Prototype JS

10.1: Prototype

- **10.1: Prototype**
- 10.2: Scriptaculous

Problems with JavaScript

- JavaScript is a powerful language, but it has many flaws:
- the DOM can be clunky to use
- the same code doesn't always work the same way in every browser
  - code that works great in Firefox, Safari, ... will fail in IE and vice versa
- many developers work around these problems with hacks (checking if browser is IE, etc.)

Prototype framework

```
<script src="http://ajax.googleapis.com/ajax/libs/prototype/1.7.0.0/prototype.js"
 type="text/javascript"></script>
```
JS

- the Prototype JavaScript library adds many useful features to JavaScript:
  - many useful extensions to the DOM
  - added methods to String, Array, Date, Number, Object
  - improves event-driven programming
  - many cross-browser compatibility fixes
  - makes Ajax programming easier (seen later)

jQuery
write less, do more.

The $ function

```
$("id")
```

- returns the DOM object representing the element with the given id
- short for **`document.getElementById("id")`**
- often used to write more concise DOM code:

```
$("footer").innerHTML = $("username").value.toUpperCase();
```

Prototype's DOM element methods

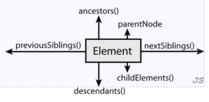
absolutize	**addClassName**	**classNames**	cleanWhitespace	clonePosition
cumulativeOffset	cumulativeScrollOffset	empty	extend	firstDescendant
getDimensions	getHeight	getOffsetParent	**getStyle**	getWidth
hasClassName	**hide**	identify	insert	inspect
makeClipping	makePositioned	match	positionedOffset	readAttribute
recursivelyCollect	relativize	**remove**	**removeClassName**	replace
scrollTo	select	setOpacity	setStyle	**show**
toggle	toggleClassName	undoClipping	undoPositioned	update
viewportOffset	visible	wrap	writeAttribute	

- categories: CSS classes, DOM tree traversal/manipulation, events, styles

Prototype's DOM tree traversal methods

method(s)	description
ancestors, up	elements above this one
childElements, descendants, down	elements below this one (not text nodes)
siblings, next, nextSiblings, previous, previousSiblings, adjacent	elements with same parent as this one (not text nodes)

```
// alter siblings of "main" that do not contain "Sun"
var sibs = $("main").siblings();
for (var i = 0; i < sibs.length; i++) {
  if (sibs[i].innerHTML.indexOf("Sun") < 0) {
    sibs[i].innerHTML += " Sunshine";
  }
}
```
JS



- Prototype strips out the unwanted text nodes
- notice that these are methods, so you need ()

Prototype's methods for selecting elements

- methods in document and other DOM objects for accessing descendents:

name	description
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)

- Prototype adds methods to the *document object (and all DOM element objects)* for selecting *groups of elements*:

getElementsByClassName	array of elements that use given class attribute
select	array of descendants that match given CSS selector, such as "div#sidebar ul.news > li"

```
var gameButtons = $("game").select("button.control");
for (var i = 0; i < gameButtons.length; i++) {
  gameButtons[i].style.color = "yellow";
}
```
JS

The $$ function

- $$ returns an array of DOM elements that match the given CSS selector
  - like $ but returns an array instead of a single DOM object
  - a shorthand for **document.select**
- useful for applying an operation to each one of a set of elements

```
var arrayName = $$("CSS selector");
```

```
// hide all "announcement" paragraphs in the "news" section
var paragraphs = $$("div#news p.announcement");
for (var i = 0; i < paragraphs.length; i++) {
  paragraphs[i].hide();
}
```

Problems with reading/changing styles

- style property lets you set any CSS style for an element

```
<button id="clickme">Click Me</button>
```
HT

```
window.onload = function() {
  $("clickme").onclick = biggerFont;
};
function biggerFont() {
  var size = parseInt($("clickme").style.fontSize);
  size += 4;
  $("clickMe").style.fontSize = size + "pt";
}
```

```
Click Me
```
outp

- **getStyle** function added to DOM object allows accessing existing styles

```
function biggerFont() {
  // turn text yellow and make it bigger
  var size = parseInt($("clickme").getStyle("font-size"));
  $("clickme").style.fontSize = (size + 4) + "pt";
}
```

```
Click Me
```
o

Setting CSS classes in Prototype

```
function highlightField() {
  // turn text yellow and make it bigger
  if (!$("text").hasClassName("invalid")) {
    $("text").addClassName("highlight");
  }
}
```
JS

- **addClassName**, **removeClassName**, **hasClassName** manipulate CSS classes
- similar to existing className DOM property, but don't have to manually split by spaces

Prototype form shortcuts

```
$F("formID")["name"]
```
JS

- gets parameter with given **name** from form with given **id**

```
$F("controlID")
```
JS

- $F function returns the **value** of a form control with the given **id**

```
if ($F("username").length < 4) {
  $("username").clear();
  $("login").disable();
}
```
JS

Stopping an event

```
<form id="exampleform" action="http://foo.com/foo.php">...</form>
```
HTML

```
window.onload = function() {
  $("exampleform").observe("submit", checkData);
};

function checkData(event) {
  if ($F("city") == "" || $F("state").length != 2) {
    alert("Error, invalid city/state.");  // show error message
    event.stop();
    return false;
  }
}
```
JS

- to abort a form submit or other event, call Prototype's stop method on the event

Classes and prototypes

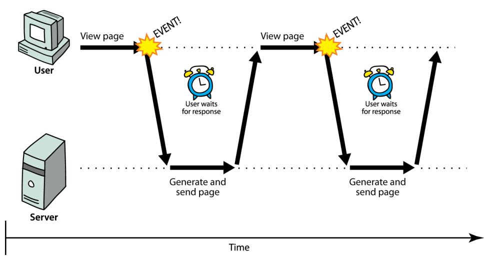
- limitations of prototype-based code:
  - unfamiliar / confusing to many programmers
  - somewhat unpleasant syntax
  - difficult to get inheritance-like semantics (subclassing, overriding methods)
- Prototype library's Class.create method makes a new class of objects
  - essentially the same as using prototypes, but uses a more familiar style and allows for richer inheritance semantics

AJAX, XML and JSON

12.1: Ajax Concepts

- **12.1: Ajax Concepts**
- 12.2: Using XMLHttpRequest
- 12.3: XML
- 12.4: JSON

Synchronous web communication

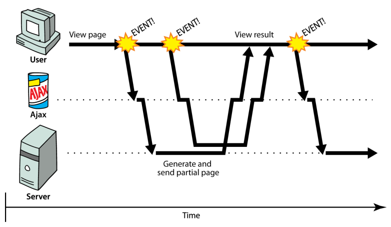


- **synchronous**: user must wait while new pages load
  - the typical communication pattern used in web pages (click, wait, refresh)

Web applications and Ajax

- web application: a dynamic web site that mimics the feel of a desktop app
  - a client–server software application in which the client (or user interface) runs in a web browser;
  - 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

Asynchronous web communication



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

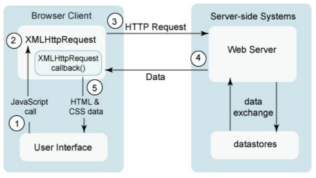
12.2: Using XMLHttpRequest

XMLHttpRequest

- 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!...

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
   1. this is often called a **callback**
   2. 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:
  - the **URL** to fetch, as a String,
  - 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"
     ...
   }
);
```

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
  }
);
```

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)

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

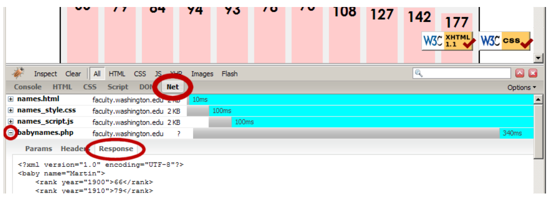
most commonly used property is **`responseText`**, to access the fetched text content

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

Debugging Ajax code



```
names.html        faculty.washington.edu  2 KB  10ms
names_style.css   faculty.washington.edu  2 KB  100ms
names_script.js   faculty.washington.edu  2 KB  100ms
babynames.php     faculty.washington.edu  ?     340ms
```

```
<?xml version="1.0" encoding="UTF-8"?>
<baby name="Martin">
    <rank year="1900">66</rank>
    <rank year="1910">79</rank>
```

- **Net** tab shows each request, its parameters, response, any errors
- expand a request with **+** and look at **Response** tab to see Ajax result

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)
- any query parameters should be passed as a **`parameters`** parameter
  - written between {} braces as a set of *name* : *value* pairs (another anonymous object)
  - get request parameters can also be passed this way, if you like

Prototype's Ajax Updater

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

- Ajax.Updater fetches a file and injects its content into an element as **`innerHTML`**
- additional (1st) parameter specifies the **`id`** of element to inject into
- **`onSuccess`** handler not needed (but **`onFailure`**, **`onException`** handlers may still be useful)

PeriodicalUpdater

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

- 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)

Ajax.Responders

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

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

12.3: XML

- 12.1: Ajax Concepts
- 12.2: Using XMLHttpRequest
- **12.3: XML**
- 12.4: JSON

The bad way to store data

```
My note:
BEGIN
  TO: Tove
  FROM: Jani
  SUBJECT: Reminder
  MESSAGE (english):
    Hey there,
    Don't forget to call me this weekend!
END
```

- we could send a file like this from the server to browser with Ajax
- what's wrong with this approach?

What is XML?

- **XML**: a "skeleton" for creating markup languages
- you already know it!
  - syntax is identical to XHTML's

```
<element attribute="value">content</element>
```

  - languages written in XML specify:
  - names of tags in XHTML: h1, div, img, etc.
  - names of attributes in XHTML: id/class, src, href, etc.
  - rules about how they go together in XHTML: inline vs. block-level elements
- used to present complex data in human-readable form
  - "self-describing data"

Anatomy of an XML file

```
<?xml version="1.0" encoding="UTF-8"?>  <!-- XML prolog -->
<note>                                  <!-- root element -->
  <to>Tove</to>
  <from>Jani</from>                     <!-- element ("tag") -->
  <subject>Reminder</subject>           <!-- content of element -->
  <message language="english">          <!-- attribute and its value -->
    Don't forget me this weekend!
  </message>
</note>
```
XML

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

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

Pros and cons of XML

- pro:
  - easy to read (for humans and computers)
  - standard format makes automation easy
  - don't have to "reinvent the wheel" for storing new types of data
  - international, platform-independent, open/free standard
  - can represent almost any general kind of data (record, list, tree)
- con:
  - bulky syntax/structure makes files large; can decrease performance
    - example: quadratic formula in MathML
  - can be hard to "shoehorn" data into a good XML format

What tags are legal in XML?

- *any tags you want!*
- examples:
  - an email message might use tags called to, from, subject
  - a library might use tags called book, title, author
- when designing an XML file, *you* choose the tags and attributes that best represent the data
- rule of thumb: data = tag, metadata = attribute

Doctypes and Schemas

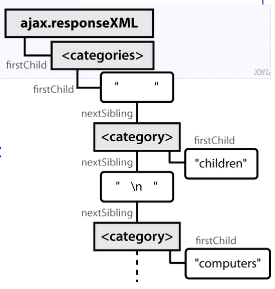
- "rule books" for individual flavors of XML
  - list which tags and attributes are valid in that language, and how they can be used together
- used to *validate* XML files to make sure they follow the rules of that "flavor"
  - the W3C HTML validator uses the XHTML doctype to validate your HTML
- for more info:
  - Document Type Definition (DTD) ("doctype")
  - W3C XML Schema
- optional — if you don't have one, there are no rules beyond having well-formed XML syntax
- (we won't cover these any further here)

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, very similar to the (X)HTML DOM

XML DOM tree structure

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



- the XML tags have a tree structure
- DOM nodes have parents, children and sib

Recall: Javascript XML (XHTML) DOM

- The DOM properties and methods* we already know can be used on XML nodes:
- properties:
  - firstChild, lastChild, childNodes, nextSibling, previousSibling, parentNode
  - **nodeName**, **nodeType**, **nodeValue**, **attributes**
- methods:
  - appendChild, insertBefore, removeChild, replaceChild
  - **getElementsByTagName**, **getAttribute**, **hasAttributes**, **hasChildNodes**
- caution: cannot use HTML-specific properties like innerHTML in the XML DOM!
- * (though not Prototype's, such as up, down, ancestors, childElements, or siblings)

Navigating the node tree

- caution: can *only* use standard DOM methods/properties in XML DOM (**NOT Prototype's**)
- caution: can't use ids or classes to use to get specific nodes (no $ or $$). Instead:

```
// returns all child tags inside node that use the given element
var elms = node.getElementsByTagName("tagName");
```
JS

Using XML data in a web page

1. use Ajax to fetch data
2. use DOM methods to examine XML:
   ***XMLnode*.getElementsByTagName("*tag*")**
3. extract the data we need from the XML:
   ***XMLelement*.getAttribute("*name*")**,
   ***XMLelement*.firstChild.nodeValue**, etc.
4. create new HTML nodes and populate with extracted data:
   document.createElement("*tag*"),
   ***HTMLelement*.innerHTML**
5. inject newly-created HTML nodes into page
   ***HTMLelement*.appendChild(*element*)**

Fetching XML using AJAX (template)

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

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

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

Analyzing a fetched XML file using DOM

```
<?xml version="1.0" encoding="UTF-8"?>
<employees>
  <lawyer money="5"/>
  <janitor name="Sue"><vacuumcleaner/></janitor>
  <janitor name="Bill">too poor</janitor>
</employees>
```
XML

- We can use DOM properties and methods on ajax.responseXML:

```
// zeroth element of array of length 1
var employeesTag = ajax.responseXML.getElementsByTagName("employees")[0];

// how much money does the lawyer make?
var lawyerTag = employeesTag.getElementsByTagName("lawyer")[0];
var salary = lawyerTag.getAttribute("money");   // "5"

// array of 2 janitors
var janitorTags = employeesTag.getElementsByTagName("janitor");
var excuse = janitorTags[1].firstChild.nodeValue;  // " too poor "
```
JS

Larger XML file example

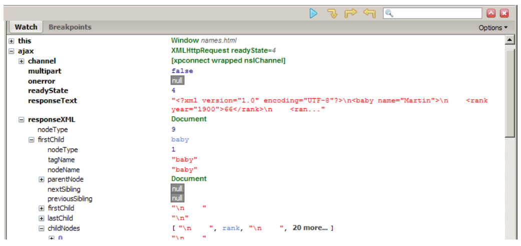
```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year><price>30.00</price>
  </book>
  <book category="computers">
    <title lang="en">XQuery Kick Start</title>
    <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 XHTML <p> tag containing data from XML
    var p = document.createElement("p");
    p.innerHTML = title + ", by " + author;
    document.body.appendChild(p);
  }
}
```
JS

Debugging responseXML in Firebug



```
⊞ this                       Window names.html
⊟ ajax                       XMLHttpRequest readyState=4
  ⊞ channel                  [xpconnect wrapped nsIChannel]
    multipart                false
    onerror                  null
    readyState               4
    responseText             "<?xml version="1.0" encoding="UTF-8"?>\n<baby name="Martin">\n    <rank
                             year="1900">66</rank>\n    <ran..."
  ⊟ responseXML              Document
      nodeType               9
    ⊟ firstChild             baby
        nodeType             1
        tagName              "baby"
        nodeName             "baby"
      ⊞ parentNode           Document
        nextSibling          null
        previousSibling      null
      ⊞ firstChild           "\n    "
      ⊞ lastChild            "\n"
      ⊟ childNodes           [ "\n    ", rank, "\n    ", 20 more... ]
        ⊞ 0                  "\n    "
```

- can examine the entire XML document, its node/tree structure

12.4: JSON

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

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~~

JSON Data Interchange Format

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 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>
```
XML

The equivalent JSON data

```
{
  "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 ofJSON.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 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

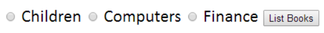
```
{ "categories": ["computers", "cooking", "finance", ...] }
```
JSON

- Supply a category query parameter to see all books in one category: http://webster.cs.washington.edu/books_json.php?category=cooking

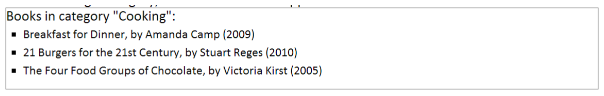
```
{
  "books": [
    {"category": "cooking", "year": 2009, "price": 22.00,
     "title": "Breakfast for Dinner", "author": "Amanda Camp"},
    {"category": "cooking", "year": 2010, "price": 75.00,
     "title": "21 Burgers for the 21st Century", "author": "Stuart Reges"},
    ...
  ]
}
```
JSON

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.



- 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);
  }
}
```

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!