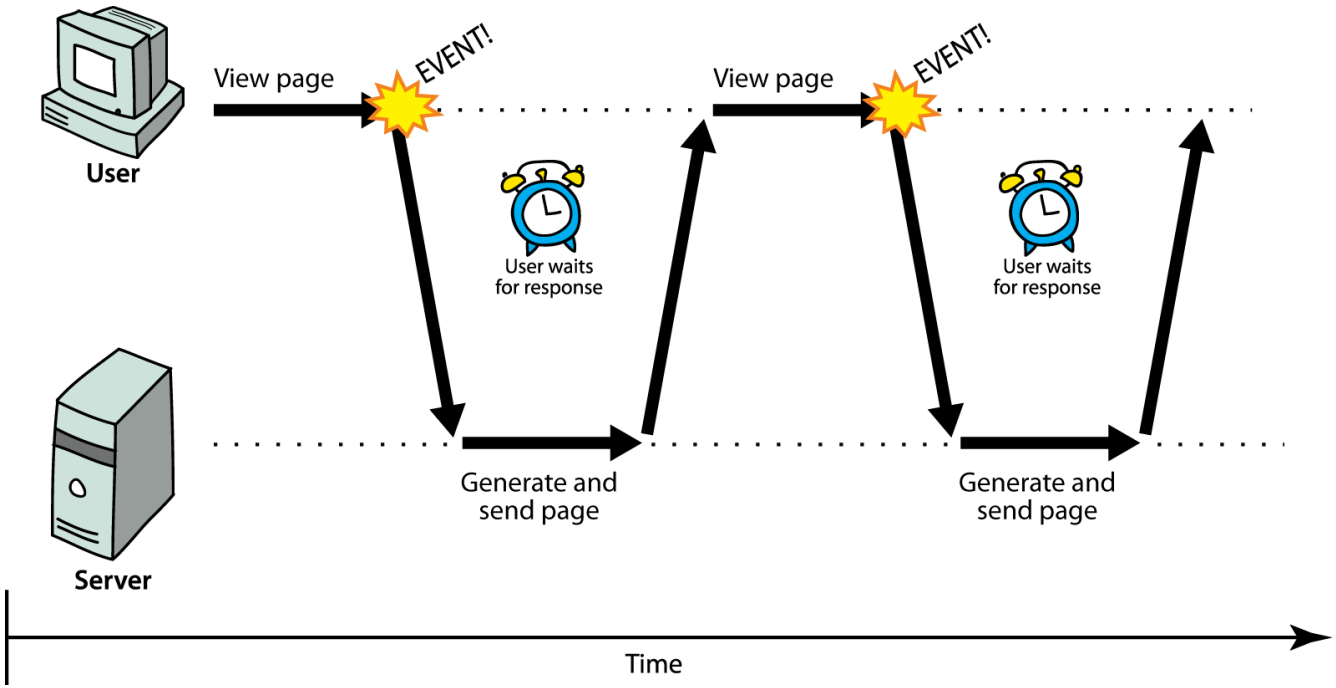
CSE3026: Web Application Development
Ajax, XML, and JSON

Scott Uk-Jin Lee

12.1: Ajax Concepts

Synchronous web communication

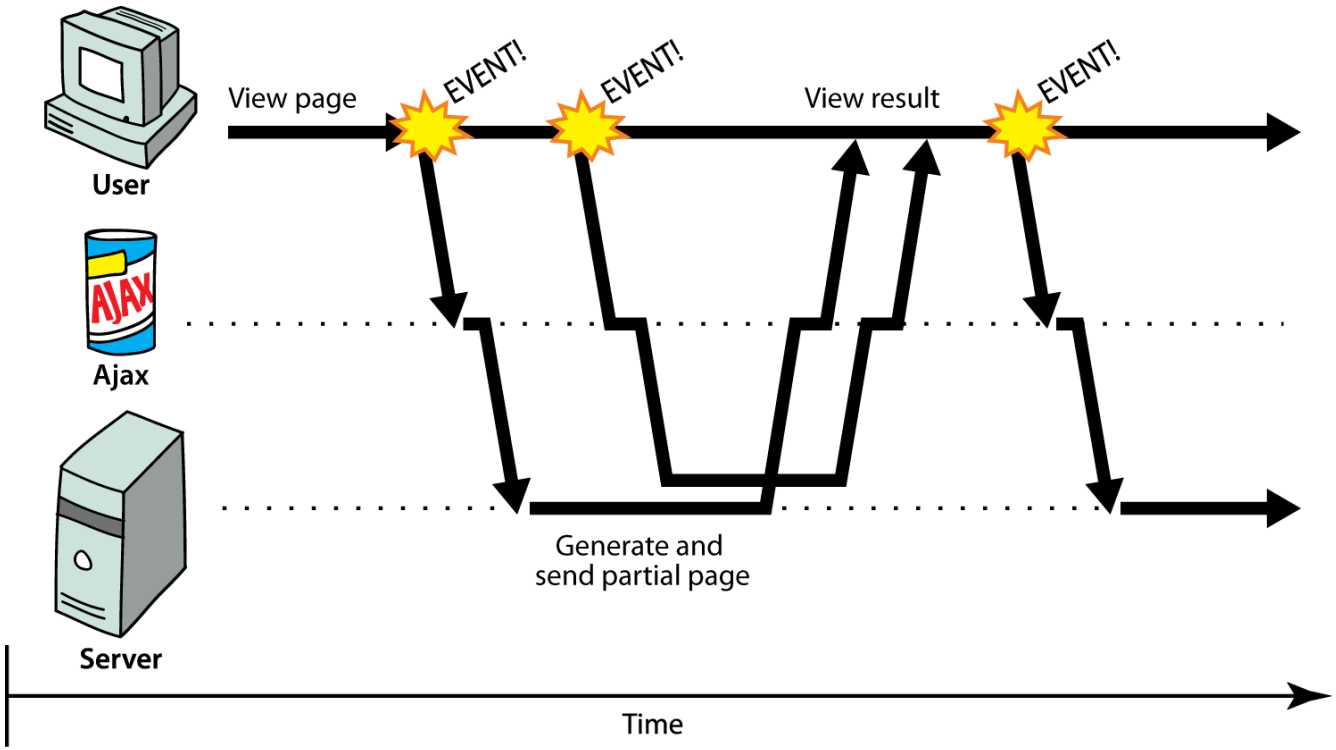


- **synchronous**: user must wait while new pages load
  - the typical communication pattern used in web pages (click, wait, refresh)

Web applications and Ajax

- **web application**: a dynamic web site that mimics the feel of a desktop app
  - presents a continuous user experience rather than disjoint pages
  - examples: Gmail, Google Maps, Google Docs and Spreadsheets, Flickr, A9
- **Ajax**: Asynchronous JavaScript and XML
  - not a programming language; a particular way of using JavaScript
  - downloads data from a server in the background
  - allows dynamically updating a page without making the user wait
  - avoids the "click-wait-refresh" pattern
  - examples: Google Suggest

Asynchronous web communication



- **asynchronous**: user can keep interacting with page while data loads
  - communication pattern made possible by Ajax

12.2: Using XMLHttpRequest

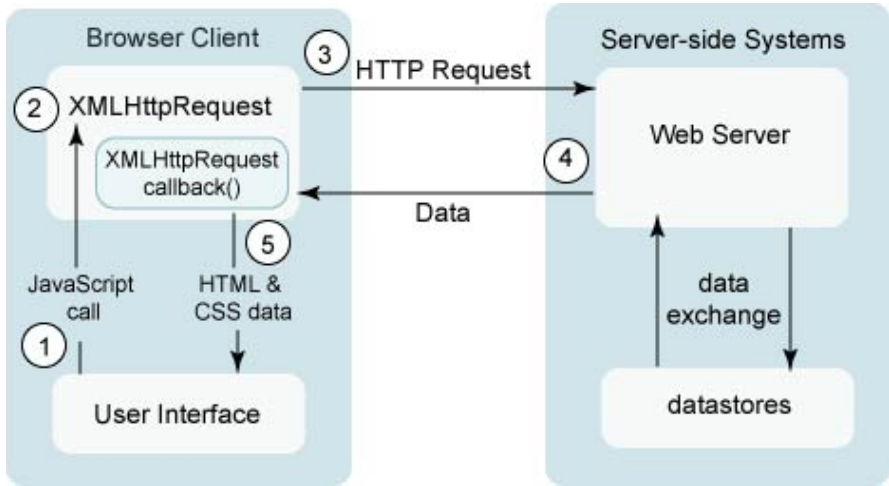
`XMLHttpRequest` (& why we won't use it)

- JavaScript includes an `XMLHttpRequest` object that can fetch files from a web server
  - supported in IE5+, Safari, Firefox, Opera, Chrome, etc. (with minor compatibilities)
- it can do this **asynchronously** (in the background, transparent to user)
- the contents of the fetched file can be put into current web page using the DOM

- sounds great!...
- ... but it is clunky to use, and has various browser incompatibilities
- Prototype provides a better wrapper for Ajax, so we will use that instead

A typical Ajax request

1. user clicks, invoking an event handler
2. handler's code creates an `XMLHttpRequest` object
3. `XMLHttpRequest` object requests page from server
4. server retrieves appropriate data, sends it back
5. `XMLHttpRequest` fires an event when data arrives
   - this is often called a **callback**
   - you can attach a handler function to this event
6. your callback event handler processes the data and displays it



Prototype's Ajax model

```
new Ajax.Request("url", {
    option : value,
    option : value,
    ...
    option : value
});
```
JS

- construct a Prototype `Ajax.Request` object to request a page from a server using Ajax
- constructor accepts 2 parameters:
  1. the **URL** to fetch, as a String,
  2. a set of **options**, as an array of *key* : *value* pairs in `{}` braces (an anonymous JS object)
- hides icky details from the raw `XMLHttpRequest`; works well in all browsers

Prototype Ajax options

option	description
`method`	how to fetch the request from the server (default `"post"`)
`parameters`	query parameters to pass to the server, if any (as a string or object)
`asynchronous`	should request be sent asynchronously in the background? (default `true`)
others: `contentType`, `encoding`, `requestHeaders`	

```
new Ajax.Request("http://www.example.com/foo/bar.txt", {
    method: "get",
    parameters: {name: "Ed Smith", age: 29},   // "name=Ed+Smith&age=29"
    ...
});
```
JS

Prototype Ajax event options

event	description
`onSuccess`	request completed successfully
`onFailure`	request was unsuccessful
`onException`	request has a syntax error, security error, etc.
others: `onCreate`, `onComplete`, `on###` (for HTTP error code ###)	

```
new Ajax.Request("http://www.example.com/foo.php", {
    parameters: {password: "abcdef"},   // "password=abcdef"
    onSuccess: mySuccessFunction
});
```
JS

Basic Prototype Ajax template

```
    new Ajax.Request("url", {
        method: "get",
        onSuccess: functionName
    });
    ...

function functionName(ajax) {
    do something with ajax.responseText;
}
```
JS

- attach a handler to the request's `onSuccess` event
- the handler takes an Ajax response object, which we'll name `ajax`, as a parameter

Ajax response object's properties

property	description
`status`	the request's HTTP error code (200 = OK, etc.)
`statusText`	HTTP error code text
`responseText`	the entire text of the fetched file, as a `String`
`responseXML`	the entire contents of the fetched file, as a DOM tree (seen later)

```js
function handleRequest(ajax) {
    alert(ajax.responseText);
}
```

- most commonly used property is `responseText`, to access the fetched text content

Handling Ajax errors

```
    new Ajax.Request("url", {
        method: "get",
        onSuccess: functionName,
        onFailure: ajaxFailure,
        onException: ajaxFailure
    });
    ...
function ajaxFailure(ajax, exception) {
    alert("Error making Ajax request:" +
          "\n\nServer status:\n" + ajax.status + " " + ajax.statusText +
          "\n\nServer response text:\n" + ajax.responseText);
    if (exception) {
        throw exception;
    }
}
```
JS

- for user's (and developer's) benefit, show an error message if a request fails

Passing query parameters to a request

```
new Ajax.Request("lookup_account.php", {
    method: "get",
    parameters: {name: "Ed Smith", age: 29, password: "abcdef"},
    onFailure: ajaxFailure,
    onException: ajaxFailure
});
...
```
JS

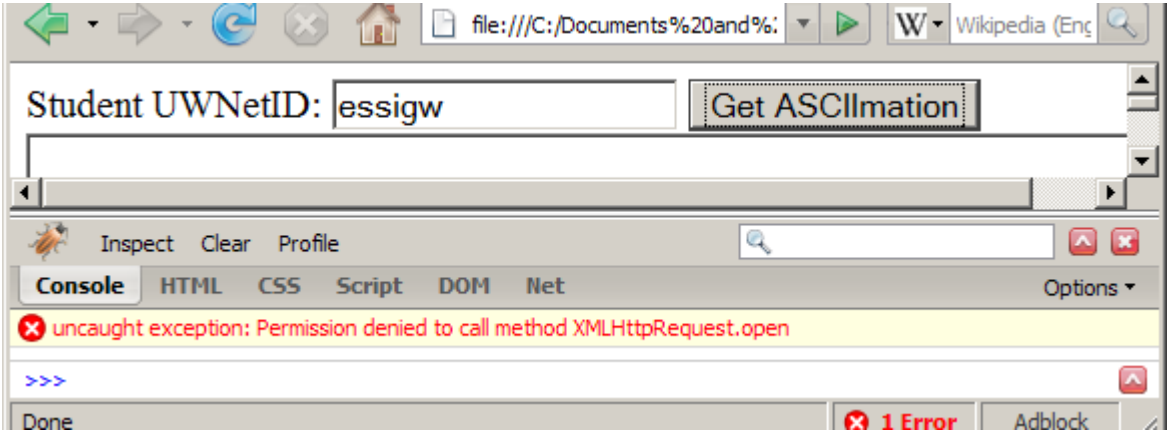
- don't concatenate the parameters onto the URL yourself with `"?" + ...`
  - won't properly URL-encode the parameters
  - won't work for POST requests
- query parameters are passed as a `parameters` object
  - written between `{}` braces as a set of *name* : *value* pairs (another anonymous object)
  - (the above is equivalent to: `"name=Ed+Smith&age=29&password=abcdef"`)

Creating a `POST` request

```
new Ajax.Request("url", {
    method: "post",   // optional
    parameters: { name: value, name: value, ..., name: value },
    onSuccess: functionName,
    onFailure: functionName,
    onException: functionName
});
```
JS

- `method` should be changed to `"post"` (or omitted; `post` is default)

`XMLHttpRequest` security restrictions



- Ajax must be run on a web page stored on a **web server**
  - *(cannot be run from a web page stored on your hard drive)*
- Ajax can only fetch files from the **same server** that the page is on
  - `http://www.foo.com/a/b/c.html` can only fetch from `www.foo.com`

Prototype's Ajax Updater

```
new Ajax.Updater("id", "url", {
    method: "get"
});
```

JS

- `Ajax.Updater` fetches a file and injects its content into an element as `innerHTML`
  - this is a common Ajax use case: "go fetch this page/file, and put its contents into an element on the page"
  - could do this with `Ajax.Request`, but `Ajax.Updater` saves you some typing and work
- additional (1st) parameter specifies the `id` of element to inject into
- `onSuccess` handler not needed (but `onFailure`, `onException` handlers may still be useful)

Ajax.Updater options

```
new Ajax.Updater({success: "id", failure: "id"}, "url", {
    method: "get",
    insertion: "top"
});
```
JS

- instead of passing a single id, you can pass an object with a `success` and/or `failure` id
  - the `success` element will be filled if the request succeeds
  - the `failure` element (if provided) will be filled if the request fails
- `insertion` parameter specifies where in the element to insert the text (top, bottom, before, after)

PeriodicalUpdater

```
new Ajax.PeriodicalUpdater("id", "url", {
    frequency: seconds,
    name: value, ...
});
```
JS

- `Ajax.PeriodicalUpdater` repeatedly fetches a file at a given interval and injects its content into an element as `innerHTML`
- `onSuccess` handler not needed (but `onFailure`, `onException` handlers may still be useful)
- same options as in `Ajax.Updater` can be passed

Ajax.Responders

```
Ajax.Responders.register({
    onEvent: functionName,
    onEvent: functionName,
    ...
});
```
JS

- sets up a default handler for a given kind of event for all Ajax requests
- can be useful for attaching a **common failure/exception handler** to all requests in one place

12.3: XML

The bad way to store data (text formats)

```
My note:
BEGIN
    FROM: Alice Smith (alice@example.com)
    TO: Robert Jones (roberto@example.com)
    SUBJECT: Tomorrow's "Birthday Bash" event!
    MESSAGE (english):
        Hey Bob,
        Don't forget to call me this weekend!
    PRIVATE: true
END
```

- Many apps make up their own custom text format for storing data.
- We could also send a file like this from the server to browser with Ajax.
- What's wrong with this approach?

XML: A better way of storing data

```
<?xml version="1.0" encoding="UTF-8"?>
<note private="true">
    <from>Alice Smith (alice@example.com)</from>
    <to>Robert Jones (roberto@example.com)</to>
    <subject>Tomorrow's "Birthday Bash" event!</subject>
    <message language="english">
        Hey Bob, Don't forget to call me this weekend!
    </message>
</note>
```

- **eXtensible Markup Language (XML)** is a format for storing nested data with tags and attributes
- essentially, it's HTML, but you can make up any tags and attributes you want
- lots of existing data on the web is stored in XML format

What is XML?

- XML is a "skeleton" for creating markup languages
  - you decide on an XML "language" of tags and attributes that you want to allow in your app
  - XML syntax is mostly identical to HTML's: `<element attribute="value">content</element>`
  - the HTML/XML tag syntax is a nice general syntax for describing hierarchical (nested) data
- when you choose to store data in XML format (or access external XML data), you must decide:
  - names of tags in HTML: `h1`, `div`, `img`, etc.
  - names of attributes in HTML: `id`/`class`, `src`, `href`, etc.
  - rules about how they go together in HTML: inline vs. block-level elements
- XML presents complex data in a human-readable, "self-describing" form

Anatomy of an XML file

```xml
<?xml version="1.0" encoding="UTF-8"?>      <!-- XML prolog -->
<note private="true">                       <!-- root element -->
    <from>Alice Smith (alice@example.com)</from>
    <to>Robert Jones (roberto@example.com)</to>
    <subject>Tomorrow's "Birthday Bash" event!</subject>
    <message language="english">
        Hey Bob, Don't forget to call me this weekend!
    </message>
</note>
```

- begins with an `<?xml ... ?>` header tag (**prolog**)
- has a single **root element** (in this case, `note`)
- tag, attribute, and comment syntax is just like HTML

Uses of XML

- XML data comes from many sources on the web:
  - **web servers** store data as XML files
  - **databases** sometimes return query results as XML
  - **web services** use XML to communicate
- XML is the *de facto* universal format for exchange of data
- XML languages are used for music, math, vector graphics
- popular use: RSS for news feeds & podcasts

What tags are legal in XML?

- *any tags you want!* examples:
  - a library might use tags `book`, `title`, `author`
  - a song might use tags `key`, `pitch`, `note`
- when designing XML data, *you* choose how to best represent the data
  - large or complex pieces of data become tags
  - smaller details and metadata with simple types (integer, string, boolean) become attributes

```
<measure number="1">
    <attributes>
        <divisions>1</divisions>
        <key><fifths>0</fifths></key>
        <time><beats>4</beats></time>
        <clef>
            <sign>G</sign><line>2</line>
        </clef>
    </attributes>
    <note>
        <pitch>
            <step>C</step>
            <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
    </note>
</measure>
```

Schemas and Doctypes

- "rule books" describing which tags/attributes you want to allow in your data
- used to *validate* XML files to make sure they follow the rules of that "flavor"
  - the W3C HTML validator uses an HTML schema to validate your HTML (related to `<!DOCTYPE html>` tag)
- these are optional; if you don't have one, there are no rules beyond having well-formed XML syntax
- for more info:
  - W3C XML Schema
  - Document Type Definition (DTD) ("doctype")

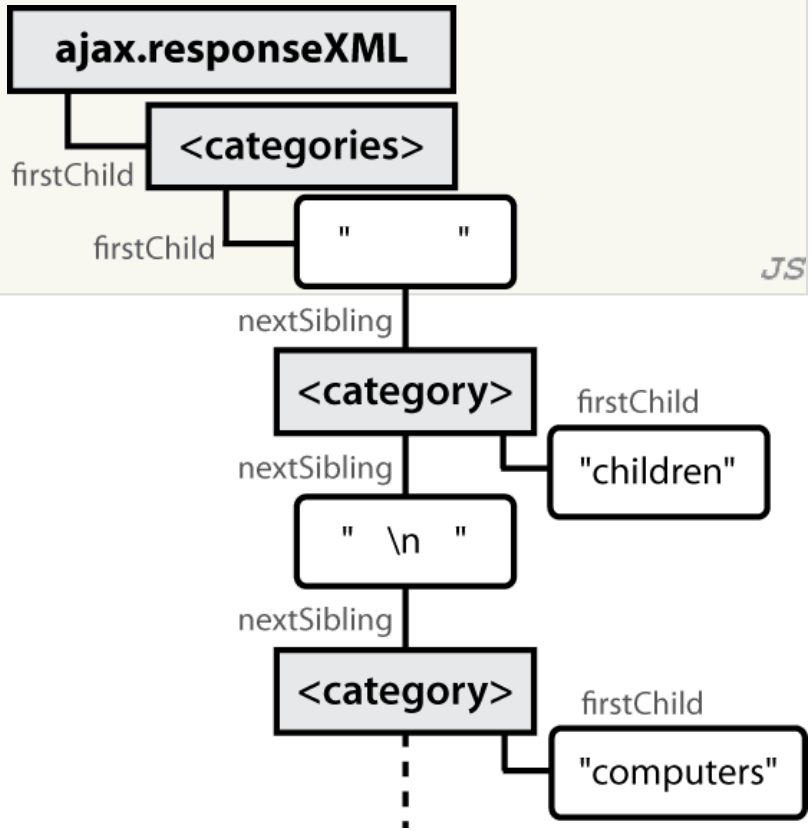
XML and Ajax

- web browsers can display XML files, but often you instead want to fetch one and analyze its data
- the XML data is fetched, processed, and displayed using Ajax
  - (XML is the "X" in "Ajax")
- It would be very clunky to examine a complex XML structure as just a giant string!
- luckily, the browser can break apart (**parse**) XML data into a set of objects
  - there is an XML DOM, similar to the HTML DOM

Fetching XML using Ajax (template)

```
    new Ajax.Request("url", {
        method: "get",
        onSuccess: functionName
    });
    ...

function functionName(ajax) {
    do something with ajax.responseXML;
}
```

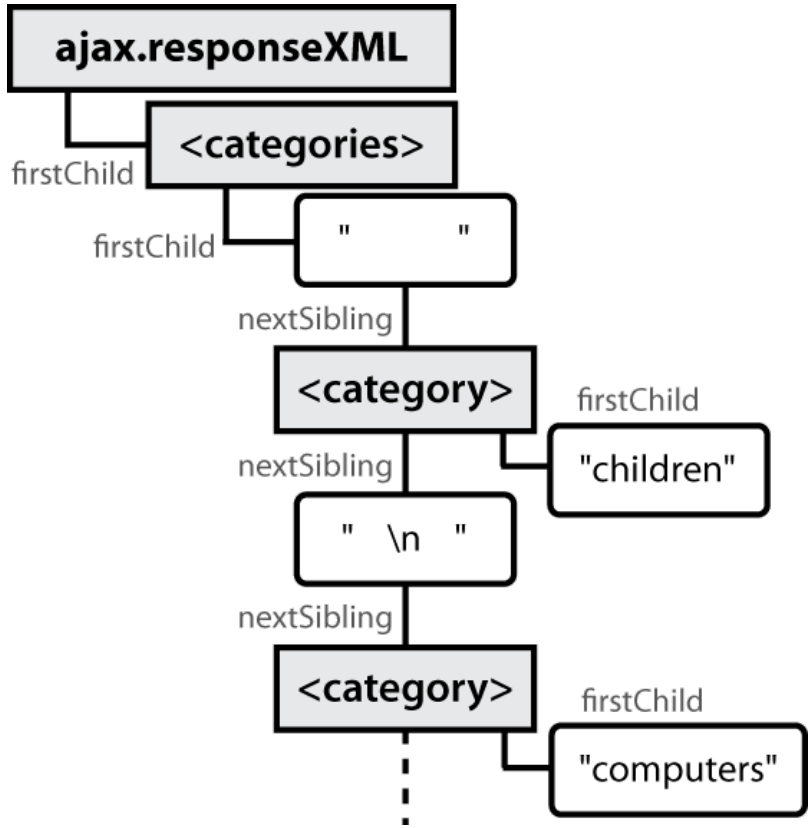


- `ajax.responseText` contains the data in plain text
- `ajax.responseXML` is a parsed XML DOM tree object

XML DOM tree structure

```
<?xml version="1.0" encoding="UTF-8"?>
<categories>
  <category>children</category>
  <category>computers</category>
  ...
</categories>
```

- the XML tags have a tree structure
- DOM nodes have parents, children, and siblings
- each DOM node object has properties/methods for accessing nearby nodes



Interacting with XML DOM nodes

To get a list of all nodes that use a given element:

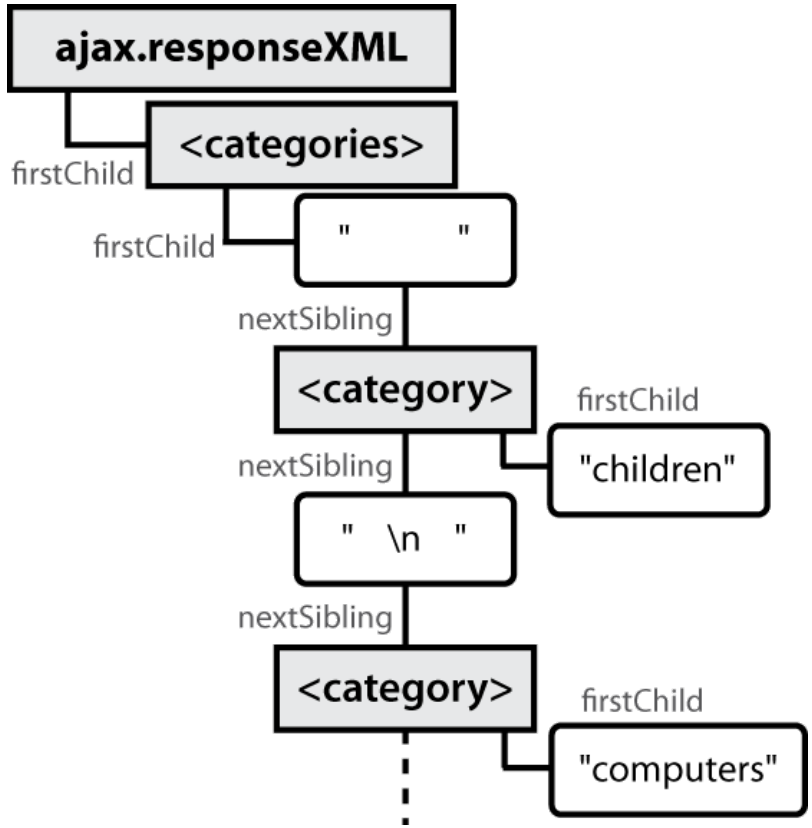
```
var elms = node.getElementsByTagName("tag");
```

To get the text inside of a node:

```
var text = node.firstChild.nodeValue;
```

To get an attribute's value from a node:

```
var attrValue = node.getAttribute("name");
```



Differences from HTML DOM

Can't get a list of nodes by id or class using `$` or `$$`:

```
var elms = ~~$$("#main li");~~   ~~$("id");~~
```

Can't get/set the text inside of a node using `innerHTML`:

```
var text = ~~$("foo").innerHTML~~;
```

Can't get an attribute's value using `.`*attributeName*:

```
var imageUrl = ~~$("myimage").src~~;
```

Full list of XML DOM properties

- properties:
  - **`nodeName`**, **`nodeType`**, **`nodeValue`**, **`attributes`**
  - `firstChild`, `lastChild`, `childNodes`, `nextSibling`, `previousSibling`, `parentNode`
- methods:
  - **`getElementsByTagName`**, **`getAttribute`**, **`hasAttribute[s]`**, **`hasChildNodes`**
  - `appendChild`, `insertBefore`, `removeChild`, `replaceChild`
- Caution:
  - can't use Prototype methods such as `up`, `down`, `ancestors`, `childElements`, or `siblings`
  - can't use HTML-specific properties like innerHTML in the XML DOM

Ajax XML DOM example

```xml
<?xml version="1.0" encoding="UTF-8"?>
<employees>
    <lawyer money="99999.00" />
    <janitor name="Ed"> <vacuum model="Hoover" /> </janitor>
    <janitor name="Bill">no vacuum, too poor</janitor>
</employees>
```

```js
// how much money does the lawyer make?
var lawyer = ajax.responseXML.getElementsByTagName("lawyer")[0];
var salary = lawyer.getAttribute("money");      // "99999.00"

// array of 2 janitors
var janitors = ajax.responseXML.getElementsByTagName("janitor");
var vacModel = janitors[0].getElementsByTagName("vacuum")[0].getAttribute("model");  // "Hoover"
var excuse = janitors[1].firstChild.nodeValue;  // "no vacuum, too poor"
```

- How would we find out the first janitor's name?
- How would we find out how many janitors there are?
- How would we find out how many janitors have vs. don't have vacuums?

Larger XML file example

```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="computers">
        <title lang="en">XQuery Kick Start</title>
        <author>James McGovern</author>
        <year>2003</year><price>49.99</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="computers">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year><price>39.95</price>
    </book>
</bookstore>
```

Navigating node tree example

```
// make a paragraph for each book about computers
var books = ajax.responseXML.getElementsByTagName("book");
for (var i = 0; i < books.length; i++) {
    var category = books[i].getAttribute("category");
    if (category == "computers") {
        // extract data from XML
        var title = books[i].getElementsByTagName("title")[0].firstChild.nodeValue;
        var author = books[i].getElementsByTagName("author")[0].firstChild.nodeValue;

        // make an HTML <p> tag containing data from XML
        var p = document.createElement("p");
        p.innerHTML = title + ", by " + author;
        document.body.appendChild(p);
    }
}
```

Pros and cons of XML

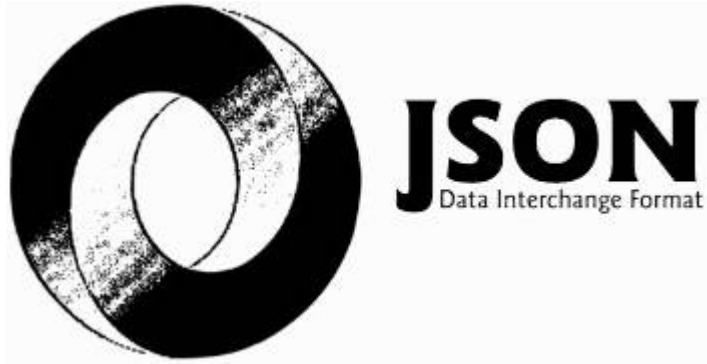
- pro:
  - standard open format; don't have to "reinvent the wheel" for storing new types of data
  - can represent almost any general kind of data (record, list, tree)
  - easy to read (for humans and computers)
  - lots of tools exist for working with XML in many languages
- con:
  - bulky syntax/structure makes files large; can decrease performance (example)
  - can be hard to "shoehorn" data into a good XML format
  - JavaScript code to navigate the XML DOM is bulky and generally not fun

12.4: JSON

JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON): Data format that represents data as a set of JavaScript objects

- invented by JS guru Douglas Crockford of Yahoo!
- natively supported by all modern browsers (and libraries to support it in old ones)
- not yet as popular as XML, but steadily rising due to its simplicity and ease of use



Recall: JavaScript object syntax

```
var person = {
    name: "Philip J. Fry",                          // string
    age: 23,                                        // number
    "weight": 172.5,                                // number
    friends: ["Farnsworth", "Hermes", "Zoidberg"],  // array
    getBeloved: function() { return this.name + " loves Leela"; }
};
alert(person.age);                                 // 23
alert(person["weight"]);                           // 172.5
```

```
alert(person.friends[2]));                          // Zoidberg
alert(person.getBeloved());                         // Philip J. Fry loves Leela
```

JS

- in JavaScript, you can create a new object without creating a class
- the object can have methods (function properties) that refer to itself as `this`
- can refer to the fields with `.`*fieldName* or `["`*fieldName*`"]` syntax
- field names can optionally be put in quotes (e.g. `weight` above)

An example of XML data

```xml
<?xml version="1.0" encoding="UTF-8"?>
<note private="true">
    <from>Alice Smith (alice@example.com)</from>
    <to>Robert Jones (roberto@example.com)</to>
    <to>Charles Dodd (cdodd@example.com)</to>
    <subject>Tomorrow's "Birthday Bash" event!</subject>
    <message language="english">
        Hey guys, don't forget to call me this weekend!
    </message>
</note>
```

The equivalant JSON data

```json
{
    "private": "true",
    "from": "Alice Smith (alice@example.com)",
    "to": [
        "Robert Jones (roberto@example.com)",
        "Charles Dodd (cdodd@example.com)"
    ],
    "subject": "Tomorrow's \"Birthday Bash\" event!",
    "message": {
        "language": "english",
        "text": "Hey guys, don't forget to call me this weekend!"
    }
}
```

Browser JSON methods

method	description
`JSON.parse(`*string*`)`	converts the given string of JSON data into an equivalent JavaScript object and returns it
`JSON.stringify(`*object*`)`	converts the given object into a string of JSON data (the opposite of `JSON.parse`)

- you can use Ajax to fetch data that is in JSON format
- then call `JSON.parse` on it to convert it into an object
- then interact with that object as you would with any other JavaScript object

JSON expressions exercise

Given the JSON data at right, what expressions would produce:

- The window's title?
- The image's third coordinate?
- The number of messages?
- The y-offset of the last message?

```
var title = data.window.title;
var coord = data.image.coords[2];
var len = data.messages.length;
var y = data.messages[len - 1].offset[1];
```

```
var data = JSON.parse(ajax.responseText);
```

```
{
    "window": {
        "title": "Sample Widget",
        "width": 500,
        "height": 500
    },
    "image": {
        "src": "images/logo.png",
        "coords": [250, 150, 350, 400],
        "alignment": "center"
    },
    "messages": [
        {"text": "Save", "offset": [10, 30]}
        {"text": "Help", "offset": [ 0, 50]},
        {"text": "Quit", "offset": [30, 10]},
    ],
    "debug": "true"
}
```

JSON example: Books

Suppose we have a service `books_json.php` about library books.

- If no query parameters are passed, it outputs a list of book categories:

```
{ "categories": ["computers", "cooking", "finance", ...] }
```

- Supply a `category` query parameter to see all books in one category:
  http://selab.hanyang.ac.kr/.../books_json.php?category=cooking

```
{
    "books": [
        {"category": "cooking", "year": 2009, "price": 22.00,
         "title": "Breakfast for Dinner", "author": "Amanda Camp"},
        {"category": "cooking", "year": 2010, "price": 75.00,
         "title": "21 Burgers for the 21st Century", "author": "Stuart Reges"},
        ...
    ]
}
```

JSON exercise

Write a page that processes this JSON book data.

- Initially the page lets the user choose a category, created from the JSON data.
  - ○ Children ○ Computers ○ Finance List Books
- After choosing a category, the list of books in it appears:

  Books in category "Cooking":
  - Breakfast for Dinner, by Amanda Camp (2009)
  - 21 Burgers for the 21st Century, by Stuart Reges (2010)
  - The Four Food Groups of Chocolate, by Victoria Kirst (2005)

Working with JSON book data

```
function showBooks(ajax) {
    // add all books from the JSON data to the page's bulleted list
    var data = JSON.parse(ajax.responseText);
    for (var i = 0; i < data.books.length; i++) {
        var li = document.createElement("li");
        li.innerHTML = data.books[i].title + ", by " +
                data.books[i].author + " (" + data.books[i].year + ")";
        $("books").appendChild(li);
    }
}
```
JS

Bad style: the `eval` function

```
// var data = JSON.parse(ajax.responseText);
var data = eval(ajax.responseText);   // don't do this!
...
```
JS

- JavaScript includes an `eval` keyword that takes a string and runs it as code
- this is essentially the same as what `JSON.parse` does,
- but `JSON.parse` filters out potentially dangerous code; `eval` doesn't
- `eval` is evil and should not be used!