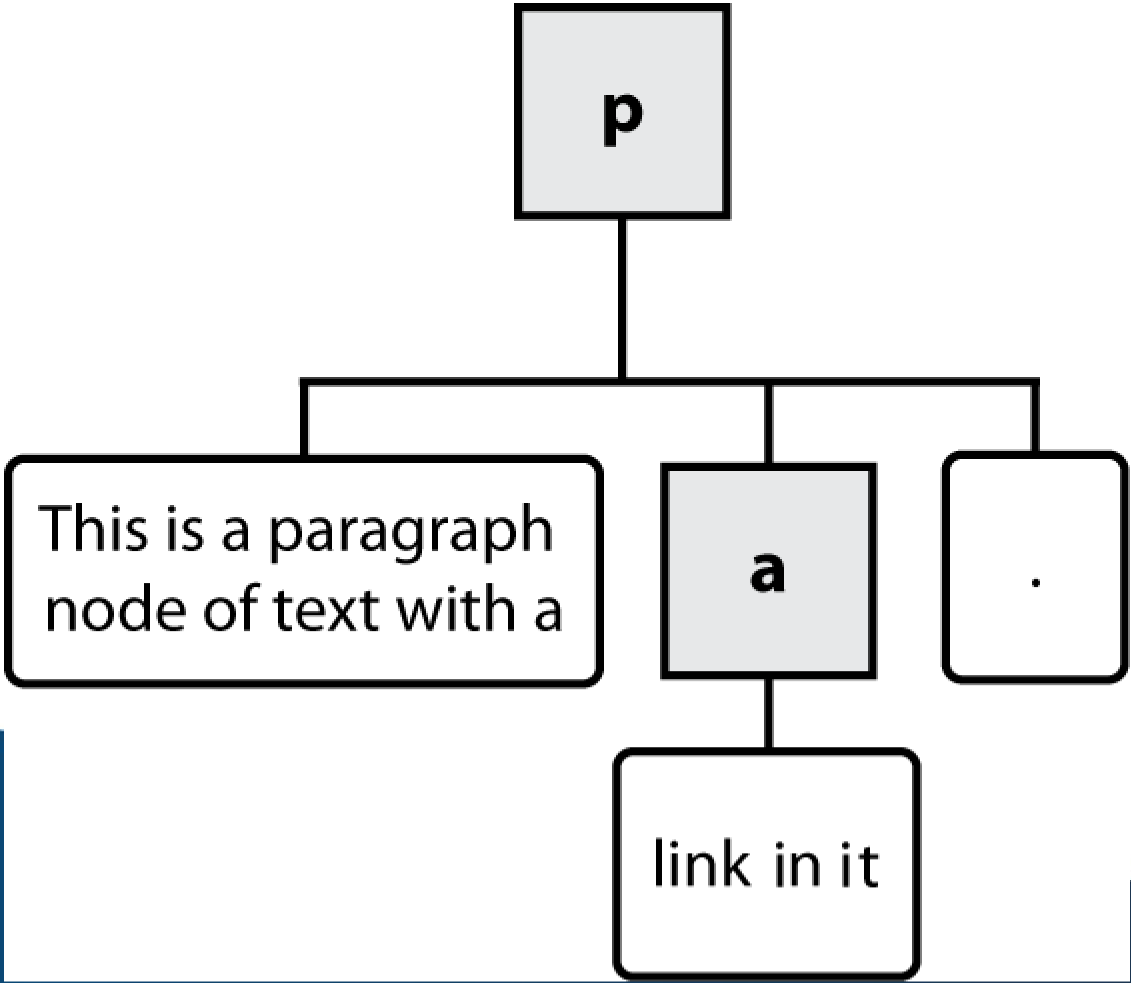
- jQuery gives two identical ways to do contextual element identification

```
var elem = $("#myid");

// These are identical
var specials = $("li.special", elem);
var specials = elem.find("li.special");
```

Types of DOM nodes

```
<p>
   This is a paragraph of text with a
   <a href="/path/page.html">link in it</a>.
</p>
```



Traversing the DOM tree

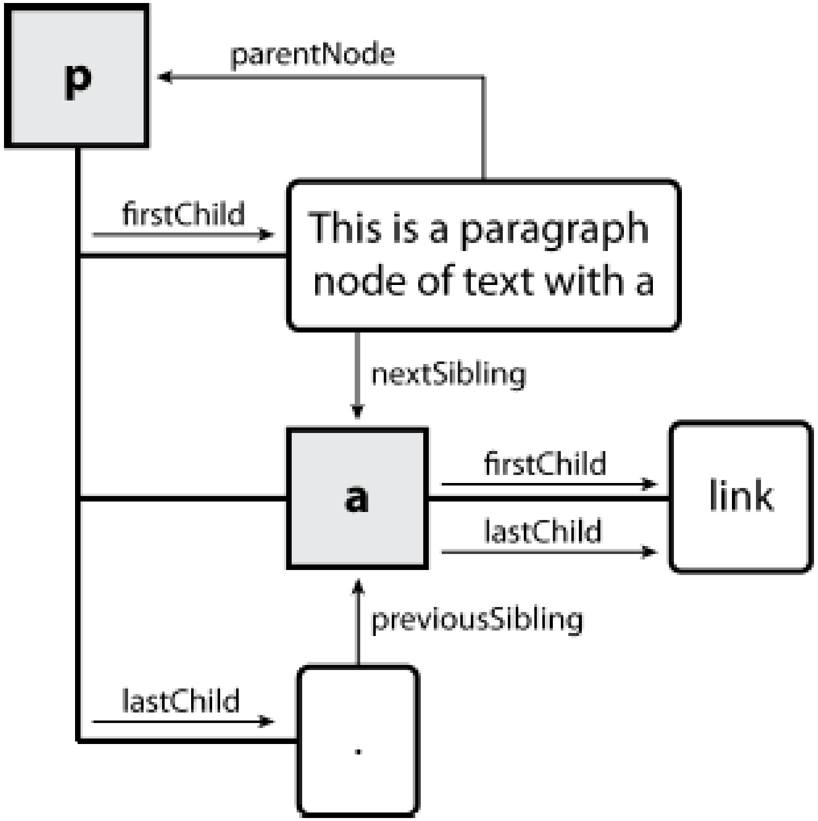
name(s)	description
firstChild, lastChild	start/end of this node's list of children
childNodes	array of all this node's children
nextSibling, previousSibling	neighboring nodes with the same parent
parentNode	the element that contains this node

- complete list of DOM node properties
- browser incompatiblity information (IE6 sucks)

DOM tree traversal example

```html
<p id="foo">This is a paragraph of text with a
<a href="/path/to/another/page.html">link</a>.</p>
```

HTML



Elements vs text nodes

```
<div>
    <p>
        This is a paragraph of text with a
        <a href="page.html">link</a>.
    </p>
</div>                                          HTML
```

- Q: How many children does the div above have?
- A: 3
  - an element node representing the <p>
  - two text nodes representing "\n\t" (before/after the paragraph)
- Q: How many children does the paragraph have? The a tag?

jQuery traversal methods

- http://api.jquery.com/category/traversing/

jQuery tutorials

- Code Academy

  http://www.codecademy.com/courses/you-and-jquery/0?curriculum_id=4fc3018f74258b0003001f0f#!/exercises/0

- Code School:

  http://www.codeschool.com/courses/jquery-air-first-flight

jQuery Example

- jQuery is developed by Google. To create the first jQuery example, you need to use JavaScript file for jQuery. You can download the jQuery file from jquery.com or use the absolute URL of jQuery file.
- In this jQuery example, we are using the absolute URL of jQuery file. The jQuery example is written inside the script tag.
- Let's see a simple example of jQuery.

- <!DOCTYPE html>
- **<html>**
- **<head>**
- **<title>**First jQuery Example**</title>**
- **<script** type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
- **</script>**

- **<script** type="text/javascript" language="javascript">
- $(document).ready(function() {
- $("p").css("background-color", "cyan");
- });
- **</script>**
- **</head>**
- **<body>**
- **<p>**The first paragraph is selected.**</p>**
- **<p>**The second paragraph is selected.**</p>**
- **<p>**The third paragraph is selected.**</p>**
- **</body>**
- **</html>**

- <!DOCTYPE html>
- **<html>**
- **<head>**
- **<title>**Second jQuery Example**</title>**
- **<script** type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js">
- **</script>**

- <**script** type="text/javascript" language="javascript">
- $(function() {
- $("p").css("color", "red");
- });
- **</script>**
- **</head>**
- <**body**>
- <**p**>The first paragraph is selected.**</p**>
- <**p**>The second paragraph is selected.**</p**>
- <**p**>The third paragraph is selected.**</p**>
- **</body>**
- **</html>**

Angular JS

A brief Introduction

Adekunle

What is AngularJS

MVC Javascript Framework by Google for Rich Web Application Development

Why AngularJS

“Other frameworks deal with HTML’s shortcomings by either abstracting away HTML, CSS, and/or JavaScript or by providing an imperative way for manipulating the DOM. Neither of these address the root problem that HTML was not designed for dynamic views”.

- Structure, Quality and Organization
- Lightweight (< 36KB compressed and minified)
- Free
- Separation of concern
- Modularity
- Extensibility & Maintainability
- Reusable Components

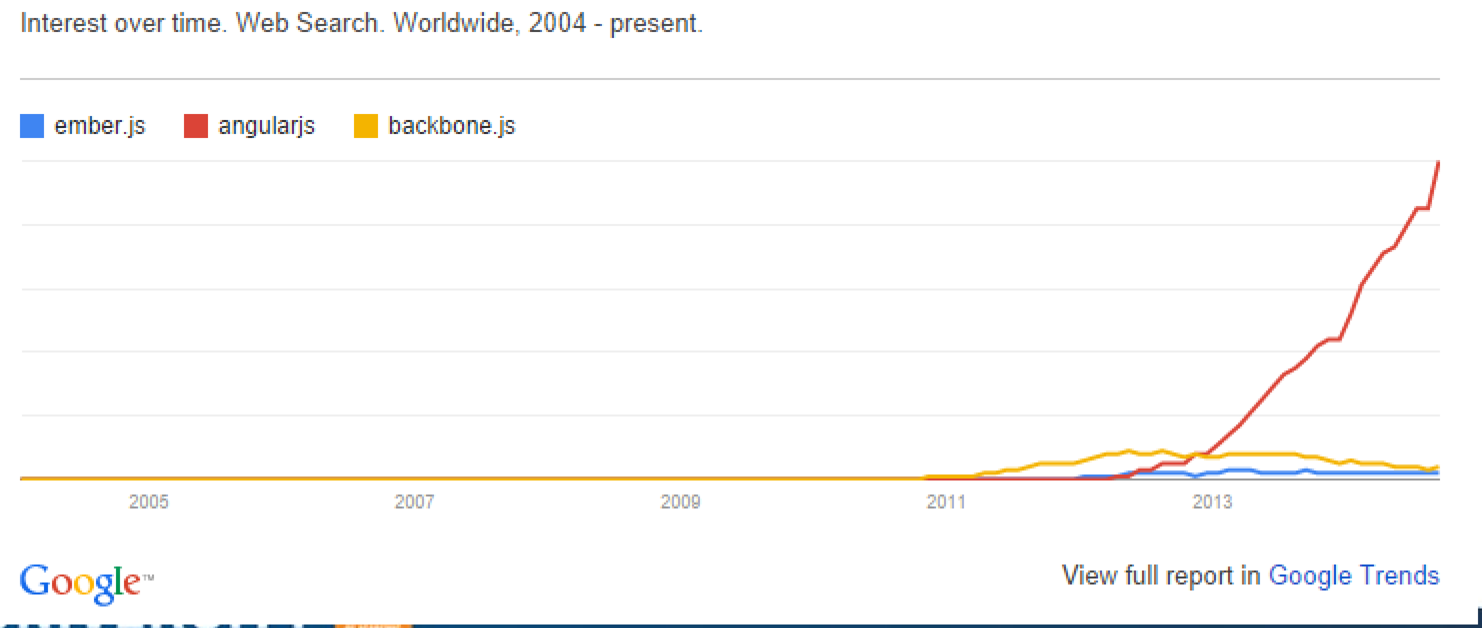
“ HTML? Build UI Declaratively! CSS? Animations! JavaScript? Use it the plain old way!”

jQuery

- Allows for DOM Manipulation
- Does not provide structure to your code
- Does not allow for two way binding

Other Javascript MV* Frameworks

- BackboneJS
- EmberJS



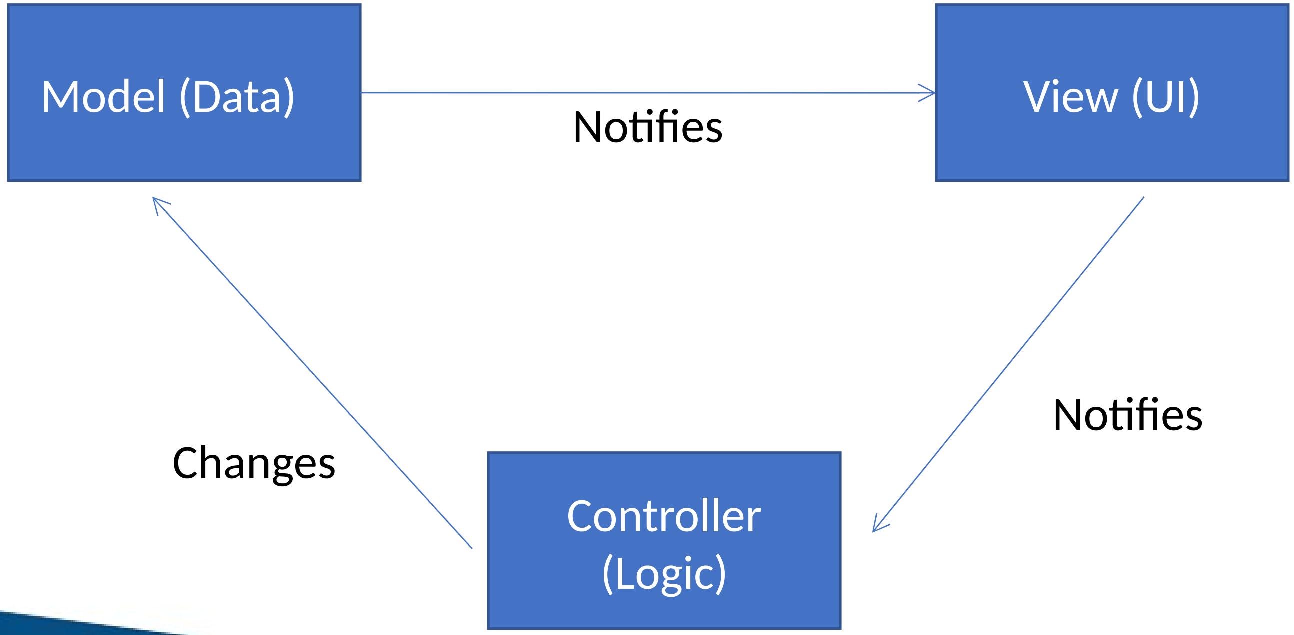
Features of AngularJS

- Two-way Data Binding – Model as single source of truth
- Directives – Extend HTML
- MVC
- Dependency Injection
- Testing
- Deep Linking (Map URL to route Definition)
- Server-Side Communication

Data Binding

```
<html ng-app>
<head>
  <script src='angular.js'></script>
</head>
<body>
  <input ng-model='user.name'>
  <div ng-show='user.name'>Hi {{user.name}}</div>
</body>
</html>
```

MVC



Model (Data)

Notifies

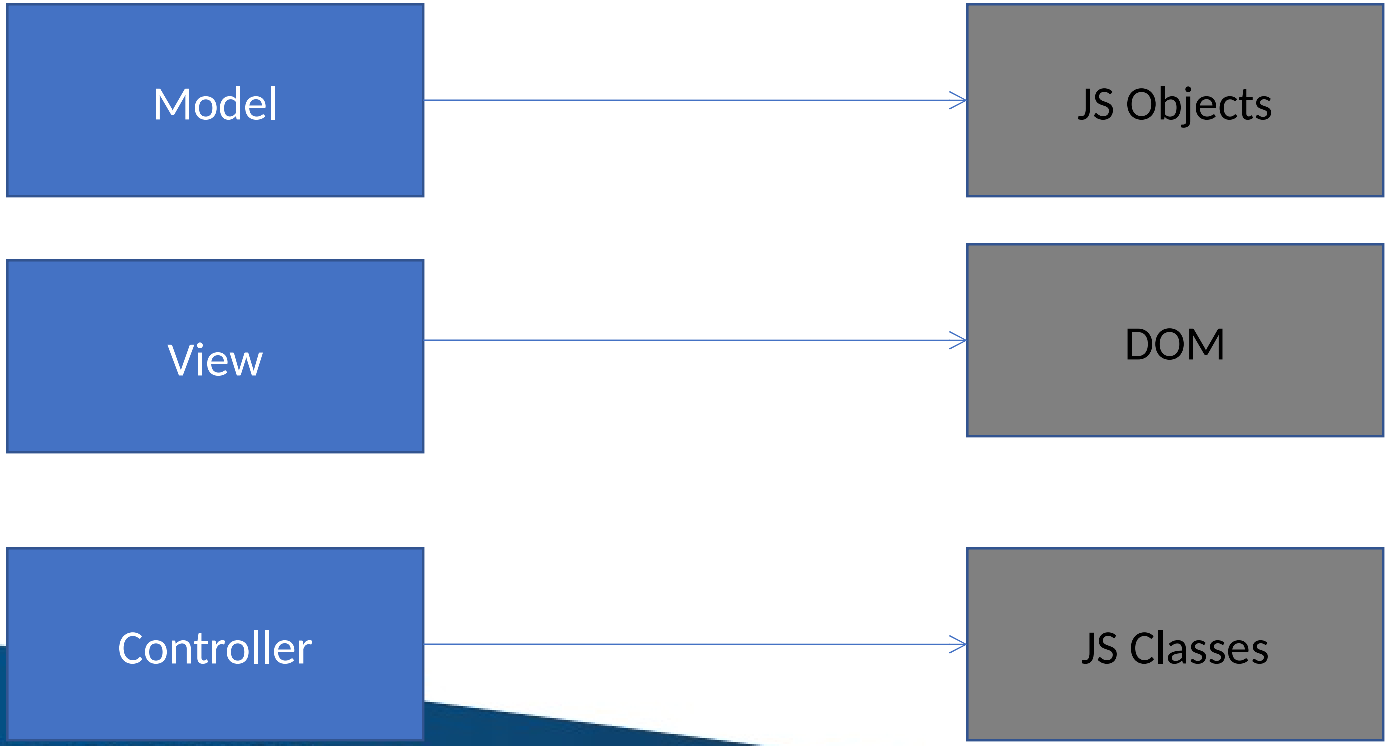
View (UI)

Notifies

Changes

Controller (Logic)

MVC



Model → JS Objects

View → DOM

Controller → JS Classes

MVC

```
<html ng-app>
<head>
  <script src='angular.js'></script>
  <script src='controllers.js'></script>
</head>
<body ng-controller='UserController'>
  <div>Hi {{user.name}}</div>
</body>
</html>

function  XXXX($scope) {
  $scope.user = { name:'Larry' };
}
```

Hello HTML

```
<p>Hello World!</p>
```

Hello Javascript

```
<p id="greeting1"></p>

<script>
var isIE = document.attachEvent;
var addListener = isIE
  ? function(e, t, fn) {
      e.attachEvent('on' + t, fn);}
  : function(e, t, fn) {
      e.addEventListener(t, fn, false);};
addListener(document, 'load', function(){
  var greeting = document.getElementById('greeting1');
  if (isIE) {
    greeting.innerText = 'Hello World!';
  } else {
    greeting.textContent = 'Hello World!';
 }
});
</script>
```

Hello JQuery

```
<p id="greeting2"></p>

<script>
$(function(){
  $('#greeting2').text('Hello World!');
});
</script>
```

Hello AngularJS

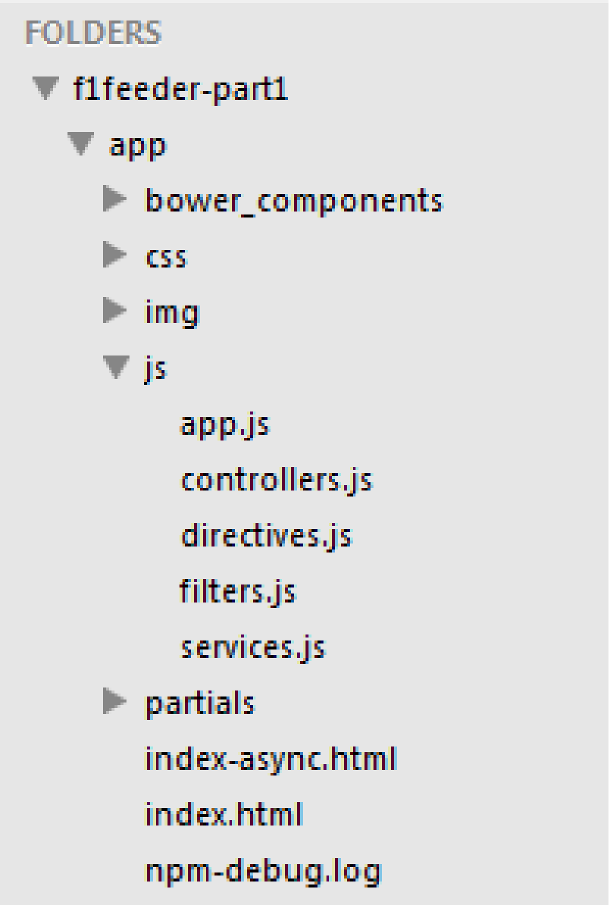
```
<p ng:init="greeting = 'Hello
  World!'">{{greeting}}</p>
```

DEMONSTRATION!!!!!

Feeder App

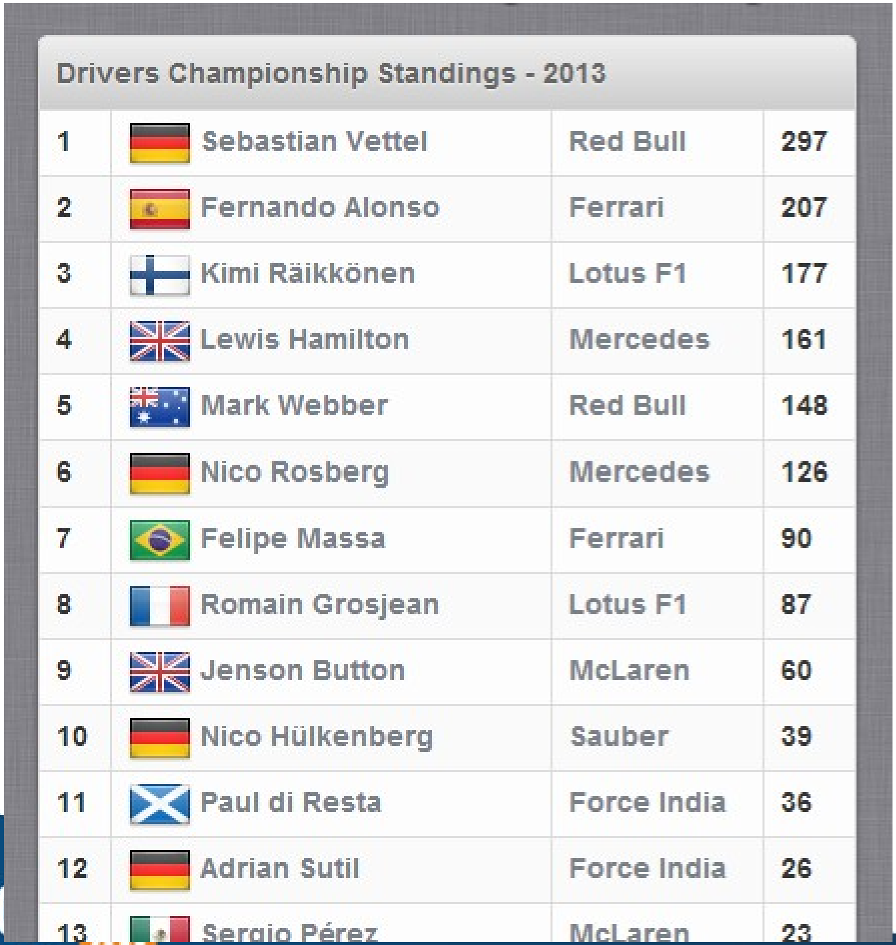
http://www.toptal.com/angular-js/a-step-by-step-guide-to-your-first-angularjs-app

App Skeleton



```
FOLDERS
▼ f1feeder-part1
  ▼ app
    ▶ bower_components
    ▶ css
    ▶ img
    ▼ js
        app.js
        controllers.js
        directives.js
        filters.js
        services.js
    ▶ partials
      index-async.html
      index.html
      npm-debug.log
```

Final View (Championship Table)



Drivers Championship Standings - 2013

1	Sebastian Vettel	Red Bull	297
2	Fernando Alonso	Ferrari	207
3	Kimi Räikkönen	Lotus F1	177
4	Lewis Hamilton	Mercedes	161
5	Mark Webber	Red Bull	148
6	Nico Rosberg	Mercedes	126
7	Felipe Massa	Ferrari	90
8	Romain Grosjean	Lotus F1	87
9	Jenson Button	McLaren	60
10	Nico Hülkenberg	Sauber	39
11	Paul di Resta	Force India	36
12	Adrian Sutil	Force India	26
13	Sergio Pérez	McLaren	23

Sample Angular Powered View

```
<body ng-app="F1FeederApp" ng-controller="driversController">
  <table>
    <thead>
      <tr><th colspan="4">Drivers Championship Standings</th></tr>
    </thead>
    <tbody>
      <tr ng-repeat="driver in driversList">
        <td>{{$index + 1}}</td>
        <td>
          <img src="img/flags/{{driver.Driver.nationality}}.png" />
          {{driver.Driver.givenName}}&nbsp;{{driver.Driver.familyName}}
        </td>
        <td>{{driver.Constructors[0].name}}</td>
        <td>{{driver.points}}</td>
      </tr>
    </tbody>
  </table>
</body>
```

Expressions

Expressions allow you to execute some computation in order to return a desired value.

- {{ 1 + 1 }}
- {{ 946757880 | date }}
- {{ user.name }}

you shouldn’t use expressions to implement any higher-level logic.

Directives

Directives are markers (such as attributes, tags, and class names) that tell AngularJS to attach a given behaviour to a DOM element (or transform it, replace it, etc.)

Some angular directives

- The ng-app - Bootstrapping your app and defining its scope.
- The ng-controller - defines which controller will be in charge of your view.
- The ng-repeat - Allows for looping through collections

Directives as Components

```
<rating max='5' model='stars.average'>

<tabs>
  <tab title='Active tab' view='...'>
  <tab title='Inactive tab' view='...'>
</tabs>

<tooltip content='messages.tip1'>
```

Adding Controllers

```
angular.module('F1FeederApp.controllers', []).
controller('driversController', function($scope) {
    $scope.driversList = [
      {
        Driver: {
            givenName: 'Sebastian',
            familyName: 'Vettel'
          },
          points: 322,
          nationality: "German",
          Constructors: [
            {name: "Red Bull"}
          ]
      },
      {
          Driver: {
          givenName: 'Fernando',
            familyName: 'Alonso'
          },
          points: 207,
          nationality: "Spanish",
```

- The $scope variable – Link your controllers and view

App.js

```
angular.module('F1FeederApp', [
  'F1FeederApp.controllers'
]);
```

Initializes our app and register the modules on which it depends

Index.html

```
<body ng-app="F1FeederApp" ng-controller="driversController">
  <table>
    <thead>
      <tr><th colspan="4">Drivers Championship Standings</th></tr>
    </thead>
    <tbody>
      <tr ng-repeat="driver in driversList">
        <td>{{$index + 1}}</td>
        <td>
          <img src="img/flags/{{driver.Driver.nationality}}.png" />
          {{driver.Driver.givenName}}&nbsp;{{driver.Driver.familyName}}
        </td>
        <td>{{driver.Constructors[0].name}}</td>
        <td>{{driver.points}}</td>
      </tr>
    </tbody>
  </table>
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-route.js"></script>
  <script src="js/app.js"></script>
  <script src="js/services.js"></script>
  <script src="js/controllers.js"></script>
</body>
</html>
```

Loading data from the server(services.js)

```
angular.module('F1FeederApp.services', []).
  factory('ergastAPIservice', function($http) {

    var ergastAPI = {};

    ergastAPI.getDrivers = function() {
      return $http({
        method: 'JSONP',
        url:
 'http://ergast.com/api/f1/2013/driverStandin
 gs.json?callback=JSON_CALLBACK'
      });
    }

    return ergastAPI;
  });
```

- $http - a layer on top of XMLHttpRequest or JSONP
- $resource - provides a higher level of abstraction
- Dependency Injection

we create a new module (F1FeederApp.services) and register a service within that module (ergastAPIservice).

Modified controller.js

```
angular.module('F1FeederApp.controllers', []).
  controller('driversController', function($scope, ergastAPIservice) {
    $scope.nameFilter = null;
    $scope.driversList = [];

    ergastAPIservice.getDrivers().success(function (response) {
        //Dig into the responde to get the relevant data
        $scope.driversList =
  response.MRData.StandingsTable.StandingsLists[0].DriverStandings;
    });
  });
```

Routes

- $routeProvider – used for dealing with routes

Modified app.js

```
angular.module('F1FeederApp', [
  'F1FeederApp.services',
  'F1FeederApp.controllers',
  'ngRoute'
]).
config(['$routeProvider', function($routeProvider) {
  $routeProvider.
  when("/drivers", {templateUrl: "partials/drivers.html", controller: "driversController"}).
  when("/drivers/:id", {templateUrl: "partials/driver.html", controller: "driverController"}).
  otherwise({redirectTo: '/drivers'});
}]);
```

Partial views

```
<!DOCTYPE html>
<html>
<head>
  <title>F-1 Feeder</title>
</head>

<body ng-app="F1FeederApp">
  <ng-view></ng-view>
  <script src="bower_components/angular/angular.js"></script>
  <script src="bower_components/angular-route/angular-route.js"></script>
  <script src="js/app.js"></script>
  <script src="js/services.js"></script>
  <script src="js/controllers.js"></script>
</body>
</html>
```

Advanced AngularJS Concept

- Dependency Injection
- Modularity
- Digesting
- Scope
- Handling SEO
- End to End Testing
- Promises
- Localization
- Filters

Useful Links

- https://angularjs.org/
- http://campus.codeschool.com/courses/shaping-up-with-angular-js/contents
- http://www.toptal.com/angular-js/a-step-by-step-guide-to-your-first-angularjs-app
- https://github.com/raonibr/f1feeder-part1