



Ajax for Java Developers

The Java Developer Education Series

Workshop Overview

- ◆ This is an in-depth course covering the use of Ajax (Asynchronous JavaScript and XML) to build Rich Internet Applications using Java on the server side
- ◆ It includes coverage of:
 - The JavaScript technology that is the foundation for Ajax
 - How to integrate Ajax with server side Java technologies
 - The role of XML in Ajax, and alternatives to using XML
 - A number of open source toolkits and technologies for Ajax
- ◆ The workshop includes many hands-on lab exercises, including a series of brief labs
 - Many of the labs follow a common fictional case study - JavaTunes, an online music store

Workshop Objectives

- ◆ At completion you should:
 - Understand the principles of interactive Web applications and how Ajax is used to create them
 - Understand how **XMLHttpRequest** works, and use it with JavaScript to update a Web page
 - Use Servlets/JSP to handle Ajax requests
 - Understand **JSON** (JavaScript Object Notation)
 - Use **JavaScript/DOM/Ajax** to manipulate Web page structure
 - Be familiar with Ajax technologies and frameworks such as **Prototype**, **script.aculo.us**, **Dojo**, and **JSON** libraries
 - Understand the basics of **CSS** and use it with Ajax
 - Use Ajax with **HTML/JSON/XML** on the client and server side
 - Use **Direct Web Remoting** (DWR) and other RPC technologies
 - Use Ajax with **JSF**
 - Understand issues with using Ajax technology

Workshop Agenda

- ◆ Session 1: **Ajax Overview**
- ◆ Session 2: **JavaScript Basics**
- ◆ Session 3: **XMLHttpRequest**
- ◆ Session 4: **Servlets and JSP for Ajax**
- ◆ Session 5: **More JavaScript for Ajax**
- ◆ Session 6: **Client Side Ajax Frameworks**
- ◆ Session 7: **Cascading Style Sheets (CSS)**
- ◆ Session 8: **JavaScript Object Notation (JSON)**
- ◆ Session 9: **XML and Ajax**
- ◆ Session 10: **DWR – Direct Web Remoting**
- ◆ Session 11: **Ajax and JavaServer Faces (JSF)**
- ◆ Session 12: **Ajax Design and Security**

Course Prerequisites

- ◆ Basic knowledge of HTML
- ◆ Practical Java and Servlet/JSP programming for the Java material
 - We'll review Servlet/JSP programming basics if required
- ◆ Some knowledge of JavaScript helpful
 - We'll review JavaScript basics if required

- ◆ This manual has been tested, and contains complete instructions, for running the labs using the following platforms:
 - **Tomcat 6 or 7**
 - **Java 6 or 7**
 - **Eclipse Java EE Edition**

- ◆ All labs have been tested on Microsoft Windows



Session 1: Ajax Overview

Rich Internet Applications
Ajax Introduction

Lesson Objectives

- ◆ Understand the current Web development trends
- ◆ Understand what Ajax is, and how it meets current needs for Web application development

Rich Internet Applications

Rich Internet Applications

Ajax Introduction

What are Rich Internet Applications

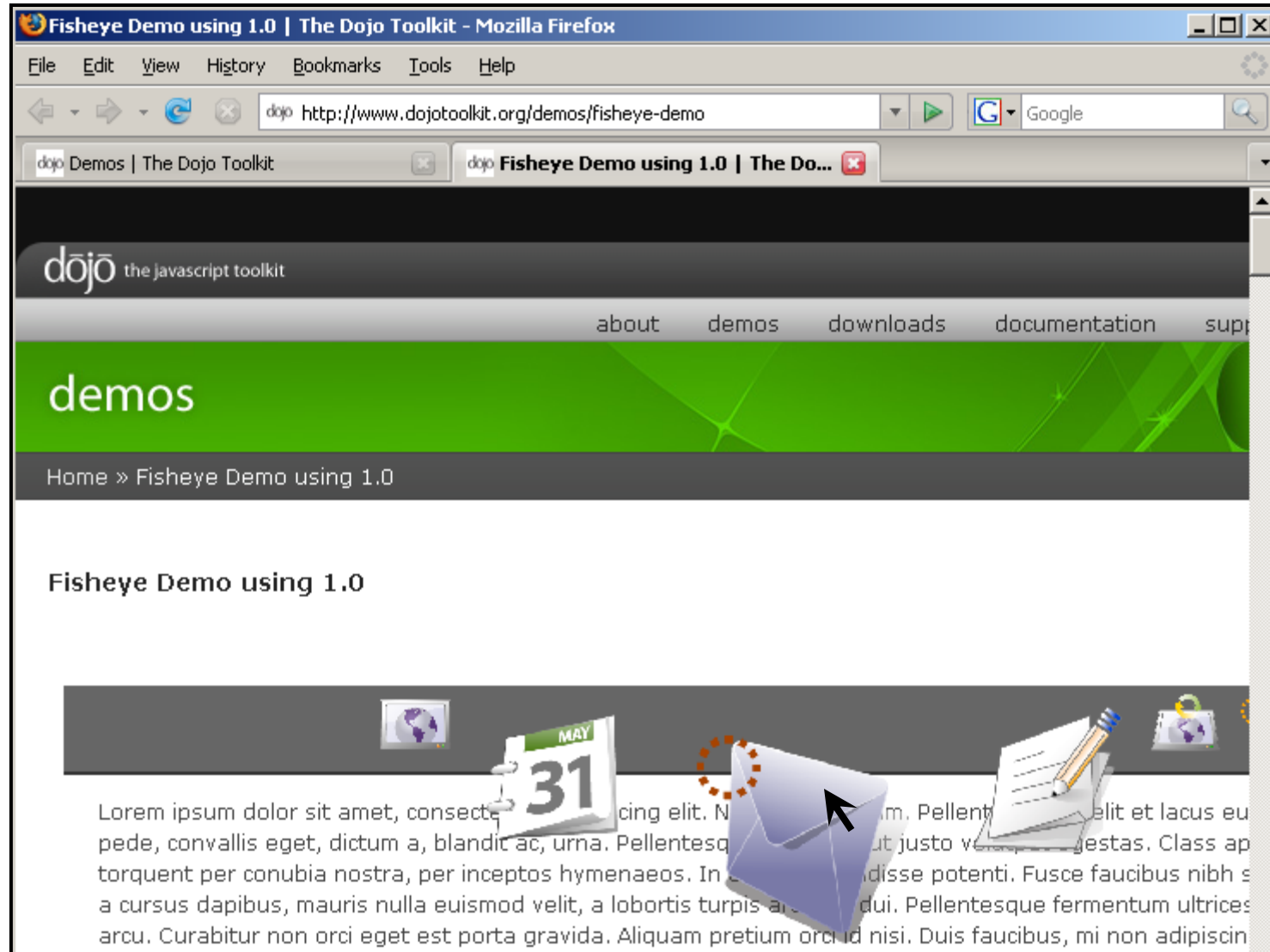
- ◆ **Rich Internet Applications (RIA)** attempt to provide the characteristics of traditional desktop applications using a browser as the client interface
 - Fast response time
 - Fast and responsive interface giving meaningful feedback to user (e.g. tooltips for icons, or table cells changing color when you hover over them)
 - Richer and more powerful selection of widgets such as sliders, drag-and-drop, calculations done on the client
- ◆ Different from traditional Web applications where user submits requests, and waits for new page from server
 - User has to wait for response from server
 - Workflow/interaction based on pages
 - Slower, less interactive, and less intuitive as you lose your current page context with each submission

RIA Technologies

- ◆ There are many technologies that support RIA. For example:
 - **Adobe Flash/Flex** – Powerful, cross platform UI technology – proprietary API requires Flash plug-in which is very widely supported
 - **Java Applets** – Full fledged programming language. Requires Java plug-in
 - **DHTML** (Dynamic HTML) – Based on programming browser with JavaScript. Supported in all major browsers.
 - **Ajax** – Adds asynchronous communication to DHTML. Supported in all major browsers
 - All these are part of what is commonly called "**Web 2.0**"
- ◆ The next slide shows an example of a widget from the Dojo toolkit that provides an animated menu using DHTML
 - Dojo is a popular open-source JavaScript framework

Dojo FisheyeList Widget Demo

- ◆ Provides a menu similar to the fish eye menu on the Mac OS



Ajax Introduction

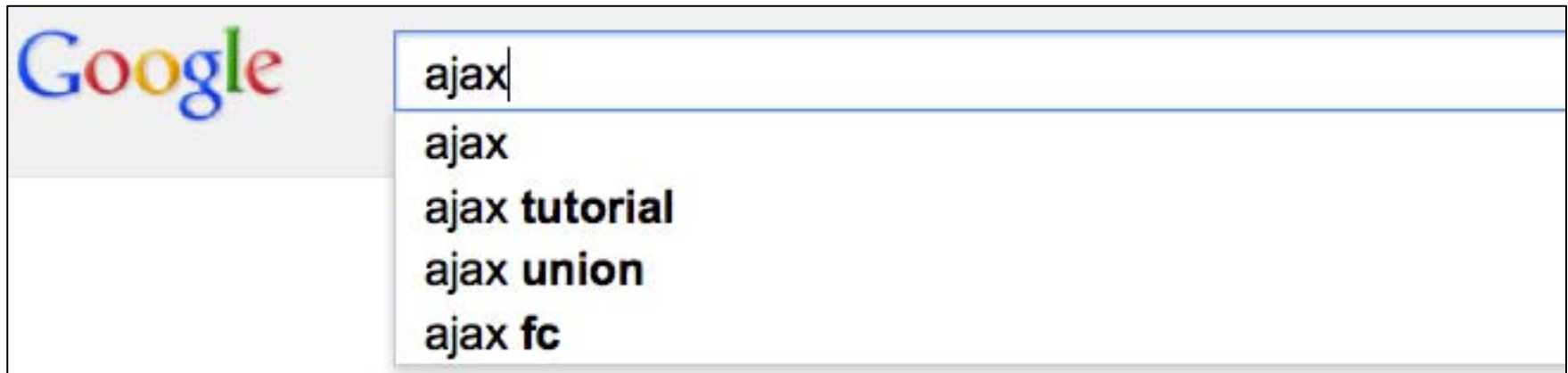
Rich Internet Applications
Ajax Introduction

What is Ajax

- ◆ **Asynchronous JavaScript and XML**
 - A web development technique for creating interactive web applications
 - Makes web applications feel more responsive by exchanging small amounts of data with the web server behind the scenes
 - Only affected parts of a web page are updated
 - The entire page does not need to be refreshed for an update
 - Intended to increase the web pages interactivity, speed, and usability
 - One of many technologies that make up RIA
- ◆ Term was first used in public in 2005 by **Jesse James Garret**
 - Based on technologies that have been in existence for years
 - What is new is the many prominent applications now being built using these technologies

Ajax Example – Google Search

- ◆ Uses Ajax to retrieve potential search matches as you type
 - Below, we show the suggestions it makes when you type ajax
 - Google search can also populate the page with results as you type
 - At bottom, we show the Ajax traffic on the XHR tab of Firebug



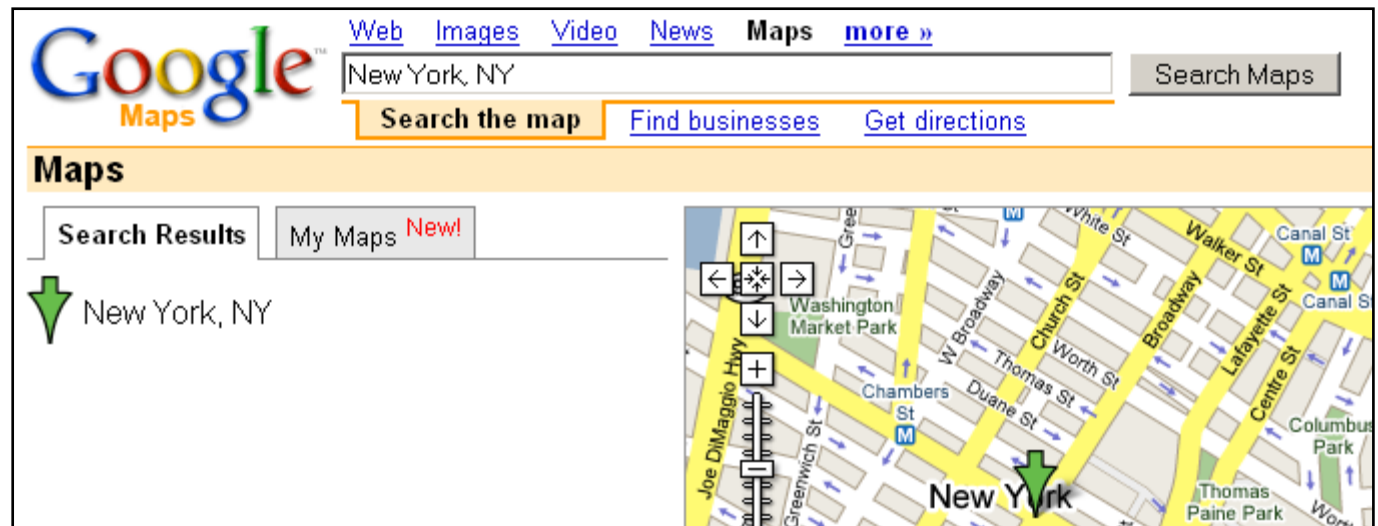
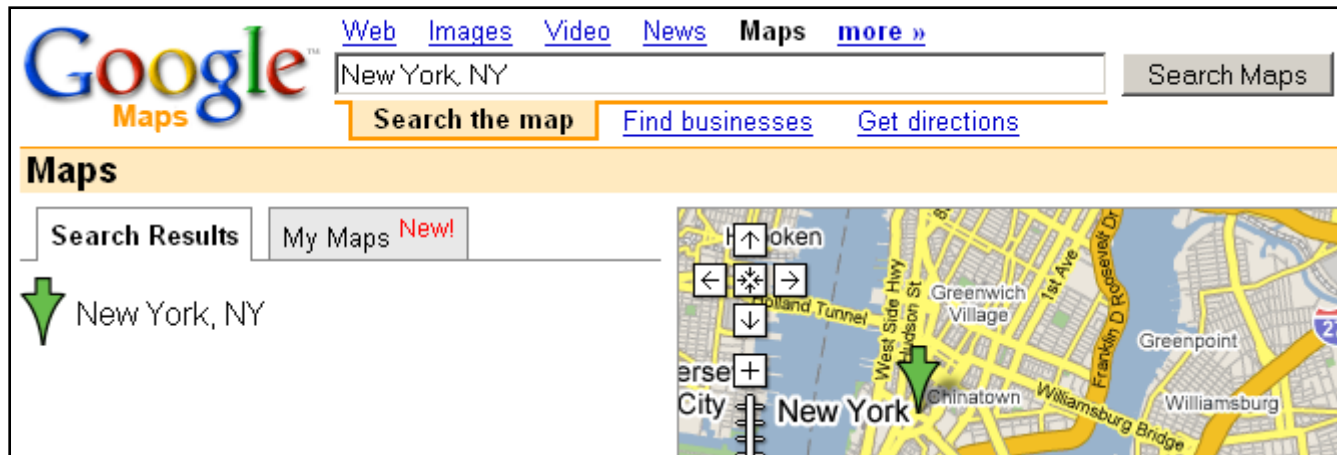
xhr	Clear	Persist	All	HTML	CSS	JS	XHR	Images	Flash	Media
URL		Status	Domain	Size	Remote IP	T				
+	GET s?hl=en&sugexq	200 OK	google.com	535 B	173.194.75.105:80					
+	GET s?hl=en&sugexq	200 OK	google.com	541 B	173.194.75.105:80					
+	GET s?hl=en&sugexq	200 OK	google.com	536 B	173.194.75.105:80					
+	GET s?hl=en&sugexq	200 OK	google.com	535 B	173.194.75.105:80					
+	GET s?hl=en&sugexq	200 OK	google.com	553 B	173.194.75.105:80					
Requests				2.6 KB						

RIA/Ajax Example – Google Maps

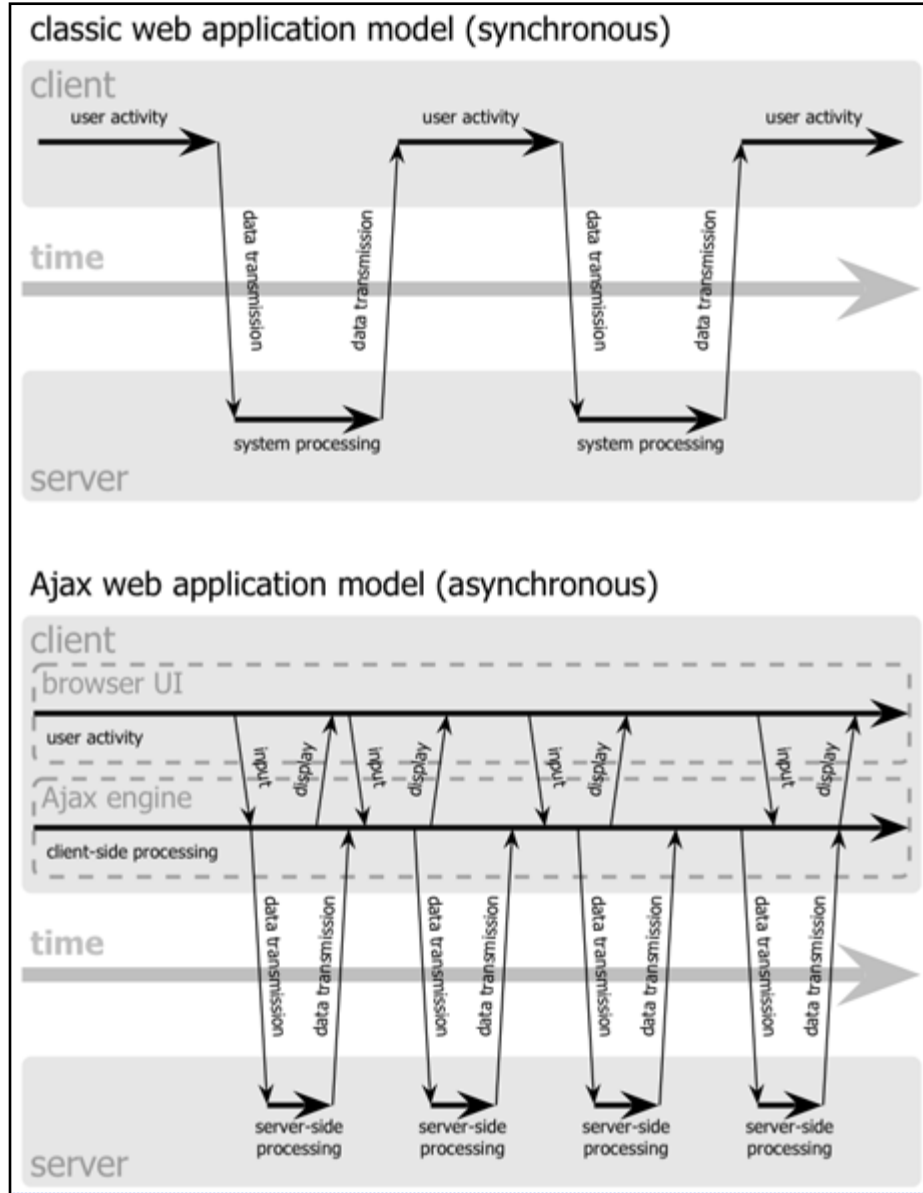
- ◆ In Google Maps, users can interactively drag the map around
 - Rather than working in a traditional way, such as clicking a submit button
 - This uses advanced JavaScript/DHTML techniques
- ◆ As the map is dragged, or you are zooming in or out, the browser is making Ajax requests
 - And downloading new data in response to the dragging
 - The data is downloaded in the background, without any interruption to the user experience
 - The new data is used to update part of the map display when it is received
- ◆ Only the map data changes
 - Other parts of the page remain the same

RIA/Ajax Example – Google Maps

- ◆ Here, we've zoomed in, and shown map (alone) updated
 - Go to <http://maps.google.com> to try dragging the map



The Ajax Difference



◆ **Classic:** Users wait for complete page refresh

– Flow interrupted

◆ **Ajax:** Data fetched in background

– Users continue to work

Diagram from Jesse James Garrets classic "name defining" paper on Ajax

Ajax, JavaScript, DHTML and More

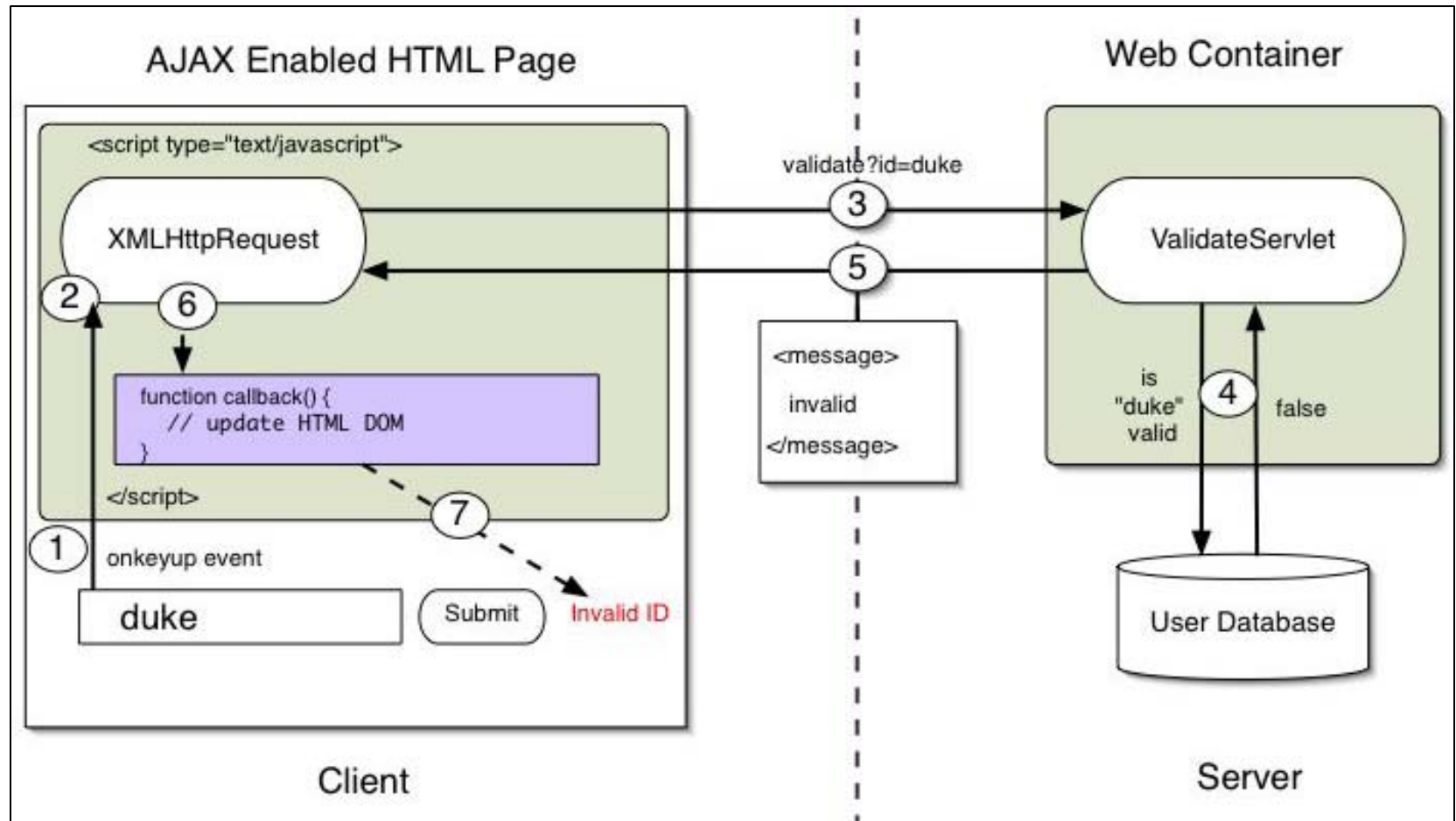
- ◆ Ajax does not refer to a single technology
 - We use it to refer to the user interface pattern described earlier
 - The updating of small parts of a web page via data retrieved via scripted HTTP requests
- ◆ Many different technologies may be used with Ajax, including:
 - **HTML/XHTML** and **Cascading Style Sheets (CSS)**: Presentation and presentation style
 - **JavaScript**: Scripting language tying technologies together
 - **XMLHttpRequest**: JavaScript object that performs asynchronous interaction with the server
 - **Document Object Model (DOM)**: Browser based object model allowing dynamic, programmatic manipulation of the web page
 - **XML/XSLT**: One data format and manipulation choice
 - **JavaScript Object Notation (JSON)**: Alternative data format

XMLHttpRequest

- ◆ **XMLHttpRequest** is a JavaScript object that is at the core of Ajax applications
 - It allows HTTP requests to be made to the server using JavaScript
 - It does **not** require a page refresh
 - You can register an event handler (a JavaScript function) that responds to the XMLHttpRequest request lifecycle
 - This event handler can be used to update the web page when data is available in a response from the request
- ◆ The server side can use any server technology
 - The difference being in the data that the server returns
 - It will return a chunk of data that satisfies the request, and not a complete web page
 - The form of the data may vary – XML, HTML, JSON are all used

XMLHttpRequest Example

- Below, XMLHttpRequest is used to validate input on each keypress
 - On the server, ValidateServlet processes the request and returns an XML document, which is processed in the browser via JavaScript



What This Course Will Focus On

- ◆ We cover the programming aspects of using Ajax
 - How JavaScript is used in conjunction with Ajax
 - How to make Ajax requests and process response data
 - How to handle Ajax requests on the server side using Java
 - How to use common Ajax frameworks that make life easier
 - These topics are the main focus of the course
- ◆ We cover usage of core presentation technologies that are commonly used with Ajax
 - HTML, DOM, DHTML and CSS
 - We cover them enough so that you understand how to use them
 - We don't cover them in depth, or show all the neat tricks that can be done with them
- ◆ The Ajax technology basket is large
 - It can't be covered in depth in one short course



Lab 1.1 – Setting Up the Environment

- ◆ **Overview:** In this lab, we will setup the lab environment, and create and deploy a simple Web application
 - The end goal is to get everything running, and use Eclipse to build and deploy a simple Web application to a Tomcat server
- ◆ **Objectives:**
 - Become familiar with the lab structure
 - Set up our Eclipse environment and Tomcat server
 - Deploy a working Web application to the Tomcat server
- ◆ **Builds on previous labs:** None
- ◆ **Approximate Time:** 25-35 minutes

- ◆ Within a lab, information only content is presented in the normal way – the same as in the student manual pages
 - Like these bullets at the top of the page
- ◆ Tasks that the student needs to perform are in a box with a slightly different look – to help you identify them
- ◆ An example appears below

Tasks to Perform

- ◆ Look at these instructions, and notice the different look of the box as compared to that above
 - Make a note of how it looks, as future labs will use this format
- ◆ OK – Now **get out your setup files**; we're ready to start working

Extract the Lab Setup Zip File

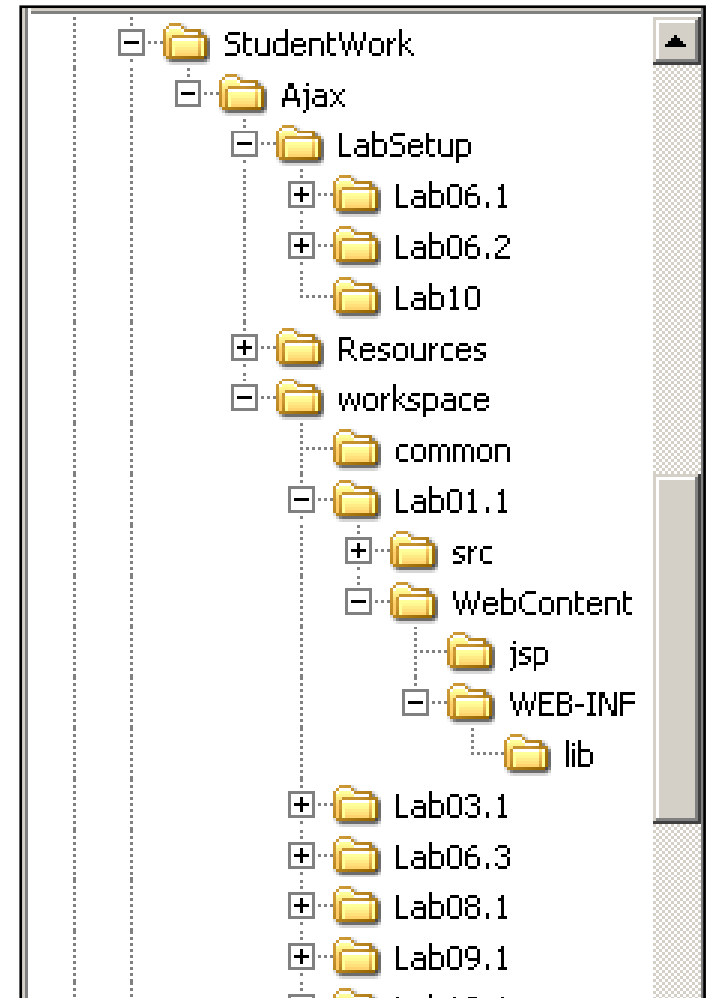
- ◆ To set up the labs, you'll need the course setup zip file *
 - It has a name like: **LabSetup_AjaxJava_Tomcat_20120920.zip**
- ◆ Our base working directory for this course will be **C:\StudentWork\Ajax**
 - This directory will be created when we extract the Setup zip
 - It includes a directory structure and files (e.g., Java files, XML files, other files) that will be needed in the labs
 - All instructions assume that this zip file is extracted to C:\. **If you choose a different directory, please adjust accordingly**

Tasks to Perform

- ◆ Unzip the lab setup file to **C:**
 - This will create the directory structure, described in the next slide, containing files that you will need for doing the labs

Lab Directory Structure

- ◆ **StudentWork\Ajax** will contain the following folders:
 - **LabSetup**: files needed for lab work
 - **LabNN**: directory for any lab with additional setup files
 - **Resources** : Extra files such as documentation
 - **workspace**: Lab directories
 - **common**: shared files
 - **LabNN** : Directory for Lab NN
 - **LabNN/build/** : compiled code (standard location for Eclipse)
 - **LabNN/src/** : Java source
 - **LabNN/WebContent/** : jsp, HTML
 - **LabNN/WEB-INF/**: web.xml



Tasks to Perform

- ◆ Make sure that you have Java installed
 - Likely installed in a folder like *C:\Program Files\Java\jdk1.7.x **
- ◆ Make sure that Tomcat is installed – likely in a directory such as **C:\apache-tomcat-7.0.30**
 - If it's been installed in a different directory, you'll need to modify the instructions in the lab to refer to your install directory
 - If it isn't installed, you'll need to download then install it
 - Download it from **<http://tomcat.apache.org/download-70.cgi>**
- ◆ Make sure that Eclipse is installed – likely in a directory such as **C:\eclipse**

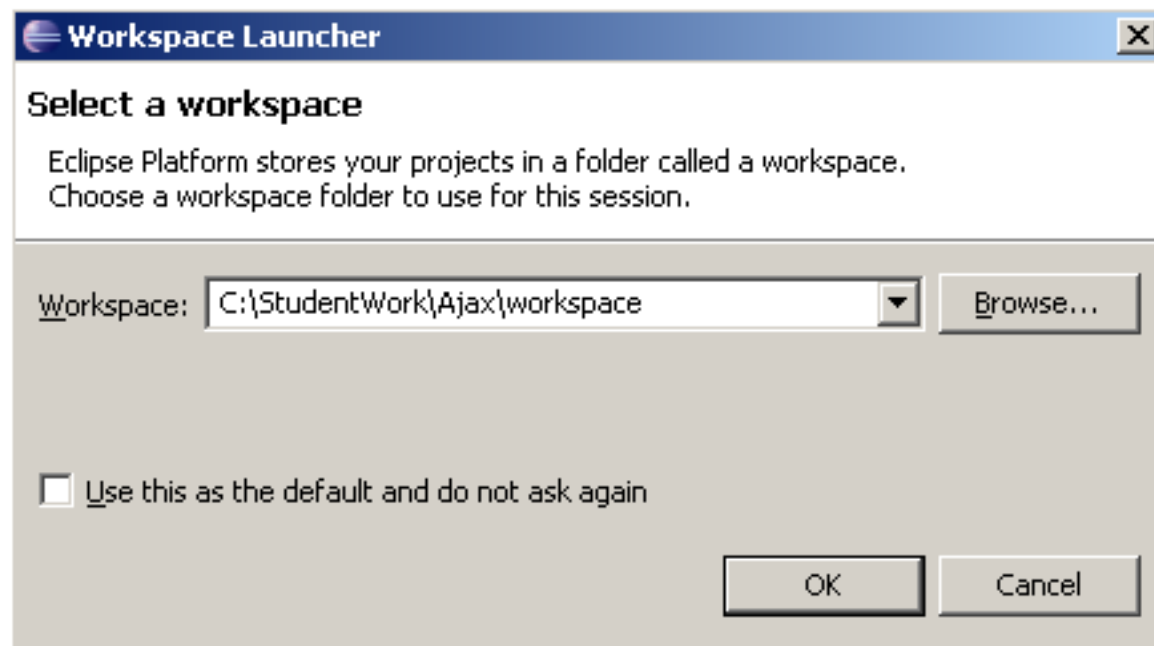
- ◆ **Eclipse** (www.eclipse.org) is an open source platform for building integrated development environments (IDEs) -
 - Used mainly for Java development - can be extended via plugins and used in other areas (e.g. C# programming)
 - Originally developed by IBM, then released into open source
- ◆ Eclipse products have two fundamental layers
 - The **Workspace** – files, packages, projects, resource connections, configuration properties
 - The **Workbench** – editors, views, and perspectives
- ◆ The remainder of this lab gives **detailed instructions on using Eclipse** to run the labs
 - The other labs **do not include detailed Eclipse instructions**
 - For these labs, you should use the same procedures to build/deploy as in this lab - refer to this lab as needed

- ◆ We'll use the **Eclipse Java EE** edition in this class
 - Has support for Java Web applications
- ◆ Eclipse organizes Java Web apps using **Dynamic Web Projects**
 - **Dynamic** Web projects contain Java EE resources such as servlets, JSP pages, plus static resources (HTML)
 - You establish project properties for the Web Project at creation time and can modify them later
 - It provides a custom editor for the *web.xml* file
- ◆ The Eclipse Web project organization is different from how the final Java EE Web application organization will be
 - It is designed to make it easy for you to work with the resources
 - When deployed, a standard WAR is build

- ◆ Organized in the following folders
 - **src**: Contains all Java source files
 - **WebContent**: Contains all Web resources
 - **WebContent\WEB-INF**: Same as Java EE WEB-INF
- ◆ Usually use **Web Perspective** or **J2EE Perspective**
- ◆ All visible elements **are not necessarily deployed** with the project
 - e.g. the src folder is not deployed - only compiled classes
 - Eclipse creates a standard WAR file when it deploys
- ◆ Before Web components are developed you must create and configure a new Web Project
 - You can specify the build path for the project to include external jar's or class files
- ◆ When Web projects are created, Eclipse automatically creates the associated Deployment Descriptor (DD)
 - **web.xml** - DD for Web project in *WEB-INF* folder

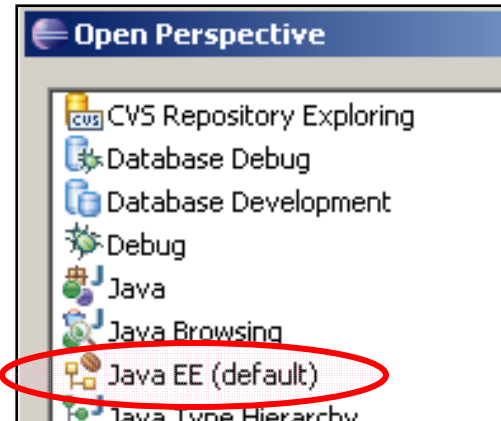
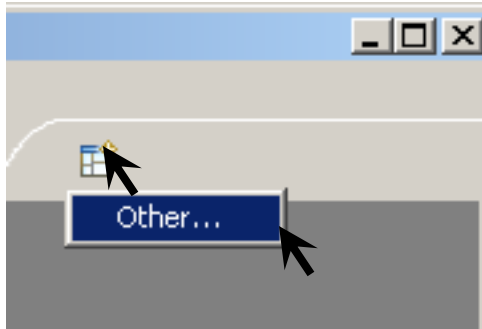
Tasks to Perform

- ◆ Make sure you have **Eclipse installed** - likely in *C:\eclipse*
- ◆ **Launch Eclipse**: Go to *c:\eclipse* and run **eclipse.exe**
 - A dialog box should appear prompting for a workspace location
 - Set the workbench location to **C:\StudentWork\Ajax\workspace**
 - If a different default Workbench location is set, change it
 - Click **OK**



Tasks to Perform

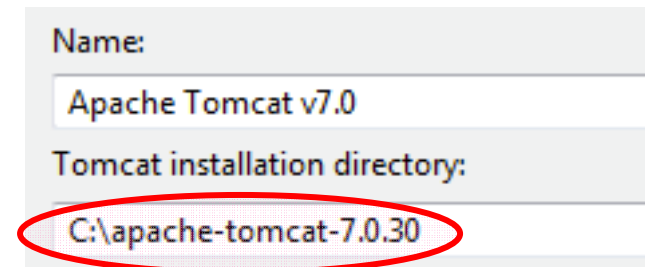
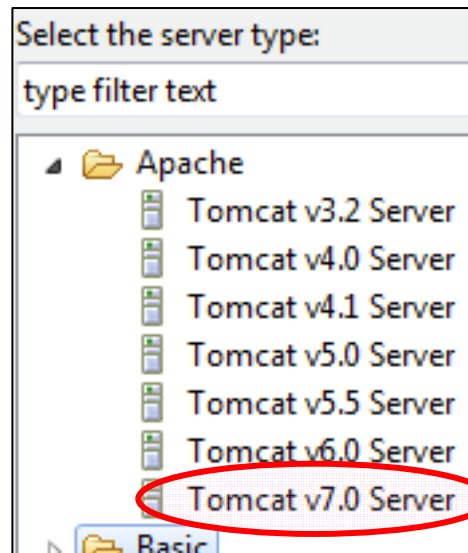
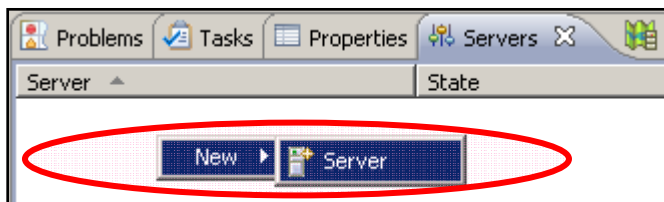
- ◆ Close the the welcome screen, (click the **X** in the upper right)
- ◆ You'll likely be in a Java EE perspective - depending on which Eclipse version you use
 - That's fine, this perspective is good for what we do
- ◆ If you need to open the Java EE perspective, (**Shouldn't need to do this now**) you can do so by clicking the Perspective icon at the top right of the Workbench, and select **Other | Java EE** (as shown below)



- ◆ We will use the Tomcat server to run our Web applications - to do this, we first need to create a server in Eclipse *

Tasks to Perform

1. Go to the Servers view, right click, and select **New | Server**
2. In the next dialog, select **Apache | Tomcat V7.0 *** and click **Next**
3. In the next dialog, browse to your **Tomcat 7 install directory**, and click **Finish** *

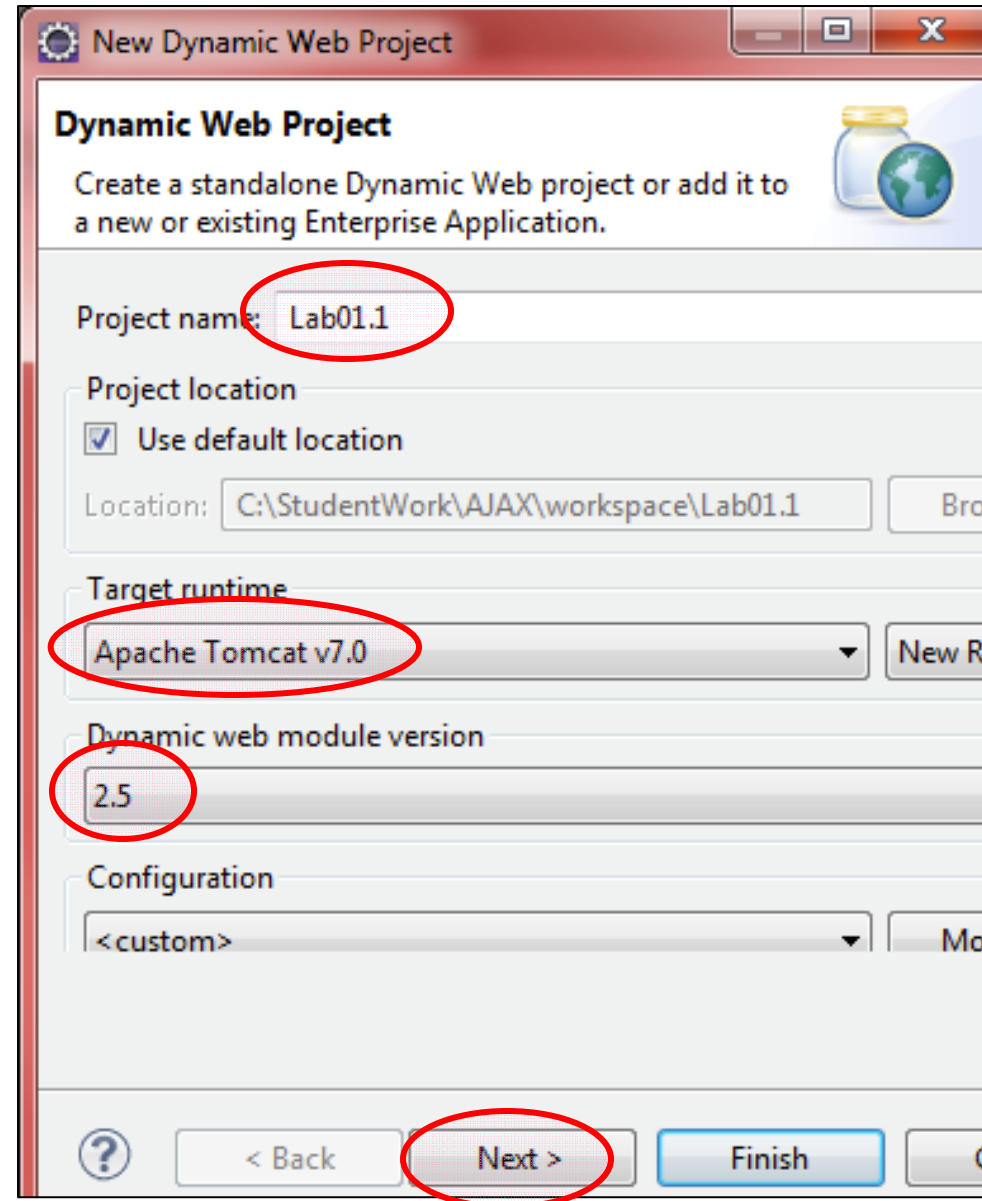
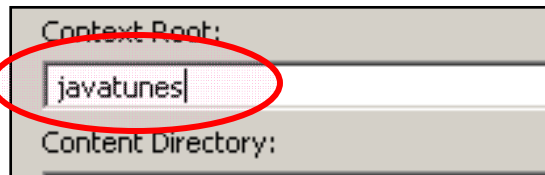


Create a Web Project

Lab

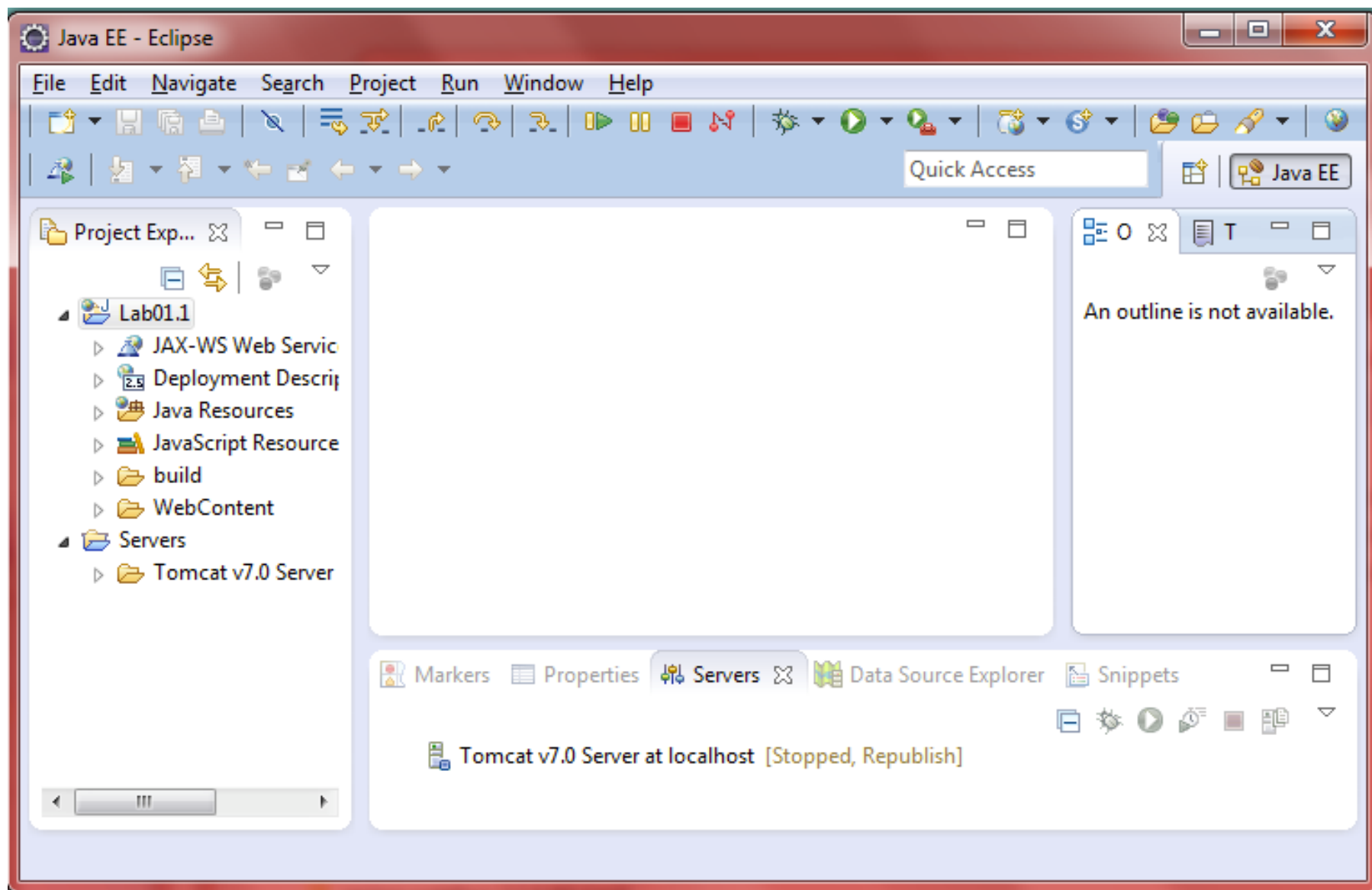
Tasks to Perform

- ◆ Create a new Dynamic Web project (as at right)
 - **File | New | Dynamic Web Project** *
 - Call it **Lab01.1** *
 - Select your Tomcat 7 server
 - Make sure the **2.5 module version** is selected *
- ◆ Click **Next** - until the Web module dialog, change the Context Root to **javatunes** (all lower case)
 - Click **Finish**



The Java EE Perspective

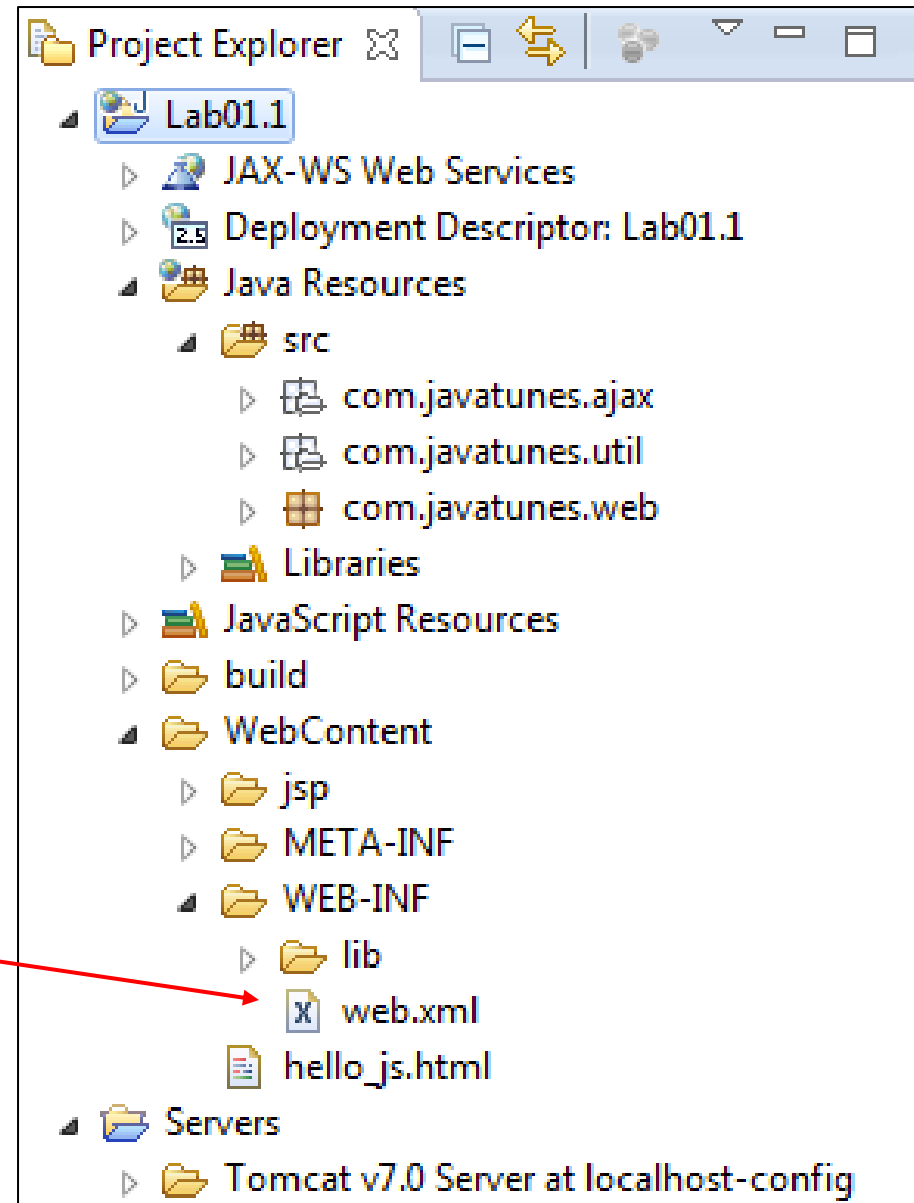
Lab



The Project Explorer View

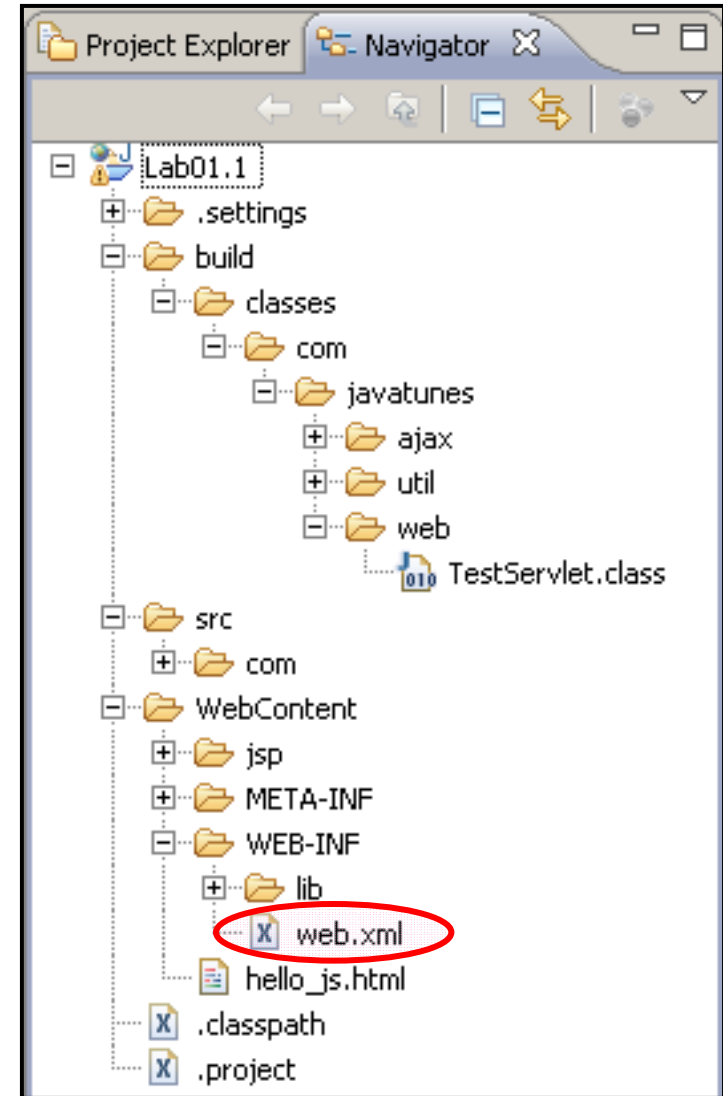
Lab

- ◆ Open the Project Explorer View
- ◆ Java EE oriented display
 - Not file oriented
 - Organized into groups based on type of project
 - Resources in a project are displayed in a view specific way
 - For example the *web.xml* deployment descriptor



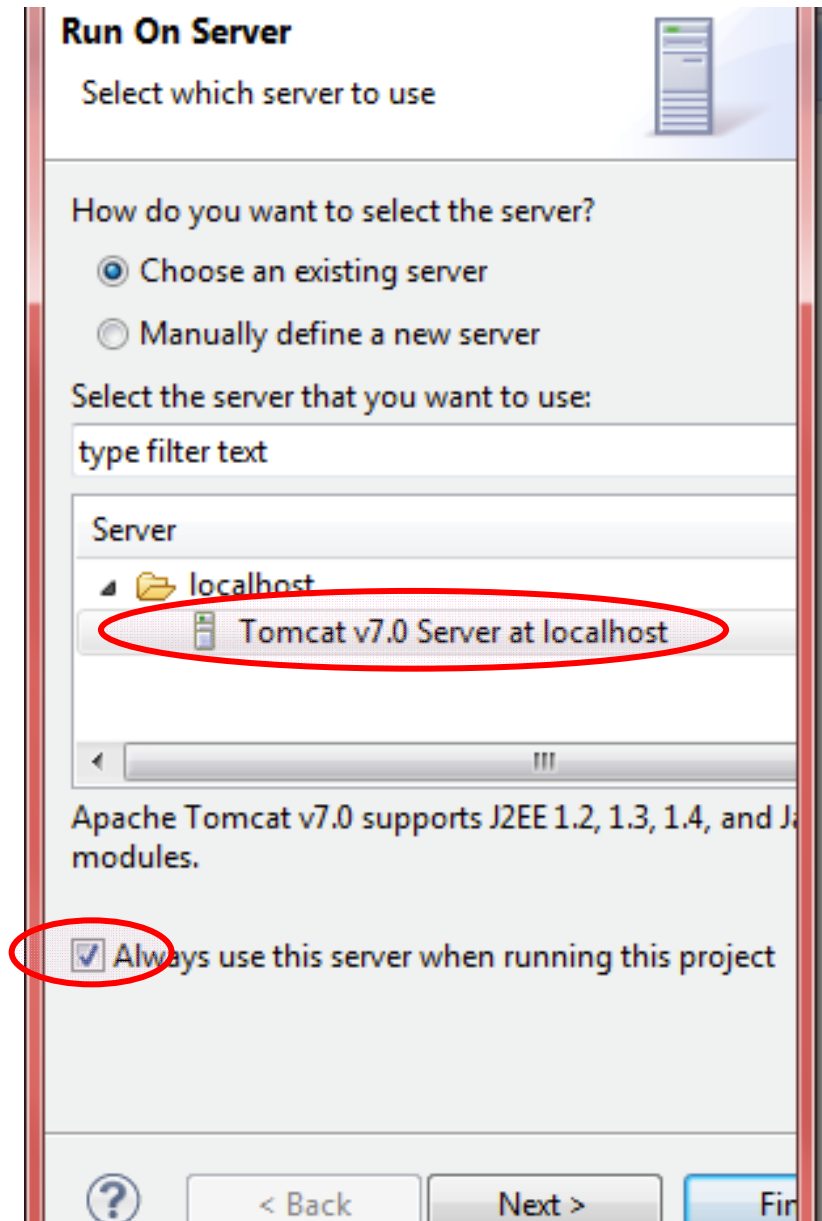
Tasks to Perform

- ◆ Open the Navigator View (**Window | Show View | Navigator**)
- ◆ Look at the Navigator view to see the Web project you just created
- ◆ File system – like view
 - Organizes Java source, Web content
 - Knows about deployment descriptors
- ◆ Note the deployment descriptor, **web.xml**, that is supplied for you
 - **Double click on web.xml** to open it for viewing and editing



Tasks to Perform

- ◆ To deploy to the server, right click on the Lab01.1 project, and select **Run As | Run on Server**
 - ◆ In the next dialog box select the existing **Tomcat v7.0** server
 - ◆ Also select **Always use this server when running this project**
 - ◆ Click **Finish**
-
- ◆ Note that running a Web app on the server will automatically start the server

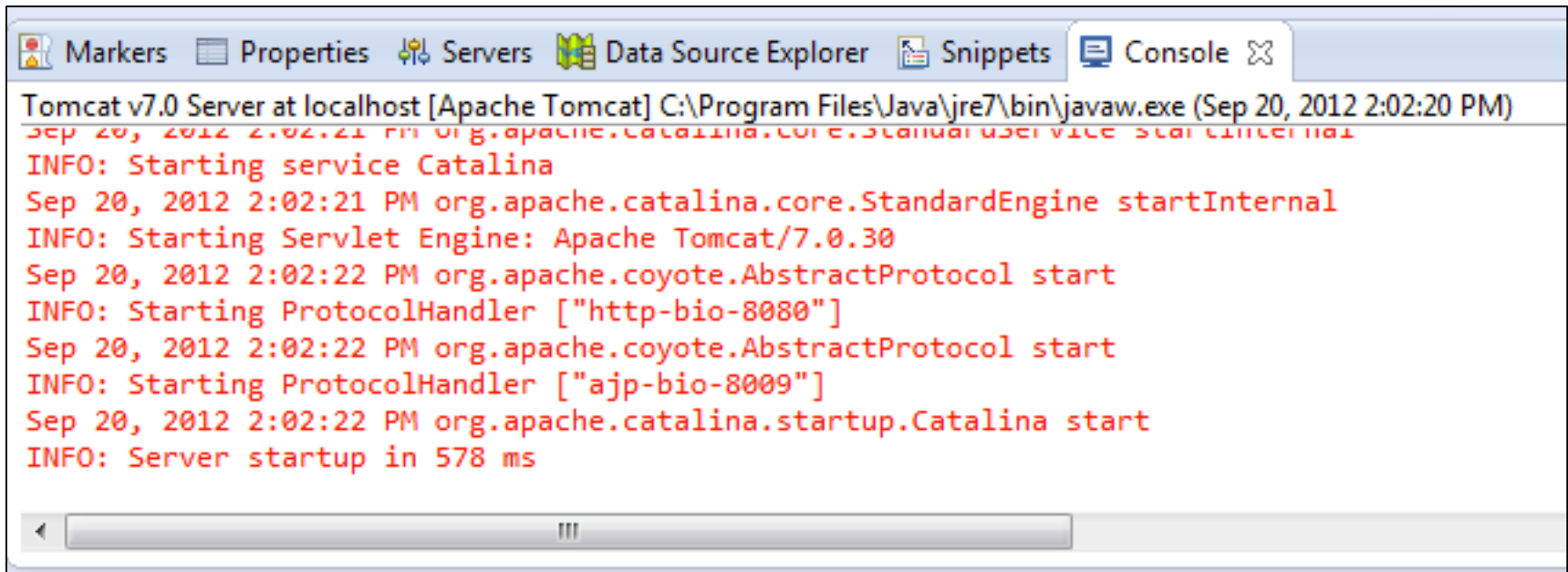


Viewing the Web Application

- ◆ Eclipse will automatically open a Web browser for you onto the Web application
 - Note that the built in browser (IE) **is sometimes misleading** because it caches Web pages, and it's hard to clear the cache
 - Also - **sometimes the browser window comes up before the server has loaded the Web app** - try a reload if a resource can't be accessed
 - If you ever feel you're having browser issues, open an external browser viewing the same URL (note that you'll see *hello_js.html* displayed in the browser, since it's specified as the welcome file in *web.xml*)
 - That's it – your Web app is up and running



- ◆ You can open the **Console** view to see output from the server startup
 - This is useful to look for exception stack traces in later labs
 - Note that server startup may take some time, especially the first time you start the server
 - You can also look at the server status in the **Servers** view



```
Tomcat v7.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jre7\bin\javaw.exe (Sep 20, 2012 2:02:20 PM)
Sep 20, 2012 2:02:21 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting service Catalina
Sep 20, 2012 2:02:21 PM org.apache.catalina.core.StandardEngine startInternal
INFO: Starting Servlet Engine: Apache Tomcat/7.0.30
Sep 20, 2012 2:02:22 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
Sep 20, 2012 2:02:22 PM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["ajp-bio-8009"]
Sep 20, 2012 2:02:22 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 578 ms
```

Important Things to Note for Eclipse

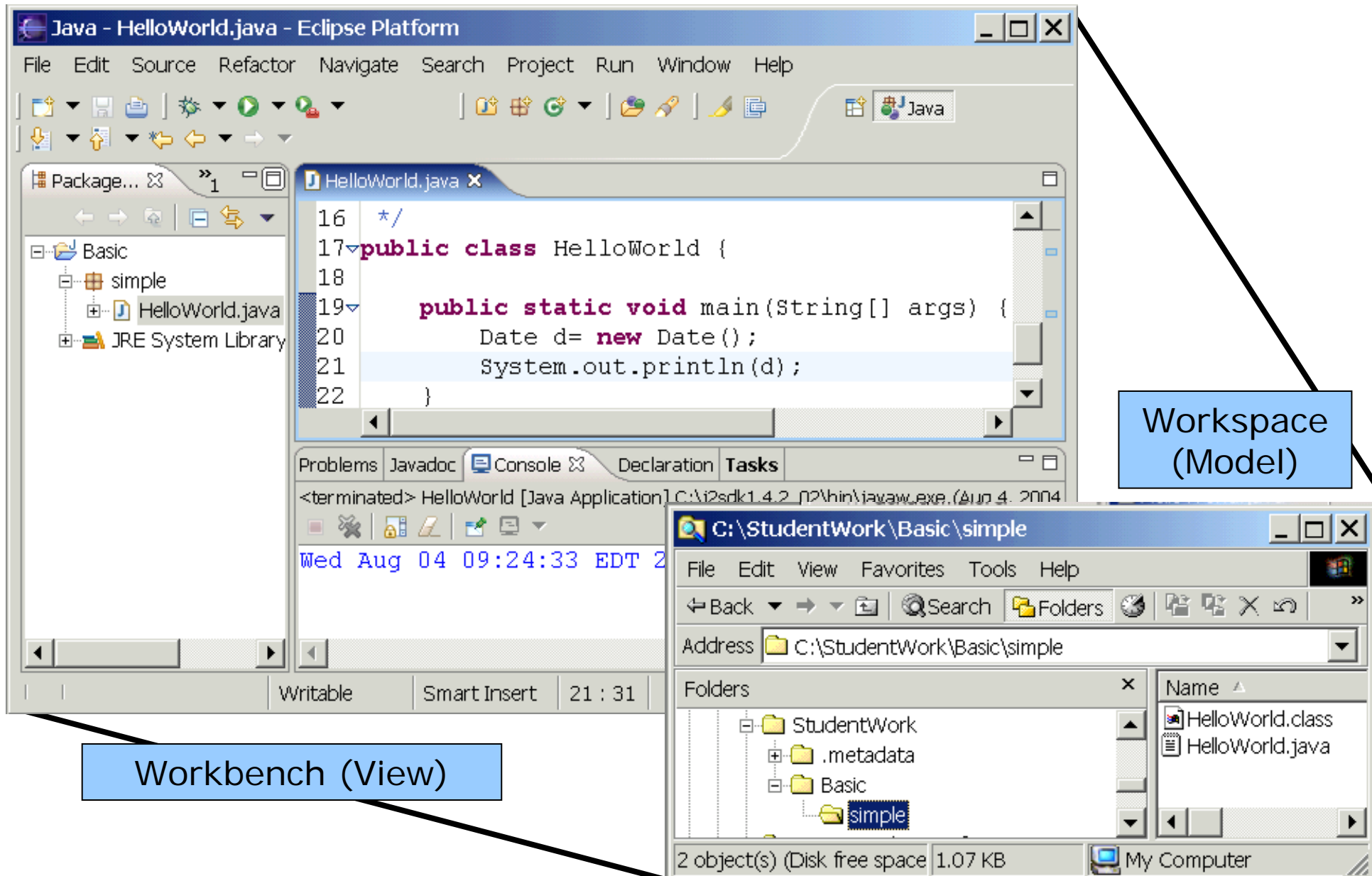
Lab

- ◆ Each lab that has a **separate lab directory** will require you to create **a new Eclipse project**
 - Sometimes several labs are done in one directory, in which case you can use the same project for all of them
- ◆ Anytime you **COPY** files into a project (e.g. from setup) you need to **Refresh** (Right click on the project, select **Refresh**)
- ◆ For anyone not familiar with Eclipse, the next few slides give a (very) brief overview of how Eclipse is structured
 - There is nothing you need to do in those slides – they are for information purposes only

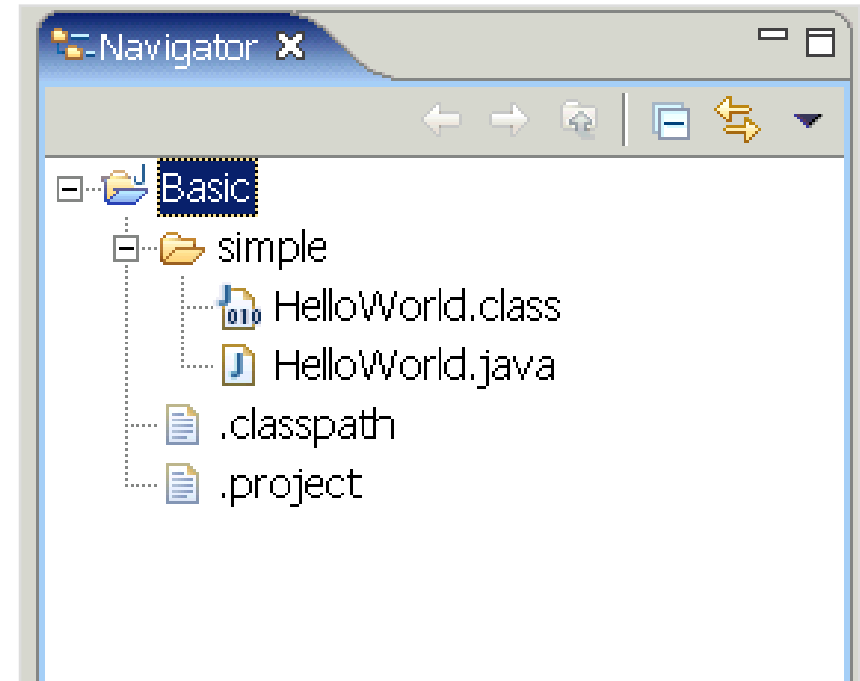
- ◆ Eclipse products have two fundamental layers
 - The **Workspace** – files, packages, projects, resource connections, configuration properties
 - The **Workbench** – editors, views, and perspectives
- ◆ The Workbench sits on top of the Workspace and provides visual artifacts that allow you to access and manipulate various aspects of the underlying resources, such as:
 - **Editor** – A component that allows a developer to interact with and modify the contents of a file.
 - **View** – A component that exposes meta-data about the currently selected resource.
 - **Perspective** – A grouping of related editors and views that are relevant to a particular task and/or role.
- ◆ You can have multiple perspectives open to provide access to different aspects of the underlying resources

Workbench and Workspace

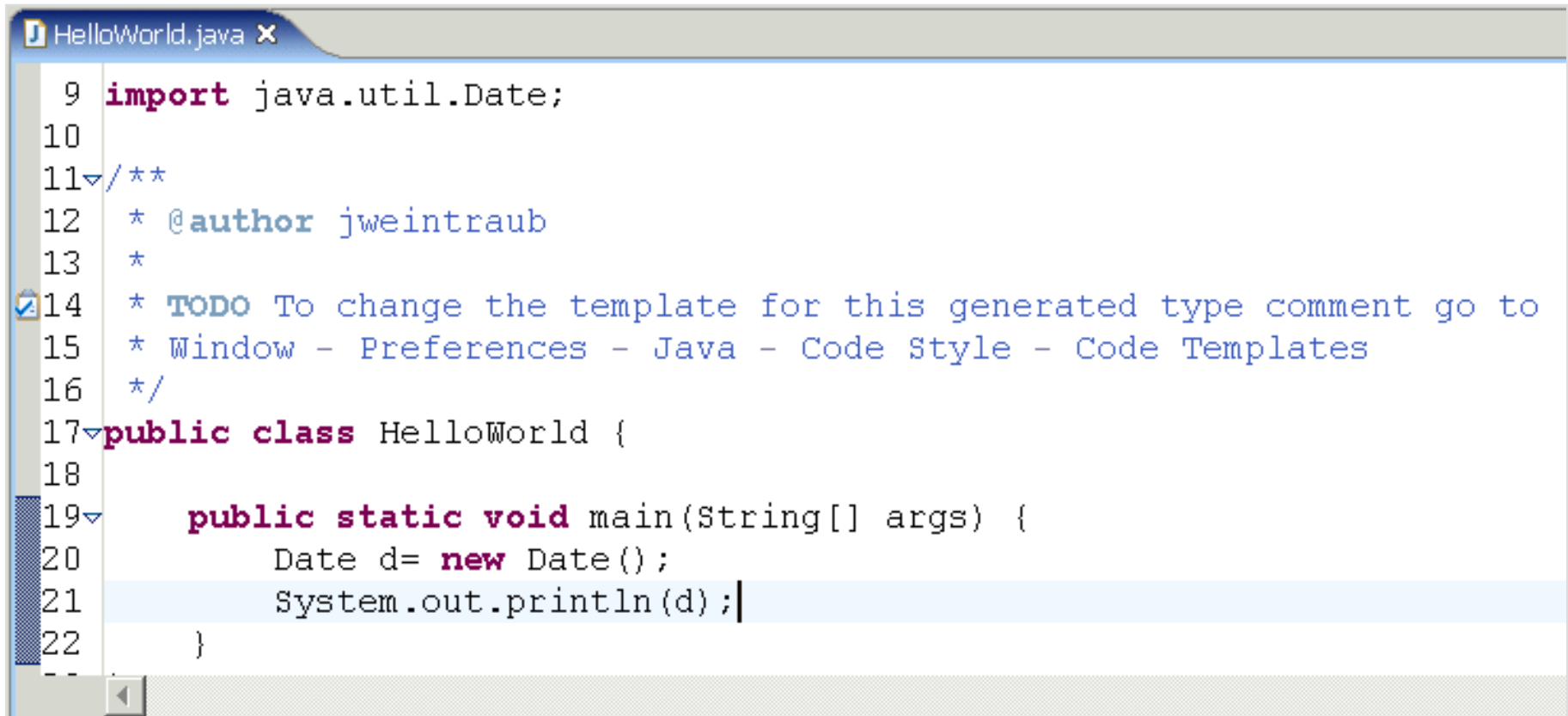
Lab



- ◆ Shows how different resources are structured
- ◆ There are three kinds of resources:
 - **Files**
 - Correspond to files on the file system
 - **Folders**
 - Like directories on the file system
 - **Projects**
 - Used to organize all your resources and for version control.
 - When you create a new project, you assign a physical location for it on the file system.
 - A third-party SCM (Source Control Manager) may be used to properly share project files amongst developers.



- ◆ There is a source editor (like this one for a .java file) for all character files. (.java, .jsp, .html, etc.)



```
9 import java.util.Date;
10
11 /**
12  * @author jweintraub
13  *
14  * TODO To change the template for this generated type comment go to
15  * Window - Preferences - Java - Code Style - Code Templates
16  */
17 public class HelloWorld {
18
19     public static void main(String[] args) {
20         Date d= new Date();
21         System.out.println(d);
22     }
23 }
```





Session 2: JavaScript Basics

JavaScript Introduction

JavaScript Functions

Accessing and Modifying HTML Elements

Lesson Objectives

- ◆ Understand what JavaScript is and its role in browsers
- ◆ Understand the basics of JavaScript and write simple programs with it
- ◆ Use JavaScript to modify a web page
- ◆ In this course, we'll cover enough JavaScript so you understand its structure and how to use it with Ajax
 - We won't cover the whole language, which is fairly large and complex, and the focus of the course is really to teach Ajax
 - We'll assume you already know programming basics



JavaScript Introduction

JavaScript Introduction

JavaScript Functions

Accessing and Modifying HTML Elements

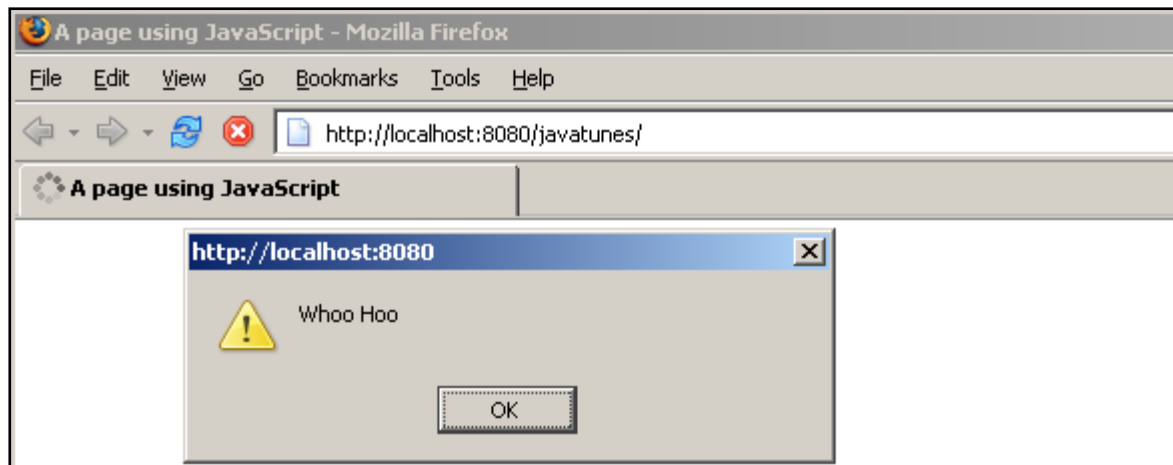
What is JavaScript

- ◆ JavaScript is a language used to program web browsers which allows you to do many things such as:
 - Access and modify the HTML elements of the page
 - Validate input data from a control
 - React to events such as a button press or page load
 - Modify images as the cursor moves over them
- ◆ JavaScript is a **scripting language**, which is loosely defined as follows:
 - Is usually an interpreted language
 - It has looser constraints than many programming languages
 - For example, JavaScript is loosely typed, so a variable can point to anything – from a number to a function
- ◆ We will focus on using JavaScript in HTML pages in a browser
 - Though it can be used in other situations

Exploring JavaScript

- ◆ Below is a web page with JavaScript and the resulting display
 - We use the JavaScript ***alert()*** that pops up a dialog window

```
<html>
  <head>
    <title>A page using JavaScript</title>
    <script type="text/javascript">
      alert("Whoo Hoo!");
    </script>
  </head>
  <body><h1>This page uses JavaScript!</h1></body>
</html>
```



Exploring JavaScript

- ◆ As you see, using JavaScript is fairly easy
- ◆ You embed the JavaScript code into an HTML web page within a **<script>** tag as shown in the previous example
 - **type="text/javascript"** declares the script language is JavaScript
 - Plain statements in the script end with a semicolon ; (see notes)
 - Scripts can go in the *<head>* or *<body>* section of a page
- ◆ Scripts are executed in the order they appear in a web page as part of the document loading and processing
 - They are processed immediately while the page loads
 - All the scripts are considered to be one program
 - For example, variables declared within one **<script>** can be accessed by code within succeeding **<script>** code

JavaScript Variables

- ◆ Working with variables in JavaScript is easy
 - They can be declared directly in a script as **global variables** accessible throughout the page they are used in
 - They may appear within a function, (covered later) as **local variables** accessible only within the function
 - The **var** keyword is used to declare them
 - Variables aren't typed, and can hold any legal value
 - Below we declare a variable, give it a string value, display it, change its value to an integer, then display it again

```
<script type="text/javascript">  
  var toDisplay = "Whoo Hoo!";  
  alert(toDisplay);  
  toDisplay = 5;  
  alert(toDisplay);  
</script>
```

JavaScript - Writing to the Web Page

- ◆ Below we write directly to the web page using JavaScript
 - This uses the **document** object, which is part of the Document Object Model that we will cover later
 - The document object represents the entire HTML document
 - **document.write()** writes directly to a document

```
<html>
  <head><title>A page using JavaScript</title></head>

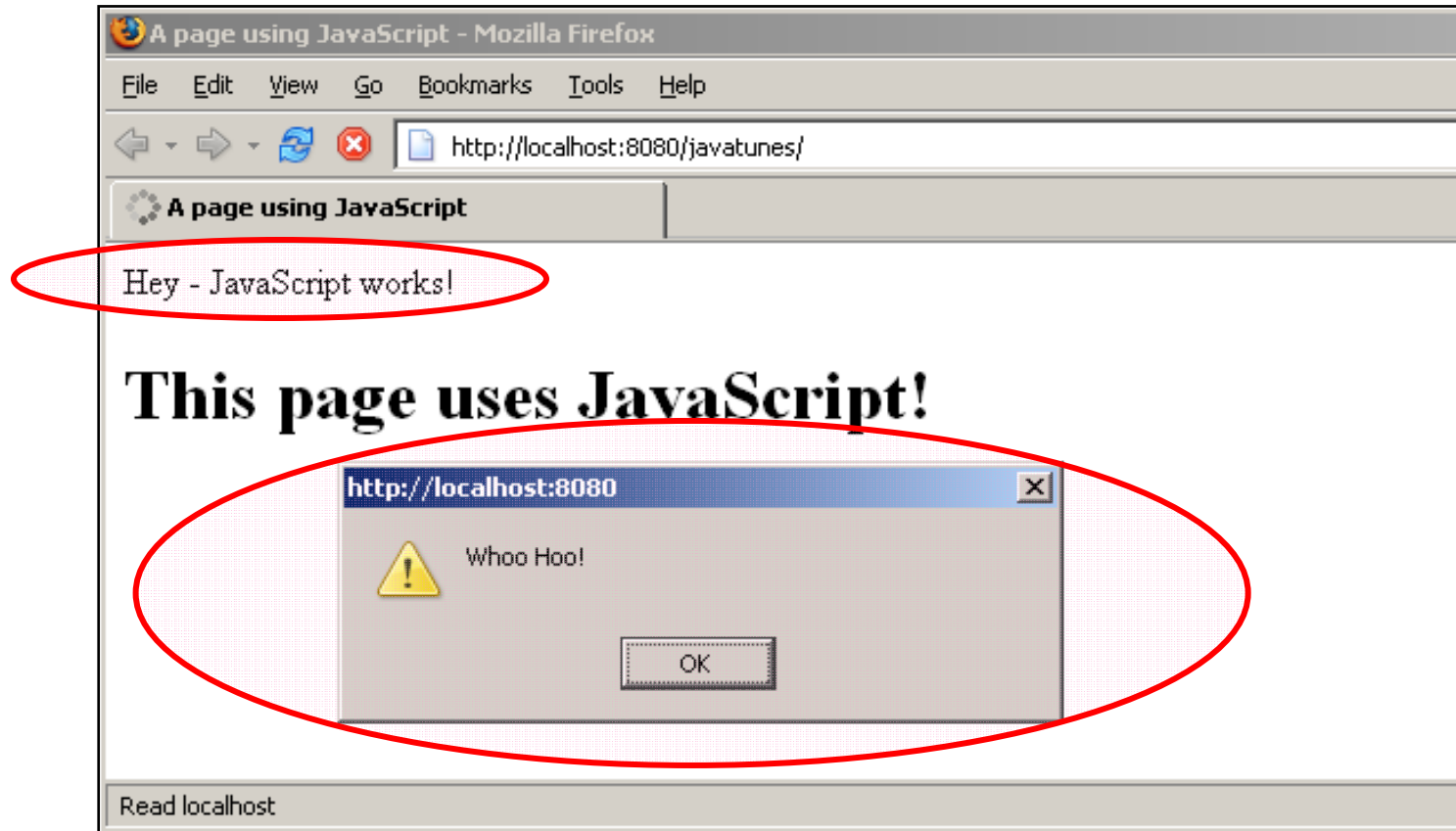
  <body>
    <script type="text/javascript">
      document.write("Hey - JavaScript works!");
    </script>

    <h1>This page uses JavaScript!</h1>

    <script type="text/javascript">
      var toDisplay = "Whoo Hoo!";
      alert(toDisplay);
    </script>
  </body>
</html>
```

JavaScript - Writing to the Web Page

- ◆ Below is the web page from the previous example
 - The text "Hey – JavaScript works" comes from the call to `document.write()`, and the alert box from the call to `alert()`
 - The rest is from the regular HTML in the page



JavaScript PopUp Boxes

- ◆ **Alert Box:** Used to make sure information seen by user
 - User has to click "OK" to proceed

alert("sometext");

- ◆ **Confirm Box:** Used to let user verify or accept something
 - User will have to click either "OK" or "Cancel" to proceed.
 - If the user clicks "OK", the box returns true. If the user clicks "Cancel", the box returns false.

var res = confirm("Do you like JavaScript");

- ◆ **Prompt Box:** Used to let the user input a value
 - User clicks "OK" or "Cancel" after entering an input value.
 - If user clicks "OK" the box returns the input value. If the user clicks "Cancel" the box returns null.

var res = prompt("sometext", "defaultvalue");

Lab 2.1 – Use JavaScript in an HTML Page

- ◆ **Overview:** In this lab you will write some simple JavaScript scripts in a web page and test the resulting browser display
 - You'll use the JavaScript capabilities from the previous section
- ◆ **Objectives:**
 - Gain experience using JavaScript in a web page
 - Work with the JavaScript we've learned so far
- ◆ **Builds on previous labs:** Lab 1.1
 - Continue working in your **Lab01.1** directory
- ◆ **Approximate Time:** 15-20 minutes

Tasks to Perform

- ◆ Open the file *hello_js.html* (in the *WebContent* folder) for editing (you can double click on it in Project Explorer View)
- ◆ Add in a `<script>` element in the `<head>` section after the `<title>` element as shown in the student manual
- ◆ Include some JavaScript code that does the following:
 - Declares and uses a variable
 - Uses `document.write()` to write something to the web page
 - Uses the `alert()` function
- ◆ After you've deployed and tested the application (described on the following slide) you can try the following **optional** JavaScript capabilities if you have time:
 - Use other popup boxes
 - Use a `<script>` with JavaScript code in the `<body>` section

- ◆ When you change something like an HTML file in your Web application, Eclipse has to publish it to the server
 - The server is set up to automatically publish every 15 sec. *
 - You can then just reload the browser to see the changes
- ◆ For some changes, (e.g. changes to a servlet) Eclipse requires you to **restart** the server (indicated by a server status of restart as shown below)
 - To restart, you can right click on the server, and select **Restart**



Tasks to Perform

- ◆ After your Web app is published, reload it in the browser, and check to see that your JavaScript is working properly



JavaScript Functions

JavaScript Introduction

JavaScript Functions

Accessing and Modifying HTML Elements

JavaScript Function Overview

- ◆ A JavaScript function defines a reusable block of code
- ◆ The syntax for creating a function is:

```
function functionname(var1,var2,...,varX) {  
    /* Some JavaScript code ...*/  
}
```

- Functions may have arguments and return values
- You don't need to specify the type of arguments or return value
- The function is usually defined in the *<head>* section or in an external file
- ◆ Functions are executed when the function is called explicitly, or by an event (covered later)
 - You can call the function anywhere in the page
 - Code in a function is not executed when the page is loaded

JavaScript Function Example

- ◆ Below, we define a very simple function in the `<head>`
 - We then call it directly from the `<body>`

```
<html>
  <head>
    <title>A page using JavaScript</title>
    <script type="text/javascript" >
      // Define a function
      function displayMessage(txt) {
        alert(txt);
      }
    </script>
  </head>

  <body>
    <h1>This page uses JavaScript!</h1>
    <script>
      // Call the function defined above
      displayMessage("Hey - JavaScript works!!");
    </script>
  </body>
</html>
```

JavaScript Function Example

- ◆ Below, we define a function that returns a value
 - We call it directly from the `<body>` and display the return value

```
<html>
  <head>
    <title>A page using JavaScript</title>
    <script type="text/javascript" >
      // Define a function
      function add(a1, a2) {
        return a1 + a2;
      }
    </script>
  </head>

  <body>
    <h1>This page uses JavaScript!</h1>
    <script>
      // Call the function defined above
      alert(add(2,3));
    </script>
  </body>
</html>
```


External JavaScript Files

- ◆ It's possible to place your JavaScript code into an external file
 - The code is then included in your HTML page using the `<script>` tag's **src** attribute
 - You can include a local file, as shown below
 - You can also include a remote file using a URL (e.g. `http://mysite.com/js/myCode.js`)

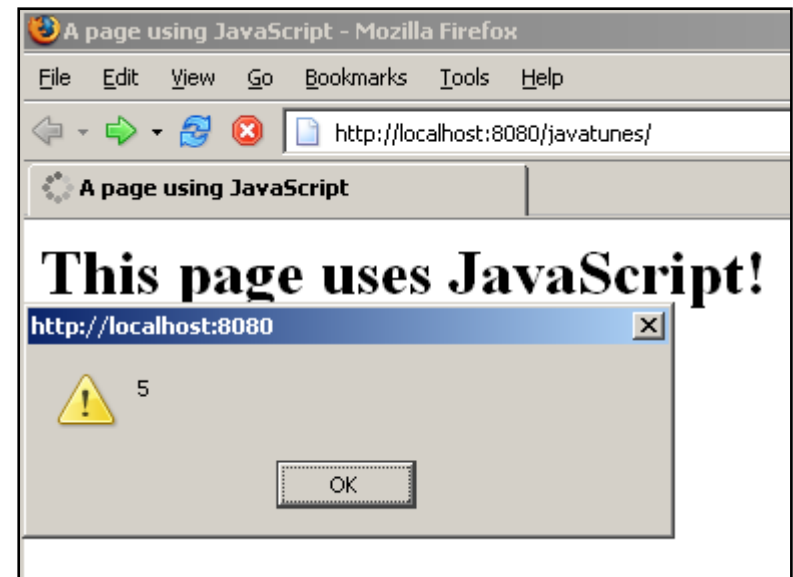
```
// file myCode.js
function add(a1, a2) {
    return a1 + a2;
}
```

```
<html>
  <head>
    <title>A page using JavaScript</title>
    <script type="text/javascript" src="js/myCode.js" >
      </script>
  </head>
  <!-- ... -->
</html>
```

Functions as Data

- ◆ In JavaScript, functions are also data
 - You can assign them to variables and then invoke them through the variable
 - The variable name is used in the same way as the function name
 - We'll see this capability used later with Ajax

```
<!-- Much detail omitted -->  
<script>  
    // Initialize var with function  
    var myFunc = add;  
    alert (myFunc(2,3));  
</script>
```



Standard JavaScript Functions

- ◆ JavaScript makes a number of global functions available
 - ***decodeURI()***: Decodes an encoded URI
 - ***decodeURIComponent()***: Decodes an encoded URI component
 - ***encodeURI()***: Encodes a string as a URI
 - ***encodeURIComponent()***: Encodes a string as a URI component
 - ***escape()***: Encodes a string
 - ***eval()***: Evaluates a string & executes it as if it was script code
 - ***isFinite()***: Checks if a value is a finite number
 - ***isNaN()***: Checks if a value is not a number
 - ***Number()***: Converts an object's value to a number
 - ***parseFloat()***: Parses a string & returns a floating point number
 - ***parseInt()***: Parses a string and returns an integer
 - ***String()***: Converts an object's value to a string
 - ***unescape()***: Decodes a string encoded by `escape()`

Lab 2.2 – Use JavaScript Functions

- ◆ **Overview:** In this lab you will place your JavaScript code in functions defined in an external file
 - You'll call these functions directly from `<script>` tags in your HTML file
- ◆ You'll also install and use the FireBug JavaScript debugger
- ◆ **Objectives:**
 - Gain experience using JavaScript functions
 - Define JavaScript functions in an external file
 - Install and use a JavaScript debugger
- ◆ **Builds on previous labs:** Lab 2.1
 - Continue working in your **Lab01.1** directory
- ◆ **Approximate Time:** 20-25 minutes

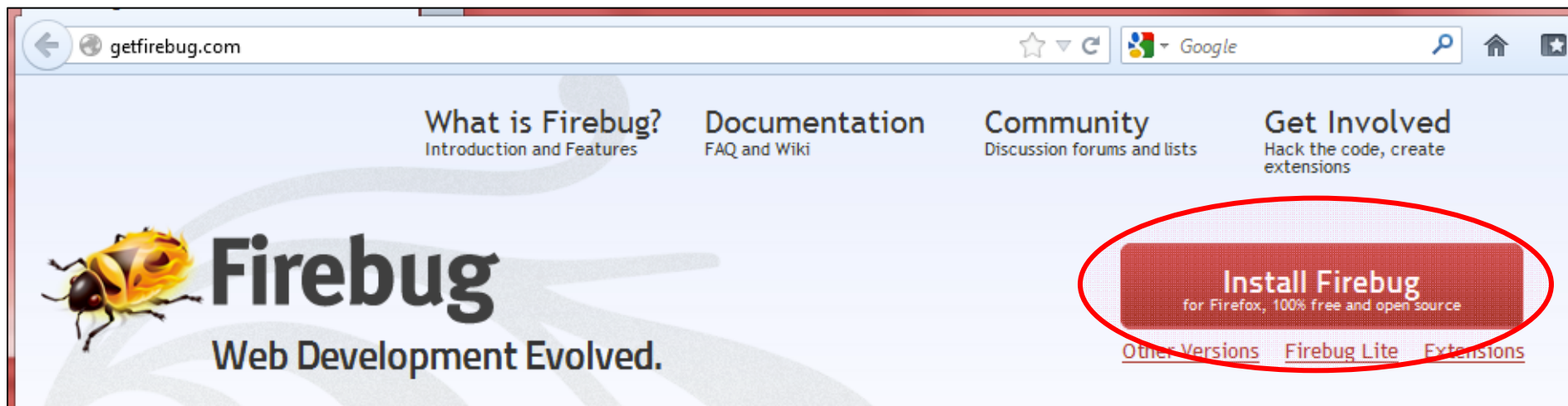
Tasks to Perform

- ◆ Create a new file *hello.js* in the *WebContent\js* directory
- ◆ Define two JavaScript functions in this file
 - One that takes a single argument, and simply displays the argument you pass in using the *alert()* function
 - One that takes two arguments, adds them, and returns the result
- ◆ Open the file *hello_js.html* from the previous lab for editing
- ◆ Modify the *<script>* element in the *<head>* section to use a *src* attribute referring to *js/hello.js*
 - Remove all the other JavaScript code in the *<script>*
- ◆ Add in a *<script>* element to the *<body>* section in *hello_js.html* that invokes your newly defined functions
- ◆ **Restart** the server and then reload the javatunes web page to see the results

- ◆ Firebug is an extension to the Mozilla Firefox browser
 - It allows you to edit, debug, and monitor JavaScript code
 - We will learn about it by using it

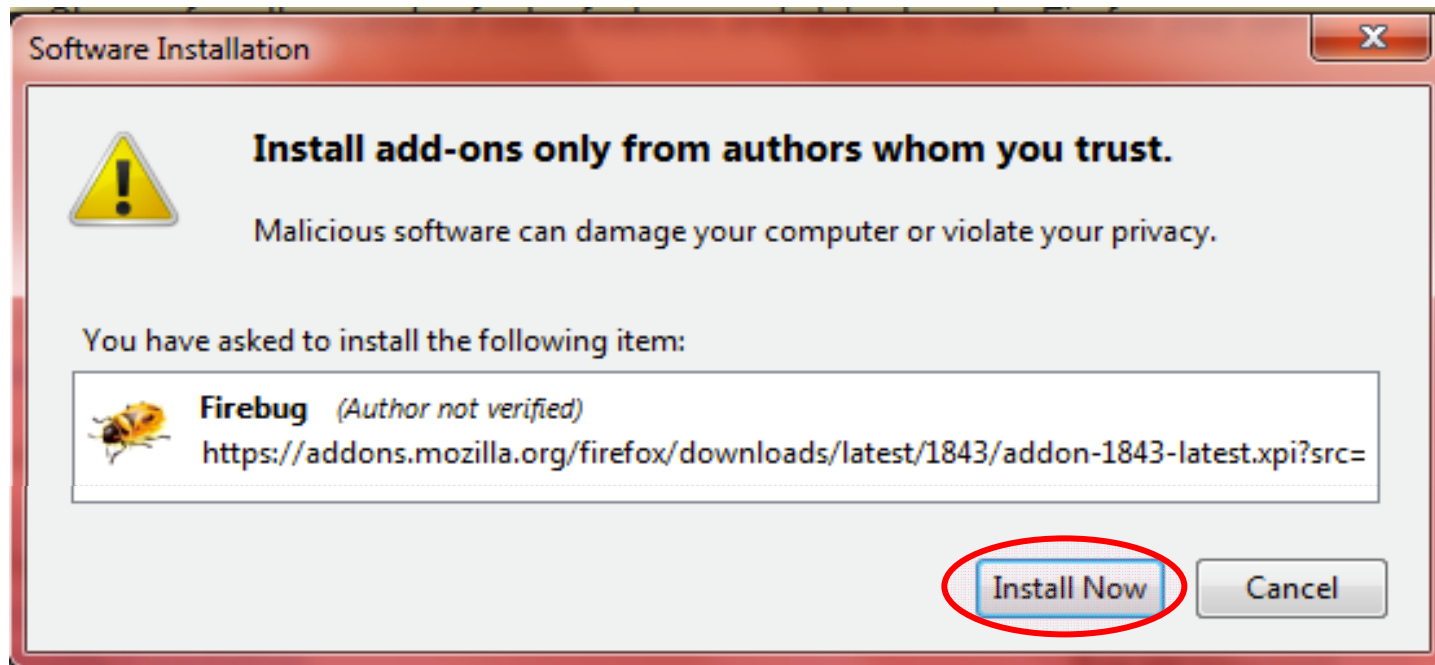
Tasks to Perform

- ◆ If you're not already using it, open the **Firefox browser**
 - Browse to **<http://www.getfirebug.com>**
- ◆ Click on the **Install Firebug** button - then on the download link for your version of Firefox (see notes)



Tasks to Perform

- ◆ Firefox should pop up a window asking you to allow the software installation
 - Click the **Install Now** button - you should be done
 - Older versions of Firefox may do this somewhat differently - just take the steps necessary to install (see notes)



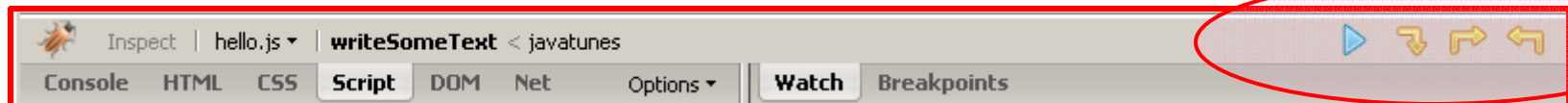
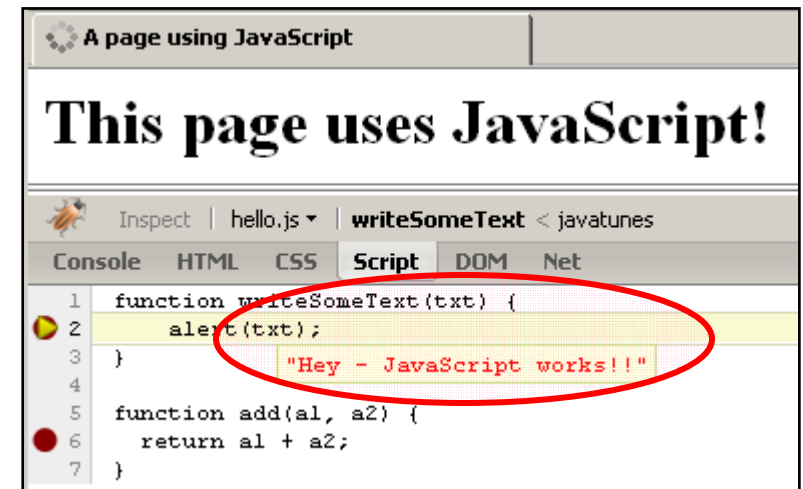
Tasks to Perform

- ◆ In Firefox, browse to localhost:8080/javatunes
- ◆ Click on the Firebug icon (on the right side of the browser) to activate Firebug for this site
- ◆ The Firebug panel should open at the bottom of your browser
 - Click the Script tab to get a JavaScript view (see notes)



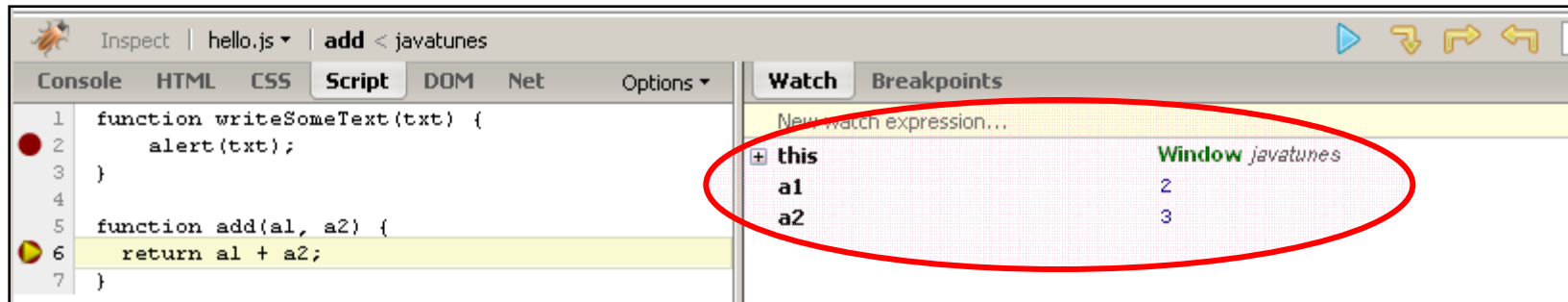
Tasks to Perform

- ◆ **Select** *hello.js* in the dropdown, as shown at right
- ◆ **Click** to the left of the number column to set breakpoints in both your functions
- ◆ **Browse** to `localhost:8080/javatunes` again
 - Move the cursor around, and you'll see when you hover over a variable or argument, the value is displayed
- ◆ Try to **Continue**, **Step Into**, **Step Over**, or **Step Out** of functions via the icons shown

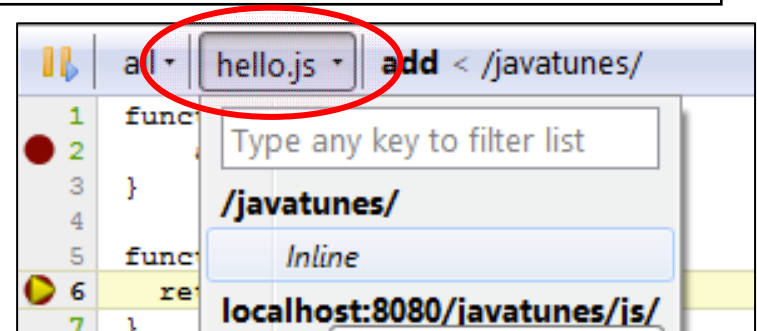


Tasks to Perform

- ◆ When you're at a breakpoint, look at the **Watch** window which contains information on variables that are active
 - Firebug allows you to change the values dynamically
 - While in your add function, double click on one of the arguments in the watch window, change its value, then continue running

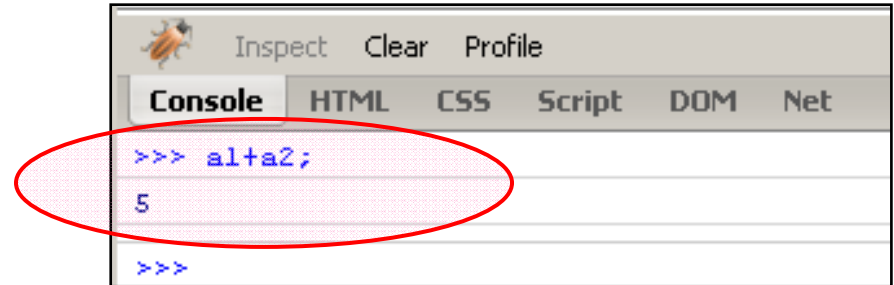
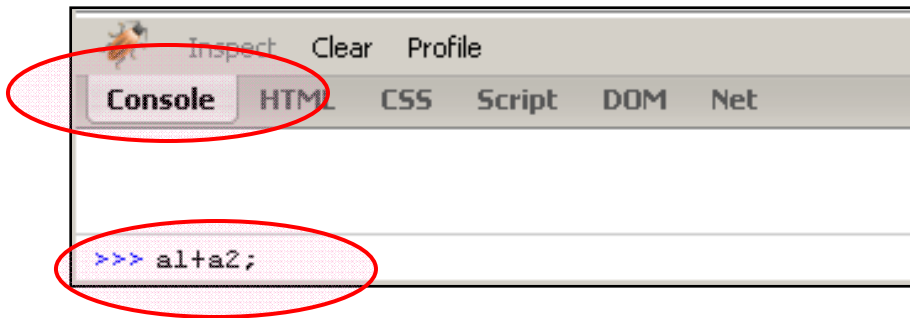


- ◆ You can change the file you're looking at via the dropdown (as shown)

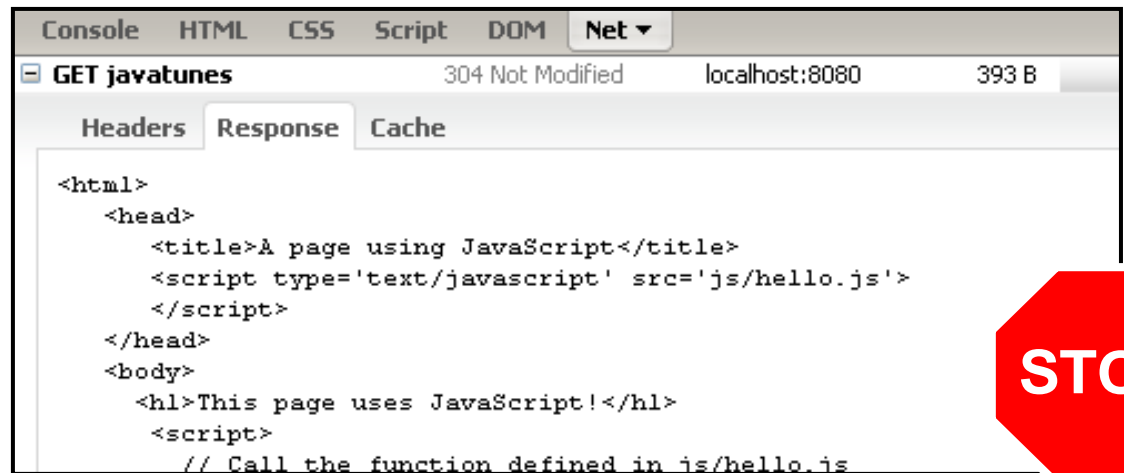
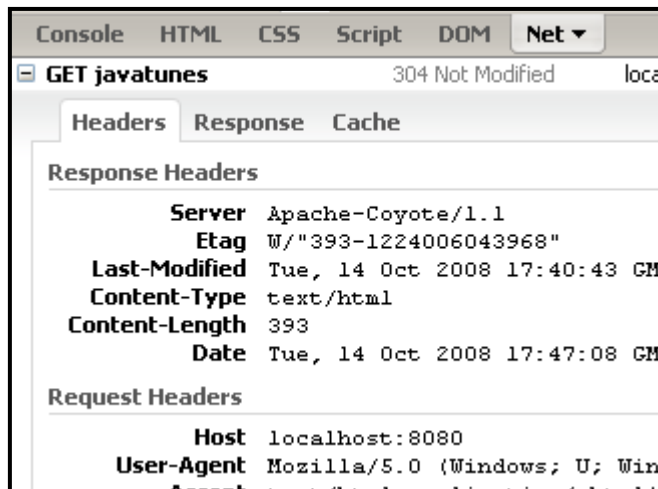


Other Firebug Capabilities

- ◆ The **Console** allows you to interactively run JavaScript code
 - Just type it in on the command area, hit return, and it's evaluated
 - Below we evaluate `a1+a2` from within the add function



- ◆ The **Net** window allows you to inspect HTTP headers & responses



Accessing and Modifying HTML Elements

JavaScript Introduction

JavaScript Functions

Accessing and Modifying HTML Elements

A More Complex HTML Page

```
<html>
  <head><title>A More Complex HTML Page</title></head>

  <body>
    <h1>This page has more HTML Elements</h1>
    <div id="testDiv">
      This text is contained in a div element
    </div>
  </body>
</html>
```

- ◆ The example includes two new HTML concepts
 - The **<div>** element defines a division or section in a document
 - By itself, it doesn't specify any presentational style
 - Often used for formatting with style sheets (covered later)
 - The **id** attribute which specifies a unique id for an element
 - This attribute can be used to access the element programmatically
 - We'll see that these allow us to access sections of the document using JavaScript

Accessing Elements via the document Object

- ◆ An HTML page is represented by a Document object
 - It is accessed via the *document* variable
 - Its properties and functions (or methods*) allow you to manipulate a page with JavaScript code (e.g. with *document.write()* seen earlier)
- ◆ The document method *getElementById()* allows you to programmatically access the element with the given id

```
var testDiv = document.getElementById("testDiv");
```

 - This is very useful when you want to modify the contents of a web page using JavaScript
 - It allows you to access a particular element in the page, as identified by its id, and then modify it
 - Useful in Ajax applications needing to modify part of a web page
 - Very often *<div>* elements with an id are used to specify sections that will later be modified by some user data

The `innerHTML` Property

- ◆ Every HTML element has an *`innerHTML`* property that represents its current HTML content
 - When you query the value of this property, you get a string representing the HTML content of the element, including any tags
 - If you set this property on an element, the browser invokes its HTML parser on the string you pass in, and replaces the element's content with what the parser returns
- ◆ *`innerHTML`* is not part of the official DOM standard
 - However, it's supported by all modern browsers
 - It is a very widely used technique for manipulating page content

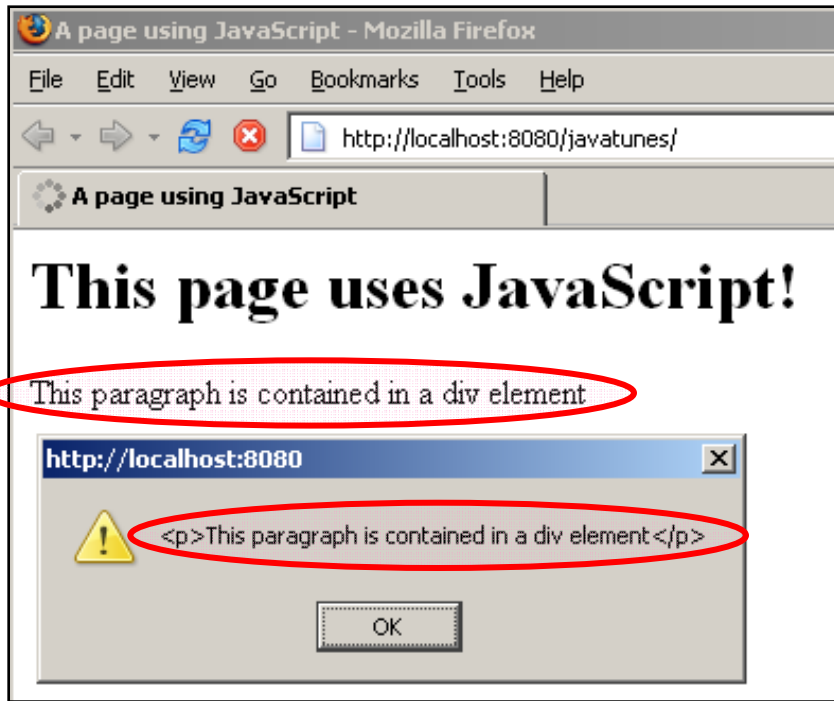
Example - Modifying an HTML Page

```
<html>
  <head><title>A More Complex HTML Page</title></head>

  <body>
    <h1>This page has more HTML Elements</h1>
    <div id="testDiv">
      <p>This paragraph is contained in a div element</p>
    </div>
    <script>
      var testDiv = document.getElementById("testDiv");
      alert(testDiv.innerHTML);
      testDiv.innerHTML="Here is some new text";
    </script>
  </body>
</html>
```

- ◆ Here, we access a div by id using *getElementById()*
 - We then display its HTML content using *innerHTML*
 - Finally, we change that content by setting *innerHTML*
 - The results in the browser are shown on the next page

Example - Modifying an HTML Page



- ◆ Notice the alert box contains all the content, including the tags
 - Once you accept the alert, the `<div>` content is replaced with the new content



Lab 2.3 – Accessing and Modifying HTML Elements

Lab 2.3 – Access/Modify HTML Elements

- ◆ **Overview:** In this lab you'll access HTML elements in a Web page from JavaScript using *document.getElementById()*
 - You'll then modify the contents of the element using *innerHTML*
- ◆ **Objectives:**
 - Gain experience defining and accessing elements in a Web page and working with the document object
 - Learn how to use innerHTML
- ◆ **Builds on previous labs:** Lab 2.2
 - Continue working in your **Lab01.1** directory
- ◆ **Approximate Time:** 15 minutes

Tasks to Perform

- ◆ Open the file *hello.js* for editing
- ◆ Write a function, *modifyContent* to modify an HTML element
 - Have this function take two arguments
 - The first arg should contain the id of an HTML element
 - The second will contain a text string
 - In the function, get the document element with the given id
 - Use *document.getElementById()* to get the element
 - Next, call *alert()* to display the elements *innerHTML* property
 - Finally modify its *innerHTML* property by setting it to the passed in text

Continued ⇒

Tasks to Perform

- ◆ Open the file *hello_js.html* for editing
 - In the `<body>`, add in a `<div>` with an id of *targetDiv* and some text content
 - After the `<div>`, add in a script that calls *modifyContent*, passing in the *"targetDiv"* id and some text
- ◆ **Restart** the server and reload the javatunes web page again
 - You should see the `<div>` content is replaced with your new text
 - If you have problems, check for JavaScript errors using Firebug



Review Questions

- ◆ What is JavaScript, and what makes it a scripting language?
- ◆ How do you write scripts in an HTML page?
- ◆ How do you write to an HTML page using JavaScript?
- ◆ What is a JavaScript function, and what does it look like?
- ◆ How can you access a specific HTML element in a page using JavaScript? How can you change its contents?

Lesson Summary

- ◆ JavaScript is a language used to program web browsers
 - It is a scripting language – i.e., loosely typed, interpreted, forgiving of mistakes
- ◆ You embed the JavaScript code into an HTML web page within a **<script>** tag
 - This can contain variable declarations, statements, etc.
- ◆ In the simplest manner, you can write to an HTML page using the **document.write()** method
 - There are other more sophisticated and useful ways also
- ◆ JavaScript code can be encapsulated into functions
 - Reusable blocks of code that take arguments and return values
- ◆ **document.getElementById()** accesses HTML elements
 - Their contents can then be changed by modifying their **innerHTML** property



Session 3: XMLHttpRequest

XMLHttpRequest Basics
Asynchronous Requests

Lesson Objectives

- ◆ Learn how the XMLHttpRequest object works
- ◆ Use XMLHttpRequest to build basic Ajax applications
- ◆ Note that we sometimes use the abbreviation XHR for XMLHttpRequest
 - To save space

XMLHttpRequest Basics

XMLHttpRequest Basics
Asynchronous Requests

More About XMLHttpRequest

- ◆ Ajax is based on making HTTP requests independently of a page reload
 - There are a number of ways this can be done
- ◆ We will focus on using *XMLHttpRequest* with JavaScript (scripting HTTP requests)
 - *XMLHttpRequest* is supported in all modern browsers
 - It communicates with a server using standard HTTP
- ◆ ***XMLHttpRequest*** allows you to make scripted HTTP requests, which generally involves:
 - **Creating** an *XMLHttpRequest* object
 - **Submitting** an HTTP request to a web server
 - **Retrieving** the response from the web server (synchronously or asynchronously)
- ◆ We'll cover each of these areas

Creating an XMLHttpRequest Object

- ◆ In many browsers, you can create an XMLHttpRequest object directly using the following code

```
var request = new XMLHttpRequest();
```

 - The *new* keyword is part of JavaScript, and it is followed here by the constructor function for *XMLHttpRequest*
- ◆ This will work in Firefox, IE 7, and most other modern browsers (Safari 2.0+, Opera 8+)
 - Unfortunately, this won't work in earlier versions of IE
- ◆ In older versions of IE *XMLHttpRequest* is an **ActiveX** object
 - You use the *ActiveXObject* constructor and the object name to create *XMLHttpRequest*
 - Unfortunately, the name of the object is different in different releases of Microsoft's XML HTTP library (**Msxml2.XMLHTTP** or **Microsoft.XMLHTTP**)
 - This makes creating XMLHttpRequest a bit complex

Creating an XMLHttpRequest Object

- ◆ There are a number of different ways to determine what your browser supports in terms of XMLHttpRequest
 - We'll use a technique that tries each of the various possibilities
 - If a possibility throws an exception, we go on to the next possibility
 - Once we've succeeded, we return the object we've created
- ◆ You can use the **try...catch** statement to handle exceptions
 - Designate blocks of protected code with the **try** keyword
 - Create an exception "handler" with the **catch** keyword – an **exception handler** is code that will be executed if an exception occurs, and therefore must "handle" the situation
- ◆ The general form for handling exceptions is:

```
try { Block }  
catch ( Argument ) { Block }
```

Creating an XMLHttpRequest Object

```
function getXMLHttpRequest() {  
    var req = false;  
    try {// Try native XMLHttpRequest object  
        req = new XMLHttpRequest();  
    } catch(e) {  
        try { // Try ActiveXObject with Msxm12.XMLHTTP  
            req = new ActiveXObject("Msxm12.XMLHTTP");  
        } catch(e) {  
            try {// Try ActiveXObject with Microsoft.XMLHTTP  
                req = new ActiveXObject("Microsoft.XMLHTTP");  
            } catch(e) {  
                req = false;  
            }  
        }  
    }  
    return req;  
}
```

- ◆ The code tries various possibilities to create XMLHttpRequest
 - If a possibility succeeds, the object created is returned
 - *try...catch* is used to catch failures and control program flow

Submitting a Request

- ◆ Once you've created an XMLHttpRequest object, you can submit a request to a server, using the XMLHttpRequest API
- ◆ Submitting a request is also a multi-step process which involves:
 - Calling the *open()* method on the XMLHttpRequest
 - Setting up the call
 - Calling the *send()* method on the XMLHttpRequest
- ◆ The *open()* method has the following signature
void open(String method, String url, boolean async, String username, String password)
 - **method**: The HTTP method to be used (GET, POST, HEAD ...)
 - **url**: The url that is the subject of the request
 - **async**: (Optional) A boolean specifying if the request should be asynchronous (default true)
 - **username, password**: (Optional) Authorization for those URLs that require them

Submitting a Request

- ◆ Once you've called *open()*, you can set properties on the XMLHttpRequest object
 - For example, setting headers or other properties
 - Note that calling *open()* just initializes XMLHttpRequest
 - It does not actually send the request
- ◆ The *send()* method causes an HTTP request to be issued
 - The method signature is
void send(Object body)
 - For "GET" request the argument should be *null*
 - For HTTP "POST" or "PUT" requests, the argument should be the body of the request that is to be passed to the server
- ◆ We'll show a very simple request shortly
 - It will make a synchronous request
 - It will use the HTTP "GET" method

XMLHttpRequest Properties

- ◆ There are a number of properties of XMLHttpRequest that are important when making requests
 - ***readonly String.responseText***: The body of the response (not including headers) that has been received from the server, or the empty string if no data has been received yet.
 - ***readonly short status***: The HTTP status code returned by the server – e.g. 200 for success or 404 for not found
 - ***readonly String.statusText***: The HTTP status code in its text form – e.g. "OK" for a status of 200
 - There are other properties that we will cover when we use them
- ◆ Once you've made a request (actually called send on XHR) you can use these properties to access the result
 - e.g., checking the status code and using the response text

Example – Submitting a Request

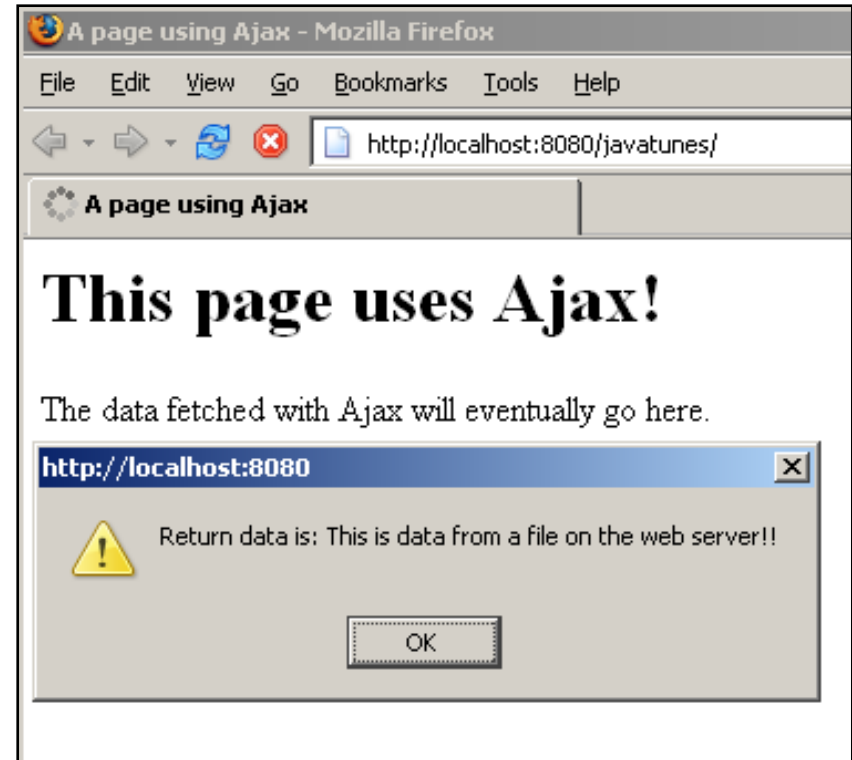
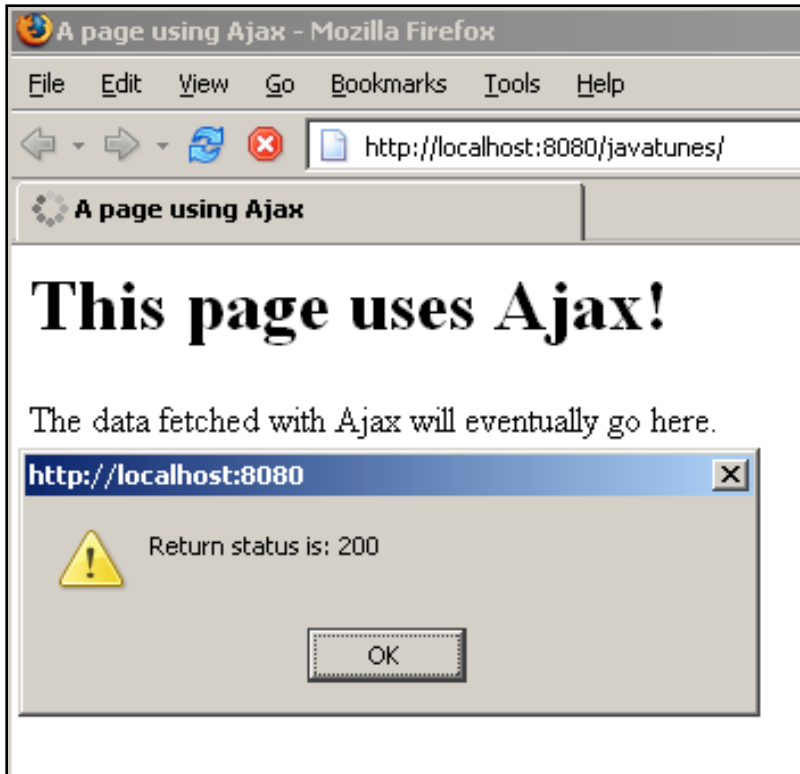
```
<html>
  <head>
    <title>A page using Ajax</title>
    <script type="text/javascript" src="js/ajax.js"></script>
  </head>

  <body>
    <script>
      var req = getXMLHttpRequest(); // Defined in js/ajax.js
      req.open("GET", "/javatunes/data.txt", false);
      req.send(null);
      alert(req.status);
      alert(req.responseText);
    </script>

  </body>
</html>
```

- ◆ We use the *getXMLHttpRequest()* function shown earlier
 - We make a synchronous GET request for a simple text file
 - When the request is done, we display some XHR properties

Result – Submitting a Request



- ◆ Below we see the status code and the text that was returned
 - The request succeeded and the contents of the data.txt file were returned as data



Lab 3.1 – Working with XMLHttpRequest

Lab 3.1 – Working with XMLHttpRequest

- ◆ **Overview:** In this lab you'll work with the *XMLHttpRequest* object to submit an HTTP request from JavaScript
 - You will just display the results of the request, and use them in a later lab
- ◆ **Objectives:**
 - Gain experience creating and working with *XMLHttpRequest*
 - Create a complete program that submits a request and uses the response
- ◆ **Builds on previous labs:** None
 - You will work in the **Lab03.1** directory for this lab
- ◆ **Approximate Time:** 20 minutes

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab03.1

- This is a new lab directory

Tasks to Perform

- ◆ Right click on the server in Servers View
 - Select **Add and Remove Projects**, and **remove** Lab01.1
- ◆ Create a new **Dynamic Web Project** called **Lab03.1** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server is selected
 - Make sure **2.5 module version** is selected
 - Remember to set the context root to **javatunes**
 - Ignore any taglib errors you may see in the JSP - we'll work with that later

- ◆ You will work with two files in **Lab03.1\WebContent**
 - **hello_ajax.html**: An HTML file that will execute a request using *XMLHttpRequest*
 - **js/ajax.js**: A JavaScript file that contains utility functions to work with *XMLHttpRequest*
- ◆ There are other files in the *WebContent* directory
 - **data.txt**: Contains data that will be sent in response to a request done using *XMLHttpRequest*
 - **WEB-INF/web.xml**: The web application deployment descriptor which specifies *hello_ajax.html* as the default file for the app
- ◆ There are also JSPs (in the *jsp* dir) and servlets (in the *src* dir)
 - These will be used in a future lab to display other pages from the JavaTunes online store
 - Ignore them until you are told to use them

Tasks to Perform

- ◆ Open the file **WebContent\js\ajax.js** for editing, and finish the function, **getXMLHttpRequest()**
 - Look for the comments with **TODO** in them
 - The skeleton of the function is complete, but you will need to add in the expressions to create the *XMLHttpRequest*
- ◆ Open the file **WebContent\hello_ajax.html** and finish the `<script>`
 - Look for the comments with **TODO** in them, and do the following:
 - Create the *XMLHttpRequest* object
 - Finish the call to the *XMLHttpRequest.open()* method to make a GET request synchronously, and to request */javatunes/data.txt*
 - Send the request
 - Retrieve and display information about the response using *alert()*

Tasks to Perform

- ◆ **Deploy** (right click on the server, click **Add and Remove...** and **Add** the Lab03.1 project - Restart the server if needed)
 - In Firefox, browse to *http://localhost:8080/javatunes*
 - Look at the results you've displayed - Did the call work?
 - If you have problems, check for errors using Firebug
- ◆ Optional:
 - Try renaming the *data.txt* file in the file system without changing your code, then redeploy and test again
 - How does the response change?
 - Try using the response text to change the display of the <div> element higher up in the file
- ◆ You've made a simple Ajax request using XMLHttpRequest
 - Soon we'll show how to make an asynchronous request



Asynchronous Requests

XMLHttpRequest Basics
Asynchronous Requests

Handling an Asynchronous Response

- ◆ *XMLHttpRequest* allows you to make asynchronous requests
 - Simply pass in **true** as the third argument to *open()*, or leave the argument out
- ◆ For asynchronous calls, *send()* returns immediately after sending the request to the server
 - When the servers response arrives, you can access the same properties as with a synchronous call to get the results
- ◆ Asynchronous calls require you to set up an event listener
 - So you will know when the response is completed
- ◆ *XMLHttpRequest* has a property and event handler that allows you to work with the response
 - **readonly short readyState**: The state of the request
 - **onreadystatechange**: Event handler function invoked each time the *readyState* property changes

The readyState Property

- ◆ *readyState* is updated during the course of the XHR request processing
 - It starts out with the value 0 when the XHR object is created, and increases to 4 when the complete HTTP response is received
 - The table below lists the possible values for *readyState*

State	Name	Description
0	Uninitialized	Initial State. XHR object just created or reset with <i>abort()</i> method
1	Open	<i>open()</i> has been called, but not <i>send()</i> . Request not yet sent
2	Sent	<i>send()</i> has been called. HTTP request sent to server. No response received yet
3	Receiving	All response headers received. Response body is being received, but not yet complete
4	Loaded	HTTP response has been fully received

onreadystatechange Event Handler

- ◆ For an asynchronous request, you need to set the *onreadystatechange* property of your XMLHttpRequest object to an event handler function

```
var req = getXMLHttpRequest();  
req.onreadystatechange = function() { /* ... */ };
```

- The function will be called any time *readyState* changes
- Generally, the event handler will check the value of *readyState* and the value of the request status and take some action
 - A handler may be called multiple times before the response is completely received, so *readyState* must be checked before processing the response
- Once *readyState* == 4, you can then check the request status, and if the request was successful (usually *status* == 200), you can use the response data

Asynchronous Request Example

```
<html>
  <head>
    <title>A page using Ajax</title>
    <script type="text/javascript" src="js/ajax.js"></script>
  </head>
  <body>
    <script>
      var req = getXMLHttpRequest(); // Defined in js/ajax.js
      req.open("GET", "/javatunes/data.txt", true); // asynchronous
      req.onreadystatechange = function() { // event handler def
        if (req.readyState == 4) {
          if (req.status == 200) {
            alert(req.responseText);
          } else {
            alert("Request unsuccessful, status = " + req.status);
          }
        }
      }
      req.send(null);
    </script>
  </body>
</html>
```

XMLHttpRequest Methods

- ◆ XHR is not standardized yet, but the following methods are a de facto standard and supported in most browsers
 - ***open***(*String method, String url, boolean async, String username, String password*):
Assigns HTTP method, destination URL, mode, name password
 - ***send***(*Object body*): Sends request with string or DOM object data
 - ***abort***(): Terminates current request, resets XHR
 - ***getAllResponseHeaders***() : Returns all headers as a string
 - ***getResponseHeader***(*String header*) : Returns value of header
 - ***setRequestHeader***(*String name, String value*):
Sets Request Headers before sending
- ◆ We've also seen the following XHR properties
 - *readyState, responseText, status, statusText*
 - There is one more, *responseXML*, that we'll look at later



Lab 3.2 – An Asynchronous Request

- ◆ **Overview:** In this lab you'll work with the *XMLHttpRequest* object to submit an asynchronous HTTP request
 - The result will be used to modify the contents of the Web page
 - This is a simple, but fully functional, Ajax application
- ◆ **Objectives:**
 - Learn how to make an asynchronous request with *XMLHttpRequest*
 - Create a fully functional Ajax based web application
- ◆ **Builds on previous labs:** Lab 3.1
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 25-35 minutes

An Asynchronous Request

Tasks to Perform

- ◆ Open the file *ajax.js* for editing, and finish the function, *makeXHRrequest(url, divId)*
 - Look for the comments with **TODO** in them
 - The skeleton of the function is complete, but you will need to add in the expressions to complete the functionality
 - The two arguments to the function contain the URL to make the request to, and the id of the element whose content will be replaced with the response from the server
- ◆ You will need to:
 - Create the *XMLHttpRequest*
 - Retrieve the element with the given ID (how did we do this?)
 - Open the request using the passed in URL, using "GET" to make an asynchronous request

Continued ⇒

Tasks to Perform

- Define and set an event handler for `onreadystatechange` that
 - Checks the *readyState* and the *status*
 - If the *readyState* is 4, and the status is 200, then it replaces the contents of the appropriate element with the response text from the XHR request (using the element's `innerHTML`)
- Send the request
- ◆ Open the file *hello_ajax.html* for editing
 - Remove the code in the script from the previous lab that made a direct `XMLHttpRequest` call and processed it
 - Add in a call to `makeXMLHttpRequest()` using `"/javatunes/data.txt"` as the URL and `"targetDiv"` as the id of the element to update
- ◆ **Restart** the server, and reload the page in Firefox
 - If you have problems, check for errors using Firebug

Tasks to Perform

- ◆ Optional:
 - Try printing out the request status each time your event handler is called. What do you see?
 - Try displaying the response headers using *alert()*



Review Questions

- ◆ What major capability does XMLHttpRequest (XHR) give us?
- ◆ How do you create an XMLHttpRequest, and what are the issues around that?
- ◆ How do you set up and submit a request using XHR?
- ◆ How do you get a response using XHR?
- ◆ What is an asynchronous request, and how do you execute one using XHR?

Lesson Summary

- ◆ XMLHttpRequest allows you to make scripted HTTP requests
 - Rather than having the browser make it on a page refresh
 - This allows us to respond to users and update a page independently of a page refresh
- ◆ There are a number of ways to create an XMLHttpRequest
 - They vary from browser to browser, and from version to version
 - This is an issue in creating browser-independent code
- ◆ You use the **open()** and **send()** methods to set up and execute an XHR request
 - When the request finishes, the response can be accessed through the XHR properties, e.g. **req.responseText**
- ◆ Requests are normally asynchronous with XHR
 - You set **onreadystatechange** to an event handler function to be invoked when the request has completed

Session 4: Servlets and JSP with Ajax

Overview of Servlets

Overview of JavaServer Pages (JSP)

Lesson Objectives

- ◆ Understand the basics of Java EE
 - Understand the basics of Servlets and JSP
 - Understand the basics of custom tags and the JSP Standard Tag Library (JSTL)
 - Understand scope in Java EE applications
 - Understand how to deploy a web application
- ◆ Use Servlets, JSP and custom tags (JSTL) to generate data for Ajax applications
- ◆ NOTE: If you are already comfortable with Servlets, JSP, and Custom Tags, **you can move quickly, or even skip the review parts of this section**
- ◆ **Do all the labs**, even if comfortable with Servlets/JSP
 - There are also a few slides before Lab 4.1 talking about Servlet use with Ajax that should be covered in all cases

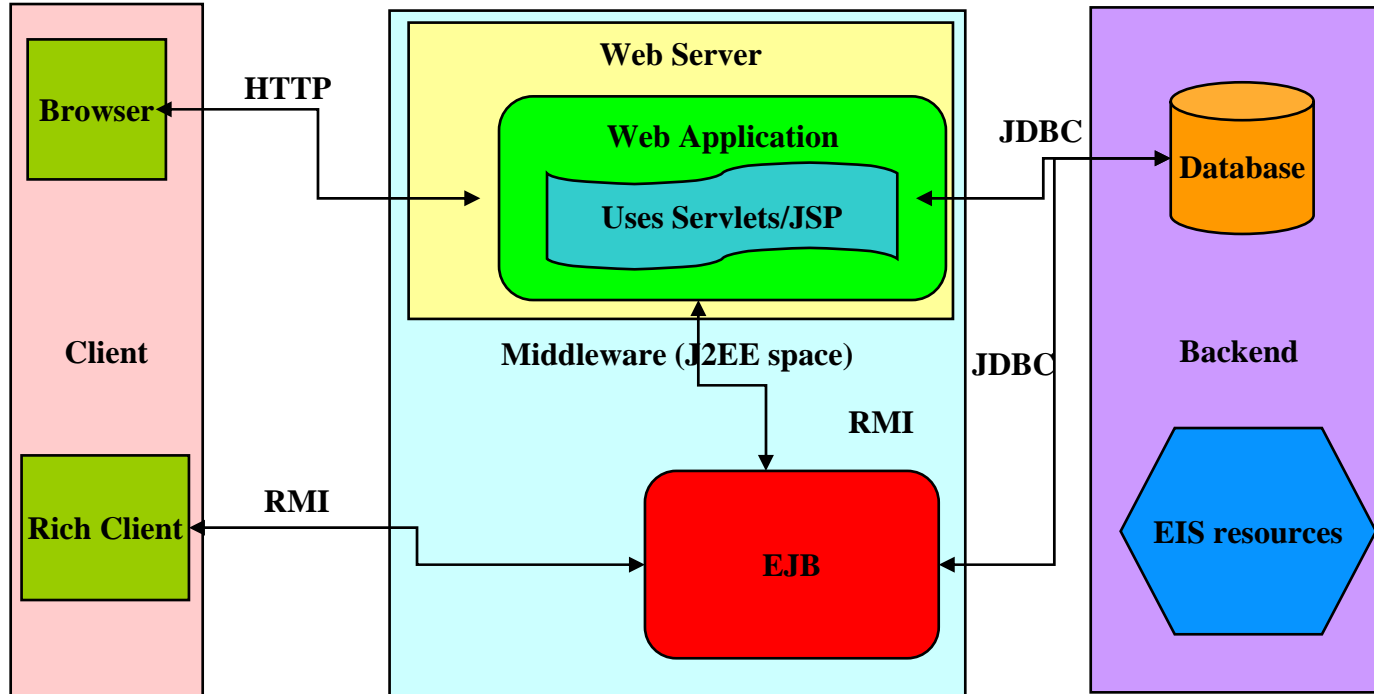
Overview of Servlets

Overview of Servlets

Overview of JavaServer Pages (JSP)

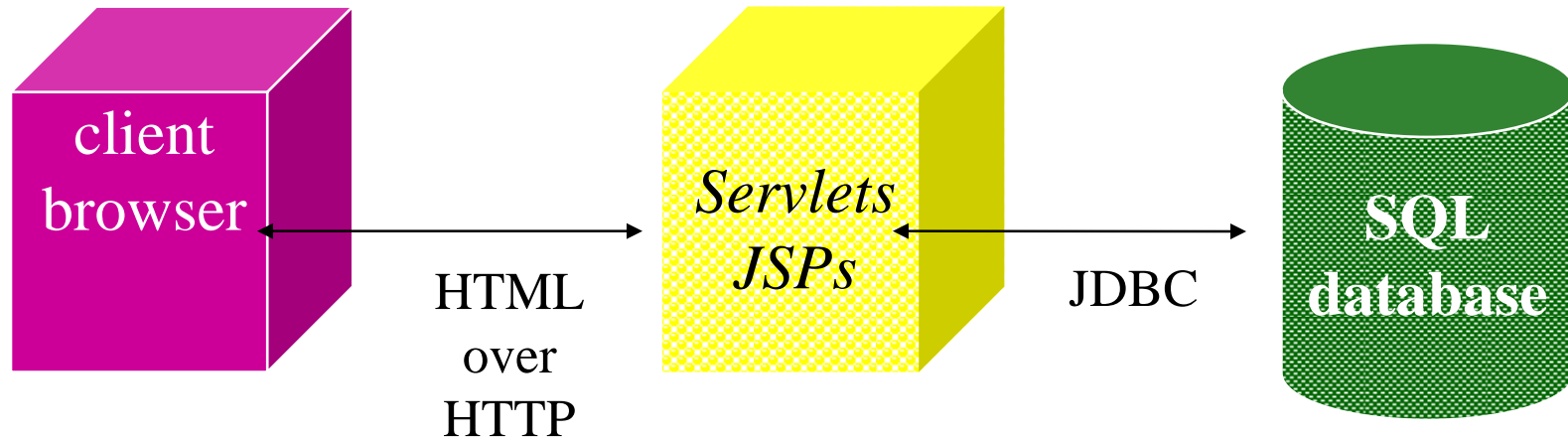
Java EE and Web Applications

- ◆ Java EE is Java's standard architecture for building scalable, distributed, reliable, Enterprise applications
 - EE is Enterprise Edition which simply means "Server Side" Java
 - These applications normally have many components:



Simple Web-centric Architecture

- ◆ A popular architecture for web-enabled systems



- ◆ Standalone clients are also possible
- ◆ Other backend information may be used
 - Sun often uses the term Enterprise Information System (EIS)

Java EE Web Applications

- ◆ **Web application** - A standard structure defined in Java EE for organizing the resources in a Java EE system
 - It is a collection of Servlets, JavaServer Pages, and other files
 - Most frequently collected inside of a Web Archive (WAR) file
 - This is the standard Java EE way to package up server programs for the Web
 - This WAR file can be deployed directly in application servers that support web applications (such as BEA WebLogic, and Red Hat JBoss*)
- ◆ Every web app has a standard directory structure and includes an XML configuration file - **web.xml**
 - Called the "Deployment Descriptor" (DD)

Web Application Structure

<Web application base directory>

 [static content files: HTML, forms, images, etc.]

 [dynamic content: JSPs]

 [other directories for content]

WEB-INF

 **web.xml**

**Web application
configuration file**

 **classes**

 [.class files: servlets and others]

 **lib**

 [JAR files]

Using Servlets

- ◆ Servlets are Java components that run on the server
- ◆ Servlets run **in response to client requests**
 - Servlets run within a **Web container** that exists as part of an application server
 - **Container** is a name for a set of services that must be available to a servlet for it to execute properly
- ◆ To **create a servlet**:
 - Subclass ***javax.servlet.HttpServlet*** class and override the ***doGet()*** method
 - Once we've created this class, you install it (in a WAR)
- ◆ When the servlet is invoked, the container executes the ***doGet()*** method whenever there is a "GET" request, and ***doPost()*** when there is a "POST" request
 - The resulting output will go back to the browser

A Simple HTTP Servlet

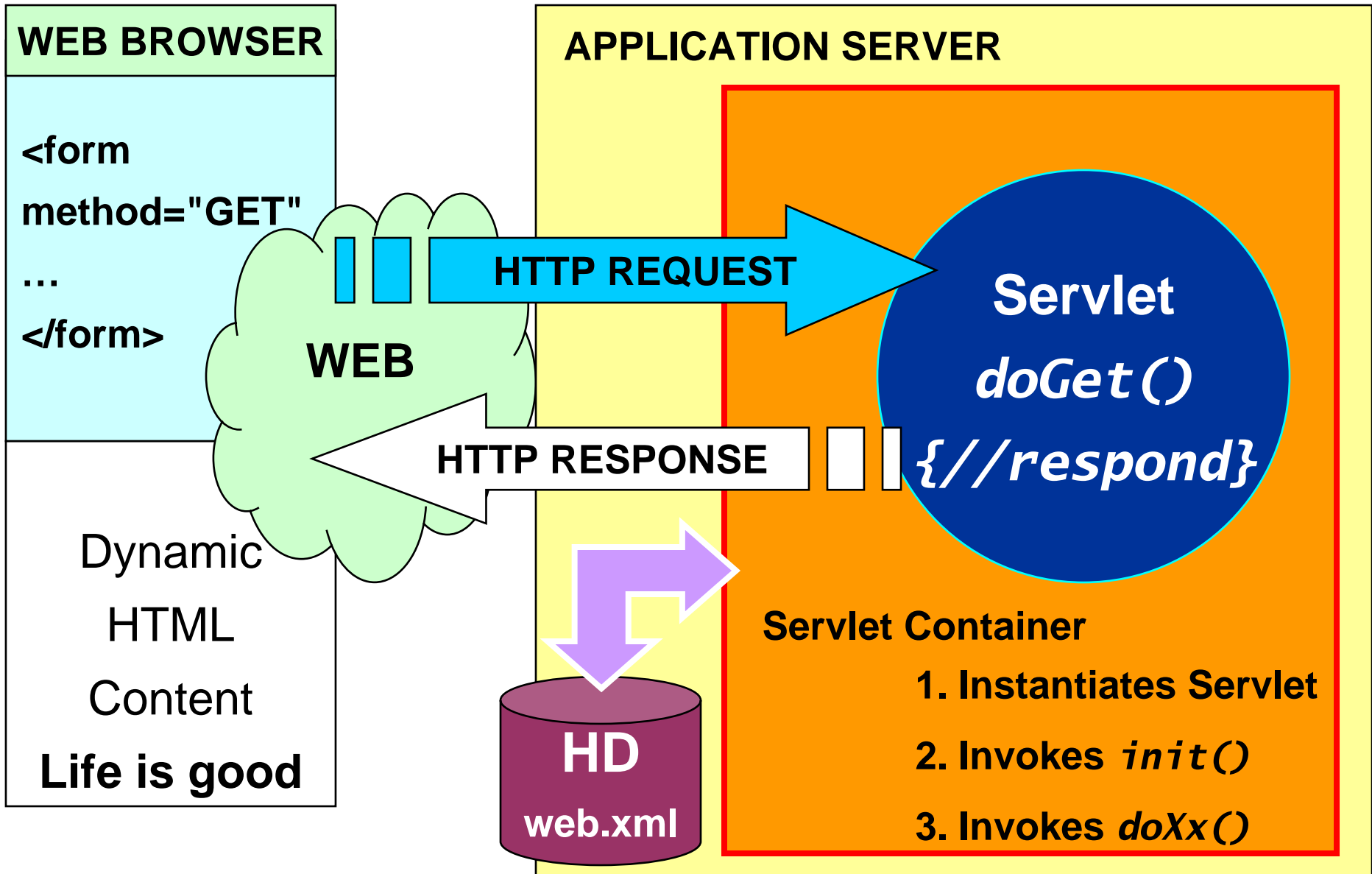
```
package com.javatunes.ajax;
import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;

public class HelloWorldServlet
extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        /* Here is where output is generated
        This gets sent back to the client */
        out.println("<b>Life is good</b>");
        out.close();
    }
}
```


How a Servlet works



The Web Archive (war) file

- ◆ Web applications are stored in **WAR** files
 - WAR file uses the ZIP file format, with a ".**war**" extension
- ◆ WAR contents include
 - JSP, html, and other content files
 - *web.xml* – web application deployment descriptor
 - Classes: servlets, JavaBeans and support classes
 - Custom tag libraries, other Java libraries
- ◆ The ***web.xml*** file is defined by the Servlet specification
 - Stored in the root of the WEB-INF directory
- ◆ Contains configuration information for the web app, including:
 - ServletContext init parameters, Session configuration
 - Servlet definitions and Servlet URI mappings
 - Mime type mappings, Error pages, Security, JNDI environment
 - Referenced resources such as EJBs or DataSources

Example web.xml file

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">

  <servlet>
    <servlet-name>HelloWorld</servlet-name>
    <servlet-class>
      com.javatunes.ajax>HelloWorldServlet
    </servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>HelloWorld</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
</web-app>
```

Deploying Web Applications

- ◆ Deployment of WAR files is application server specific
- ◆ For many application servers, you can copy WAR files to a specific directory, and the server will automatically deploy them
- ◆ For others, you must use deployment tools to deploy WAR files in the application server
 - Many app servers allow you to do both

Servlets and Ajax

- ◆ Servlets can be used to respond to Ajax requests
 - They are the preferred way to handle HTTP requests, and the mechanics (for the servlet) of handling an Ajax request are basically the same as handling a normal HTTP request
 - To the servlet, it is just another HTTP request
 - The servlet will have to generate a response appropriate for the Ajax request, which may be more complicated than handling a normal request
- ◆ Responses to Ajax requests can return many kinds of data
 - Plain text, HTML, XML
- ◆ Ajax doesn't necessarily require XML data
 - This was one of the early ways it was used, but it is not restricted to XML
 - In fact, very often HTML or JSON data is simpler and faster
 - We'll work with all of these kinds of response data in the course

Accessing the Servlet Using Ajax

- ◆ For an HTTP GET request, the access with Ajax is simple
 - Simply supply the servlet URL when making an XHR request
*req.open("GET", **"/javatunes/hello"**, true);*
- ◆ When using an HTTP **POST**, there is a little bit more to do
 - You need to set a request header to specify that the body of the request contains form data
 - You also need to send the form encoded data
 - The example below shows how you'd do that, and send two parameters (firstname and lastname) to the servlet

```
// Open an HTTP POST connection to the hello servlet
// Assume req is a variable initialized with an XMLHttpRequest object
req.open("POST", "/javatunes/hello", true);
// Specify that the body of the request contains form data
req.setRequestHeader("Content-Type",
    "application/x-www-form-urlencoded");
// ... Other code omitted
// Send form encoded data
req.send("firstname=Jane&lastname=Programmer");
```

A Servlet Handling a Post Request

- ◆ Below, we've used a *doPost()* method, and also accessed the parameters sent with the request using *getParameter()*

```
// Imports, package statements omitted
public class HelloWorldServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        String name = request.getParameter("firstname") + " " +
                       request.getParameter("lastname");

        /* Here is where output is generated
        This gets sent back to the client */
        out.println("Hi " + name + "<br/>");
        out.println("<b>Life is good</b>");
        out.close();
    }
}
```

Lab 4.1 – Generating Ajax Data with a Servlet

- ◆ **Overview:** In this lab you'll work with a simple servlet to generate data in response to an Ajax generated request
 - The response data will be very simple, and just serve to illustrate the principals

- ◆ **Objectives:**
 - Practice working with Java EE Web applications
 - Learn how to use a servlet to generate data in response to an Ajax request

- ◆ **Builds on previous labs:** Lab 3.2
 - Continue working in your **Lab03.1** directory

- ◆ **Approximate Time:** 30-40 minutes

Tasks to Perform

- ◆ Open the file **src\com\javatunes\ajax\AjaxServlet.java** for editing, and finish the method, **doGet()**
 - Look for the comments with **TODO** in them
- ◆ The skeleton of the method is complete, but you will need to add in the code to complete the following functionality
 - Set the content type ("text/plain" or "text/html" will work)
 - Get a *PrintWriter* from the response
 - Print some content (keep it simple) using the *PrintWriter*
 - Close the *PrintWriter*
- ◆ Open the file **WEB-INF\web.xml**
 - The skeleton of this file is done, and you'll need to fill in details
 - Look for **TODO** comments in the file

Continued ⇒

Tasks to Perform

- ◆ In *web.xml*, you'll need to:
 - Remove the comments around the `<servlet>` and `<servlet-mapping>` elements
 - Replace the TODOs in those elements with appropriate values to map *AjaxServlet* to */ajax*
- ◆ Open the file *hello_ajax.html*
 - Change the URL for your existing Ajax call to access your servlet (*/javatunes/ajax*)
- ◆ **Restart** Tomcat and browse back to your web app
 - You should see the data from the servlet in the resultant page
- ◆ You are now accessing dynamic data generated by the servlet, instead of a static text file

Tasks to Perform

- ◆ In this part, we will modify our code to support **POST**
- ◆ Open *ajax.js*, and modify the *makeXHRrequest()* function to have the following signature
function makeXHRrequest(method, url, divID, postArg)
 - ***method***: The HTTP method ("GET" or "POST")
 - ***url***: URL to make the request to (as before)
 - ***divID***: id of the element whose content is changed (as before)
 - ***postArg***: An argument for a POST call (if needed)
 - After getting the XHR object, call *open()* using the *method* arg that was passed in, rather than hard coding GET or POST
 - Before calling *send()*, check if the *method=="POST"*, and if so
 - Set the request header needed for POST arguments
 - When you call *send()* use *postArg* for the post arguments

Tasks to Perform

- ◆ Open *hello_ajax.html*, and do the following:
 - Modify the existing call to *makeXHRrequest* to include a first argument whose value is "GET" (the HTTP method)
 - Add another *<div>* element with an *id="postDiv"*
 - Add in another call to *makeXHRrequest()*, using "POST" as the HTTP method, */javatunes/ajax* as the URL, *postDiv* as the element id, and the following as the post (last) argument:
"firstname=Jane&lastname=Programmer"
- ◆ Open *AjaxServlet.java*, and add a *doPost()* method
 - To start, copy your *doGet()* method, and change the name
 - Add code to *doPost()* to retrieve the *firstName* and *lastName* parameters
 - Have *doPost()* create some HTML output that includes the value of the above two parameters
- ◆ **Restart/Reload**, and look at the results in the browser

STOP

Overview of JavaServer Pages (JSP)

Overview of Servlets

Overview of JavaServer Pages (JSP)

What is a JSP?

- ◆ A JSP is like an HTML page on steroids
 - Lets you mix regular HTML with dynamic content
 - Allows you to mix in dynamically generated content with regular (static) HTML
 - It can include content sent to the JSP - as we will see later
- ◆ A JSP consists of two types of data:
 - **Template Data** consists of regular text, HTML, XML ...
 - It doesn't change from request to request
 - **Dynamic Data** is generated anew for each request
 - Tags, scripting elements, and standard actions can all generate dynamic data
 - They are evaluated for each request and enable the JSP to produce different output as appropriate

A Very Simple JSP - simple.jsp

```
<HTML>
  <HEAD>
    <TITLE>
      The time of day
    </TITLE>
  </HEAD>
  <BODY>
    <CENTER>
      Current date: <%= new java.util.Date() %>
    </CENTER>
  </BODY>
</HTML>
```

◆ It's mostly HTML

- Except for the Java code between the `<%= %>`
- We'll soon see ways to do this with tags also

JSPs Look Like HTML

- ◆ JSPs are HTML with some extra tags

<HTML> </HTML>

- ◆ Some of the extra tags are like this:

<jsp: . . . />

- ◆ JSTL (JSP Standard Tag Library) tags might look like this:

<c:out . . . />

- ◆ Other JSP tags have the following beginning and ending:

<% . . . %>

- ◆ JSP files have a *.jsp* extension
 - Which is how the server knows to treat them like JSPs

JSP Expressions

- ◆ Expression syntax:

<%= any valid Java expression %>

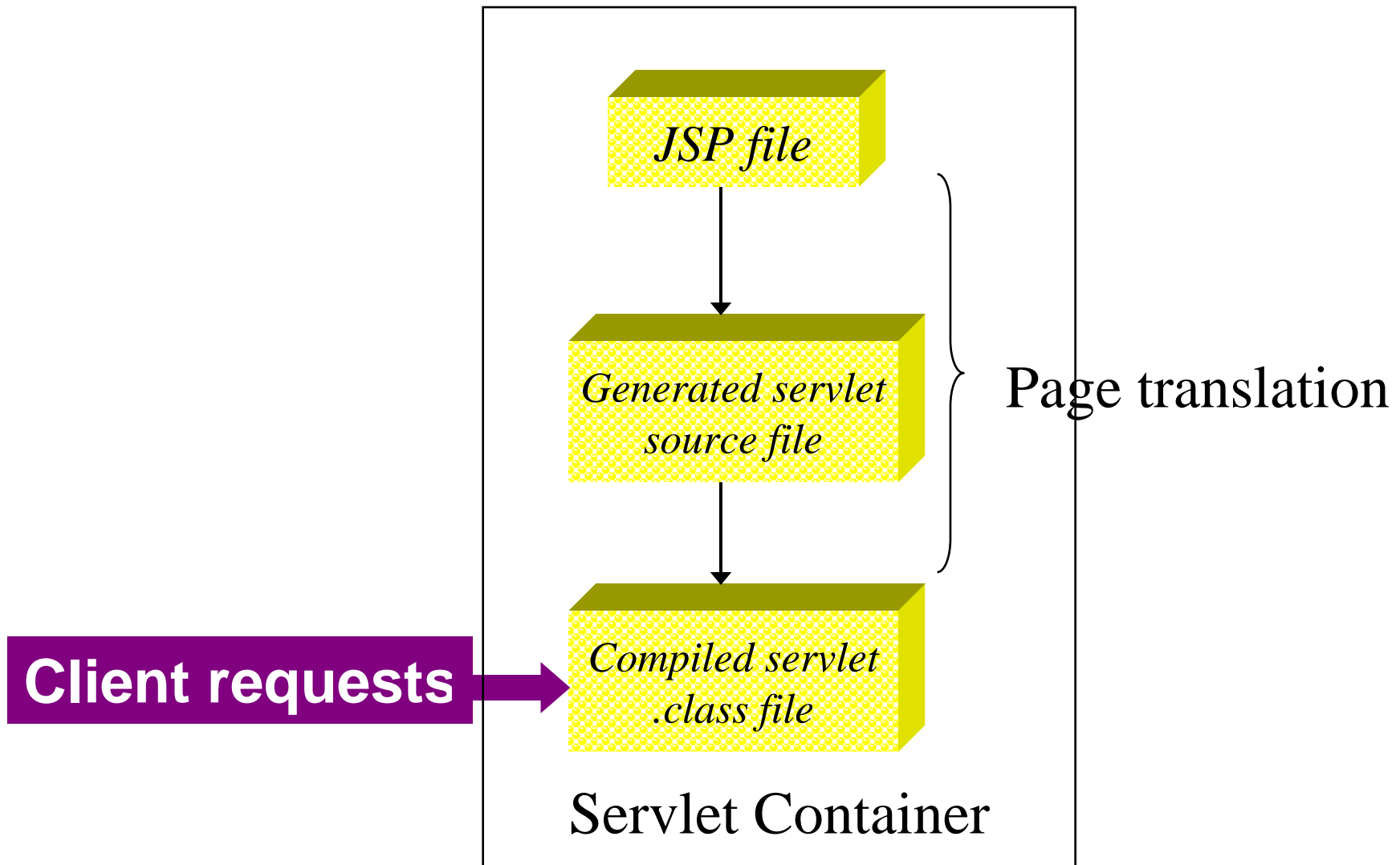
1 + 1 = <%= 1+1 %>

- ◆ Expressions are used to insert values directly into the output
 - The syntax is like a parameter to a println
 - No semicolon used at end of the expression
- ◆ The expression is evaluated, converted into a string, and then inserted into the page
 - The evaluation is done at run time!
 - The expression has access to all information about the current request

JSPs are Really Servlets

- ◆ They look like HTML, but behind the scenes, they are converted to a servlet
 - JSPs are syntactic sugar, a convenient way to work in a primarily HTML world. They are servlets in disguise.
- ◆ When a client first requests a JSP:
 - The server creates a servlet source file from the JSP
 - The generated servlet source code is compiled
 - The server runs the servlet
- ◆ In subsequent requests:
 - The server just runs the generated servlet
 - The server can regenerate the servlet if the JSP changes
- ◆ We don't really need to deal with this to use JSP
 - But it's good to know what's really going on

Lifecycle of a JSP



Object Buckets or Scopes

- ◆ Servlets/JSPs must share data in order to cooperate
- ◆ The Servlet API offers three main buckets, or **scopes**, for holding shared objects (see *javax.servlet*)
 - The *ServletRequest* or **request** scope
 - The *ServletContext* or **application** scope
 - The *HttpSession* or **session** scope
- ◆ To access the scope, you use the associated object
 - For example, *ServletRequest* for the request scope
 - You use the ***setAttribute()*** and ***getAttribute()*** methods to put things on and take things off the scope

```
request.setAttribute("president", "Abraham Lincoln");
```

Predefined JSP variables - Implicit Objects

- ◆ There are some important variables that are always available (**implicit objects**), and may be referenced directly
 - **request** - The *HttpServletRequest* associated with the request
 - **response** - The *HttpServletResponse* associated with the request
 - **application** - The *ServletContext*
 - **session** - The session object. we will talk about this later
 - **out** - The *PrintWriter* used to sent output to the client using *print()*, *println()*, etc.
- ◆ For example, to display the user's browser in a JSP page:
`<%= request.getHeader("User-Agent") %>`

Working with `<jsp:useBean>`

- ◆ Built in JSP tag
- ◆ Defines a bean in the scope of the page
- ◆ A Way to map the model (the bean) into the view (JSP)

```
<jsp:useBean id="myutil"  
  class="com.javatunes.util.SearchUtility"  
  scope="request" />
```

- ◆ The above maps a bean of type ***SearchUtility*** into ***request*** scope, with the name ***myutil***
- ◆ It instantiates (creates) the bean if it does not already exist (null) on that scope

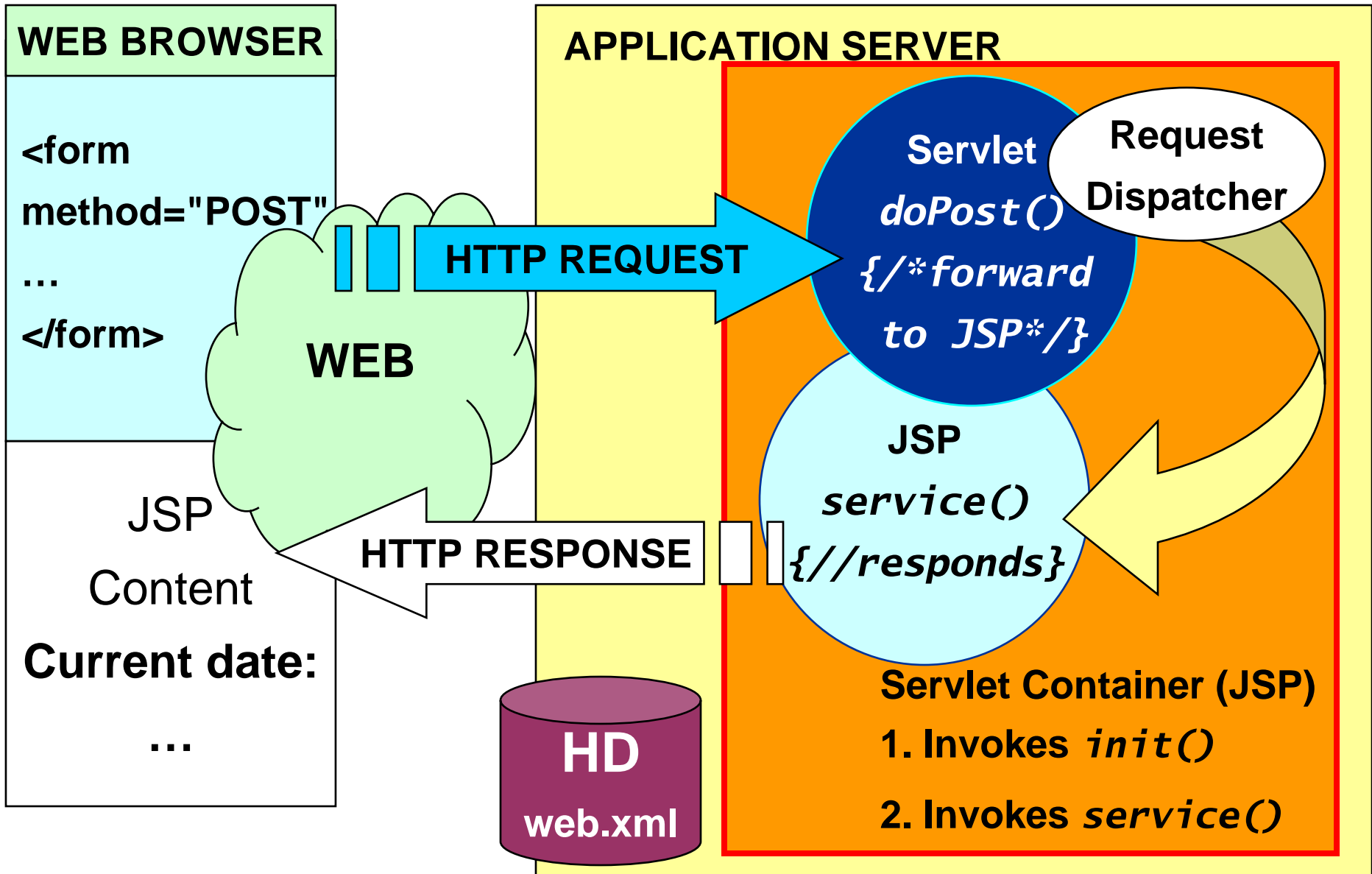
More <jsp:useBean>

```
<jsp:useBean id="myBean"  
  class="com.javatunes.util.SomeBean"  
  scope="request" />
```

- ◆ The above **<jsp:useBean>** tag would become* this code:

```
Object o = request.getAttribute("myBean");  
com.javatunes.util.SomeBean myBean = null;  
if (o != null) {           //casting needed  
    myBean = (com.javatunes.util.SomeBean) o;  
} else {                   // instantiation if null  
    myBean = new com.javatunes.util.SomeBean();  
}
```


How a Servlet works with a JSP



Issues With JSP

- ◆ Most web applications have a number of JavaServer Pages (JSP) to manage the client VIEW (remember MVC)
 - JSP forms
 - JSP display pages (for rendering requests for data, etc.)
 - Whatever else the view needs to accomplish
- ◆ In the past, JSPs have made wide use of **JSP scriptlets**
 - Allows virtually ANY Java code to be written directly into the JSP page itself
 - Powerful and way cool
 - Hard to maintain (readability, complexity, etc.)
 - Steep learning curve for non-Java programmers

Custom Tags

- ◆ Sun's solution to the Java code in JSPs is **Custom Tags**
- ◆ A tag is actually a JSP reference (of sorts) to a **Java object** that gets instantiated on the back end to “handle the tag”
 - When the Application Server sees the tag, it makes sure that the tag handler class is instantiated, passes any attributes over, and invokes appropriate methods on the tag handler object
 - Generally, in the generated servlet for the JSP
 - Someone has to write a tag handler class for each custom tag
 - Beyond the scope of this course
- ◆ Some Custom Tag Capabilities
 - **Insert text into a page**: (generally from Java Beans)
 - **Flow of control**: Control presentation based on conditions
 - **Iteration**: Without Java scripting
 - **Nesting**: Custom tags can be nested within each other

Custom Tags and Tag Libraries

- ◆ Custom tags are implemented using Java objects
 - Called **tag handlers**, and they implement the tag functionality
- ◆ A **tag library** is used to hold a collection of custom tags, and usually consists of:
 - A **TLD** (Tag Library Descriptor) - An XML descriptor file
 - A **taglib URI** uniquely identifying the tag library
 - A **JAR** file which holds the .class files and the TLD
- ◆ Called by a simple tag in the JSP, as with **c:out** shown below
<c:out value="Hello Custom Tag World" />
 - The Container invokes the Java object transparently when it sees the tag in a JSP page

The JSTL

- ◆ Sun has specified a standard tag library for JSP
- ◆ **JSTL** - **J**avaServer Pages **S**tandard **T**ag **L**ibrary
 - Contains an extensive set of tag libraries - we'll use the **core** tag library - containing conditional logic, loops, output, URL, etc
- ◆ Some of the core library tags we will use are:
 - **<c:out>**: Outputs a value into the JSP page
 - **<c:forEach>**: Basic iteration tag
- ◆ JSTL also defines an **Expression Language** (EL) that simplifies access to Java language constructs in JSP
 - JSTL EL expressions have the general form:
`${expression}`
 - EL values will often be the properties of JavaBeans
`${beanName.propertyName}`

taglib Directive in JSP

- ◆ A **taglib** directive is used in a JSP to notify the container that the page relies on a tag library
 - Makes the tags in the library available for use in the page
- ◆ The taglib directive specifies two things
 - **uri**: Declares which tag library is being used (see note)
 - **prefix**: Abbreviation for the library **in this JSP page only**
- ◆ Below, we show a taglib directive for the JSTL core library
 - And some examples of using the *c:out* tag

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<c:out value="Any text rendered at runtime" />
<c:out value="${someBean.someProperty}" />
```

Example - Custom Tags in a JSP file

- ◆ The example shows some sample JSTL usage
 - It assumes there is a bean "student" in scope *

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
      prefix="c" %>
```

You can use tags in many different ways...

```
<c:out value="${student.name}" />
```


Or something more complex, such as this...


```
<c:choose>
```

```
  <c:when test="${student.martialArt == 'Aikido'}">
    Morihei Ueshiba created Aikido...
```

```
  </c:when>
```

```
  <c:when test="${student.martialArt == 'Judo'}">
    Professor Jigoro Kano founded Judo...
```

```
  </c:when>
```

```
  <c:when test="${student.martialArt == 'Jeetkundo'}">
    Bruce Lee made up his own style of no-style...
```

```
  </c:when>
```

```
  <c:otherwise>
```

```
    Maybe try Tai-Chi, it's fun...
```

```
  </c:otherwise>
```

```
</c:choose>
```

A Servlet and JSP Cooperating

- ◆ Servlets can work with other resources
- ◆ A *RequestDispatcher* is used to send the request from one resource to another within a servlet container
 - You get a *RequestDispatcher* from the *ServletContext*
- ◆ Use *RequestDispatcher.forward()* to forward a request to another resource
 - Pass along the same *request* and *response* objects
 - The fragment below shows a servlet putting data on the request, then forwarding to a JSP page

```
List cartObj = // Initialized somehow
request.setAttribute("cart", cartObj);
String otherResource = "/cartDisplay.jsp";
ServletContext ctx = getServletContext();
RequestDispatcher rd =
    ctx.getRequestDispatcher(otherResource);
rd.forward(request, response);
```


The `<c:forEach>` tag

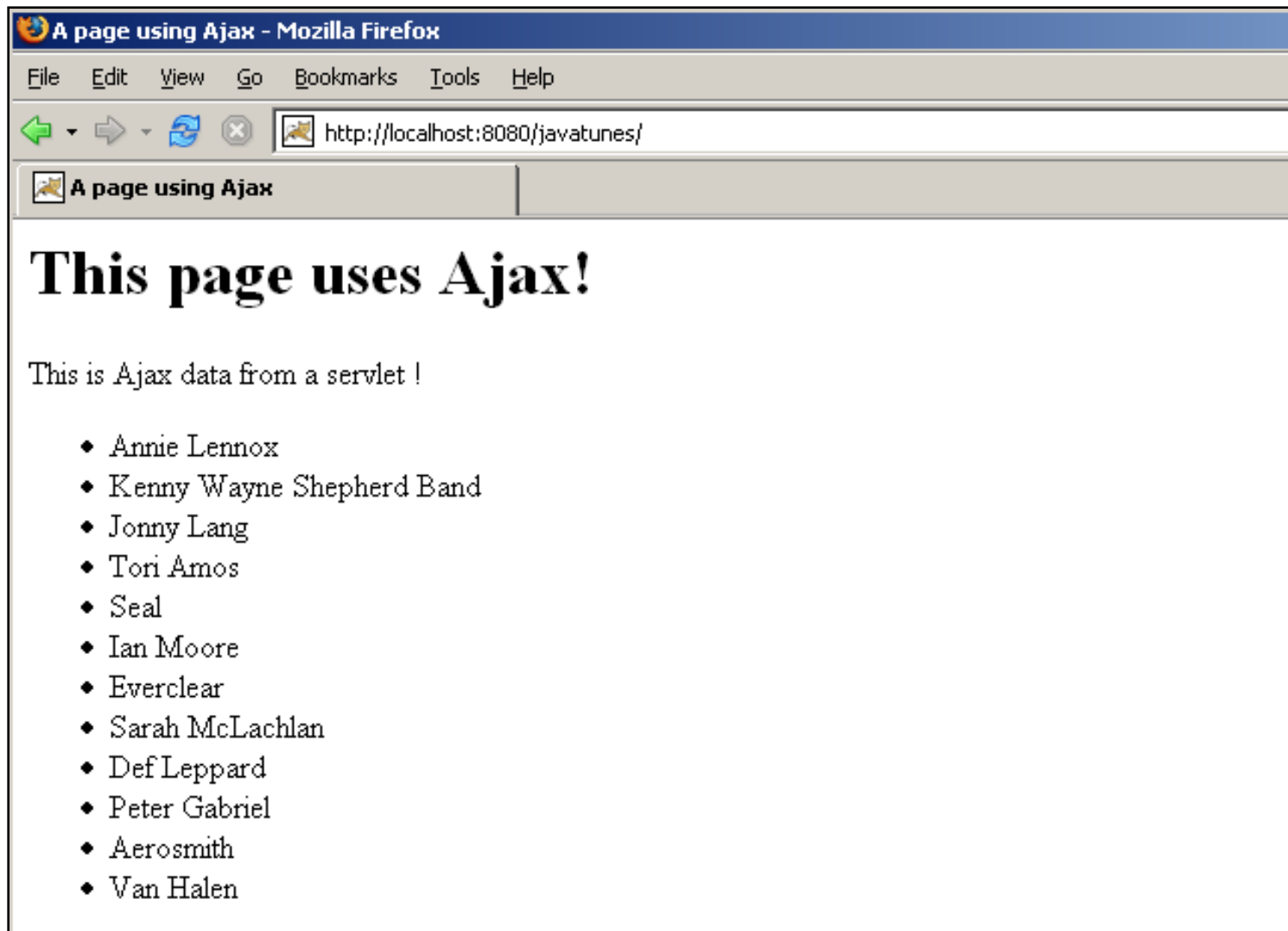
- ◆ `<c:forEach>` is a JSTL tag for iteration over a collection
 - We show an example below, using the following tag attributes
 - *items*: Refers to the collection itself
 - *var*: Variable name to access current item in the loop
 - It uses the cart put on the request in the servlet example
 - Assume the cart holds beans with *title* & *price* properties

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
      prefix="c" %>
<!-- Much detail omitted -->
<table>
  <c:forEach items="${cart}" var="item">
    <tr>
      <td><c:out value="${item.title}" /></td>
      <td><c:out value="${item.price}" /></td>
    </tr>
  </c:forEach>
</table>
```



Lab 4.2 – Generating Ajax Data with a JSP

- ◆ **Overview:** In this lab you'll work with a simple JSP to generate data in response to an Ajax generated request
 - The response data will be an unordered list, generated by a JSP page with data from a servlet
- ◆ **Objectives:**
 - Learn how to use a JSP to generate HTML using data from a servlet in response to an Ajax request
- ◆ **Builds on previous labs:** Lab 4.1
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 30-40 minutes
- ◆ Note that this lab uses some functionality from JavaTunes
 - JavaTunes is an online music store that is used in our labs



- ◆ In this lab, you'll use two utility classes supplied in the setup
 - These classes are part of the JavaTunes online store
- ◆ ***com.javatunes.util.MusicItem***
 - A JavaBean-style value class that represents music items
 - It has an artist property (*getArtist()*/*setArtist()* methods) that is of type *String*
- ◆ ***com.javatunes.util.SearchUtility***
 - Used to search for items in the store, and contains the following public method (among others):
 - ***static Collection<MusicItem> findByArtist(String artist)***
 - *artist* - search string
 - returns a collection of *MusicItem* that match the argument (matches substrings of artists of items in the store)

Tasks to Perform

- ◆ Open the file *hello_ajax.html*, and modify the POST call to *makeXHRrequest()* to have a POST (last) argument of **"artist=a"**
- ◆ Open the file ***AjaxServlet.java*** to edit, and modify ***doPost()*** by commenting/removing the old code, & doing the following:
 - Retrieve the request parameter named **"artist"**
 - Call *SearchUtility.findByArtist()* with the retrieved parameter and put the results of that call on the request with the name **"suggestions"**
 - See notes for some sample code
 - Get a *RequestDispatcher* & forward to */jsp/ajaxData.jsp*
 - The data this servlet puts on the request, will be used by a JSP to generate an unordered list in response to an Ajax request

Tasks to Perform

- ◆ Open the file *WebContent/jsp/ajaxData.jsp* and do the following
 - Look for the **TODOs** in the file
 - Complete the `<%@ taglib .. %>` element to use the core JSTL tag library with a prefix of `c` (see earlier examples)
 - Complete the `<forEach ...>` so that you are using `"suggestions"` as the collection, and a variable named `item`
 - Within the `<forEach>` print out the artist property of each item within ` ... ` tags (Note the `` outside the `forEach`)
 - Use `<c:out>` to access the artist property
- ◆ **Restart** Tomcat and browse to your application
 - You should see an HTML unordered list generated by the JSP



Review Questions

- ◆ What is a Servlet?
- ◆ When creating a Servlet, what class do you extend? What method do you need to define?
- ◆ What is a JSP?
- ◆ In which cases do you use a JSP? A Servlet?
- ◆ What are the 3 scopes in the Servlet API? What is the difference between these scopes?
- ◆ How are web applications packaged?
- ◆ How are web applications deployed?

Lesson Summary

- ◆ Servlets are Java components that run on the server, in response to client requests
- ◆ When creating a servlet, you extend *HttpServlet* and implement the *doGet()* or *doPost()* method
- ◆ A JSP is a document centric view of a Servlet
 - A JSP becomes a Servlet.
- ◆ The three scopes in the Servlet API are **request**, **session** and **application**
 - The JSP API adds a scope called **page**
 - Request scope is for the duration of the request
 - Session scope is for the entire user session (has a timeout).
- ◆ Web applications are packaged into WAR files.



Session 5: More JavaScript and Ajax

Browser Event Model
JavaScript Objects and Arrays
Classes in JavaScript

Lesson Objectives

- ◆ Understand the basics of the event model in browsers
- ◆ Use events to trigger Ajax requests and update a web page
- ◆ Understand the basics of JavaScript objects
- ◆ Be able to use JavaScript objects to better structure your Ajax programming

Browser Events

Browser Event Model

JavaScript Objects and Arrays

Classes in JavaScript

Event Based Programming

- ◆ User interfaces need to respond to user actions
 - Submitting a form is just one possible way to allow a user to act
 - It is a coarse-grained and relatively inflexible action
- ◆ JavaScript allows you to define **event handlers** at a fine grain
 - Event handlers consist of JavaScript code defined as attributes of HTML elements
 - They are automatically invoked by the web browser when the associated event occurs
 - For example, you define an event handler for when a button is clicked by defining JavaScript code for its **onclick** attribute
 - In the example below, every time the button is clicked, a message is displayed via *alert()*

```
<input type="button" name="myButton" Value="Click Me"  
      onclick="alert('myButton was pressed');"/>
```

Event Handlers

- ◆ Event handlers are simply arbitrary strings of JavaScript code
 - Multiple statements must be separated by a semicolon
`<input type="button" name="myButton" value="Click Me" onclick="numButtonClicks++; alert('click # ' + numButtonClicks);"/>`
 - This can be cumbersome if you have multiple statements
- ◆ It's more common to define a function for multiple statements, and then call the function in the event handler

```
<!-- Much detail omitted -->
<script>
    var numButtonClicks = 0;
    function buttonClicked() {
        numButtonClicks++;
        alert ("button click number " + numButtonClicks);
    }
</script>
<!-- Much detail omitted -->
<form>
    <input type="button" name="myButton" value="Click Me"
        onclick="buttonClicked()"/>
</form>
```

Defined Browser Events

- ◆ There are many events that may occur in browsers
 - For example, a mouse click, moving the mouse over a link or image, submitting an HTML form
- ◆ Here is a list of the standard events that exist in all browsers
 - **onabort**: Image loading is interrupted
 - **onblur**: Element loses input focus
 - **onchange**: Field loses focus, and its content changed
 - **onclick**: Mouse clicks an object
 - **ondblclick**: Mouse double-clicks an object
 - **onerror**: An error occurs when loading a document or an image
 - **onfocus**: An element gets focus
 - **onkeydown**: A keyboard key is pressed
 - **onkeypress**: A keyboard key is pressed or held down
 - **onkeyup**: A keyboard key is released

Defined Events

- ***onload***: A page or an image is finished loading
- ***onmousedown***: A mouse button is pressed
- ***onmousemove***: The mouse is moved
- ***onmouseout***: The mouse is moved off an element
- ***onmouseover***: The mouse is moved over an element
- ***onmouseup***: A mouse button is released
- ***onreset***: The reset button is clicked
- ***onresize***: A window or frame is resized
- ***onselect***: Text is selected
- ***onsubmit***: The submit button is clicked
- ***onunload***: Document or frameset unloaded

Form Validation

- ◆ Event handlers are often used for form validation
- ◆ They can be used to check individual fields, as shown below
`<input type="text" id="email" onchange="validateEmail();">`
- ◆ The *onsubmit* handler is often used to validate an entire form
 - Note also that the event handler return value here is used
`onsubmit="return validateForm();"`
 - The return tells the browser to check event handler return code, and if it is false, to NOT take the default action (submit the form)

```
<!-- Much detail omitted -->
<script>
    function validateForm() {
        // Validate form and return true/false based on validation
    }
</script>
<!-- Much detail omitted -->
<form onsubmit="return validateForm();">
    <!-- Form elements omitted -->
</form>
```

onload and onunload Events

- ◆ **onload** is fired after the page has completely loaded
 - The document is parsed, all scripts have run, and all auxiliary content like images have loaded
 - You can register an *onload* handler by setting the *onload* attribute of `<body>`
 - When it is triggered, the document is fully loaded and parsed, and JavaScript can manipulate any part of the document
 - *onload* handlers should not call *document.write()*, as that would overwrite the contents of the entire document
- ◆ **onunload** is fired when a user navigates away from a page
 - Set the *onunload* attribute for `<body>` to set up an *onunload* handler
 - It can be used to undo effects of an *onload* handler
 - For example, closing windows that were opened by *onload*

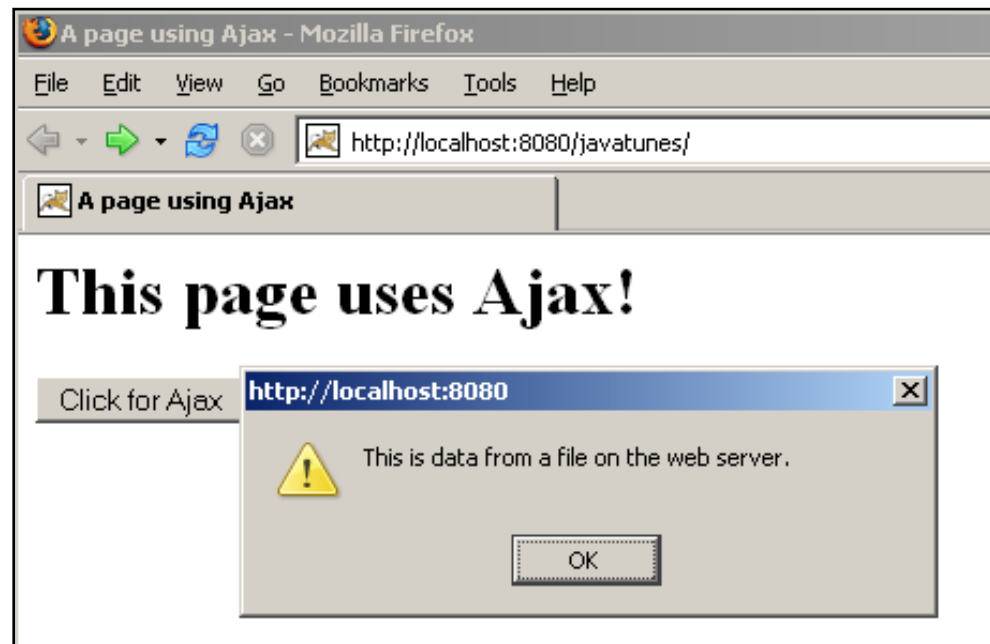
Using Ajax and Events

- ◆ It's simple to include Ajax calls in an event handler
 - They are normal functions, and you can include any JavaScript code you want, including Ajax requests

```
<!-- Much detail omitted -->
<script>
    function callAjax() {
        var req = getXMLHttpRequest(); // Defined in js/ajax.js
        req.open("GET", "/javatunes/data.txt", false);
        req.send(null);
        alert(req.responseText);
    }
</script>
<!-- Much detail omitted -->
<input type="button" name="ajaxButton" value="Click for Ajax"
        onclick="callAjax()"/>
```

Ajax and Events

- ◆ When the button is clicked, the event handler is called
 - It makes an Ajax request, and displays the data – though of course in a real application you would update part of the page
- ◆ This type of event handling lies at the core of making responsive user interfaces with Ajax
 - In response to browser events, without a page refresh, you can update the web page with data from the server





Lab 5.1 – Use Events to Trigger Ajax

- ◆ **Overview:** In this lab you will write a JavaScript event handler that invokes an Ajax request
 - You'll use the Ajax code you've written from previous labs
- ◆ **Objectives:**
 - Gain experience using events in a web page
 - See how Ajax and events can dynamically modify a portion of a web page with server data in response to a user action
- ◆ **Builds on previous labs:** Lab 4.2
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 15 minutes

Tasks to Perform

- ◆ Open the file *hello_ajax.html* from the previous lab for editing
- ◆ Add in a `<form>` with a button control immediately after the `<div>` with the id "testDiv"
 - Remove the *makeXHRrequest()* call that does an HTTP GET from the script in the HTML body, and add it in as the event handler to the button control
 - Remember that this call should use HTTP **GET**, with a URL of */javatunes/ajax*, and a div id of **testDiv**
- ◆ **Restart** Tomcat, reload in Firefox and try your application
 - When you press the button, your div element should have its content replaced with the Ajax content
 - If you have problems, check for errors using Firebug



JavaScript Objects and Arrays

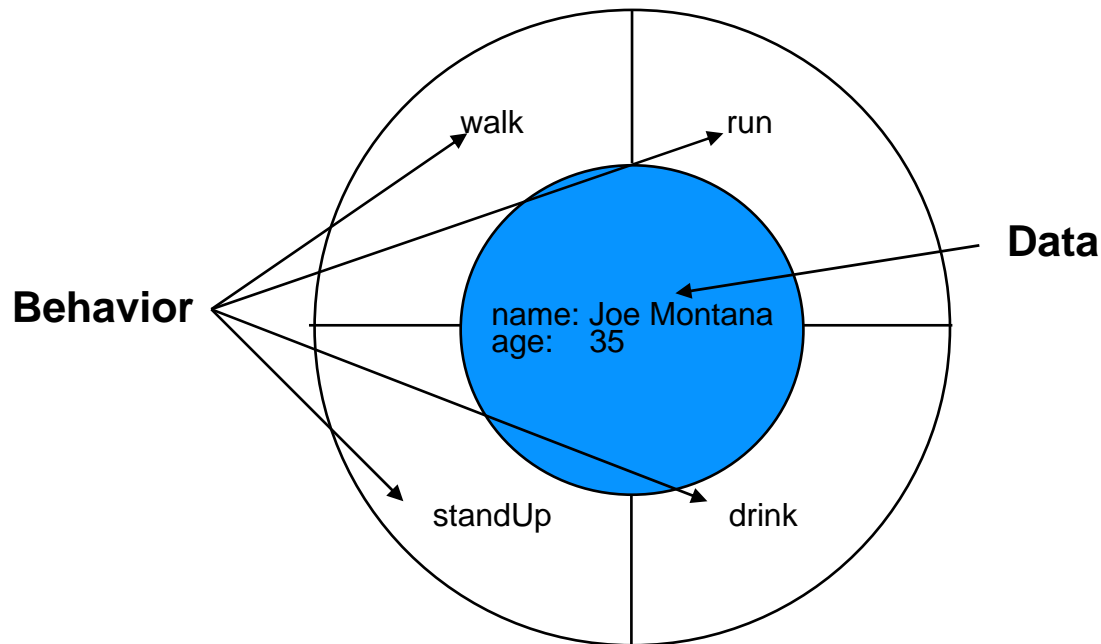
Browser Event Model

JavaScript Objects and Arrays

Classes in JavaScript

JavaScript Objects

- ◆ JavaScript has objects with properties and methods
 - An objects is a representation of a concept, abstraction, or thing
 - Objects have properties and methods that are used to manipulate the object and its properties
 - For example a person might have a name and age, and can behave in various ways



Creating JavaScript Objects

- ◆ Objects aggregate multiple properties and methods into a single unit – making them much easier to work with
 - The properties are name/value pairs
 - A method is just a JavaScript function attached to an object
- ◆ The simplest way to create an object is with an **object literal**
 - This is a comma separated list of **property** name/value pairs enclosed in braces
 - You generally initialize a variable with the property
 - You can then access the properties using dot notation

```
var empty = {};           // An empty object
var point = {x:0, y:0};   // Object with x, y properties
alert(point.x);           // Display x property of aPoint
```

Working with Objects as Function Args

- ◆ You can also pass objects as function arguments, or return them, just as you do simple values like strings or numbers
 - Below we pass an object representing an HTML `<div>` element

```
<html>
  <head>
    <title>A page using JavaScript</title>
    <script type="text/javascript" >
      function displayContent(obj) { // Pass an object as arg
        alert(obj.innerHTML);      // Use the object
      }
    </script>
  </head>

  <body>
    <div id="targetDiv"><p>Paragraph in a div element</p></div>
    <script>
      var targetDiv = document.getElementById('targetDiv');
      displayContent(targetDiv); // Pass the object as an arg
    </script>
  </body>
</html>
```

Working with Objects as Function Args

- ◆ If a function takes multiple arguments, it is often clearer to have the arguments be objects
 - Below, we show an example of a function that expects objects (with x and y properties) as arguments, and then an example of calling the function with object literals
 - We can do this more easily, and more formally, with classes, which we cover soon

```
function isASquare(pt1, pt2) {  
    // Check if length and width are the same  
    if ( (pt1.x-pt2.x) == (pt1.y-pt2.y) ) {  
        alert("The points make a square");  
    }  
    else {  
        alert("The points DON'T make a square");  
    }  
}  
  
isASquare( {x:4, y:2}, {x:6, y:4} );
```

Working with Object Properties

- ◆ There are a number of ways to work with object properties
- ◆ You can loop through all the properties with a *for/in* loop
for (variable in object) { / code */ }*
 - The body of the loop is executed once for each property, and the variable takes on the name of the property
- ◆ You can check for a properties existence using *in*
- ◆ You can delete a property using *delete*

```
var obj = {x:0, y:1};

for (var propName in obj) { // Loops through props x and y
    alert(propName);
}

// If obj has a property named x, then set it to 1
if ("x" in obj) obj.x=1;

delete obj.x; // Remove the property from the object
```

Arrays in JavaScript

- ◆ An array is an ordered and indexed collection of values
 - Each **element** in the array has a numerical **index**
 - Elements in the array can be of any type
- ◆ You can create an array using an array literal, as shown below

```
var empty = []; // Array with no elements
var oneToFive = [1,2,3,4,5]; // Array with numbers 1 to 5
var points = [ {x:0,y:1}, {x:1,y:2} ] // Array with two objects
```

- ◆ You can access elements of an array using the **[]** operator and a numerical index
 - Indexing starts at 0, as in Java
 - This is used for both reading and writing

```
var firstNumber = oneToFive[0]; // Read the first element
points[1] = {x:2, y:2}; // Change the second element
```

Working with Arrays

- ◆ You can add an element to an array by assigning a value to it
 - Elements don't have to be contiguous
- ◆ All arrays have a **length** property specifying the number of elements in the array
 - This is always one larger than the largest index in the array
 - It is often used for looping

```
points[2] = {x:3, y:5}; // Add another point to end of array

for (var i=0; i<points.length; i++) { // Iterate through array
    alert(points[i].x + "," + points[i].y);
}

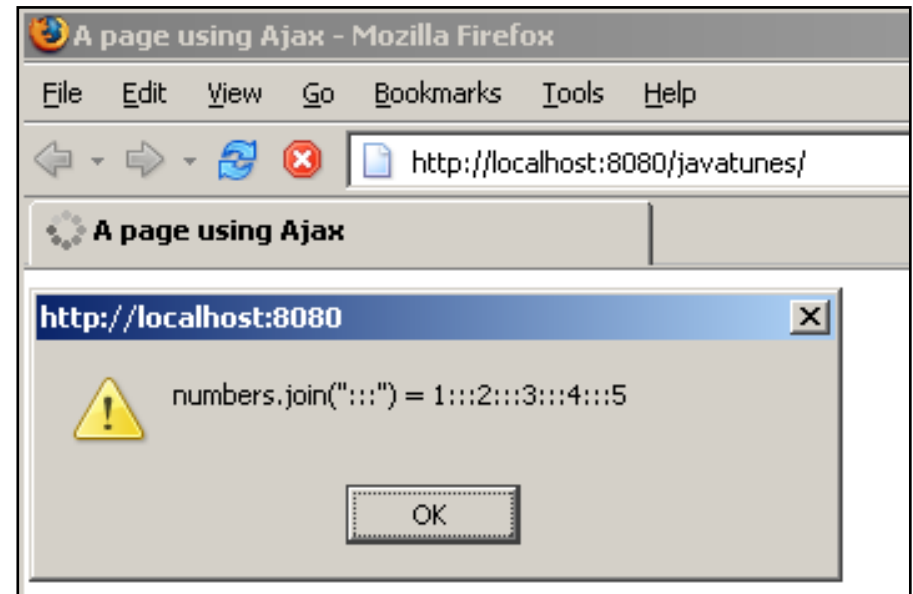
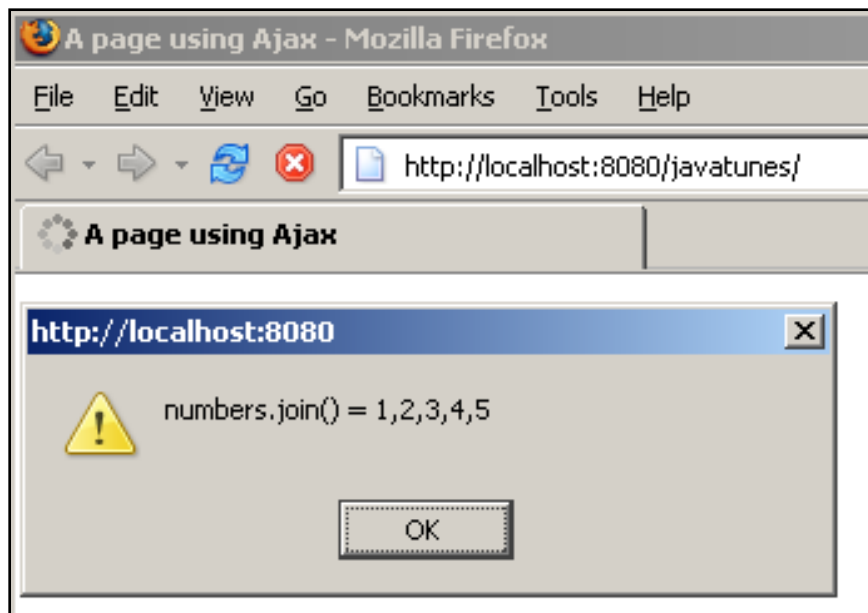
// If the array is not contiguous, check for existence
points[5] = {x:6, y:6}; // points[3],points[4] undefined
for (var i=0; i<points.length; i++) {
    if (points[i]) alert(points[i].x + "," + points[i].y);
}
```

Array Methods

- ◆ All arrays have the following methods defined
 - **concat()**: Joins two or more arrays and returns the result
 - **join()**: Puts all the elements of an array into a string. The elements are separated by a specified delimiter
 - **push()**, **pop()**: Treats array like a stack. Adds element to end of array and returns length, or removes and returns last element
 - **reverse()**: Reverses the order of the elements in an array
 - **shift()**, **unshift(e1)**: Treats array like a LIFO. *shift()* removes & returns first element. *unshift()* adds e1 to start of array & returns length
 - **slice()**: Returns selected elements from an existing array
 - **sort()**: Sorts the elements of an array
 - **splice()**: Removes and adds new elements to an array
 - **toString()**: Converts an array to a string and returns the result
 - **valueOf()**: Returns the primitive value of an Array object

join() Example

```
var numbers = [1,2,3,4,5];  
alert('numbers.join() = ' + numbers.join());  
alert ('numbers.join(":::") = ' + numbers.join(':::'));
```



Objects as Arrays

- ◆ JavaScript objects can be treated like arrays in two ways
- ◆ Objects are "associative" arrays – you can access object properties using array syntax and the name of the property
- ◆ You can use the [] operator and positive indexes to hold items in an object, much like an array, except that:
 - The length property isn't defined or updated as for arrays
 - The special array methods (e.g. join()) are not available

```
var point = {x:1, y:2};  
alert (point['x']); // Access property using [] and property name  
alert (point['y']);  
  
var obj = {length:2}; // Define our own length property  
obj[0] = 0; // Treat object like array  
obj[1] = 1;  
for (var i=0; i<obj.length; i++) { // Iterate through array  
    alert(obj[i]);  
}
```

Lab 5.2 – Using Objects

- ◆ **Overview:** In this lab you will use an object to pass arguments to a function
 - This is a common technique in many JavaScript libraries
 - You'll modify the functions written in previous labs
- ◆ **Objectives:**
 - Gain experience working with JavaScript Objects
- ◆ **Builds on previous labs:** Lab 5.1
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 20-30 minutes (30-40 minutes with optional callback portion)

Tasks to Perform

- ◆ Open the file *ajax.js* from the previous lab for editing
 - Modify the signature of *makeXHRrequest()* so it takes one argument (an object called *args*)
 - The argument should be an object that has these properties
 - ***method***: The HTTP method (e.g. "GET" or "POST")
 - ***ajaxURL***: The URL for the ajax request
 - ***divID***: The id of the div element to modify
 - ***param***: Parameters for a request (if any)
 - Modify the body of the method so that it now accesses the needed arguments through the single object literal argument (e.g. *args.method*)

Tasks to Perform

- ◆ Open the file *hello_ajax.html* from the previous lab for editing
 - Modify all calls to *makeXHRrequest()* to use an object literal as the argument
 - You don't need to change the values of the arguments, you should just encapsulate them in an object
 - Below the example from the student manual of using an object literal as a method arg is repeated – remember, you'd need to use an object literal with properties called *method*, *ajaxURL*, etc.
- ◆ **Restart** Tomcat, reload in Firefox, and test

```
function isASquare(pt1, pt2) {  
    // ...  
}  
isASquare( {x:4, y:2}, {x:6, y:4} );
```

Tasks to Perform

- ◆ **Optional:** Modify the *makeXHRrequest()* argument object to include a property (named *callback*) holding a callback method, rather than the *divID* you've been passing in
 - The callback method itself should expect one argument, the *responseText* from the XHR call
 - The callback method should use this argument to update a div
 - Below is an example of how you might call the new version of *makeXHRrequest()*
 - Callbacks are one way to integrate libraries with user code

```
<script>
  var arg = {method: "POST", ajaxURL: "/javatunes/ajax",
    param: "artist=a",
    callback: function(txt) {
      document.getElementById("postDiv").innerHTML = txt; } };
  makeXHRrequest( arg );
</script>
```

STOP

Classes in JavaScript

Browser Event Model
JavaScript Objects and Arrays
Classes in JavaScript

Classes in JavaScript

- ◆ We've seen how JavaScript allows you to define objects that hold a set of properties
- ◆ JavaScript also uses **constructors** and **prototypes** to create new types – similarly to the object-oriented idea of a class
 - A **class** is a **blueprint** of an object, but not the actual object
 - For example, every Person might have a name, age, etc
 - The definition of a person, the *Person* class, is not a person
- ◆ We call actual objects **instances** of a given type
 - You, the people taking this course, are instances of the *Person* type (one would hope)
 - Creating instances of a type is called **instantiating** that type
- ◆ Each instance has its own values for the properties that are characteristic of its type
 - You all have your own particular names, ages, etc.

JavaScript Constructors

- ◆ A **constructor** (or constructor function) is a function that is designed to initialize a newly created instance
 - It usually has arguments and sets properties of the new object
- ◆ In the example below, we define a constructor for Person
 - It initializes two properties, name and age, which means that every person will have a name and age property
 - The **this** keyword and the **.** (dot) operator is used to access the instance's properties
 - *this* refers to the object itself, and its use is required to access properties of an object in a constructor
 - The constructor will be used with the **new** operator, covered next

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}
```

The new Operator

- ◆ The **new** operator creates a new instance of an object
 - It must be followed by the name of a (constructor) function
 - The new operator creates a new object with no properties
 - It then invokes the constructor function, and passes the newly created object in as the value of the *this* keyword
 - The constructor function executes, and the result is a new instance (initialized by the code in the constructor function)
 - Below, the example initializes a variable with a new Person instance and displays its name property (using the . operator)

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}  
  
var you = new Person("John", 25);  
alert (you.name);
```



More on Constructors

- ◆ Creating a class in JavaScript is different from languages like Java or C#
 - You actually never define a class explicitly, you define a function that you use to initialize an object
 - Defining that function implicitly defines the class – with the same name as the function
- ◆ There is nothing special about these functions to indicate they are constructor functions
 - Except that you define them to initialize an object
 - For example, the code below is legal – though doesn't make much sense

```
function foo() { alert ("foo is called"); }  
new foo(); // Displays "foo is called"
```

The Object Class

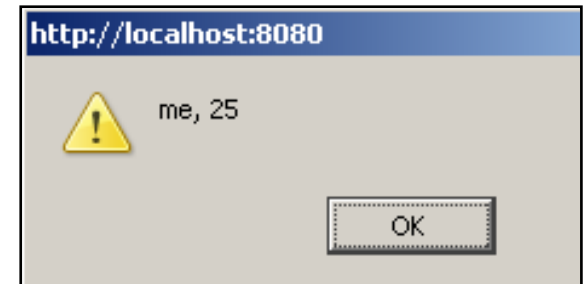
- ◆ Built-in base JavaScript object type
 - All JavaScript objects inherit from *Object* (have the methods that are defined for *Object*)
- ◆ Object has a number of useful methods, including:
 - ***toString()***: Returns a string representation of the object
 - ***valueOf()***: Returns the primitive value of the object, if any. Called, for example, if you try to add a number and an object
 - Other methods dealing with inheritance
- ◆ Object also has the following properties
 - ***constructor***: The constructor function that initializes the object
 - ***prototype***: Allows the addition of properties to the object
- ◆ You can create a new *Object* with no properties using the *Object* constructor

```
var obj = new Object();    // Create empty object
```

The prototype Property

- ◆ The **prototype** property holds properties that are shared by all instances of a class
 - It is created when you define the constructor function for a class
 - It can be accessed through the constructor function
- ◆ Usually used to hold methods of a class
 - Below we define a *toString()* method for Person (see note)
 - When the method is called, "*this*" is set to refer to the invoking object

```
function Person(name, age) {  
    this.name = name;  
    this.age = age;  
}  
Person.prototype.toString = function() {  
    return this.name + ", " + this.age; }  
var me = new Person("me", 25);  
alert(me.toString());
```



Properties of the prototype

- ◆ You can define **properties** of the prototype object which are shared by all instances (useful for declaring constants)
 - When you read a property of an object, if it isn't defined in the object, then the prototype is checked for the property
- ◆ You can't write to a prototype property through an instance
 - When you write a property through an object, JavaScript doesn't use the prototype property, but defines one for the instance
 - However, if you write the property directly through the prototype, then it is changed for all instances using it

```
var me = new Person("me", 25);
var you = new Person("you", 25);
Person.prototype.RETIRE_AGE = 65;
alert(me.RETIRE_AGE);           // Displays 65
alert(you.RETIRE_AGE);          // Displays 65
me.RETIRE_AGE = 62;           // You probably won't do this
alert(me.RETIRE_AGE);           // Displays 62
alert(you.RETIRE_AGE);          // Displays 65
```

A More Complete Class

```
function Person(name, age) {           // Constructor function
    this.name = name;
    this.age = age;
}
Person.prototype.RETIRE_AGE = 65; // Common property
Person.prototype.toString = function() { // Method def
    return this.name + ", " + this.age;
}
Person.prototype.isRetired = function() { // Method def
    return (this.age >= this.RETIRE_AGE);
}

var me = new Person("Me", 25); // Create instance
var you = new Person("You", 65); // Create instance
alert (me.isRetired()); // Displays false
alert (you.isRetired()); // Displays true
```

- ◆ The Person class above shows a simple class definition
 - It illustrates using constructors, prototypes, and defining methods

Modules and Namespaces

- ◆ JavaScript is being used in complex ways today, especially with Ajax
 - More and more libraries are being written and widely used
 - That raises the problem of using libraries from different places that use the same names – e.g. both have a Person class
 - There is no formal way in JavaScript to protect against this
- ◆ A common convention is to use a JavaScript object as a namespace
 - You then define all your global definitions within the namespace
 - This keeps them out of the global namespace, and helps prevent them from being overwritten by other libraries
 - When writing libraries, you should minimize the definitions put in the global namespace
 - Very often there is only one (the top level name in the namespace)

An Example of Using Namespaces

```
// File - js/javatunes/Person.js
var javatunes;           // Declare namespace var
if (!javatunes) javatunes = {}; // Check if namespace defined

javatunes.Person = function (name, age) {           // Constructor
    this.name = name;
    this.age = age;
}
javatunes.Person.prototype.RETIRE_AGE = 65;         // Property
javatunes.Person.prototype.toString = function() { // Method def
    return this.name + ", " + this.age;
}
javatunes.Person.prototype.isRetired = function() { // Method def
    return (this.age >= this.RETIRE_AGE);
}
```

```
<script type="text/javascript" src="js/javatunes/Person.js"></script>
<script>
    var me = new javatunes.Person("Me", 25);           // Create instance
    var you = new javatunes.Person("You", 65);         // Create instance
    alert (me.isRetired());                             // Displays false
</script>
```

- ◆ We show the javatunes namespace – just an object with definitions

Utility Modules

- ◆ Sometimes we want to define a set of utility functions or properties as one module
 - We can do this using an object as the module, and setting the functions or properties to be properties of the module object itself
 - This achieves something like static definitions in Java/C#
- ◆ Below we show a simple Math module

```
// File - js/javatunes/Math.js
var javatunes;
if (!javatunes) javatunes = {};

javatunes.Math = {}
javatunes.Math.abs = function(val) { return val >= 0 ? val : -val; }
javatunes.Math.max = function(x,y) { return x >= y ? x : y; }
```

```
<script type="text/javascript" src="js/javatunes/Math.js"></script>
<script>
    alert (javatunes.Math.abs(-5));
    alert (javatunes.Math.max(2,3));
</script>
```



Lab 5.3 – Creating Modules

- ◆ **Overview:** In this lab you will move your XHR related functions into a module
 - You'll create a javatunes namespace, and XHR module
 - You'll move the functions written in previous labs into this module
- ◆ **Objectives:**
 - Gain experience working with JavaScript modules
- ◆ **Builds on previous labs:** Lab 5.2
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 15-20 minutes

Create javatunes/XHR Module

Tasks to Perform

- ◆ Create a directory *WebContent/js/javatunes*
 - Move the *ajax.js* file into this directory
 - Rename the *ajax.js* file to *XHR.js* (Short for **X**ML**H**ttp**R**equest)
- ◆ Open the file *XHR.js* for editing
 - Add a namespace declaration for javatunes, as in the examples
 - Create a variable *javatunes.XHR* and initialize it with an empty *Object* instance
 - Modify the *getXMLHttpRequest()* function definition to be called the *javatunes.XHR.create()* method
 - Modify the *makeXHRrequest()* method to be called the *javatunes.XHR.request()* method, and to use *javatunes.XHR.create()*
 - The method definitions remain the same – but they are methods of the *javatunes.XHR* object now

Use javatunes/XHR Module

Tasks to Perform

- ◆ Open the file *hello_ajax.html* from the previous lab for editing
 - Modify the `<script>` in the `<head>` to include the *js/javatunes/XHR.js* file instead of *js/Ajax.js*
 - Modify all calls to *makeXHRrequest()* to use *javatunes.XHR.request()*
- ◆ **Restart** Tomcat, reload in Firefox, and test
 - Everything should run as before
 - Check for errors in Firebug if you have problems



Review Questions

- ◆ What is a JavaScript event handler, and what is its use?
- ◆ What is a JavaScript object, and how is it used?
- ◆ What is a JavaScript array, and how is it used?
- ◆ How are classes used in JavaScript?

Lesson Summary

- ◆ JavaScript event handlers consist of JavaScript code associated with HTML elements that are automatically invoked by the browser when the associated event fires
 - e.g., the **onclick** event handler fired upon a button press
 - These are very useful to trigger things like an Ajax request
- ◆ JavaScript objects are structures that have data and methods
 - They collect properties and their associated methods into a single unit – making them much easier to work with
 - Objects can be passed as arguments or returned from functions
- ◆ An array is an ordered and indexed collection of values
 - You access array elements using **[]** and a numerical index
 - They have a **length** property holding the number of elements
- ◆ You can create new JavaScript types with **constructors** and **prototypes**
 - These have much of the same functionality as classes in Java

Session 6: Client Side Frameworks

Framework Overview

Prototype Overview

script.aculo.us Overview

Other Frameworks and Libraries

Lesson Objectives

- ◆ Gain a broad overview of what client-side Ajax frameworks are available
- ◆ Explore a few of the major frameworks in more depth
 - We will not go into detail on all of the frameworks
 - There is a lot of information involved in each framework
- ◆ We'll give you a good idea of what the major frameworks provide, and how to use them

Framework Overview

Framework Overview

Prototype Overview

script.aculo.us Overview

Other Frameworks and Libraries

No Need to Reinvent the Wheel

- ◆ Developers have been working with Ajax for quite a while
 - There are a number of open source frameworks available that can help you quickly get started using Ajax
 - It's not necessary, these days, to start from scratch and write a whole library of Ajax related JavaScript code
 - These are available, and usually better than what you might write yourself
 - There are many kinds of frameworks available for use with Ajax
- ◆ We will cover client-side JavaScript frameworks in this session
 - These are the simplest to use, and will provide the quickest immediate help to you
 - We'll look at other kinds of frameworks in other sessions

Capabilities of Client Side JavaScript Libraries

- ◆ Offer many different capabilities
 - Handling remote communication, Ajax enabled widgets and components, utility functions, sophisticated event handling
- ◆ May also provide a large selection of DHTML capabilities
 - For example, drag and drop
 - We DON'T cover these, but focus on the Ajax capabilities
- ◆ **Server side agnostic:** In general, these libraries can be used with any type of server side technology (Java, .NET, PHP ...)
 - They're used on the client side – how requests are handled on the server is not their concern
- ◆ **Can use multiple libraries:** It's usually possible to mix and match to use the best capabilities of different libraries
 - Typically use namespaces to make combining libraries easier
 - Some are built on top of others – e.g. script.aculo.us & Prototype

General Library Capabilities

- ◆ **Handles remote communication:** Provides an abstraction layer hiding *XMLHttpRequest* handling
 - Provides options for graceful degradation (such as *IFrame*) for when XHR not available
- ◆ **Handles browser incompatibilities:** Transparently deals with all browser specific code
- ◆ **Widgets and components:** Provides palette of widgets
 - Common Ajax enabled widgets, e.g. an autocomplete textfield
- ◆ **Utility functions:** Provides JavaScript utility functions
 - Make your life easier while programming Ajax
- ◆ **Sophisticated event handling:** Provides higher level abstraction of dealing with event handling
- ◆ **Improved navigation:** For use with Ajax
 - Support for working back button, bookmarking

General Library Capabilities

- ◆ **Ajax enabled widgets**: Provides Ajax enabled version of popular widgets, such as an autocomplete textfield
 - Also provides DHTML/Web 2.0 widgets such as drag-and-drop
- ◆ **JavaScript Utilities**: Utilities for many common needs
 - DOM tree navigation, Timers,
- ◆ **Maturity**: Many available that are mature with many revision cycles done
 - Usually well structured and organized, some use OO programming models, and they can help you write better code
- ◆ These libraries can save you a lot of time and effort
 - Many are proven in the market and continually being improved
 - Developer communities can provide help
 - They are generally easy to use (much easier than writing them)

Some Client Side JavaScript Libraries

- ◆ **Prototype**: Low level utility library, including Ajax support
 - Used by many other libraries to handle low level functionality
- ◆ **script.aculo.us**: Add-on to Prototype
 - Ajax enabled controls, animation, drag and drop
- ◆ **Dojo**: Large, mature DHTML toolkit with Ajax support
 - Many capabilities, well organized, with sophisticated structure
- ◆ **Yahoo User Interface Library (YUI)**: JavaScript utilities and controls supporting DHTML and Ajax
- ◆ These are some of the major open source toolkits
 - There are too many to list - you can easily find them on the web
 - There are also many server-side specific toolkits (PHP, Java, .NET) – we'll cover some of the Java ones later
- ◆ We'll look at Prototype, script.aculo.us and Dojo in more detail
 - We won't cover any of the other libraries

Prototype Overview

Framework Overview

Prototype Overview

script.aculo.us Overview

Other Frameworks and Libraries

About Prototype

- ◆ "JavaScript Framework that aims to ease development of dynamic web applications"
 - Great toolkit for JavaScript and Ajax development
 - Provides tools for writing better JavaScript and Ajax code and making common tasks easier
- ◆ Provides tools in the following key areas
 - **JavaScript and DOM extensions** making it easier to create OO like JavaScript classes and work with web pages dynamically
 - **Ajax tools** to make it easier to make Ajax calls in an easy and cross-browser compatible way
 - **JSON** encoding and parsing support
- ◆ Used as the basis for many other toolkits
 - Such as script.aculo.us
- ◆ See it at: <http://www.prototypejs.org>

Utility Methods and DOM Extensions

- ◆ Let's face it, Ajax (and all DHTML) is all about JavaScript
 - And about manipulating the DOM web page elements and your own objects using JavaScript
 - This can be a pain in the neck
- ◆ Prototype includes a **library of utility methods** making many of the common actions on these elements easier to program
 - These are encapsulated into the Prototype *Element* object
 - Includes methods to work with CSS, to work with the DOM tree of a Web page, work with dimensions of an element, hide and show elements, display debugging information, and more ...
- ◆ It also provides a very neat mechanism to **extend the DOM** elements in the web page
 - So that these utility methods can be called directly on them
 - This makes your code much cleaner and shorter

Utility Methods of Element Class

- ◆ There are many – we show a few here to get started
 - **hide(element)**: Hides and returns element
 - **show(element)**: Displays and returns element
 - **getHeight(element)**: Returns computed height of element
 - **getWidth(element)**: Returns computed width of element
 - **inspect(element)**: Returns debug oriented string
- ◆ It's very easy to use these methods

```
<html> <head>
  <!-- Include the prototype library -->
  <script type="text/javascript" src="js/prototype/prototype.js">
</script> </head>
</head>
<body>
  <div id="targetDiv"><p>Paragraph in a div element</p></div>
  <script>
    Element.hide('targetDiv'); // Hide the above div.
  </script>
</body> </html>
```

The `$()` Utility Function

- ◆ Prototype provides a number of useful utility functions
- ◆ `$` (the dollar function) is one of the most basic and useful
 - It wraps `document.getElementById()`, to allow easy retrieval of elements
 - If provided with a string, `$` returns the element in the document with matching id, otherwise it returns the passed element

```
var targetEl = $('targetDiv');
```
- ◆ All elements returned by `$` are **extended** with Prototype DOM extensions – adding additional methods to the elements
 - These are basically the same methods as in the `Element` class
 - It makes it even easier and more natural to use these methods
 - Below, we retrieve the element with id `targetDiv`, then call the `hide()` method, which is one of the Prototype DOM extensions

```
$('targetDiv').hide();
```

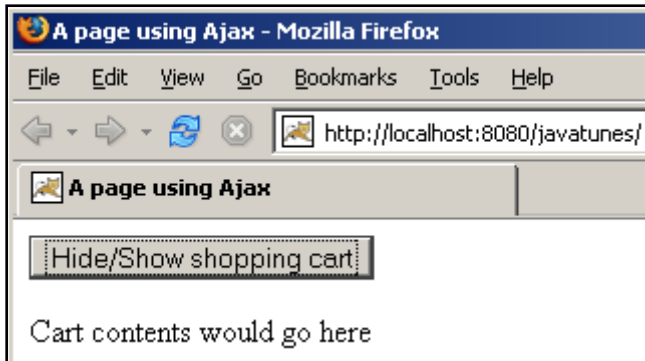
Using \$ and the DOM Extensions

- ◆ Below, we hide/show a shopping cart in response to a button click – notice how simple the code is
 - This isn't Ajax, but Prototype makes JavaScript programming much easier – and a lot of Ajax is about JavaScript

```
<html> <head>
  <!-- Include the prototype library -->
  <script type="text/javascript" src="js/prototype/prototype.js">
</script> </head>
</head>
<body>

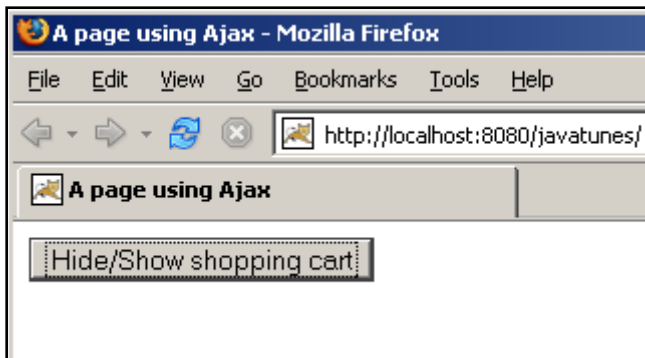
  <form>
    <input type="button" value="Hide/Show shopping cart"
      onclick="$('shoppingCart').visible() ?
        $('shoppingCart').hide() : $('shoppingCart').show();" />
  </form>
  <div id="shoppingCart">Cart contents would go here</div>
</body> </html>
```

Using \$ and the DOM Extensions

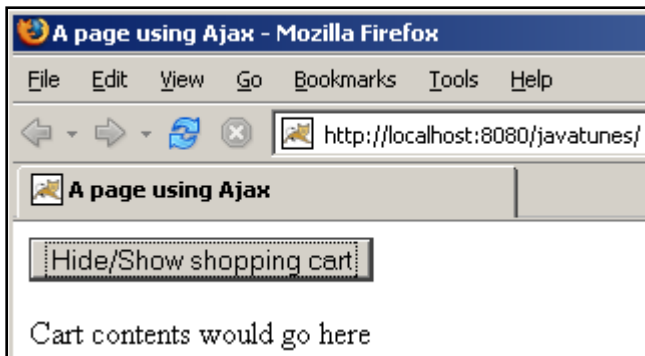


- ◆ Here are some pictures of how this works

- Of course, the shopping cart could be as complex as you want



- ◆ After the first click, the div is hidden



- ◆ After the second click, the div is visible again

Prototype Ajax Support

- ◆ Prototype has a number of objects to deal with Ajax communication, which are listed below
 - They make Ajax programming much simpler
- ◆ ***Ajax.PeriodicalUpdater***: Periodically performs an AJAX request and updates a container's contents based on the response text
- ◆ ***Ajax.Request***: Initiates and processes an AJAX request.
- ◆ ***Ajax.Responders***: A repository of global listeners notified about every step of Prototype-based AJAX requests
- ◆ ***Ajax.Updater***: Performs an AJAX request and updates a container's contents based on the response text
- ◆ We'll look at the *Ajax.Request* object in some more detail
 - We won't look at all of these – once you get started with Prototype, it's fairly easy to pick up the rest

Ajax.Request

- ◆ **Ajax.Request** is a general-purpose AJAX requester
 - Handles the request lifecycle, boilerplate code and things like browser compatibility, and lets you plug in callback functions
 - Takes options which are common to most of the Ajax objects
- ◆ You create a requester through the **new** operator
 - As soon as the object is created, it initiates the request, then goes on processing it throughout its lifecycle

new Ajax.Request(url[, options])
- ◆ The options object contains all the options for the call
 - These can include callback functions, as well as normal options
 - Examples of common options are:
 - **method**: 'GET', 'POST', etc.
 - **parameters**: Parameters for the request
 - **onSuccess**: Callback function for a successful request

Direct Ajax vs. Ajax.Request Example

```
<!-- Much detail omitted throughout this example-->
<script> // Call Ajax using our own code
    function callAjax() {
        var req = getXMLHttpRequest(); // Defined in js/ajax.js
        req.open("GET", "/javatunes/data.txt", false);
        req.send(null);
        alert(req.responseText);
    }
</script>
<input type="button" name="ajaxButton" value="Click for Ajax"
        onclick="callAjax()"/>
```

```
<!-- Same example using Ajax.Request -->
<script>
    function callAjax() {
        new Ajax.Request("/javatunes/data.txt",
            { method: "GET", onSuccess: function(transport) {
                alert(transport.responseText); }
            } ) ;
    }
</script>
<input type="button" name="ajaxButton" value="Click for Ajax"
        onclick="callAjax()"/>
```

Ajax.Request - Additional Options

- ◆ There are a number of other options you can use, including:
 - **asynchronous**: Default *true*
 - **content-type**: Default *'application/x-www-form-urlencoded'*
 - **encoding**: Default *'UTF-8'*
 - **method**: Default *'post'*
 - **parameters**: Default *''*
 - **postBody**: Default *none*
 - **requestHeaders**:
 - **evalJS**: auto-evaluate JavaScript (default *true*)
 - **evalJSON**: auto-evaluate JSON (default *true*)
 - **sanitizeJSON**: prevent eval of malicious JSON (default *true*)
- ◆ There are a number of callbacks available (see notes)
 - *onCreate*, *onComplete*, *onException*, *onFailure*, *onInteractive*, *onLoaded*, *onLoading*, *onSuccess*, ...
 - Callbacks are invoked with two parameters, the *XMLHttpRequest* object, and the result of evaluating the X-JSON header, if any

Ajax.Updater

- ◆ *Ajax.Updater* is a specialization of *Ajax.Request*
 - It updates a container's contents based on the response text
 - It takes the container to update as an additional argument
 - It takes all the options that *Ajax.Request* does
- new Ajax.Updater(container, url[, options])*
- Note how simple the example below is
 - By default the complete contents of the container are replaced
 - You can insert the response text into the container at a specified insertion point using the insertion option and an insertion position

```
<div id="testDiv">This will be updated with Ajax data</div>
<script>
    new Ajax.Updater("testDiv", "/javatunes/data.txt",
        { method: "GET" } );
</script>
```

Other Prototype Capabilities

- ◆ We will go over some of this at a high level only here
- ◆ Utility Methods
 - ***\$A***: Accepts an array-like collection and returns its equivalent as an actual Array object.
 - ***\$F***: Returns the value of a form control
 - ***\$w***: Splits a string into an Array, treating all whitespace as delimiters
 - ***Try.these***: Accepts an arbitrary number of functions and returns the result of the first one that doesn't throw an error
- ◆ **Form object**: Module for all things form related
 - enable/disable form elements
 - quickly traverse form and retrieve specific form elements
 - Manipulate keyboard focus of the form

Other Prototype Capabilities

- ◆ **Class Object**: Module for class based OOP
 - Allows you to more easily create classes in JavaScript
 - Modeled after a Ruby class
- ◆ **Event Object**: Module for handling events
 - Deals with many subtle browser specific issues
 - Provides a standardized list of key codes
 - Helps deal with IE memory leaks
 - Makes life a lot easier when dealing with events
- ◆ **String**: Prototype enhances the String object with a number of very useful methods
 - Everything from stripping off trailing whitespace to parsing a string
 - Look at the documentation !

Much More Capability

- ◆ More than we can go into in this course
- ◆ Let's take a brief look at the documentation
 - It is available on your setup disk in pdf form
 - It is also available on Prototype's web site:

<http://www.prototypejs.org/api>

- ◆ Take a few minutes to look these documents over

Lab 6.1 – Using Prototype

Lab 6.1 – Using Prototype

Lab

- ◆ **Overview:** In this lab you will replace your code using the XHR object directly with code using Prototype
 - You'll use your existing *XHR.request* method, but the internals will become calls using Prototype functions
- ◆ **Objectives:**
 - Gain experience using Prototype
- ◆ **Builds on previous labs:** Lab 5.3
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 30-40 minutes

Modify the `XHR.request()` method

Tasks to Perform

- ◆ Open the file *XHR.js* from the previous lab for editing
- ◆ Replace the implementation of the *javatunes.XHR.request* method with code that uses *Ajax.Updater* as follows:
- ◆ Create an *Ajax.Updater* object (using *new*) and pass in the following arguments
 - The container (passed in as *args.divID*)
 - The URL (passed in as *args.ajaxURL*)
 - An options object containing the following options
 - method, passed in as *args.method*
 - parameters, passed in as *args.param*
 - Note that the *parameters* value can't be null, and *args.param* may be null if it isn't passed in, so use the ternary operator for this
 - For the value of the parameters option, use an empty string if there are no parameters: *(args.param ? args.param : "")*

Tasks to Perform

- ◆ **Copy** the directory *LabSetup\Lab06.1\prototype* to your *Lab03.1\WebContent\js* directory
 - **Refresh** the project if needed (see notes)
 - This directory contains the *prototype.js* file, which holds the complete Prototype implementation
- ◆ Open the file *hello_ajax.html* for editing
 - Add in a `<script>` with *`src="js/prototype/prototype.js"`*
 - This will include the prototype library
- ◆ **Restart** Tomcat, reload in Firefox, and test

Continued ⇒

Tasks to Perform

- ◆ Try using the `$()` method and `hide/show`
 - Add in a form with an input button and an `onClick()` method that toggles the visibility of one of your `<div>` elements
 - It should use `$()` to access the `<div>` element
 - It should use the `visible()`, `hide()`, and `show()` methods to toggle the visibility of the element
 - Look at the example in the student manual slides
 - **Restart** Tomcat, reload in Firefox, and test
- ◆ **Optional:** Use `Ajax.Request` instead of `Ajax.Updater`
 - Look at the example in the student manual slides
 - You'll need to specify a callback function for `onSuccess`
 - The callback function will update your div element



script.aculo.us Overview

Framework Overview

Prototype Overview

script.aculo.us Overview

Other Frameworks and Libraries

- ◆ JavaScript library built on top of Prototype
 - Provides dynamic visual effects (Opacity, Scale, MoveBy, Highlight, and Parallel)
 - Provides various controls (Drag and Drop, Autocompletion, In Place Editing)
 - Included with Ruby on Rails
- ◆ We'll look at its Autocompleter which is Ajax powered
 - We're not going to look deeply at the rest of the library
 - There are a couple of other Ajax powered controls, such as the editors, but a lot of it is not Ajax related
- ◆ As usual, we will focus on the Ajax portion of the toolkit

Using script.aculo.us

- ◆ You need to include the scriptaculous JavaScript library
 - These are contained in a number of JavaScript files - *scriptaculous.js*, *builder.js*, *effects.js*, *dragdrop.js*, *slider.js* and *controls.js*
 - Each file contains an area of functionality in the library
 - You also need to include Prototype, which scriptaculous uses
 - You include it, as usual, with `<script src=...>`
 - You can include the complete library, or limit the scripts that are loaded by specifying them in a comma separated list

```
<script language="javascript" src="js/prototype/prototype.js"></script>  
<!-- Include entire scriptaculous library -->  
<script language="javascript" src="js/scriptaculous/scriptaculous.js"> </script>
```

```
<script language="javascript" src="js/prototype/prototype.js"></script>  
<!-- Include effects and controls portions -->  
<script language="javascript"  
  src="js/scriptaculous/scriptaculous.js?load=effects,controls"> </script>
```


The Scriptaculous Autocompleter

- ◆ This autocompleter allows for server powered auto-completing text fields – powered by Ajax
 - It is a fairly simple component which requires that the server send back HTML in the form of a
 - It does not support more advanced functionality such as JSON or XML data
 - It's very easy to use, and in many cases all you need
- ◆ The autocompleter requires at a minimum three things
 - A text input field that will be "auto-completed"
 - A <div> where the auto-complete display will appear
 - JavaScript code creating the auto-complete object, of the form shown below

```
new Ajax.Autocompleter( id_of_text_field,  
                        id_of_div_to_populate, url, options);
```

Autocompleter Example

- ◆ The example below creates an autocomplete field for inputting the name of a fruit
 - The input field has an id = **auto_fruit**
 - The <div> for autocomplete display has an id = **fruit_choices**
 - The *Ajax.Autocompleter* constructor takes the two ids, the url for the server side, and an options object (empty in this example)
 - The autocompleter takes care of all the display and hiding of the autocomplete choices

```
<form>
  <input type="text" id="auto_fruit" name="fruit"/>
</form>
<div id="fruit_choices"></div>
<script language="javascript">
  new Ajax.Autocompleter("auto_fruit", "fruit_choices",
                        "/javatunes/fruit", {});
</script>
```

Autocompleter Example

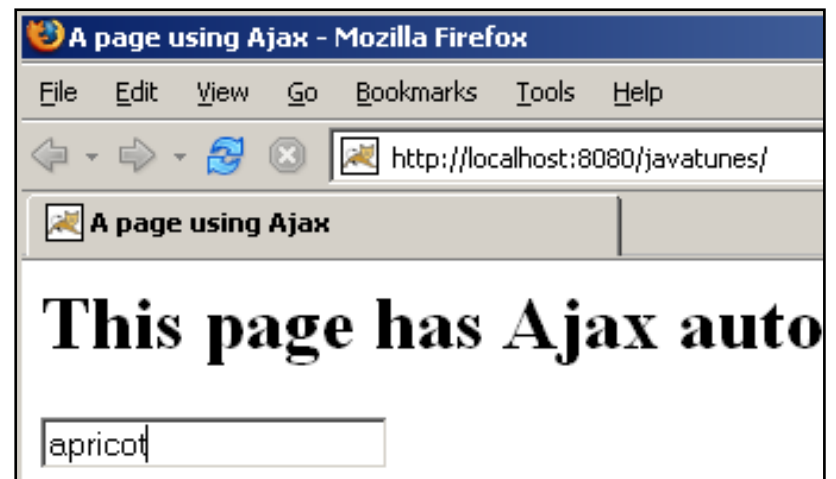
- ◆ Each time a character is entered, the autocompleter sends a request to the url with a parameter *fruit="current_input"*
 - The HTML that is returned becomes the content of the `<div>` for the autocomplete display
 - Below is a very simple servlet that generates HTML markup for the autocomplete based on the value of the fruit parameter

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    String fruit = request.getParameter("fruit");
    out.println("<ul>");
    if (fruit.equals("a")) {
        out.println("<li>apple</li><li>apricot</li><li>avocado</li>");
    } else {
        out.println("<li>some other fruit</li><li>another fruit</li>");
    }
    out.println("</ul>");
    out.close();
}
```

Autocompleter Example

- ◆ In the image at right, you can see the autocomplete choices for the letter a
 - If you click on one of the choices, it will be entered into the textfield
- ◆ In the second image, we see the result after the "apricot" item has been selected
 - We'll see a little later how CSS can make the display better



Lab 6.2 – Using the `script.aculo.us` `Ajax.Autocompleter`

- ◆ **Overview:** You'll use the script.aculo.us autocompleter to display possible artists for a search in JavaTunes
 - The search will be entered in a textbox with autocomplete
 - The completion terms will be generated with the same Servlet/JSP generation of a list that we've made in previous labs
 - You'll need to add in the script.aculo.us libraries and the code for the autocompleter
- ◆ **Objectives:**
 - Gain experience using script.aculo.us
 - See an autocompleter working
- ◆ **Builds on previous labs:** Lab 6.1
 - Continue working in your **Lab03.1** directory
- ◆ **Approximate Time:** 15-20 minutes

Tasks to Perform

- ◆ **Copy** the directory *LabSetup\Lab06.2\scriptaculous* to your *Lab03.1\WebContent\js* directory
- ◆ Open the file *hello_ajax.html* for editing, and add in a `<script>` with the following src attribute *:
`src="js/scriptaculous/scriptaculous.js?load=effects,controls"`
- ◆ Add in a **`<form>`**, with the following attributes:
`method="post" action="/javatunes/search"`
 - Add an `<input>` element within the form having attribute/values of `id="auto_artist"`, and `name="artist"`
 - Add a submit button to the form also, so we can actually search
`<input type="submit"/>`
 - We've included functionality, so that when you submit the search the results will be displayed

Tasks to Perform

- ◆ Add in a `<div>` below the form with an *id* of *artist_choices*
 - Add in a `<script>` that creates an *Ajax.Autocompleter* with constructor arguments of: the `<input>` id, the `<div>` id, the URL */javatunes/ajax*, and an empty options object (see the book example)
- ◆ **Restart** Tomcat, reload in Firefox, and test
 - *AjaxServlet* & *ajaxData.jsp* from previous labs will work without any changes to generate the `` needed for the completions
 - Browse to *http://localhost:8080/javatunes* to view your page
- ◆ Type the letter "a" into the textbox – you should see a number of completions displayed below the textbox
 - Choose one, and submit the form and see the results
 - We've included simple functionality that displays the results
- ◆ **Optional:** Try using some of the options that are passed in to the *Ajax.autocompleter* (see notes in the student manual)

STOP

Other Frameworks and Libraries

Framework Overview

Prototype Overview

script.aculo.us Overview

Other Frameworks and Libraries

Some Well Known Frameworks

- ◆ We will review these frameworks briefly here
- ◆ **Dojo**: Popular DHTML toolkit, with support for Ajax
 - Large set of functionality and widgets
- ◆ **YUI** (Yahoo User Interface Toolkit): Popular DHTML toolkit
 - Large set of functionality and widgets, support for Ajax
- ◆ **Google Ajax Search API**: Proprietary Google framework for accessing its search service using Ajax
 - Allows you to put Google search in your web pages using JavaScript
- ◆ **Google Maps API**: Proprietary Google framework for accessing its mapping service
 - Uses IFrames for access, rather than Ajax as its core technology
 - Still interesting and we'll take a quick look at it

- Dojo Functionality -

- ◆ Sophisticated framework with a lot of capability
 - We'll touch on the high points of the features
- ◆ Powerful abstract wrapper (*dojo.xhr**) around XHR
- ◆ Large selection of DHTML widgets
 - Support for creating custom widgets
- ◆ Handles browser JavaScript incompatibilities
- ◆ Advanced event handling system
 - "Aspect oriented" type functionality which allows you to attach code to traditional DOM events, and to arbitrary events such as calls to a particular function
- ◆ Abstracted interface to JavaScript classes which supports subclassing and extension of existing classes
- ◆ Dojo libraries are organized in modules and classes, which you import, e.g. *dojo.require("dijit.io.script");*

dojo.xhr* Functions

- ◆ Powerful abstract wrapper around underlying transport
 - Contains generic API for doing network I/O
 - Hides browser differences in XMLHttpRequest
 - Supports different types of response data (text, HTML, JSON, XML, JavaScript)
 - Has back and forward button support -
 - Transparent form submission over XHR (you can pass in a form node, and the library extracts the data from the form elements)
 - Advanced error handling with error callback functions
 - Using the library is very easy
- ◆ Most functionality is exposed in the function *dojo.xhr()*
dojo.xhr(method: String, args: dojo.__XhrArgs, hasBody)
 - There are also *xhrGet* and *xhrPost* convenience functions for making GET and POST requests

Using `dojo.xhrGet()`

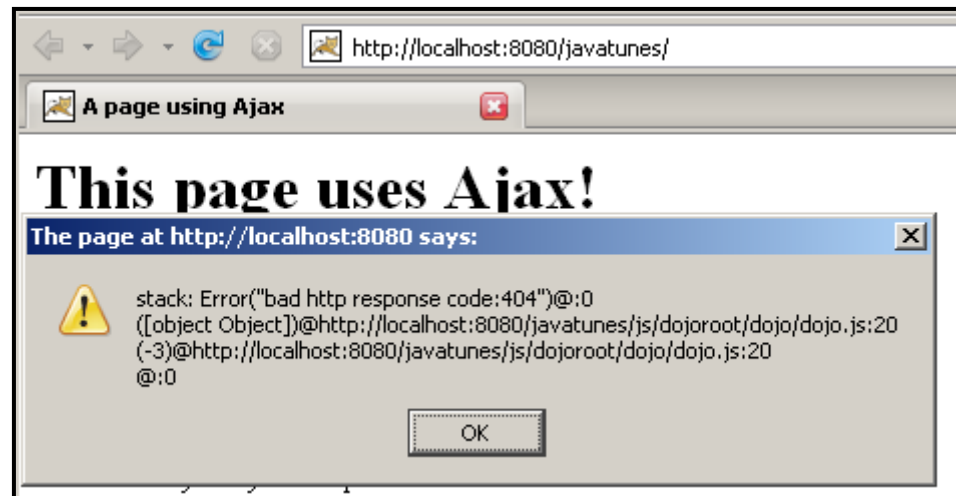
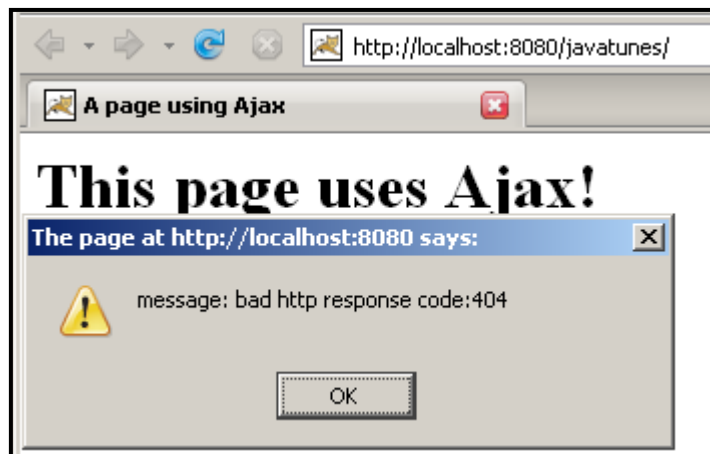
- ◆ ***function** (***/*dojo.__XhrArgs*/ args***)* is a convenience function for making HTTP GET requests using XHR
 - Arguments are passed as properties of the args object, and the function generally requires at least two arguments
 - ***url***: The destination of the request you are making
 - ***load***: Callback function to handle the returned data
 - Other arguments are available, such as:
 - ***timeout***: Milliseconds to wait for the response.
 - ***content***: props with string values - serialized as name1=value1 and passed with the request

```
<!-- Load the general Dojo library and the io library -->  
<script type="text/javascript" src="js/dojoroot/dojo.js"></script>  
<script type="text/javascript">  
    // Call dojo.xhrGet()  
    dojo.xhrGet( { url: "/javatunes/data.txt",  
                  load: function(response, ioArgs) { alert(response); } });  
</script>
```

dojo.xhrGet Error Handling

- ◆ The example below shows a simple error handler for xhrGet
 - We trigger it by trying to access a non-existent URL, and show two of the resulting error response properties at bottom

```
<script type="text/javascript">
  dojo.xhrGet( { url: "/javatunes/doesNotExist.txt",
    load: function(response, ioArgs) { alert(response); },
    error: function(response, ioArgs) {
      for (e1 in response) { alert(e1 + ": " + response[e1]); } }
  });
</script>
```



Some Issues with Dojo

- ◆ It has been perceived to be big and slow in the past
 - The later releases of Dojo (1.0+) address these issues
- ◆ Documentation is relatively immature
 - The organization is somewhat complex, and it can be hard to find the information you need - Work is also continuing on this, with a brand new manual being written for the 1.x releases
 - Version 1.8 attempted to address this comprehensively
- ◆ It's fairly large and bulky, and can be hard to learn
- ◆ Components are not that consistent across the toolkit
 - Making them harder to learn and use
- ◆ However, it does have a LOT of functionality
 - You'll have to decide if its capabilities outweigh its shortcomings

- Yahoo User Interface Toolkit (YUI) -

- ◆ YUI is an open source JavaScript/CSS library for creating Rich Web applications
 - Support for DOM scripting, DHTML, CSS, and Ajax
 - It has a large number of widgets, with support for all major browsers
- ◆ YUI supplies functionality in a number of areas
 - A **core JavaScript engine** with DOM and event functionality
 - **JavaScript utilities**, that for example support Drag and Drop
 - **Widgets** - for example a Calendar / Date Picker widget
 - **CSS** components
 - **Developer tools** (e.g. the YUI Configurator)
- ◆ YUI has extensive documentation, including a User's Guide, Cheat Sheet, full API documentation, and examples for each component

The YUI Dom Class

- ◆ Provides helper methods for DOM elements, similar to the way Prototype does, including the ones listed below *
- ***get (el)***: Returns element(s) in the page
- ***getChildren(node)***: Returns the child elements of node
- ***getFirstChild(node)***: Returns first child of node
- Many other traversal methods ...
- ***getX, getY, getXY (el)***: Position of el in page coordinates
- ***setX, setY, setXY (el, ...)***: Set position of el
- ***getDocumentHeight, getDocumentWidth***: Current size of actual document
- ***getViewportHeight, getViewportWidth***: Current size of viewable area of page (excludes scrollbars)

A Simple Application with YUI

- ◆ Below, we include the needed YUI libraries *, define a button and define a div (with id=shoppingCart) that contains an image
 - We'll show you how to manipulate these with YUI next

```
<html>
<head>
<script type="text/javascript"
src="http://yui.yahooapis.com/combo?2.6.0/build/yahoo-dom-
event/yahoo-dom-event.js">
</script>

<script type="text/javascript">  /* Not Shown */ </script>
</head>

<body>
<div id="shoppingCart"></div>
<hr/>
<button id="demo-move">Move</button>
</body>
</html>
```

A Simple Application with YUI

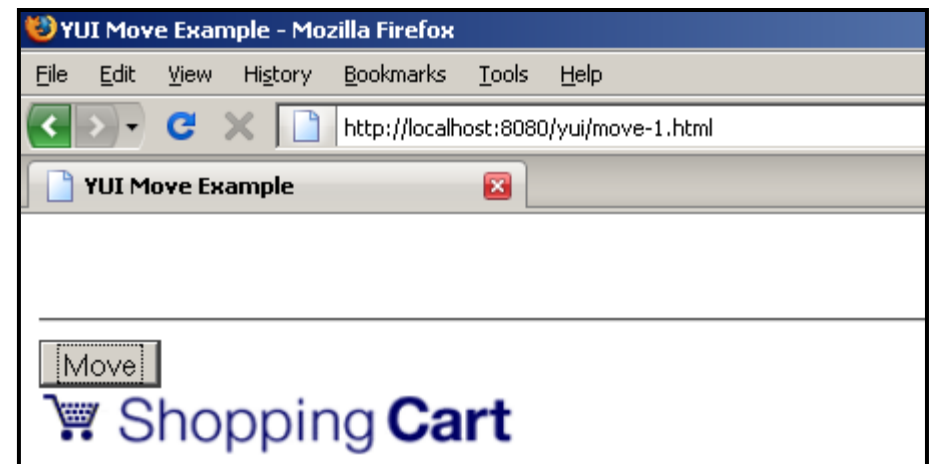
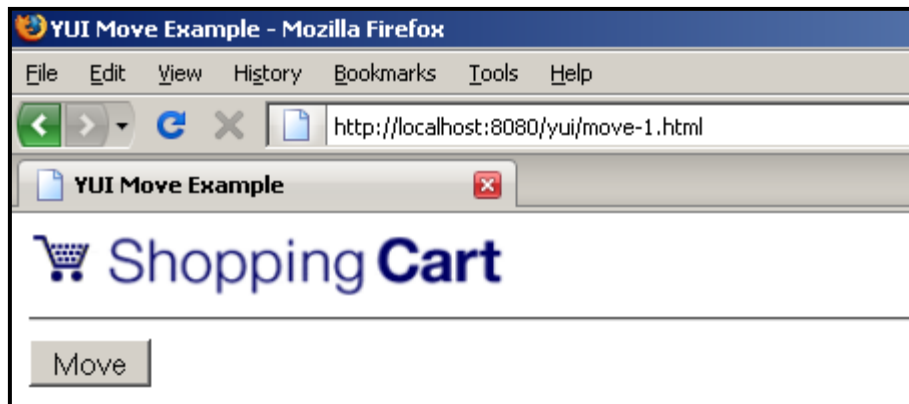
- ◆ In the script below, we define a function *move()* that:
 - Gets the region that the button (with id=demo-move) occupies *
 - Moves the div (with id=shoppingCart) below the button
- ◆ It also uses the YUI Event class * to associate the move function with click events on the button
- ◆ When the button is clicked, this code will move the div to below the button

```
<script type="text/javascript">
  var move = function() {
    var reg =
      YAHOO.util.Region.getRegion(YAHOO.util.Dom.get('demo-move'));
    YAHOO.util.Dom.setXY('shoppingCart', [reg.left,reg.bottom] );
  };

  YAHOO.util.Event.on('demo-move', 'click', move);
</script>
```

A Simple Application with YUI

- ◆ Below left, we show the page before the button is clicked
 - Below right, we show the page after the button is clicked
 - You can see how easy YUI makes this sort of thing



YUI and Ajax

- ◆ YUI provides a simple interface to doing Ajax with XHR
 - Similar to the Prototype functionality
- ◆ The ***YAHOO.util.Connect*** class provides the *asyncRequest* method for asynchronous XHR requests
 - ***asyncRequest (method, uri, callback, postData)***
 - ***method <string>***: HTTP transaction method (e.g. GET or POST)
 - ***uri <string>***: Fully qualified path of resource
 - ***callback <callback>***: User-defined callback function or object
 - ***postData <string>***: POST body (for POST requests)
 - The method returns the connection object
 - The callback function is invoked when the request completes
- ◆ We show an example of this on the following page

YUI Ajax Example

- ◆ Below, we show code segments that define **loadUrl** to hold the URL of our Ajax source, and **callback** to hold an array of functions that defines success and failure functions to be used in the Ajax call

```
<script type="text/javascript"
  src="http://yui.yahooapis.com/2.6.0/build/yahoo-dom-event/yahoo-dom-event.js">
</script>
<script type="text/javascript"
  src="http://yui.yahooapis.com/2.6.0/build/connection/connection-min.js">
</script>

<script type="text/javascript">
  var loadUrl = "/yui/ajax";           // A servlet returning text
  var callback = {
    success: function(o) {            // Success
      YAHOO.util.Dom.get('testDiv').innerHTML = o.responseText;
    },

    failure: function(o) {
      alert("AJAX doesn't work");      // FAILURE
    }
  }
</script>
```

YUI Ajax Example

- ◆ Below, we show the use of ***Connect.asyncRequest*** to make the XHR call to the *loadUrl*, using the *callback* array to handle the results of the call

```
<body>
  <h1>This page uses Ajax!</h1>

  <div id="testDiv">
    The data fetched with Ajax will go here.
  </div>

  <form>
    <input type="button" value="Click for Ajax"
      onclick="YAHOO.util.Connect.asyncRequest('GET', loadUrl,
        callback, null)"/>
  </form>
</body>
```

YUI Ajax Example

- ◆ Below, we see that the text in the div has been replaced after the button is clicked
 - It's very simple to do XHR with YUI
- ◆ The next few slides show some non-Ajax YUI controls



TabView and TreeView

TabView Control: Build from Markup

This demonstrates how to build a TabView from markup.

[View example in new window.](#)

Tab One Label

Tab Two Label

Tab Three Label

Tab Three Content

TreeView Control: Default TreeView

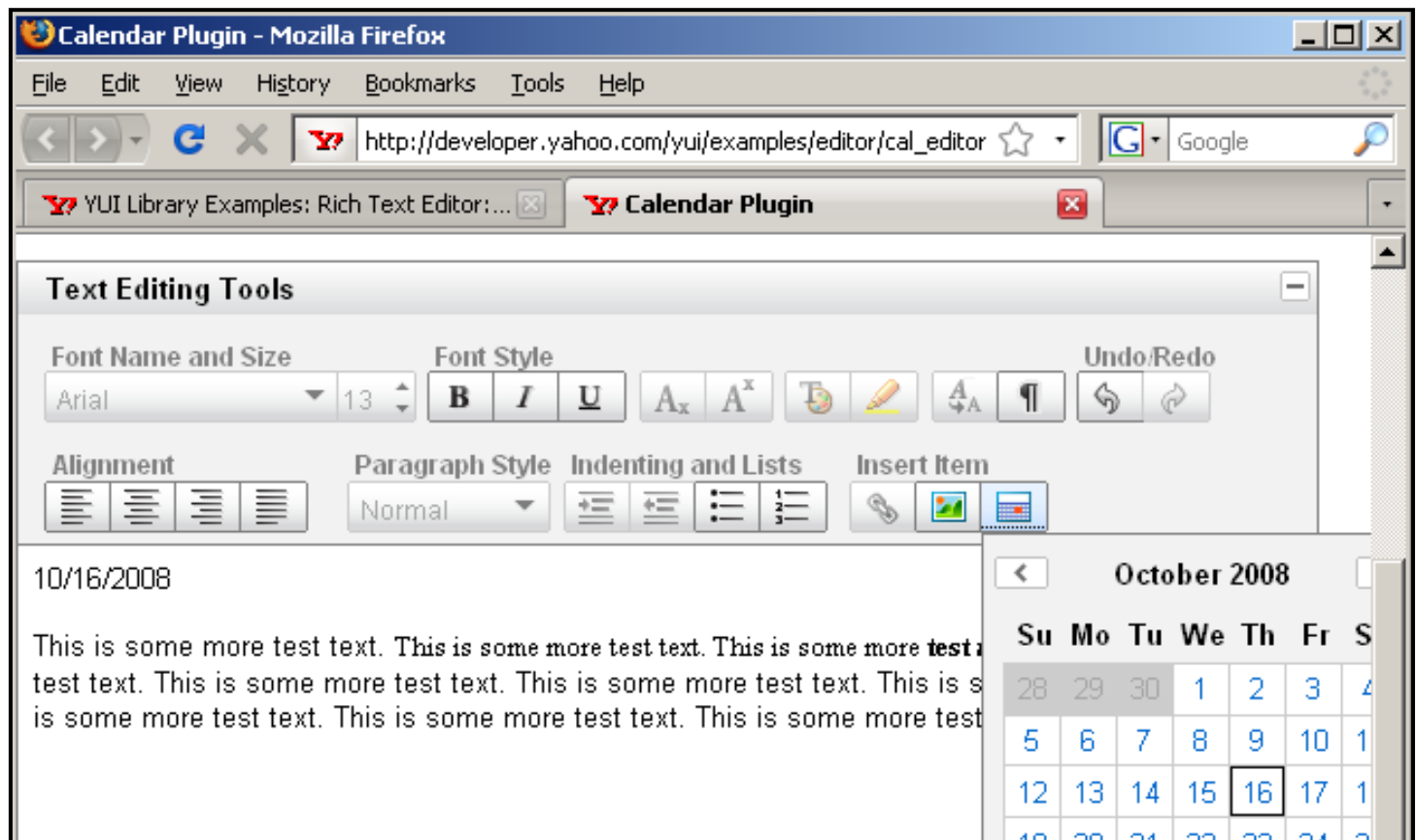
In this simple example you see the default presentation for the [TreeView Control](#). Click on labels or on the expand/collapse icons for each node to interact with the TreeView Control.

[View example in new window.](#)

- + label-0
- + label-1
- label-2
 - label-2-0
 - label-2-1

Rich Text Editor and Calendar Control

- ◆ The example below shows the YUI rich text editor
 - It allows you to place a WYSIWYG editor to edit and style text
 - In the example below, we show it in conjunction with the YUI calendar control used to insert a date into the text



Other YUI Capabilities

- ◆ YUI is a very, very large library
- ◆ For example, it has a DataSource object, which lets you provide data to a widget from a variety of sources
 - e.g. using XHR, or using a database
- ◆ It has a large number of widgets
 - Calendars, color pickers, image croppers, paginators, etc.
 - More than we want to go into in this course
- ◆ Many of the widgets are RIA widgets which do not necessarily relate to Ajax
 - Though many of them that require data, can get that data using Ajax (e.g. with an XHRDataSource)
- ◆ We've shown a small sampling of the widgets available - look at YUI's extensive documentation for more info



Lab 6.3 – Using YUI

- ◆ **Overview:** In this lab you will view some simple Web applications that use YUI
 - You will not have to do any coding for this lab
- ◆ **Objectives:**
 - Become more familiar with YUI
- ◆ **Builds on previous labs:** none
- ◆ **Approximate Time:** 10-15 minutes
- ◆ **Note** - Look at the html source after running the examples to see how simple it is to use YUI

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab06.3

- This is a new lab directory

Tasks to Perform

- ◆ Create a new **Dynamic Web Project** called **Lab06.3** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server and **2.5 module version** are selected
 - **Set the context root** to **yui**
- ◆ Right click on **move-1.html** (under WebContext) and select **Run As | Run on Server** - You may need to restart the server
 - move-1.html file contains the single button example of moving an image that we showed in the slides - try it in your browser
- ◆ You can also browse to **move-2.html** (two button example of moving an image) and **hello_ajax.html** (Ajax example from slides)

A red octagonal stop sign with the word "STOP" in white capital letters in the center.

STOP

- The Google Ajax Search API -

- ◆ Allows you to put Google search into your web pages using JavaScript
 - Can embed a simple search box in your own web page
 - Submission of search returns results from Google search engine
- ◆ To use it in a Web page you load the Google AJAX Search API javascript library

`http://www.google.com/uds/api?file=uds.js&v=1.0&key=XXX)`

 - You need to get a key from Google, which is valid only for a particular directory on a particular website
- ◆ The library provides a number of JavaScript objects
 - These are used to make searches
 - Under the hood XHR and other I/O methods are used
 - The search results are displayed in an element you supply
 - An example is shown on the following pages

Google Search API Code

- ◆ Only detail relevant to Google Search API is shown

```
<head>
  <link href="http://www.google.com/uds/css/gsearch.css" type="text/css"
        rel="stylesheet"/>
  <script src="http://www.google.com/uds/api?file=uds.js&v=1.0&key=XXX"
        type="text/javascript"></script>
  <script language="Javascript" type="text/javascript">
    //<![CDATA[
      function OnLoad() {
        // Create a search control
        var searchControl = new GSearchControl();
        // Add in a full set of searchers
        searchControl.addSearcher(new GwebSearch());
        // Tell the searcher to draw itself and tell it where to attach
        searchControl.draw(document.getElementById("searchcontrol"));
        // Execute an initial search
        searchControl.execute("Google");
      }
      GSearch.setOnLoadCallback(OnLoad);
    //]]>
  </script>
</head>
<body>
  <div id="searchcontrol"/>
</body>
```


Overview of Code Example

- ◆ The Web page includes the Google library, as well as a display stylesheet, with link and script elements
 - A JavaScript script then does the following:
- ◆ Creates a Google object – GSearchControl

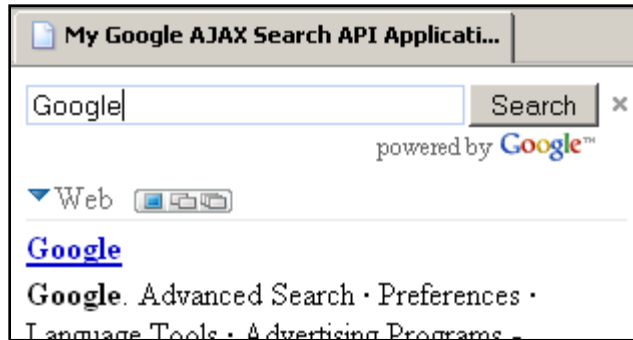
```
var searchControl = new GSearchControl();
```
- ◆ The search control is initialized to do a Web search

```
searchControl.addSearcher(new GwebSearch());
```
- ◆ The display element for results is set

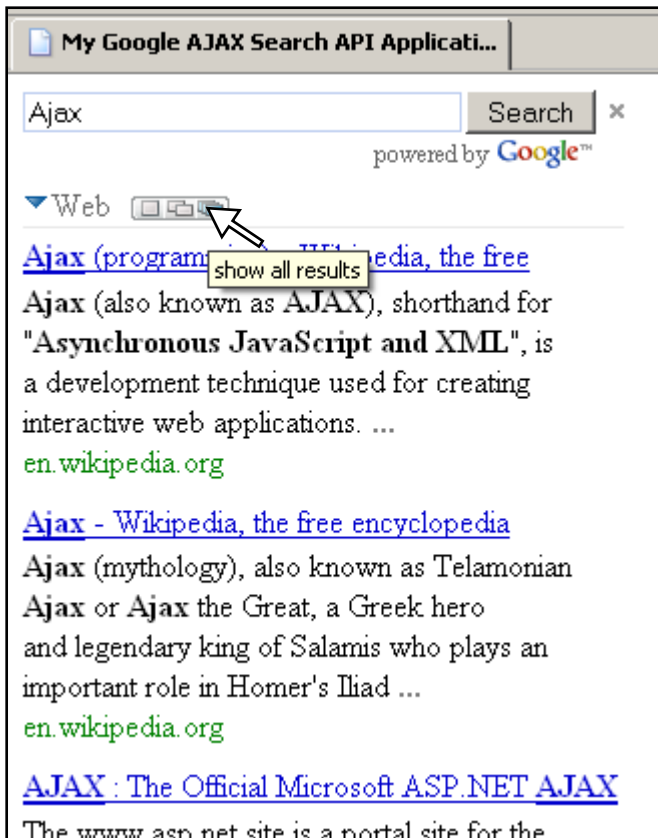
```
searchControl.draw(document.getElementById("searchcontrol"));
```
- ◆ An initial search is set up for the word "Google"

```
searchControl.execute("Google");
```
- ◆ That's all you need – see the screen shots in the next slides
 - You can see the resulting web page, and search results

Google Search Page Display



- ◆ Initial search for "Google"
 - By default only shows the first search item



- ◆ Search for "Ajax"
 - All results shown by clicking on the toolbar icon for "show all results"

Much More Capability

- ◆ You can add many more searchers in addition to a Google Web searcher
 - A **local** searcher, for which you can set the location
 - A **video** searcher to search for videos
 - A **blog** searcher to search blogs
 - And more
- ◆ You can also add styling via CSS
 - There is a whole section of the documentation that describes how to customize the look and feel using CSS
- ◆ You **can't** control how the results are displayed
 - You can only give the id of the element to place the results in, and some stylistic information via CSS
 - You don't get the results and display them yourself
 - Google wants to have total control over the display

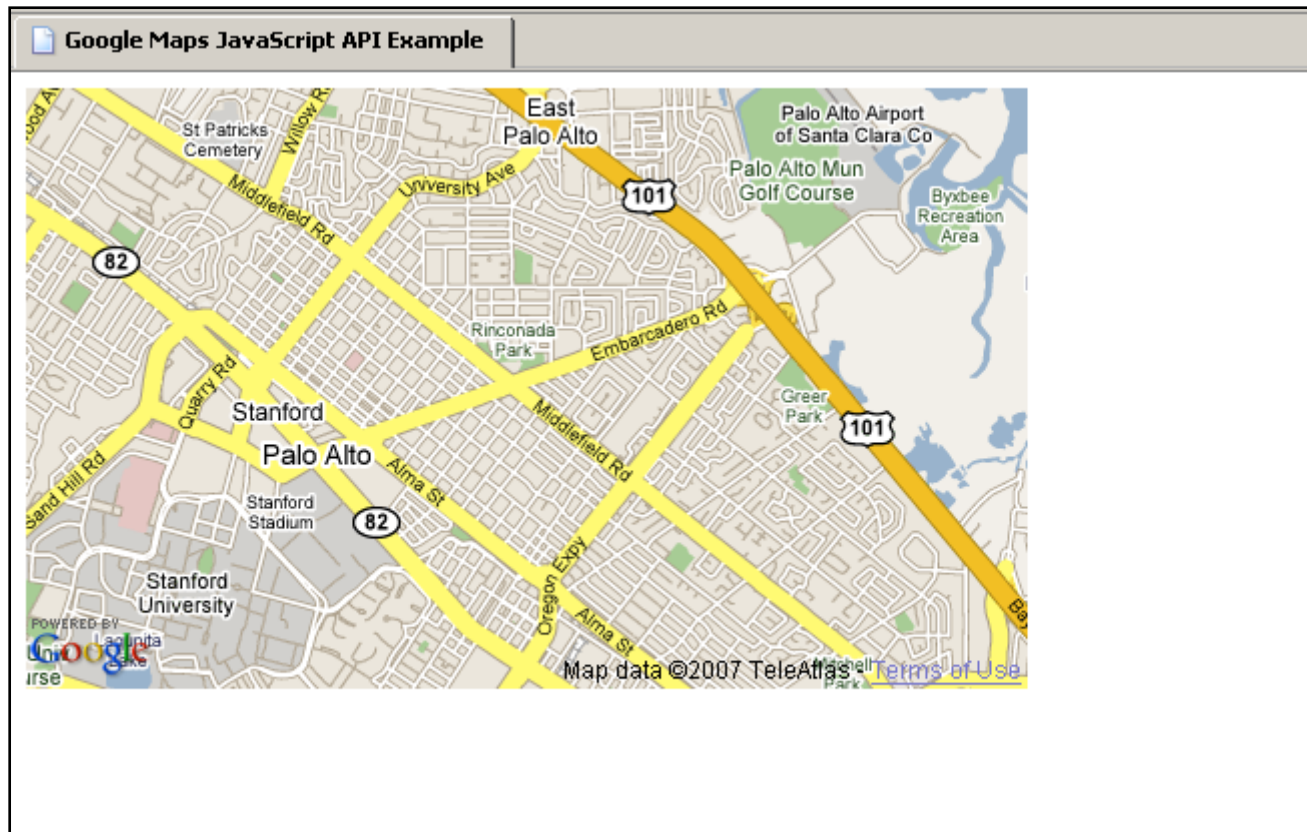
- Google Maps API -

- ◆ Works similarly to Search API
 - You include a JavaScript library, and use its objects
 - Below we show a simple usage, that displays a map centered near Palo Alto, CA

```
<head>
  <script src="http://maps.google.com/maps?file=api&v=2&key=XXX"
    type="text/javascript"></script>
  <script type="text/javascript">
    //
    function load() {
      if (GBrowserIsCompatible()) {
        var map = new GMap2(document.getElementById("map"));
        map.setCenter(new GLatLng(37.4419, -122.1419), 13);
      }
    }
    //]]&gt;
  &lt;/script&gt;
&lt;/head&gt;
&lt;body onload="load()" onunload="GUnload()"&gt;
  &lt;div id="map" style="width: 500px; height: 300px"&gt;&lt;/div&gt;
&lt;/body&gt;</pre></div><div data-bbox="0 967 59 990" data-label="Page-Footer"><p>20120920</p></div><div data-bbox="171 967 514 990" data-label="Page-Footer"><p>Copyright © 2007-12 LearningPatterns Inc. All rights reserved.</p></div><div data-bbox="658 965 881 986" data-label="Page-Footer"><p>Session 6: Client Side Frameworks</p></div><div data-bbox="968 967 997 988" data-label="Page-Footer"><p>284</p></div>
```

Maps API Display

- ◆ Below is the resulting map display
 - It has all of the normal Google map functionality
 - For example, you can scroll the map by sliding it
 - You can also add things like zoom controls, etc.

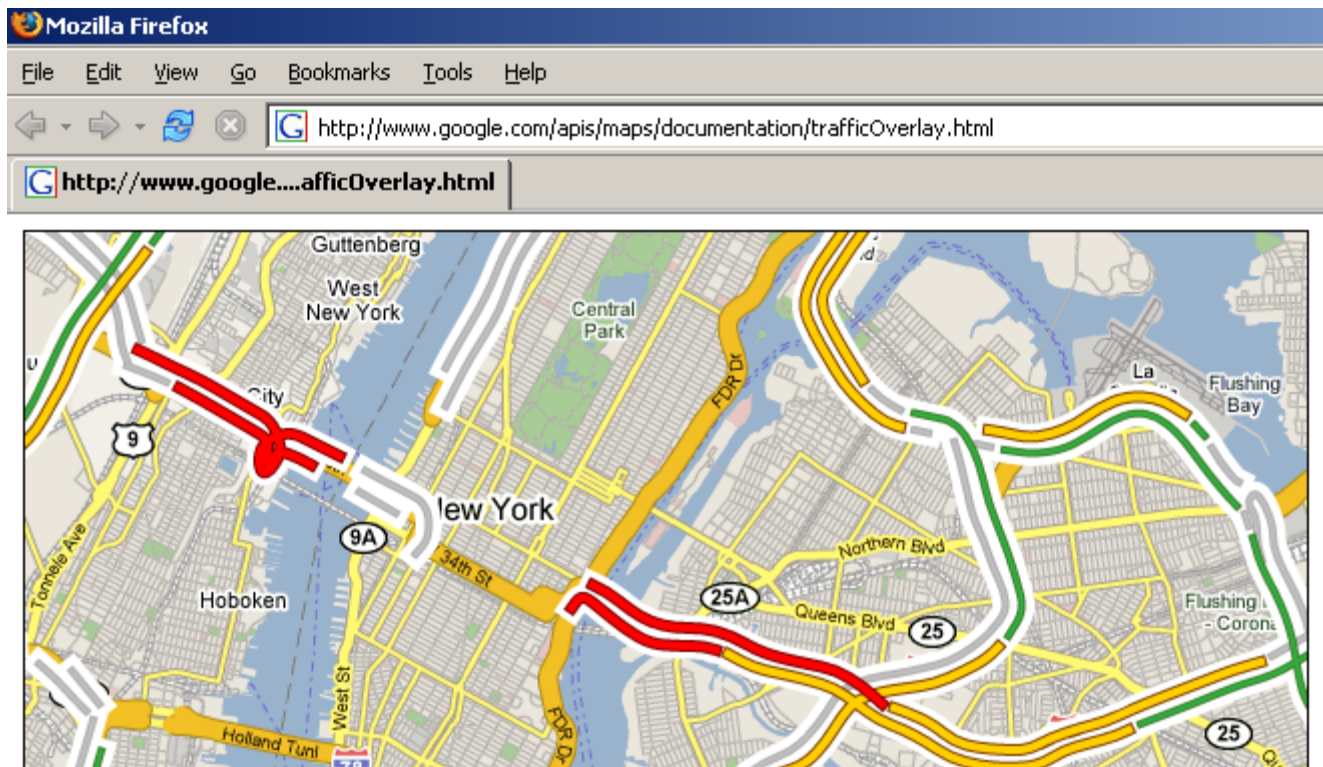


Additional Google Maps API Capability

- ◆ There is an enormous amount of additional capability
- ◆ A lot of it allows you to create "mashups" of your data and the map
 - Mashups combine your own content with content source from a third party – for example the Google Maps content
- ◆ Many possibilities for creating mashups with the Maps API
 - You can create your own overlay on the map by supplying icons and locations
 - You can also create an info window that pops up on the map with your own information
 - You can add geographic information using the KML and GeoRSS data formats
 - You can add traffic information using *GTrafficOverlay*
- ◆ Much capability – we don't go into details here

Traffic Overlay Example

- ◆ The traffic overlay is added using Google JavaScript objects
`var trafficInfo = new GTrafficOverlay();`
`map.addOverlay(trafficInfo);`



Review Questions

- ◆ Why would we bother with Ajax frameworks?
 - What capabilities do they give us?
- ◆ What capabilities does the **Prototype** library provide?
- ◆ What capabilities does the **script.aculo.us** library provide?

Lesson Summary

- ◆ Ajax frameworks make the boilerplate Ajax code much easier
 - They can take care of things such as handling the XHR communication details, browser differences, etc.
- ◆ Prototype is a JavaScript Framework that aims to ease the development of dynamic web applications
 - It provides a large number of useful JavaScript utilities, such as `$()`, `hide/show`, etc.
 - It also has a convenient Ajax interface in ***Ajax.Request***, ***Ajax.Updater***, and other classes
- ◆ ***script.aculo.us*** is a library built over Prototype that provides a more extensive set of widgets and capabilities
 - For example, an autocompleting Ajax-enabled input textbox
- ◆ There are many other frameworks with different capabilities
 - For example, Dojo, Google Search/Maps API, etc.



Session 7: Cascading Style Sheets

Lesson Objectives

- ◆ Understand the basics of what Cascading Style Sheets (CSS) are, and what they do
- ◆ Learn the basics CSS formatting language
- ◆ Create simple CSS style sheets
- ◆ Use CSS in conjunction with Ajax by scripting CSS
- ◆ We will not go deeply into the details of CSS at all
 - The purpose of the section is to show how to script CSS for use with Ajax - which is fairly simple
 - We give a brief overview of CSS for those who haven't used it
 - More coverage of CSS itself is beyond the scope of the course



Cascading Style Sheets

Issues with Formatting in HTML

- ◆ HTML lets you declare both document structure & formatting
 - Structure with tags like `<h1>`, `<h2>`, ... to specify headings, `<p>` to specify paragraphs, `<table>` to create tables
 - Formatting with tags like `` or `<i>`, and tag attributes like `` and `<table border="1">`
- ◆ HTML tags *should* be used to encode document structure
 - Where are the paragraphs, the hypertext links, the tables?
 - What is the overall document structure?
- ◆ HTML Tags are frequently misused for their formatting
 - h1, h2, etc. are intended to describe top-level, second-level and third-level headlines
 - Frequently used to provide “big, bold text” and used out of order

Issues with Formatting in HTML

- ◆ Using HTML tags for formatting leads to sites that are difficult to create and maintain
 - Formatting information embedded in each page, and distributed across all the pages
 - Making changes to a sites look-and-feel involves changing the formatting in each page
- ◆ HTML offers only rudimentary formatting capabilities
 - Font color can be set, but not background color
 - Limited control over font sizes
 - Many other shortcomings
- ◆ **Cascading Style Sheets** address these issues

Cascading Style Sheets (CSS)

- ◆ **CSS** helps separate out the formatting information from the structural information
 - You specify the structural information with HTML tags like `<p>`
 - You use CSS to specify a style sheet that specifies how the structure elements should be displayed
- ◆ CSS **style sheets** let you specify how different elements are displayed
 - For example, display `<h1>` content in bold, centered, upper-case 24-point letters
- ◆ Some of the goals for CSS are:
 - Make formatting of web sites simpler and easier
 - Provide a richer formatting capability than HTML
 - Allow styles to be accessed across multiple pages or websites
 - Provide flexibility in formatting

Declaring Style Information

- ◆ With CSS, styles are specified as name/value pairs separated by semicolons

font-weight: bold; color: red; font-style: italic;

- There are a large number of CSS style attributes
- The style sheet language is also fairly complex
- Covering them all is beyond the scope of this course

- ◆ Styles can be specified in separate style sheets, inline in an HTML page, or using the style attribute of an HTML element
 - Below we apply the styles (inline) to a specific <p> element (which is not really that useful, but it's simple)

```
<p style="font-weight: bold; color: red;  
        font-style: italic ">
```

This is bold red italic text

```
</p>
```


Style Sheets

- ◆ Style sheets let you separate out formatting from structure
 - Style sheets group all the style information in a single place
- ◆ A style sheet consists of a set of **style rules** which contain a **selector** and a set of **style attributes** in curly braces
 - A **selector** specifies what elements the rule applies to
 - The **style attributes** define the formatting to apply
- ◆ Below we show a style rule that applies the styling we saw earlier for a single `<p>` element to all `<p>` elements

```
p {  
    font-weight: bold;  
    color: red;  
    font-style: italic;  
}
```

Using Style Sheets

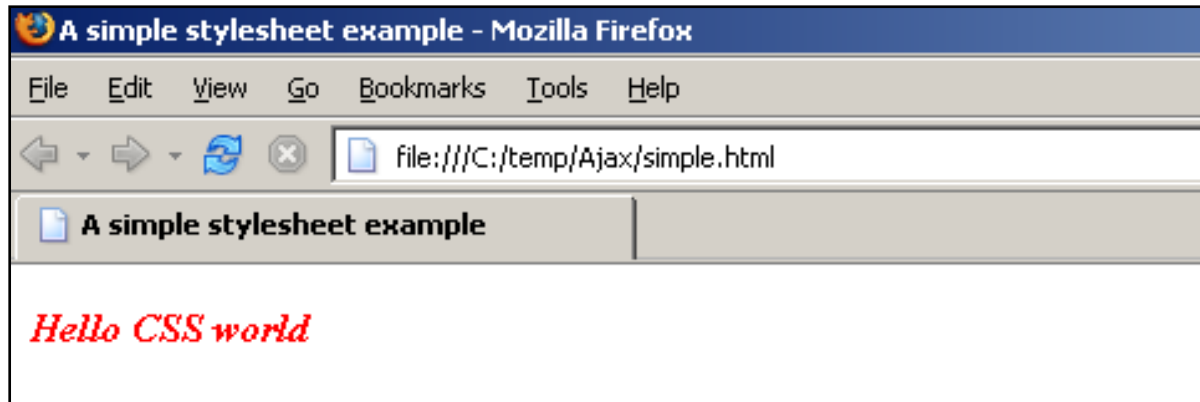
- ◆ You can declare style sheets internally in an HTML document, or in an external file
- ◆ External style sheets are declared using the <link> tag in the head section (see notes)

```
<html>
  <head>
    <title>A simple stylesheet example</title>
    <link href="simple.css" type="text/css" rel="stylesheet"/>
  </head>

  <body> <p>Hello CSS world</p> </body>
</html>
```

```
/* simple.css */
p {
  font-weight: bold;
  color: red;
  font-style: italic;
}
```

Resulting Display



- ◆ You can also declare the style sheet internally in the <head> tag with the same result

```
<head>
  <title>A simple stylesheet example</title>
  <style type="text/css">
    p {
      font-weight: bold;
      color: red;
      font-style: italic;
    }
  </style>
</head>
```

The class Selector

- ◆ You can define classes for HTML elements, and declare styles based on those classes
 - Using . (dot) and the name of the class in the selector

```
/* simple.css */  
p {font-weight: bold; color: red; font-style: italic; }  
.right {text-align: right}      /* Or p.right - see note */  
.center {text-align: center}
```

```
<body>  
  <p class="center">Hello CSS world</p>  
  <p class="right">To be Continued !</p>  
</body>
```

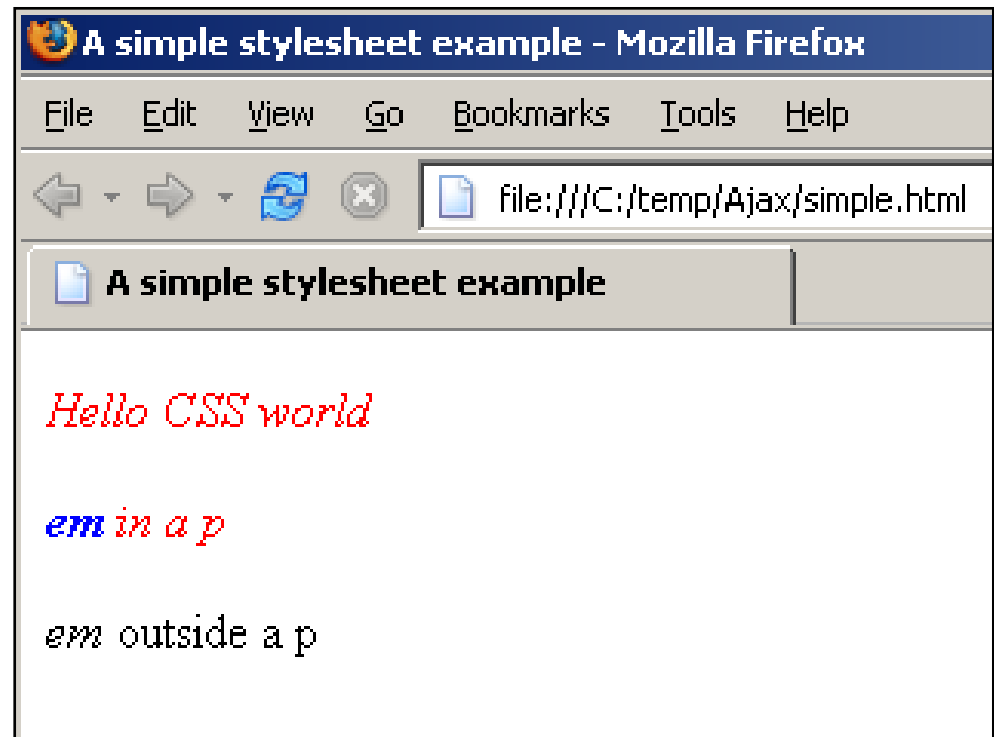


Descendant Selectors

- ◆ Used to select elements that are descendants of another element in the document tree
 - For example, you may wish to select `` elements that are within a `<p>` element, as shown in the example below

```
/* simple.css */  
p {  
    color: red;  
    font-style: italic  
}  
p em {  
    font-weight: bold;  
    color: blue;  
}
```

```
<body>  
  <p>Hello CSS world</p>  
  <p><em>em</em> in a p</p>  
  <em>em</em> outside a p  
</body>
```

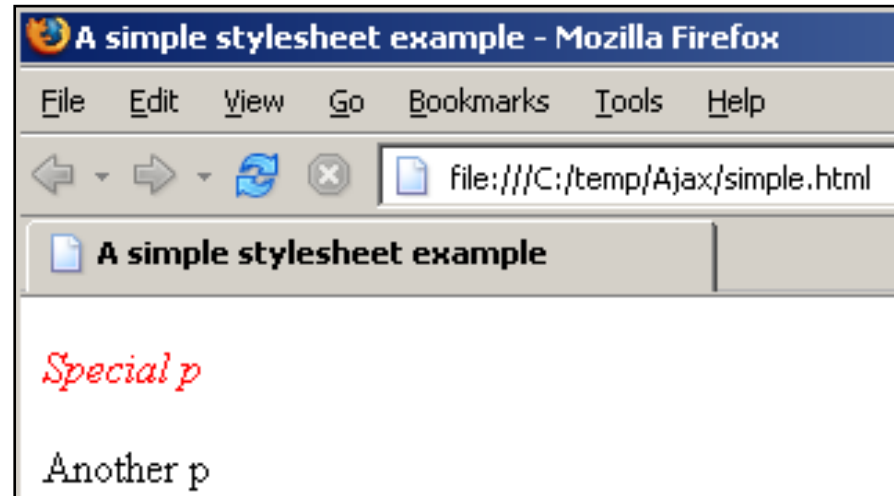


ID Selectors

- ◆ You can also define styles based on the id of an element
 - Using # in the selector and the name of the id

```
#special {  
    color: red;  
    font-style: italic  
}
```

```
<body>  
<p id="special">Special p</p>  
<p> Another p</p>  
</body>
```



- u There are other things you can do with CSS Selectors
 - For example, specify a style for an element with a particular attribute
 - It's beyond the scope of the course – we have the basics

display and visibility Style Properties

- ◆ The display and visibility properties are useful in Ajax
 - You often want to control the visibility of an object dynamically – e.g. make it visible and display it using Ajax data
 - These two properties allow you to do that
- ◆ **Visibility** sets if an element should be visible or invisible
 - The element still takes up space in the layout, but isn't displayed
 - It can have the following values:
 - **visible**: The element is visible
 - **hidden**: The element is invisible
 - **collapse**: For use in tables
- ◆ Display can also be used to make an element invisible
 - When it has the value "none"
 - The element then does NOT take up space in the layout

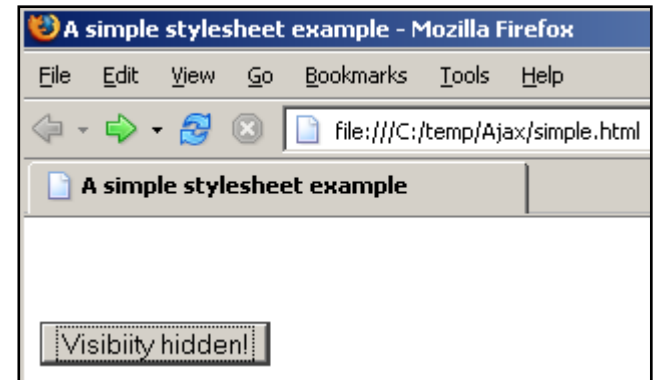
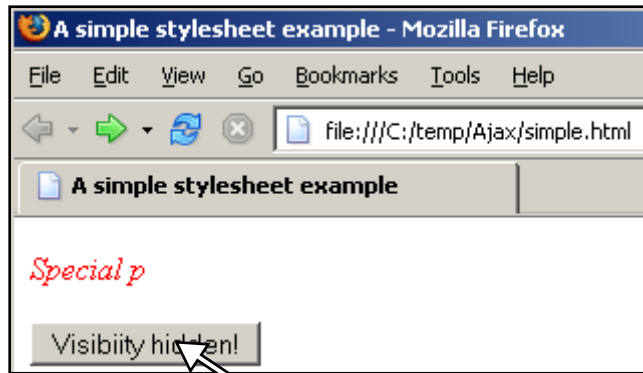
Other Style Properties

- ◆ There are many style properties available in CSS
 - We summarize the general categories here, and each category has a number of properties – see a CSS reference for details
- ◆ **Background**: define the background effects of an element
- ◆ **Text**: Define appearance of text
- ◆ **Font**: Define the font in text
- ◆ **Border**: Define the borders around an element
- ◆ **Outline**: Draw lines around elements outside the border
- ◆ **Margin**: Defines the space around an element
- ◆ **Padding**: Defines space between the border and the content
- ◆ **List**: Define appearance of the list and the list item markers
- ◆ **Table**: Set the layout of a table

Scripting Styles

- ◆ It's very easy to script element styles using JavaScript
 - Every element has a **style** property available, which allows you to access and modify the elements inline styles
 - This style object has a property for every style that is defined
 - Below is an example that shows this

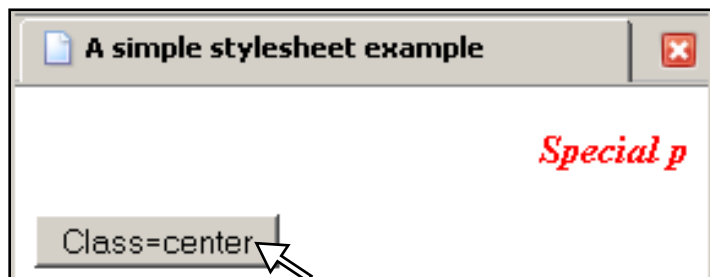
```
<body>  
<p id="special">Special p</p>  
  
<input type="button" value="Visibility hidden!"  
onclick="document.getElementById('special').style.visibility='hidden';"/>  
  
</body>
```



Scripting Classes

- ◆ It's also possible to script CSS classes
 - This is often easier than scripting individual style properties
- ◆ The *className* attribute of an HTML element is used to script its HTML class, as in the example below

```
<body>  
<p id="special" class="right">Special p</p>  
  
<input type="button" value="Class=center"  
onClick="document.getElementById('special').className='center';"/>  
</body>
```



The "Cascading" in CSS

- ◆ Multiple style sheets "cascade" into one
 - You might specify styles in different places that apply to the same element
- ◆ All the styles that apply to an element will cascade into one virtual style
 - All the styles specified are applied to the element
 - If styles from different places overlap, the following rules are used to choose the style used (with 4 being the highest priority)
 1. Browser Default
 2. External Style Sheet
 3. Internal Style Sheet (inside the <head> tag)
 4. Inline Style (inside an HTML element)
 - An inline style (inside an HTML element) has the highest priority, which means that it will override a style declared anywhere else

Lab 7.1 – Using CSS to Modify Ajax.AutoCompleteter

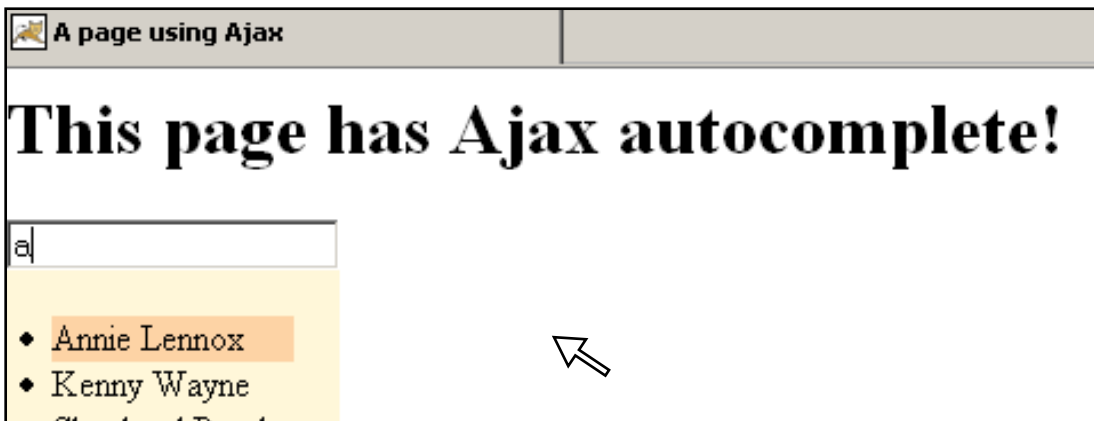
- ◆ **Overview:** In this lab you'll use CSS to customize the display of the script.aculo.us autocompleter output
 - You'll add a class to the <div> that holds the output, and then create a CSS style sheet that applies to that div
- ◆ **Objectives:**
 - Gain experience using CSS with Ajax
- ◆ **Builds on previous labs:** Lab 06.2
 - Continue working in your **Lab03.1** directory
 - Note that we are no longer using the YUI lab directory - that was done just for illustration of some YUI capabilities
- ◆ **Approximate Time:** 15-20 minutes

Tasks to Perform

- ◆ Open the file *hello_ajax.html* for editing
 - Add in a link to a stylesheet, `css/ajax.css`
 - Add in a class attribute of *autocomplete* to the div holding the autocomplete output (the one with *id="artist_choices"*)
 - Save this file
- ◆ Create a `css` directory under your *WebContent* directory
- ◆ Create a stylesheet file (*ajax.css*) under in the `css` directory, and add the following to the stylesheet
 - A selector for all `` elements within the autocomplete div:
div.autocomplete ul
 - Within that selector, add the following properties
margin:0px;
padding:20px;
background-color: #fff7d9;

Tasks to Perform

- Add a selector for all selected `` elements within `` elements elements within the autocomplete div:
`div.autocomplete ul li.selected`
- Within that selector, add the following property
`background-color: #fdd3a5;`
- ◆ **Restart Tomcat**
 - Browse to `http://localhost:8080/javatunes` to view your page
- ◆ Type the letter "a" into the textbox – you should see the completions formatted using the CSS styles



Review Questions

- ◆ What are Cascading Style Sheets (CSS) and what capabilities do they give us?
- ◆ What is a style rule?
- ◆ How do you use styles in a Web page?
- ◆ How do you script styles with JavaScript?

Lesson Summary

- ◆ **CSS** lets you define **style sheets** that separate out the formatting information from the structural information
 - This helps make formatting of web sites easier, provides a richer formatting capability than HTML, and allows styles to be accessed across multiple pages or websites
- ◆ Style sheets are composed of **style rules** which contain:
 - **Selectors** that specify what elements the rule applies to
 - **Style attributes** that define the formatting to apply

```
p {  
    font-weight: bold;  
    color: red;  
    font-style: italic;  
}
```

Lesson Summary

- ◆ You can declare style sheets internally in an HTML document within a `<style>`, or in an external file using the link tag
`<link href="simple.css" type="text/css" rel="stylesheet"/>`
- ◆ It's very easy to script element styles with JavaScript using the *style* object and its properties
`document.getElementById("special").style.visibility="hidden";`
- ◆ You can also script an element's class with JavaScript using its `className` attribute
`document.getElementById("special").className="left";`

Session 8: JSON (JavaScript Object Notation)

JSON Overview

JSON on the Server

Autocomplete Example Using JSON

Other JSON Tools

Lesson Objectives

- ◆ Understand what JSON is, and its role in Ajax
- ◆ Learn how to use JSON in Ajax programs on both the client and server side

JSON (JavaScript Object Notation) Overview

JSON Overview

JSON on the Server

Autocomplete Example Using JSON

Other JSON Tools

What is JSON

- ◆ Key Ajax idea is alternative to page refresh in the browser
 - Implies data interchange between the browser and the server
 - The question arises as to what we use to exchange the data
- ◆ **JSON** (JavaScript Object Notation) is a lightweight data interchange format
 - It is based on the JavaScript object/array structure
- ◆ JSON is a simple, easily understandable text format
 - It is easily readable by people
 - It is very easy to work with using JavaScript in a browser
 - It is independent of the programming language
 - Can be generated from many server environments
 - Generally the client side will be JavaScript in a browser
 - It is very familiar to anyone who has worked with JavaScript

Review of JavaScript Literals

- ◆ An object literal defines the properties of an object and their values
 - It is a comma separated list of **property** name/value pairs enclosed in braces
 - The values can be simple values, other objects, or arrays
 - The examples show one object with x & y properties with numeric values, and another object with string and numeric values
 - JSON uses this same object literal notation to transfer data
 - JavaScript is fairly flexible in defining object literals, but JSON is stricter, basically requiring the format shown in the examples below

```
var point1 = {  
    "x":2,  
    "y":4  
};  
var aPerson = {  
    "name":"Jane",  
    "age":30  
};
```

Arrays and More Complex Objects

- ◆ Array literals in JavaScript are composed of zero or more elements enclosed in square brackets (`[]`)
 - The elements may be simple values, objects, or other arrays

```
// Array containing simple strings (State abbreviations)
var states = [ "AL", "AK", ... "WY" ];
// Array containing two objects with x/y values
var points = [
    {"x":0, "y":1},
    {"x":1, "y":2}
];
// Object with a string property and an array property
var line = {
    "description" : "A line with three point" ,
    "points" : [
        {"x":5, "y":6},
        {"x":6, "y":7}
    ]
}
```


JSON Details

- ◆ JSON is a text-based data interchange format
 - For example, the JSON representation of the points var on the previous slide containing an array of two objects is the string:

```
[ {"x":0, "y":1}, {"x":1, "y":2} ]
```
- ◆ JSON is a subset of the JavaScript object notation
 - It has more limitations than a JavaScript object
 - A JSON value MUST be a number, string (in quotes), object, array, or one of the following three literals: *false null true*
 - The literal names MUST be lowercase
 - Numbers may have exponents,
 - Hex and octal numbers are not allowed
 - NaN and Infinity are not allowed
 - Strings must appear in quotes
 - There is no native JSON date/time literal format

Creating JSON Strings in JavaScript

- ◆ There is no native support for creating JSON text
 - However, the reference implementation, available at <http://www.json.org/json2.js>, provides a JSON object with a ***stringify*** method to convert the following to a JSON string:
array, boolean, date, number, object, string
 - The value to be evaluated can't contain cyclical references
 - Illegal values in the value will be ignored
 - The default conversion for dates is to an ISO string
 - This allows you to turn objects into JSON strings with a single method call, as shown in the example below

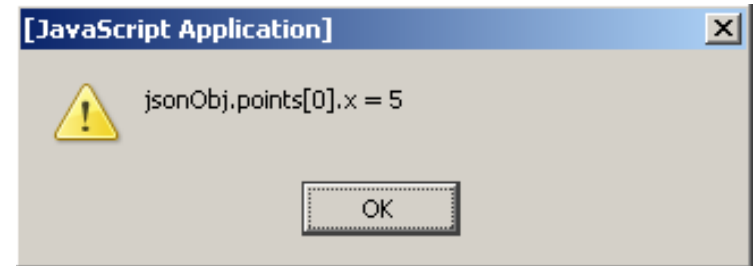
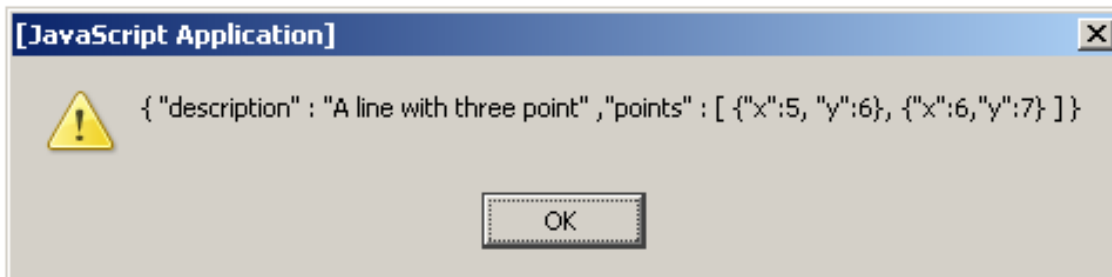
```
// Include json2.js before this
var states = new Array();
states.push("AL");
states.push("AK");
alert(JSON.stringify(states));
```



Parsing JSON Strings in JavaScript

- ◆ Since any valid JSON string is a valid JavaScript literal, it is trivial to parse JSON strings
 - JavaScript has an ***eval()*** function that takes a string of valid JavaScript code, and evaluates the expression
 - When used on a JSON string, it converts it into a value of the appropriate type (e.g. an object or an array)

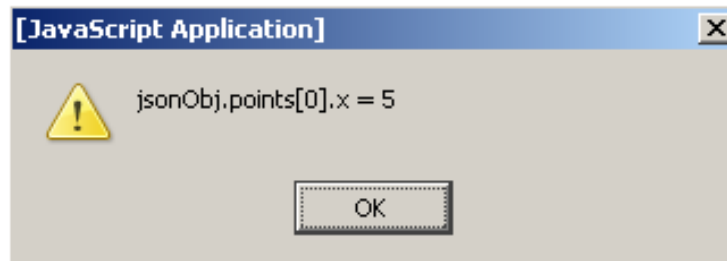
```
var jsonText = '{ "description" : "A line with two points" , ' +  
    '"points" : [ {"x":5, "y":6}, {"x":6,"y":7} ] }';  
alert(jsonText);  
var jsonObj = eval( "(" + jsonText + ")" );  
alert("jsonObj.points[0].x = " + jsonObj.points[0].x);
```



Parsing Strings with `JSON.parse()`

- ◆ `json2.js` adds a **`JSON.parse()`** method that validates a passed in string as valid JSON, and only then calls `eval()`
- ◆ Using `eval()` directly can be dangerous
 - `eval()` will evaluate whatever you pass, be it JSON strings or other JavaScript code (e.g. executable methods)
 - Input from untrusted sources can include potentially dangerous JavaScript code

```
var jsonText = '{ "description" : "A line with three point" , ' +  
    '"points" : [ {"x":5, "y":6}, {"x":6,"y":7} ] }';  
var jsonObj = JSON.parse(jsonText);  
alert("jsonObj.points[0].x = " + jsonObj.points[0].x);
```



JSON on the Server

JSON Overview

JSON on the Server

Autocomplete Example Using JSON

Other JSON Tools

Creating JSON Text on the Server

- ◆ There are many ways to generate a JSON string on the server
 - There is no standard way to do this now – JSON is not standardized yet in any way
 - We will look at a couple of tools that make this fairly easy
- ◆ **json-lib** is a java library for transforming beans, maps, collections, Java arrays, and XML to JSON and back again
 - Supplied as a Java class library in the *net.sf.json* package
 - It includes support for the following type conversions

JSON		Java
string	<=>	java.lang.String, java.lang.Character, char
number	<=>	java.lang.Number, byte, short, int, long, float, double
true false	<=>	java.lang.Boolean, boolean
null	<=>	null
function	<=>	net.sf.json.JSONFunction
array	<=>	net.sf.json.JSONArray (object, string, number, boolean, function)
object	<=>	net.sf.json.JSONObject

JSONObject and JSON

- ◆ The *JSONObject* class is the most basic way to use JSON-lib
 - It represents an unordered collection of name/value pairs
 - Its external form is the JSON text for an object
 - *put(String key, Type value)* methods are used to add name/value pairs to the instance

```
new JSONObject().put("JSON", "Hello, World!").toString();
```

produces the string

```
{"JSON": "Hello, World"}
```

- There are many versions of the put method to add in the different support types - e.g. *put(String key, boolean value)*

JSONArray

- ◆ The *JSONArray* type allows you to build JSON arrays easily
 - It represents an ordered sequence of values
 - Its external form is the JSON text for an array
 - *add (Object value)*, and *add (int index, Object value)* methods can be used to append or place values on the array
 - The example below, produces the JSON text:

[{"age":30,"name":"Jeremy Programmer"}, {"age":35,"name":"Felicia Manager"}]

```
JSONObject json_1 = new JSONObject();
JSONObject json_2 = new JSONObject();
json_1.put("name", "Jeremy Programmer");
json_1.put("age", 30);
json_2.put("name", "Felicia Manager");
json_2.put("age", 35);
JSONArray jarray_1 = new JSONArray();
jarray_1.add(json_1);
jarray_1.add(json_2);
System.out.println(jarray_1);
```


Creating JSON Text from POJOs

- ◆ The ***JSONObject.fromObject()*** method makes it very easy to create JSON text from a POJO

static JSONObject fromObject(Object bean)

- Let's look at an example using a simple fruit class with properties for an id, name, pickDate, and price (see notes for the class def)
- The code below produces the JSON text:

```
{ "price":2.99,  
  "pickDate":{"month":5,... much more data, see notes ... },  
  "name":"apples",  
  "id":1  
}
```

```
Fruit f = new Fruit(new Long (1), "apples", new Date(),  
                    new BigDecimal("2.99"));  
JSONObject f_1 = JSONObject.fromObject(f);  
System.out.println(f_1);
```

Creating JSON Text From Collections

- ◆ You can create JSON from a collection using JSONArray's method:

boolean addAll(Collection collection)

- Let's look at an example using a collection of fruit instances
- The code below produces the JSON text:

```
[  
  { "price": 2.99, "pickDate": { "month": 5, ...}, "name": "apples", "id": 1 },  
  { "price": 0.99, "pickDate": { "month": 5, ...}, "name": "oranges", "id": 2 }  
]
```

```
Fruit f1 = new Fruit(new Long (1), "apples", new Date(), new  
BigDecimal("2.99"));  
Fruit f2 = new Fruit(new Long (2), "oranges", new Date(), new  
BigDecimal(".99"));  
ArrayList fruit = new ArrayList();  
fruit.add(f1);  
fruit.add(f2);  
JSONArray a_1 = new JSONArray();  
a_1.addAll(fruit);  
System.out.println(a_1.toString(2));
```

Dealing with Dates

- ◆ json-lib doesn't have special serialization support for dates
 - A date value is treated like any other object, and its properties are just extracted and included in the generated text
 - This is partly because JavaScript doesn't have any date literal format defined - though the **ISO 8601** date format is becoming fairly standard for JSON (e.g. 2008-01-01T13:41:21Z)
- ◆ json-lib lets you define custom serialization behavior for generating JSON text using the *JsonConfig* and *JsonValueProcessor* classes
- ◆ *JsonValueProcessor* defines the *processArrayValue* and *processObjectValue* methods to provide custom processing
 - You create a class that implements the interface and methods
- ◆ *JsonConfig* lets you specify that your processor should be applied to objects of a particular type via the method:
registerJsonValueProcessor

Custom Date Serialization

- ◆ The following code, registers a processor for dates
 - It generates JSON in ISO 8601 form
 - You can see, that it's fairly easy to add this custom behavior

```
// code fragment - only relevant detail not shown ...
```

```
JsonConfig jsonConfig = new JsonConfig();  
jsonConfig.registerJsonValueProcessor(Date.class,  
    new JsonValueProcessor() {  
        public Object processArrayValue(Object value, JsonConfig jsonConfig) {  
            return processDate(value);  
        }  
        public Object processObjectValue(String key, Object value,  
                                         JsonConfig jsonConfig) {  
            return processDate(value);  
        }  
    }  
);
```

```
private String processDate(Object dateObj) {  
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss");  
    return sdf.format((Date)dateObj);  
}
```

Custom Date Serialization

- ◆ You use the *JsonConfig* object to specify custom behavior
 - The code fragment below will result in date serialization as shown at bottom
 - The JSON date text is generated by *processDate()* in the previous slide

```
ArrayList fruit = new ArrayList();  
// Much code omitted - fruit list created as before  
JSONArray a_1 = new JSONArray();  
// Pass in the config object to customize serialization  
a_1.addAll(fruit, jsonConfig); // jsonConfig from previous slide  
System.out.println(a_1.toString(2));
```

```
[  
  {  
    "id": 1,  
    "name": "apples",  
    "pickDate": "2008-03-18T10:38:36",  
    "price": 2.99  
  }, ... remaining detail omitted  
]
```

JSONSerializer

- ◆ The *JSONSerializer* class makes it very easy to convert Java objects to JSON text
 - It takes care of the details of figuring out if you have a collection, array, etc, and generates the JSON for you
 - The core functionality is contained in the methods:
static JSON toJSON(Object object);
static Object toJava(JSON json);
 - Both methods also have versions taking a *JsonConfig* object
- ◆ It returns an object that implements the *JSON* interface, which includes the following methods
 - *boolean isArray()*: Returns true if object is a JSONArray, else false
 - *int size()*: Returns number of props in an object or the size of array
 - *String toString(int indentFactor)*: Make prettyprinted JSON
 - *String toString(int indentFactor, int indent)*: Make prettyprinted JSON
 - *Writer write(Writer writer)*: Write contents as JSON text to writer

JSONStringer

- ◆ *JSONStringer* is a convenient way to create JSON strings
 - Often easier than using *JSONObject*
 - The code at bottom will produce the JSON string:
{"JSON": "Hello, World!"}
- ◆ Subclass of *JSONBuilder* - includes the following methods
 - *JSONBuilder value*(Type xx): Append a value
 - *JSONBuilder key*(String s): Append a key
 - *JSONBuilder object*()/*endObject*(): Start/end an object
 - *JSONBuilder array*()/*endArray*(): Start/end an array

```
myString = new JSONStringer()  
    .object()  
    .key("JSON")  
    .value("Hello, World!")  
    .endObject()  
    .toString();
```

Other json-lib Capabilities

- ◆ The json-lib library has a lot more capability
 - For example, there is functionality to generate Java objects from JSON text – for use where clients send data in the form of JSON
- ◆ We've given you a start and a feel for how json-lib is used
 - That's all we'll cover in the course on this tool
- ◆ Take some time to look at the documentation
 - It is included in the Setup, under Resources\json-lib
 - There is a documentation zip file in that directory which contains the javadoc for the library
 - Unzip this file to your computer, and browse the documentation for a few minutes
 - Look especially at the three classes, *JSONObject*, *JSONArray*, *JSONSerializer*, and the *JSON* interface

Autocomplete Example Using JSON

JSON Overview

JSON on the Server

Autocomplete Example Using JSON

Other JSON Tools

An Autocomplete Example With JSON

- ◆ Let's take a look at an example that will use JSON data generated in a servlet to generate an autocomplete suggestion box in the client
 - This will be based on a simple example using *Fruit* objects
- ◆ There are a number of parts to this:
 - **An input field** in an HTML page that has which will include autocomplete functionality, including making an Ajax request on each keypress for autocomplete suggestions
 - **A servlet** which generates JSON text based on the current input in the input field
 - **JavaScript code** which takes the JSON response from the servlet, and generates and displays the autocomplete suggestions
- ◆ We'll look at each of these individually

An Input Field Generating Ajax Requests

- ◆ In the example below, we show an input field that generates Ajax requests which return JSON text as the response
 - Note the **onkeyup** event handler for the input box – this will trigger the Ajax request when a character is entered
 - We use our *XHR.makeRequest* function to make an Ajax request, and pass in a callback function that expects JSON text as an argument
 - The callback function calls another function, **doCompletion()**, passing in as an argument a JSON object created from the response

```
<input type="text" id="autoTBox"
  onkeyup="XHR.makeRequest( {
    method: 'POST',
    ajaxURL: '/javatunes/json', postArg: 'fruit='+this.value,
    callback: function(txt) { doCompletion(JSON.parse(txt)); }
  } )"; />

<div id="suggestDiv" style="display:none"
  class="autocomplete"> </div>
```

Producing JSON in a Servlet

- ◆ The example below shows a servlet that creates the JSON string in response to a request, and sends it as a response
 - The response is JSON text representing an array of fruit objects
 - Each object has price, name, and id properties, giving JSON similar to:
[{ "price": 2.99, "name": "apples", "id": 1 }, { ... }, { ... }]

```
public class JSONServlet extends HttpServlet {  
  
    public void doPost(HttpServletRequest req,  
        HttpServletResponse res) throws ServletException, IOException {  
  
        response.setContentType("application/json"); // JSON MIME type  
        PrintWriter out = response.getWriter();  
        String fruitString = req.getParameter("fruit");  
        ArrayList fruit = // Initialized somehow using fruitString  
        String[] excludes = { "pickDate" };  
        JSON jsonFruit = JsonSerializer.toJSON(fruit, excludes);  
        jsonFruit.write(out);  
        out.close(); // Close the PrintWriter  
    }  
}
```

JavaScript Code Constructing Suggestions

- ◆ *doCompletion()* creates the suggestions by iterating through the JSON array, and creating a `` from it

- For each element of the array, it creates an `` of the form:

```
<li class="" onmousedown="doMouseClicked('apples');"
    onmouseout="this.className=''"
    onmouseover="this.className='selected'">apples</li>
```

```
<script>
function doCompletion(jsonObj) {
    var el = document.getElementById("suggestDiv");
    var txt = "<ul>"
    for (var cur=0; cur<jsonObj.length; cur++) {
        txt += '<li '
        + 'onmouseover="this.className=\'selected\';" '
        + 'onmouseout="this.className=\'\';" '
        + 'onmousedown="doMouseClicked(\' ' + jsonObj[cur].name + '\');" '
        + '>' // end <li> tag
        + jsonObj[cur].name + '</li>';
    }
    txt += "</ul>";
    el.innerHTML = txt;
    el.style.display="";
}
```

Accessing the JSON Data We Want

- ◆ In the previous JavaScript code, there are two key places to look at:
- ◆ *doCompletion()* receives a JavaScript object constructed from the JSON text

- This object will be an array of fruit objects, each with price, name, and id properties, and the following code iterates over this array:

```
for (var cur=0; cur<jsonObj.length; cur++) {
```

- ◆ In the body of the for, we construct a string representing an ** that includes the name property of the current fruit

- This value is accessed with the following expression:

```
jsonObj[cur].name
```

- This expression is used to add the fruit name as an argument to the *doMouseClicked* function, and as the content of the **

```
+ 'onmousedown="doMouseClicked(\' ' + jsonObj[cur].name + '\');
```

```
+ '>' // end <li> tag
```

```
+ jsonObj[cur].name + '</li>';
```

arg to doMouseClicked()

Contents of ...

JavaScript Code Constructing the Suggestions

- ◆ The *doMouseClick()* function is called in response to a mouse click on an item in the suggestion list
 - It receives the content of the `` as an argument
 - It makes the suggestion list invisible (by setting the style *display:none*)
 - It sets the text of the input field to be the text from the ``
- ◆ That's it – you've looked at all the parts
 - Let's see how it runs

```
function doMouseClick(val){  
    document.getElementById("suggestDiv").style.display="none";  
    //update text box with selected item  
    var autoTBox = document.getElementById("autoTBox");  
    autoTBox.value = val;  
}  
</script>
```

Autocomplete at work

- ◆ Here, we show the autocomplete list
 - Below, we also show the json text returned from the servlet, and the `` generated from the JSON



POST http://localhost:8080/javatunes/json (16ms)

Headers Post Response

```
[{"price":2.99,"name":"apples","id":1},{ "price":0.99,"name":"oranges","id":2}]
```



```
<li class="" onmousedown="doMouseClicked('apples');" onmouseout="this.className="" onmouseover="this.className='selected'">apples</li>
```

```
<li onmousedown="doMouseClicked('oranges');" onmouseout="this.className="" onmouseover="this.className='selected'">oranges</li>
```


Other JSON Tools

JSON Overview
JSON on the Server
Autocomplete Example Using JSON
Other JSON Tools

The JSON Universe

- ◆ JSON is still maturing, with lot of development going on
 - There is also not a lot of standardization yet
 - json-lib is fairly well known, but there are many more tools
 - Here are a few
- ◆ **JSON-RPC** – Allows JavaScript DHTML web applications to call remote methods in a Java Application Server via Ajax
- ◆ **JSON-taglib** - JSP 2.0 tag library used to render JSON (JavaScript Object Notation) data from within JSP code
 - This is an alternative to creating the JSON string in the servlet using Java code
- ◆ **JsonT** – A transformer library to convert JSON into other formats
 - For example, into HTML
 - <http://goessner.net/articles/jsont>



[Optional] Lab 8.1 – Generating and Using JSON Data

- ◆ **Overview:** In this lab you will create your own (very simple) autocomplete textbox
 - The autocomplete functionality will use JSON data from a servlet to generate the autocomplete list
 - You will write the code to both generate the JSON data in the servlet, and to generate the list from the data using JavaScript
- ◆ **This lab is challenging**, since it puts a lot of pieces together
 - You may skip it if you feel short of time
- ◆ **Objectives:**
 - Gain experience generating JSON data using Java
 - Gain experience using JSON data with JavaScript
- ◆ **Builds on previous labs:** none
- ◆ **Approximate Time:** 35-45 minutes

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab08.1

- This is a new lab directory

Tasks to Perform

- ◆ Right click on the server in Servers View
 - Select **Add and Remove Projects**, and **remove** Lab03.1
- ◆ Create a new **Dynamic Web Project** called **Lab08.1** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server is selected
 - Make sure **2.5 module version** is selected
 - Remember to set the context root to **javatunes**

- ◆ You'll work with one file in **Lab08.1\WebContent**
 - **hello_ajax.html**: An HTML file that will have an autocomplete suggestion list created from JSON data from a servlet
- ◆ You'll work with one file in **Lab08.1\src\com\javatunes\ajax**
 - **JSONServlet.java**: A Java servlet that generates JSON data in response to an XHR request
- ◆ There are other files in the lab that will generate a response to the search so you can see it work – these are supplied to you
 - *SearchServlet.java* does the actual search
 - *searchResults.jsp* displays the search results
 - We've seen these in previous labs

- ◆ *JSONServlet* generates a JSON array holding autocomplete data that is generated for a request parameter called *keyword*
 - We've given you a skeleton file to start with (see notes)

Tasks to Perform

- ◆ Open the file *JSONServlet.java* for editing
- ◆ First, import *JSONSerializer* (in package *net.sf.json*)
 - We'll need this to generate our JSON data
- ◆ Go to the *doPost()* method (see the next slide for code examples)
 - Set the content type to *application/json*
 - Generate the JSON data with the *JSONSerializer.toJSON()* method on the results collection
 - This method returns a JSON object, which has a *write()* method - use this method to write the JSON object to the servlet writer
 - You can also print it to the console if you want to see the JSON string *
 - Compile cleanly (*build compile*) and you're done with the servlet

- ◆ Below is a code fragment from the doPost() method
 - At TODO 1, you simply need to use the correct string value to set the content type
 - At TODO 2, you need to use JsonSerializer to convert the results collection to a JSON instance
 - Once you do that, you can use its write() method to write itself out to the PrintWriter out.

```
String artist = request.getParameter("artist");  
List results = (List)SearchUtility.findByArtist(artist);  
  
// TODO 1: Set content type to application/json  
response.setContentType("TODO");  
PrintWriter out = response.getWriter();  
  
// Send back some JSON  
// TODO 2: Use the JsonSerializer to convert the results to JSON,  
// and write them out to the Servlet's PrintWriter
```


Tasks to Perform

- ◆ Open the file *hello_ajax.html* for editing
 - Look for the TODO comments – there are quite a few of them
- ◆ First, include *js/json2.js* into the file in the head section
 - This includes the JSON library that we'll be using
- ◆ Next look at the following HTML code in *hello_ajax.html*
 - This is the code for our autocomplete textbox
 - We'll go over it piece by piece

```
<form action="/javatunes/search" method="POST">
  <input type="text" id="autoTBox" name="artist" autocomplete="off"
    onkeyup='XHR.makeRequest( {method: "POST", ajaxURL: "/javatunes/json",
      postArg: "artist=" + this.value,
      callback: function(txt) {
        doCompletion( /*TODO - add in the argument */ )
      }
    } )'; />
  <input type="submit" name="Submit" value="Search"/>
</form>
```

- ◆ The `<form ...>` element is a normal HTML element
 - The `/javatunes/search` action goes to the simple search servlet we've provided – we're not concerned with the servlet
 - There is a submit button that submits the form
- ◆ The `<input ...>` element is a normal HTML element
 - The attributes below specify information that we'll need for our lab to work
 - `type="text" id="autoTBox"` are as normal
 - `name="artist"` names the request parameter for submission
 - `autocomplete="off"` turns off built-in browser autocomplete *
- ◆ The **onkeyup** attribute is part of the autocomplete functionality
 - When a key is pressed, then released, this event is fired
 - We've defined `onkeyup` to perform an Ajax request, get a JSON string containing suggested completions from the response, and then display the completions within the HTML page

```
onkeyup='XHR.makeRequest( {method: "POST", ajaxURL: "/javatunes/json",  
    postArg: "artist="+ this.value,  
    callback: function(txt) {  
        doCompletion( /*TODO - add in the argument */ )  
    }  
}
```

- ◆ In the *onkeyup* event, we make an Ajax request using the utilities we've written in earlier labs
 - The request is to */javatunes/json* – which maps to the *JSONServlet* you just completed, and which returns the autocomplete data in a JSON array string
 - The *postArg*, which will be passed with the request, includes the text entered into the input textbox (retrieved via *this.value*)
 - The callback function, which is called with the response data from the Ajax request, calls a JavaScript function, *doCompletion()*, which will generate the suggest dropdown
 - You need to complete the call to this function

```
onkeyup='XHR.makeRequest( {method: "POST", ajaxURL: "/javatunes/json",
    postArg: "artist="+ this.value,
    callback: function(txt) { doCompletion( /*TODO - add argument */ ); }
} )'; />

```

Tasks to Perform

- ◆ Finish the **doCompletion()** call, which requires 1 argument
 - The argument needs to be a JSON object constructed from the JSON string passed to the callback function (and called txt)
 - Generate this with **JSON.parse()** from the JSON library, which is called directly on the string passed to the callback (txt)
- ◆ Note that the div with the *id="suggestDiv"* is the suggestion div – it will be filled with the suggestions

The doCompletion Function

```
function doCompletion(jsonObj) {  
    var el = document.getElementById("suggestDiv");  
    var txt = "<ul>"  
    for (var cur=0; cur<jsonObj.length; cur++) {  
        txt += '<li' + 'onmouseover=this.className="selected" '  
            + 'onmouseout=this.className="" '  
            + 'onmousedown="doMouseClicked(\' ' + /* TODO add artist */ + '\');"'  
            + '>' // end <span> tag  
            + /*TODO - add in artist*/ + '</li>'  
        }  
        // TODO: Set the el contents to txt  
        // TODO: Set the el to be visible *  
    }  
}
```

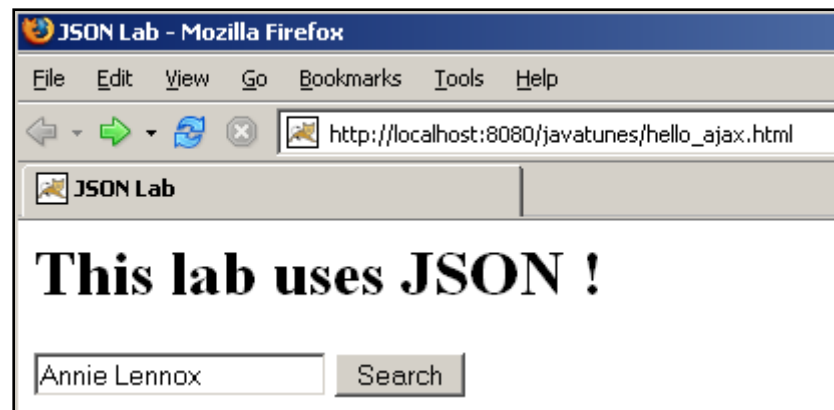
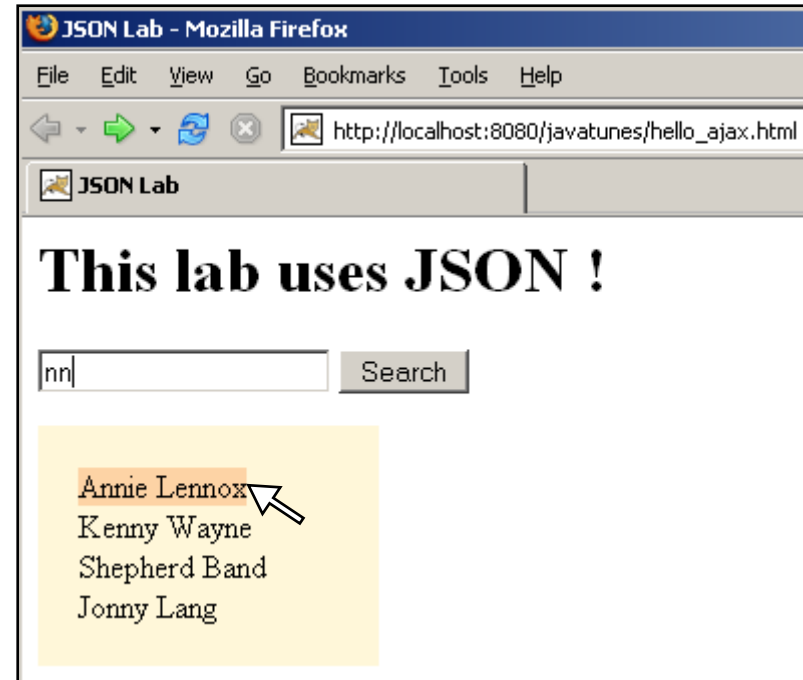
- ◆ `doCompletion()` is called when the Ajax request succeeds, and it:
 - Gets the suggestion div, and creates a string which has a `` element
 - Iterates through the JSON array passed in as the first arg
 - For each object in the array, it creates a string with an `` a number of events, and the `` content set to the artist in the current object, then adds this string to the text being generated (see note for example)
 - Fills the suggestion div with the generated text, and makes it visible

Tasks to Perform

- ◆ Complete the *doCompletion(jsonObj)* function:
 - First, you must complete the code that generates the `` element text by including the artist property of the current object being processed in the loop
 - Right now this is just a placeholder - "TODO - add in artist"
 - Change this to be the artist property in the current object in the JSON array (the current object is accessed as `jsonObj[cur]`)
 - Secondly, set the contents of the suggestion div (using innerHTML) to be the string you just created
 - Lastly, make the suggestion div element visible *
- ◆ Complete the *doMouseClick(val)* function:
 - First get the suggestion div, and make it invisible *
 - Next, set the search textbox contents to the argument that is passed in (val) (the id of the textbox is "autoTBox")

Run the Application

- ◆ **Deploy** (right click on project, **Run As | Run on Server**)
 - You'll probably need to restart the server
- ◆ Here, we see the drop down with some choices
 - Notice the highlighting on the item the mouse is over
- ◆ Here, we see how the textfield is filled after we've selected an entry



Review Questions

- ◆ What is JSON, and how is it used?
- ◆ What is the format of JSON data based on, and what does it look like?
- ◆ How do you work with JSON using JavaScript?
- ◆ How do you work with JSON using Java?

Lesson Summary

- ◆ **JSON** (JavaScript Object Notation) is a lightweight data interchange format using text strings based on JavaScript
 - It uses the array/object syntax of representing data, e.g.:
`[{"x":0, "y":1}, {"x":1, "y":2}]`
- ◆ The reference implementation for JSON gives you JavaScript tools to work with it
 - ***JSON.stringify()*** is provided to create JSON from primitive types, arrays, and objects
 - ***JSON.parse()*** is provided to parse JSON text
- ◆ **json-lib** provides ways to work with JSON from Java
 - ***JSONObject***, ***JSONArray***, and ***JSONSerializer*** provide easy ways to create JSON strings for various Java types

Session 9: XML and Ajax

XML Overview

Working with XML

Autocomplete Example Using XML

XML Versus JSON

Lesson Objectives

- ◆ Understand what XML is, and its role in Ajax
- ◆ Learn how to use XML in Ajax programs on both the client and server side
- ◆ Understand how JSON and XML compare as a data interchange format

XML Overview

XML Overview

Working with XML

Autocomplete Example Using XML

XML Versus JSON

What is XML?

- ◆ XML stands for **eXtensible Markup Language**
- ◆ XML is a **meta-markup language** for representing **data**
 - XML **documents** therefore contain markup and data -- the markup is generally in the form of **tags**
 - Both markup and data are in plain text
- ◆ XML is **extensible** because there is no predefined set of tags
 - Instead, you create your own tags to represent your own data
 - This is why we call it a meta-markup language instead of a markup language (like HTML, which has a fixed set of tags)
- ◆ XML is **platform-neutral**, standardized, and widely adopted as a mechanism for representing data
 - There is a set of rules that apply to XML documents
 - Since the data format is standardized, heterogeneous systems can exchange XML data without knowing anything about each other
 - We thus consider XML documents to be **portable**

Benefits of XML - Example XML Document

- ◆ Without knowing anything about XML, you can probably figure out what the XML document below is about

```
<?xml version='1.0'?>

<customer ID='67183625'>
  <name>    Leanne Ross    </name>
  <street>   1475 Cedar Avenue </street>
  <city>     Fargo        </city>
  <state>    ND           </state>
  <zipcode>  58103        </zipcode>
</customer>
```

- ◆ XML documents describe the data that they contain
 - To contrast, what are the following flat files describing?

```
67183625,Leanne Ross,1475 Cedar Avenue,Fargo,ND,58103
```

```
CD516/90125/Yes/1983-10-16/11.97/11.97
```

The Underlying Theme of XML

- ◆ XML's main reason for existence is to **represent data**
 - In a portable way
- ◆ This “block of data” can be retrieved, modified, stored, etc.
 - By heterogeneous applications -- Application A can generate XML and Application B can read it
 - And Applications A and B don't even know about each other
- ◆ Or it can be exchanged between systems, across a network
 - Systems that are written in different languages, running on different platforms, and are not coupled to one another
 - This **ubiquitous data interchange format** is probably the most important benefit that XML provides
- ◆ We will look at how XML works with Ajax for data interchange
 - This is a very small piece of XML functionality
 - We will not delve into any of the other uses/issues of XML

JavaTunes Purchase Order Document - Body

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>

<!-- root or document element - order - contains the entire body -->
<order ID='67183625' dateTime='2001-10-03 09:50'>
  <customer>
    <name>Leanne Ross</name>
    <street>1475 Cedar Avenue</street>
    <city>Fargo</city>
    <state>ND</state>
    <zipcode>58103</zipcode>
    <shipper name='FedEx' accountNum='893-192' />
  </customer>
  <item ID='CD509'>
    <name>Surfacing</name>
    <artist>Sarah McLachlin</artist>
    <releaseDate>1997-12-04</releaseDate>
    <listPrice>17.97</listPrice>
    <price>13.99</price>
  </item>
</order>
```

Diagram annotations:

- attribute**: Points to the `dateTime` attribute of the `order` element.
- comment**: Points to the `<!-- root or document element - order - contains the entire body -->` comment.
- element**: Points to the `<item ID='CD509'>` element.
- text node**: Points to the text value `13.99` inside the `<price>` element.

The Document Body and Elements

- ◆ The body is the “main” part of the document, and is required
 - The body is contained entirely within the **document element**, also called the **root element**
 - In our JavaTunes purchase order, that is the **order** element
 - The XML Declaration at the top declares this to be an XML doc
- ◆ The fundamental component of the body is the **element**
- ◆ Elements contain the document’s **content**
 - This content is usually **character data** or **child elements**

```
<!-- the name element contains character data -->
<name>Leanne Ross</name>

<!-- the person element contains child elements -->
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

Attributes

```
<!-- attributes in a start-tag -->
<person ssn='987-65-4321' gender='F'>
  <name>Leanne Ross</name>
  <age>25</age>
</person>

<!-- attributes in an empty-element-tag -->
<shipper name='FedEx' accountNum='893-192'/>
```

- ◆ An attribute is additional information **about** or **associated with** an element
 - Attributes can appear in start-tags or empty-element-tags
 - **Attributes are considered to be markup**, since they are defined to be information **about** an element, not part of the **content** of an element
- ◆ Attribute values must be quoted with either single- or double-quotes
- ◆ An element cannot have two attributes with the same name
- ◆ Attributes can appear on an element in any order

Working with XML

XML Overview

Working with XML

Autocomplete Example Using XML

XML Versus JSON

Working with XML and Ajax

- ◆ We will focus on working with XML in the context of Ajax requests, and will look at two common scenarios
 - Working with XML documents on the client
 - Generating XML documents on the server
 - At a high level, it's fairly straightforward
- ◆ On the client, the mechanics of making an Ajax request are the same as we've seen previously
 - However, the **responseXML** property of *XMLHttpRequest* is used to access the response data, instead of *responseText*
 - Working with the XML document is somewhat complicated
- ◆ On the server, most of the boilerplate code is similar to generating JSON
 - A servlet can receive the request and generate the response
 - An XML document is generated, rather than JSON text

Accessing XML With Ajax

- ◆ Below, we show how you can access the XML response data
 - This example is based on our earlier module to do Ajax calls
 - Note the access to the XML document using **req.responseXML**
 - This version of *makeXMLRequest* works with a callback function that does the actual processing of the XML document

```
XHR.makeXMLRequest = function (arg) {  
    var req = XHR.create();  
    if (req) {  
        req.open(arg.method, arg.ajaxURL);  
  
        req.onreadystatechange = function() {  
            if (req.readyState == 4 &&  
                req.status == 200) {  
                arg.callback(req.responseXML);  
            }  
        }  
    }  
    // Remainder of function not shown ...  
}  
}
```

Working With XML Documents

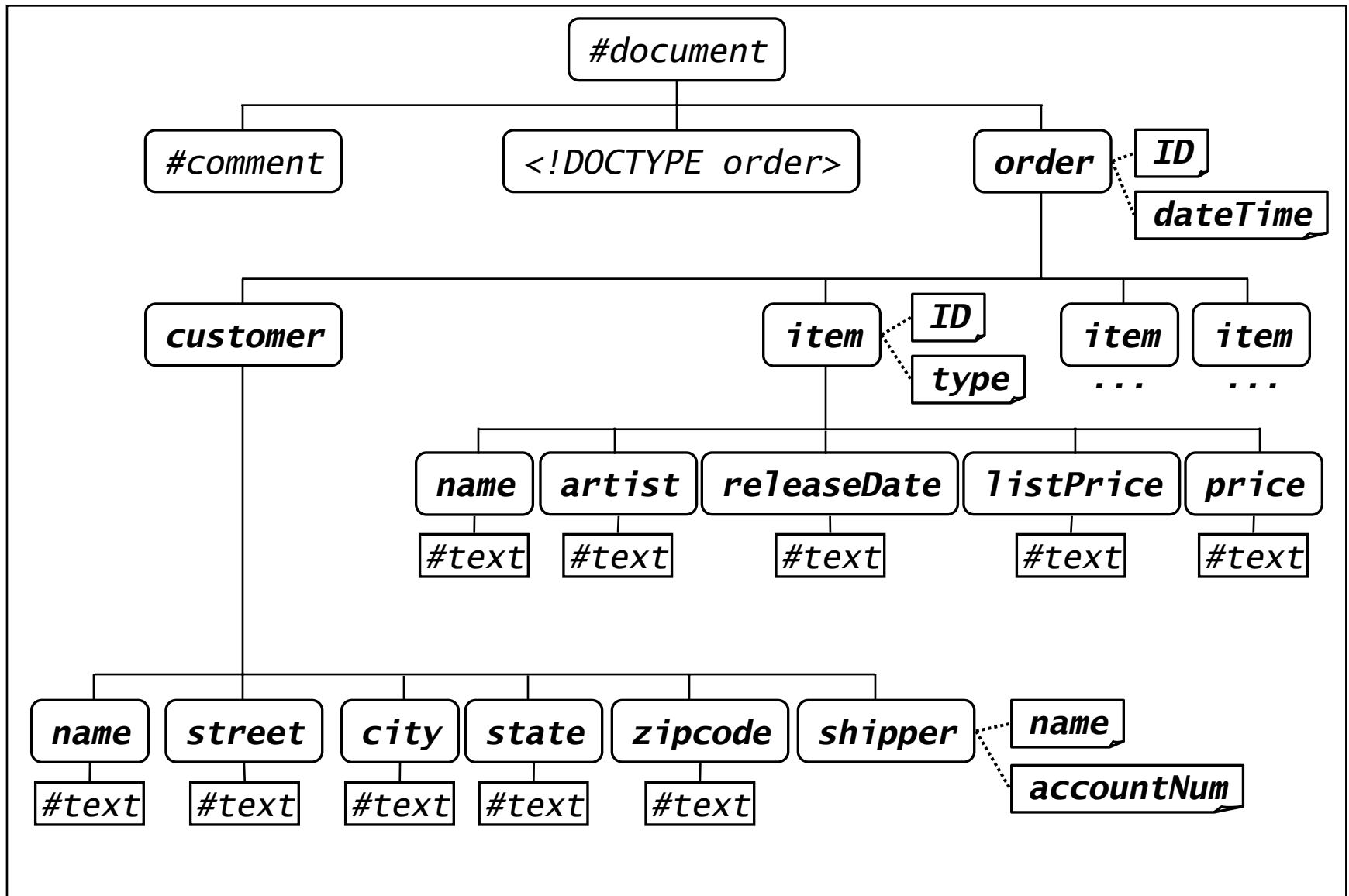
- ◆ Working with XML documents from JavaScript requires some knowledge of the W3C **Document Object Model (DOM)**
 - This is a logical model and API for representing and accessing structured documents
- ◆ The DOM organizes a document into a tree of **nodes**
 - **Everything** in the XML document is a node in the tree
 - The tree includes elements and attributes, as well as comments, the document type declaration (`<!DOCTYPE>`), and so on
- ◆ Each node has a **type**, a **name**, and a **value**
 - The following pages show a sample JavaTunes order document , and the resulting DOM tree showing nodes and their names
 - Some nodes don't have explicit names, so they are given pseudo-names, e.g., `#document`, `#comment`, and `#text`

JavaTunes Purchase Order Document

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<!-- JavaTunes order XML document -->
<!DOCTYPE order SYSTEM '... '>

<order ID='67183625' dateTime='2001-10-03 09:50'>
  <customer>
    <name>Leanne Ross</name>
    <street>1475 Cedar Avenue</street>
    <city>Fargo</city>
    <state>ND</state>      <!-- must use abbreviation -->
    <zipcode>58103</zipcode>
    <shipper name='FedEx' accountNum='893-192' />
  </customer>
  <item ID='CD509'>
    <name>Surfacing</name>
    <artist>Sarah McLachlin</artist>
    <releaseDate>1997-12-04</releaseDate>
    <listPrice>17.97</listPrice>
    <price>13.99</price>
  </item>
  <!-- Additional item nodes ... -->
</order>
```

JavaTunes Order as a DOM Tree



More About the W3C DOM

- ◆ In the DOM, everything in an XML document is a **node**
 - The entire document is represented with a **document node**
 - Every tag is an **element node**
 - The texts contained in the elements are **text nodes**
 - Every attribute is an **attribute node**
 - Comments are **comment nodes**
- ◆ Nodes have a **hierarchical relationship** to each other.
 - Every node, except for the document node, has a **parent** node
 - Most element nodes have **child** nodes
 - Nodes are **siblings** when they share a parent – e.g. the customer and item nodes share the order parent node
 - You can traverse a document by traveling along these relationships

Traversing a Document with JavaScript

- ◆ There are a number of element properties that allow you to traverse the document when using JavaScript
 - **childNodes**: All child nodes of an element
 - **parentNode**: Parent node of the element
 - **firstChild**: First child node of an element (often useful to get the text content of an element)
 - **lastChild**: Last child node of an element
 - **nextSibling**: Sibling node immediately following a node
 - **previousSibling**: Sibling node immediately preceding a node
 - **documentElement**: Available on document node only, and gives the root **element** of the XML document
- ◆ The following code accesses the `<customer>` element of our sample doc, (given a var `xmlDoc` holding the document node)

```
xmlDoc.documentElement.childNodes[1]
```

Getting Node Information

- ◆ Nodes have properties containing information about the node
 - **nodeName**: Name of the node
 - tag name for an element, attribute name for an attribute node, #text for a text node, #document for the document node
 - **nodeValue**: Value of the node
 - Contains the text for text nodes, contains the attribute value for attribute nodes, not available on document and element nodes
 - **nodeType**: The type of a node, with the values shown below
 - Element: 1, Attribute: 2, Text: 3, Comment: 8, Document: 9
- ◆ For example, to get the customer name in our sample document, you could use the following expression

xmlDoc.documentElement.childNodes[1].childNodes[1].firstChild.nodeValue

Finding Nodes in a Document

- ◆ The DOM includes the ability to find nodes by name also
 - The *findElementsByTagName* method returns all the nodes with a given tag name that are descendants of the invoking node
 - For example, the following code returns an array containing the single customer element in the purchase order doc

```
var customers = xmlDoc.getElementsByTagName("customer");
```

- If we wanted to get the customers name from our sample document, using the above array, we could do the following

```
var nameNodes = customers[0].getElementsByTagName("name");  
var name = nameNodes[0].firstChild.nodeValue;
```

- Note that the following will get ALL the name elements in the doc, including those under both *<customer>* and *<item>*

```
var names = xmlDoc.getElementsByTagName("name");
```

White Space Handling and Other Issues

- ◆ White space such as blank characters and newlines between elements is significant in the structure of an XML document
 - White space in the document becomes text nodes in the DOM tree
 - When navigating the document, you need to take these potential white space nodes into account
 - We've seen this in the access of the customer name via the expression:
`xmlDoc.documentElement.childNodes[1].childNodes[1].firstChild.nodeValue`
 - We use `childNodes[1]`, rather than `childNodes[0]`, because the first node is a text node representing white space
 - White space handling may vary in different browsers
- ◆ As you can see, extracting information from XML documents is somewhat complex and tedious
 - It requires dealing with low level details of the document structure itself

- Creating XML Documents on the Server -

- ◆ Generating XML on the server is not that difficult, and there are many ways to do it
 - For example, you can use a technology like JAXB (Java Architecture for XML Binding) to bind XML documents to Java objects
 - You can also use special purpose tools like the XML support in some databases
- ◆ We will look at one simple way to generate XML using JSP
 - This uses the same basic ideas used to generate HTML in a JSP
 - However, we generate XML elements instead of HTML elements
 - This is one easy way to generate XML documents in response to HTTP requests
 - We'll use the same fruit example we used for JSON

Producing XML With a Servlet and JSP

- ◆ The example below shows a servlet that creates a collection of fruits based on the input, puts the collection on the request, and then forwards to a JSP

```
public class XMLServlet extends HttpServlet {  
  
    public void doPost(HttpServletRequest req,  
        HttpServletResponse res) throws ServletException, IOException {  
  
        String fruitString = request.getParameter("fruit");  
        List fruitList = // Initialized somehow - not shown in example  
        // Set up request attribute  
        request.setAttribute("suggestions", fruitList);  
        // Forward to /jsp/xmlData.jsp for display  
        this.getServletContext().  
            getRequestDispatcher("/jsp/xmlData.jsp").  
            forward(request, response);  
    }  
}
```

The JSP Generating the XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<%@ page contentType="text/xml"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:choose>
  <c:when test="\${empty suggestions}">
    <fruits></fruits>
  </c:when>
  <c:otherwise>
    <fruits>
      <c:forEach items="\${suggestions}" var="fruit">
        <fruit>
          <id><c:out value="\${fruit.id}"/></id>
          <name><c:out value="\${fruit.name}"/></name>
          <price><c:out value="\${fruit.price}"/></price>
        </fruit>
      </c:forEach>
    </fruits>
  </c:otherwise>
</c:choose>
```


The JSP Generating XML

- ◆ In the JSP on the previous page, there are a few important things to note
 - Note the **XML declaration** at the top. This should be the first thing in the document
 - Note the **page declaration** with the content type set to **text/xml**
 - This lets the JSP container know that the response is XML, and not HTML
- ◆ The rest of the JSP page uses the "*suggestions*" list put on the request by the servlet
 - It first generates a `<fruits>` element as the root element
 - It then goes through the completion list, and generates `<fruit>` elements for each fruit in the list
 - Each fruit element has subelements with data for that item
- ◆ That's all there is to it – we'll show the generated XML later

Autocomplete Example Using XML

XML Overview

Working with XML

Autocomplete Example Using XML

XML Versus JSON

An Autocomplete Example With XML

- ◆ Let's take a look at an example that will use an XML document created with a servlet/JSP to generate an autocomplete suggestion box in the client
 - This will be based on the previous fruit example
- ◆ There are a number of parts to this:
 - **An input field** in an HTML page that has which will include autocomplete functionality, including making an Ajax request on each keypress for autocomplete suggestions
 - **A servlet** which works with a **JSP** to generate an **XML document** text based on the current input in the input field
 - **JavaScript code** which takes the xml document from the servlet, and generates and displays the autocomplete suggestions
- ◆ We'll look at each of these individually

An Input Field Generating Ajax Requests

- ◆ In the example below, we show an input field that generates Ajax requests which return an XML document as the response
 - Note the **onkeyup** event handler for the input box – this will trigger the Ajax request when a character is entered
 - We use *XHR.makeXMLRequest* to make an Ajax request, and pass in a callback function expecting an XML document as an argument (the XML is extracted from the response via the **xmlResponse** property *)
 - The callback function calls another function, **doCompletion()**, passing in as arguments the XML document from the response, and the autocomplete div to be filled in by *doCompletion*

```
<input type="text" id="autoTBox"
      onkeyup='XHR.makeXMLRequest( {
        method: "POST",
        ajaxURL: "/javatunes/xmlData", postArg: "fruit="+this.value,
        callback: function(xmlDoc) {
          doCompletion(xmlDoc ) } } )'; />
<div id="suggestDiv" style="display:none"
    class="autocomplete"> </div>
```

XML Document From Servlet/JSP

- ◆ The XML document generated from our previous Servlet/JSP example will have the form shown in the sample below
 - Our JavaScript will have to traverse it to get our completion data

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<fruits>
  <fruit>
    <id>1</id>
    <name>apples</name>
    <price>2.99</price>
  </fruit>
  <fruit>
    <id>2</id>
    <name>oranges</name>
    <price>0.99</price>
  </fruit>
</fruits>
```

JavaScript Code Constructing the Suggestions

- ◆ **doCompletion()** creates the suggestions by extracting needed information from the XML document, and creating a `` from it
 - For each fruit, it uses the fruit name to create an `` of the form :
`<li class="" onmousedown="doMouseClicked('apples');" onmouseout="this.className="" onmouseover="this.className='selected'">apples`

```
<script>
function doCompletion(xmlDoc) {
    var el = document.getElementById("suggestDiv");
    var txt = "<ul>"
    var names = xmlDoc.getElementsByTagName("name"); // Get fruit name elements
    for (var cur=0; cur<names.length; cur++) {
        txt += '<li ' + 'onmouseover="this.className=\'selected\';" '
        + 'onmouseout="this.className=\'\';" '
+ 'onmousedown="doMouseClicked(\' ' + names[cur].firstChild.nodeValue + '\');" '
        + '>' // end <li> tag
        + names[cur].firstChild.nodeValue + '</li>';
    }
    txt += "</ul>";
    el.innerHTML = txt;
    el.style.display="";
}
}
```

Accessing the XML Nodes We Want

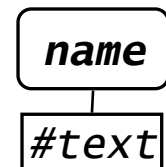
- ◆ In the previous JavaScript code, there are two key expressions that work with the XML document
- ◆ We first get an array of name elements (subelements of the fruit element) from the passed in XML doc with this statement

```
var names = xmlDoc.getElementsByTagName("name");
```

- ◆ We access the text node of each name element and get its value with the following expression

```
names[cur].firstChild.nodeValue
```

- Looking at the node tree for a name element, we can see that the first child is the text node, and its value contains the text, so the expression above gives us the text of the name element - which is what we want



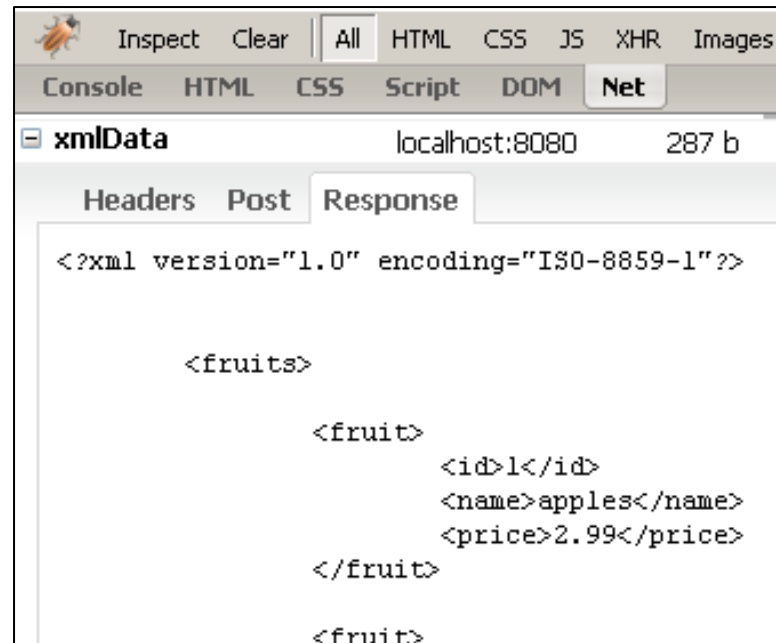
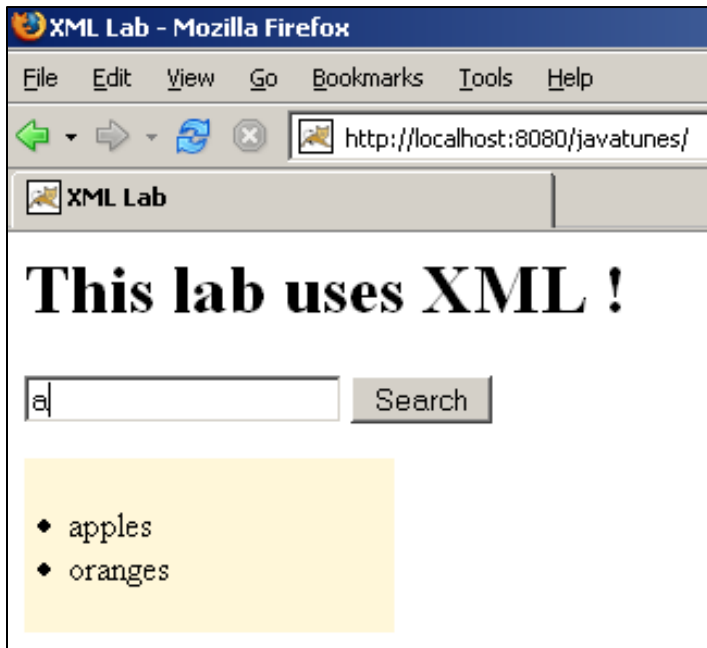
Using the Suggestions

- ◆ The *doMouseClick()* function (shown below) is called in response to a mouse click on an item in the suggestion list
 - It receives the text of the `` as an argument
 - It makes the suggestion list invisible (by setting the style *display:none*)
 - It sets the text of the input field to be the text from the ``
- ◆ That's it – you've looked at all the parts
 - Let's see how it runs

```
function doMouseClick(val){  
    document.getElementById("suggestDiv").style.display="none";  
    //update text box with selected item  
    var autoTBox = document.getElementById("autoTBox");  
    autoTBox.value = val;  
}  
</script>
```


Autocomplete at work

- ◆ Below, we show the suggest list, some of the XML returned from the servlet, and the `` generated from the XML



```
<ul>
  <li onmousedown="doMouseClicked('apples');" onmouseout="this.className='';" onmouseover="this.className='selected';"> apples </li>
  <li onmousedown="doMouseClicked('oranges');" onmouseout="this.className='';" onmouseover="this.className='selected';"> oranges </li>
</ul>
```

XML Versus JSON

XML Overview
Working with XML
Autocomplete Example Using XML
XML Versus JSON

XML and JSON for Data Interchange

- ◆ We've seen that both XML and JSON can be used for data interchange when working with Ajax
 - Let's examine the strengths and weaknesses of each
- ◆ **JSON Pros:**
 - It is a lightweight, **easy to use** format, understandable by people
 - It is designed for exchanging data
 - It is fairly easy to create on the server
 - It is extremely easy to use in browsers, because you manipulate JavaScript arrays and objects which are basically type business objects
 - Browsers can parse it very quickly and cheaply
 - Working with JSON in JavaScript is very natural, easy, and usually fairly straightforward – navigating objects generated from JSON is exactly like navigating any other JavaScript object
 - JSON supports Unicode

XML and JSON for Data Interchange

◆ JSON Cons:

- JSON use is not as widespread as XML
- More limited than XML (e.g. , no "JSON Schema Language")
- Doesn't have a formal date type (yet)

◆ XML Pros:

- Already used in many environments
- There is a lot of capability and flexibility
 - XML Schema, XPATH, XSLT

◆ XML Cons:

- Browsers don't implement XML standards uniformly
- Browser based XML implementations are generally fairly slow
 - Significant overhead in parsing incoming XML and creating a DOM tree. The larger the data, the more relevant this becomes.

Continued →

XML and JSON for Data Interchange

◆ XML Cons (continued):

- An XML document is more complex and verbose than JSON
- The DOM tree basically gives you untyped data, unless you use a schema and some sort of databinding tool
- Working with the XML DOM tree in JavaScript is reasonably complex, and not very natural
 - Traversing / navigating the DOM tree can be a pain in the neck
- Working with higher level technologies is not easy
 - XSLT, for example, has varied support among different browsers
- Really designed for creating documents, not exchanging data

◆ Don't forget HTML - If your needs are simple enough, passing HTML from the server may work fine

Summary

- ◆ It seems clear that for browser-server communication, JSON is an excellent, easy to use format
 - It is usually significantly easier to use than XML
 - It's wonderful for serializing & transferring simple data structures
 - For wider uses, it may be too limited or too immature
- ◆ XML may be a useful choice for you if you have other considerations or needs
 - For example, if you've already invested heavily in using XML in your system
 - If the rest of your system is using XML, and you have a set of XML tools (on browser and server), then XML may be better for you
 - Another possible scenario is when you are using the data for more than browser-server communication
 - If you are storing or sharing the data, then XML may be a better format because it is widely understood

Lab 9.1 – Generating and Using XML Data

- ◆ **Overview:** In this lab you will once again create your own (very simple) autocomplete textbox
 - This time, the autocomplete functionality will use XML data from a servlet to generate the autocomplete list
 - This is very similar to what you did in the JSON lab
 - This lab is also fairly complex
- ◆ **Objectives:**
 - Gain experience using XML data with JavaScript
- ◆ **Builds on previous labs:** none
- ◆ **Approximate Time:** 20-30 minutes

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab09.1

- This is a new lab directory

Tasks to Perform

- ◆ Right click on the server in Servers View
 - Select **Add and Remove Projects**, and **remove** Lab08.1
- ◆ Create a new **Dynamic Web Project** called **Lab09.1** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server is selected
 - Make sure **2.5 module version** is selected
 - Remember to set the context root to **javatunes**

- ◆ You'll work with three files in **Lab09.1\WebContent**
 - **hello_ajax.html**: An HTML file that will have an autocomplete suggestion list created from an XML document from a servlet
 - **jsp/xmlData.jsp**: Works with the servlet to generate XML
 - **js/ajax.js**: Makes the actual Ajax call, extracts the XML data
- ◆ The lab is based on our JavaTunes application
 - It provides suggestions for a search input textfield (as for the JSON lab)
 - **XMLServlet.java** receives an XHR request (that includes the current input in the search field), and puts a collection of *MusicItem* on the request scope with the name "suggestions"
 - The servlet then forwards to *xmlData.jsp* to generate an xml doc
 - This servlet is **given to you**, and you don't need to change it

- ◆ You will want `xmlData.jsp` to use the servlet data, and generate an XML document of the form shown below

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<items>
  <item>
    <id>501</id>
    <title>Diva</title>
    <artist>Annie Lennox</artist>
  </item>
  <item>
    <id>503</id>
    <title>Trouble is...</title>
    <artist>Kenny Wayne Shepherd Band</artist>
  </item>
  <item>
    <id>504</id>
    <title>Lie to Me</title>
    <artist>Jonny Lang</artist>
  </item>
</items>
```

Tasks to Perform

- ◆ Open the file `WebContent\jsp\xmlData.jsp` * and do the following
 - Look for the **TODOs** in the file
 - Finish the `<% page %>` element by setting its *contentType* attribute to *text/XML*
 - Complete the `<forEach ...>` so that you are using *"suggestions"* as the collection, and a variable named *item*
 - Within the `<forEach>` create an `<item> ... </item>` element
 - Within this element, include the following elements
 - `<id>` containing the id property of the current item
 - `<title>` containing the title property of the current item
 - `<artist>` containing the artist property of the current item
 - Use `<c:out>` to access the properties of the item

Tasks to Perform

- ◆ Open the file `WebContent\js\ajax.js` and do the following
 - Look at the `XHR.makeXMLRequest()` function, and go to the line calling the callback function `arg.callback(TODO)`;
 - We want to pass the XML doc to the callback, so add the request's **responseXML** property as an argument to the callback
 - That's all that needs to be done in `ajax.js`
- ◆ Open the file `WebContent\hello_ajax.html` for editing
 - Find the form that we'll generate suggestions for, and look at the **onkeyup** event handler – very similar to the JSON lab's

```
onkeyup='XHR.makeRequest( {method: "POST", ajaxURL: "/javatunes/xmlData",
    postArg: "artist="+ this.value,
    callback: function(xmlDoc) { TODO }
} )'; />
<input type='submit' name='Submit' value='Search' />
</form>

<div id="suggestDiv" style="width:170px; visibility:hidden;
background:#fff7d9" class="autocomplete"> </div>
```

Tasks to Perform

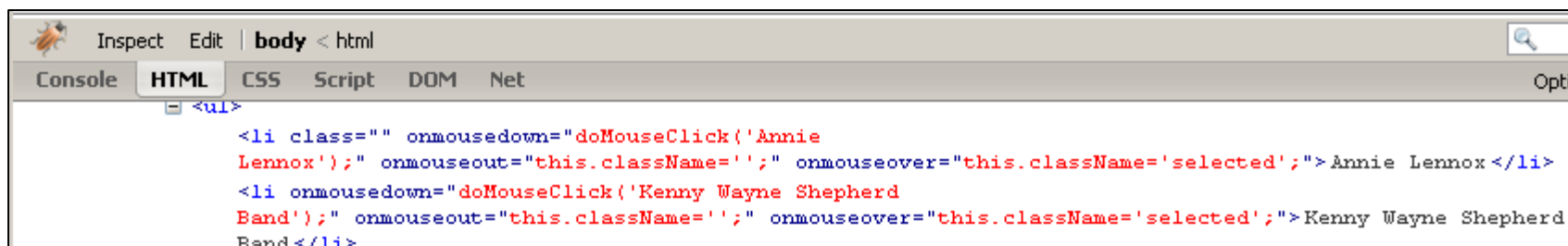
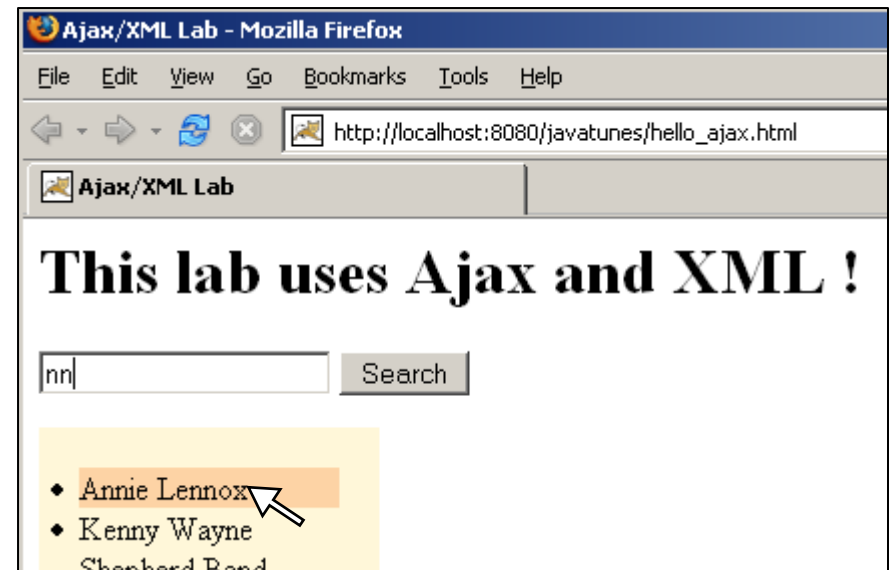
- ◆ In the **onkeyup** event handler, finish the callback function by adding code which does the following:
 - Invokes the **doCompletion()** function with 1 argument
 - The argument is just the XML doc passed to the callback function – You don't need to do any processing on it
 - That's all that needs to be done in the event handler
- ◆ Next, look for the **doCompletion(xmlDoc)** function *
 - This function works much the same way as in the JSON lab
 - It generates the `` list with the suggestions for completing the search expression entered into the input textbox
 - It is passed an **XML document** (as opposed to a JSON object)

Tasks to Perform

- ◆ Look for the TODOs in the *doCompletion()* function
 - Find the following line of code, and initialize the artists var with all the artist elements in the document
var artists = TODO:
 - Use *document.getElementsByTagName()* to do this
- ◆ Find the code shown below within the *for* loop, and replace the TODOs with the text value of the current artist element
 - + *'onmousedown="doMouseClicked(\' ' + TODO + '\');*
 - + *'>' // end tag*
 - + *TODO + '';*
 - *artists[cur]* will access the current artist element
 - Look at the fruit example in the manual for accessing the text

Run the Application

- ◆ **Deploy** (right click on project, **Run As | Run on Server**, restart server)
 - Below, we show the XML doc in the response, the resulting drop down display, and the generated HTML for the suggestions
 - If you have problems, check all of these using FireBug



STOP

Review Questions

- ◆ What is XML, and what is it used for?
- ◆ How do you access XML data from XMLHttpRequest?
- ◆ How do you work with XML data using JavaScript?
- ◆ How does XML compare to JSON in terms of Ajax?

Lesson Summary

- ◆ XML is a **meta-markup language** for representing **data**
 - XML is **platform-neutral**, standardized, and widely adopted to exchange data between different parts of a system
- ◆ **XHR.responseXML** can be used to access an XML response
 - This provides a DOM tree representing the XML document
- ◆ You can access XML elements in a document using **getElementsByTagName()**
 - This returns an array of elements with the given name – often the easiest way to get what you need from an XML document
- ◆ The W3C **Document Object Model (DOM)** can be used to navigate directly from node to node in an XML document
 - This is often very useful, but somewhat cumbersome
- ◆ JavaScript based JSON is usually easier to use than XML
 - If you're already heavily using XML, then it may be better for you

Session 10: DWR (Direct Web Remoting) and Other Technologies

DWR Overview
Working with DWR
Other Technologies

Lesson Objectives

- ◆ Understand what DWR (Direct Web Remoting) is, and how it simplifies Ajax programming
- ◆ Learn the basics of DWR
- ◆ Use DWR to make Ajax requests to the server

DWR Overview

DWR Overview

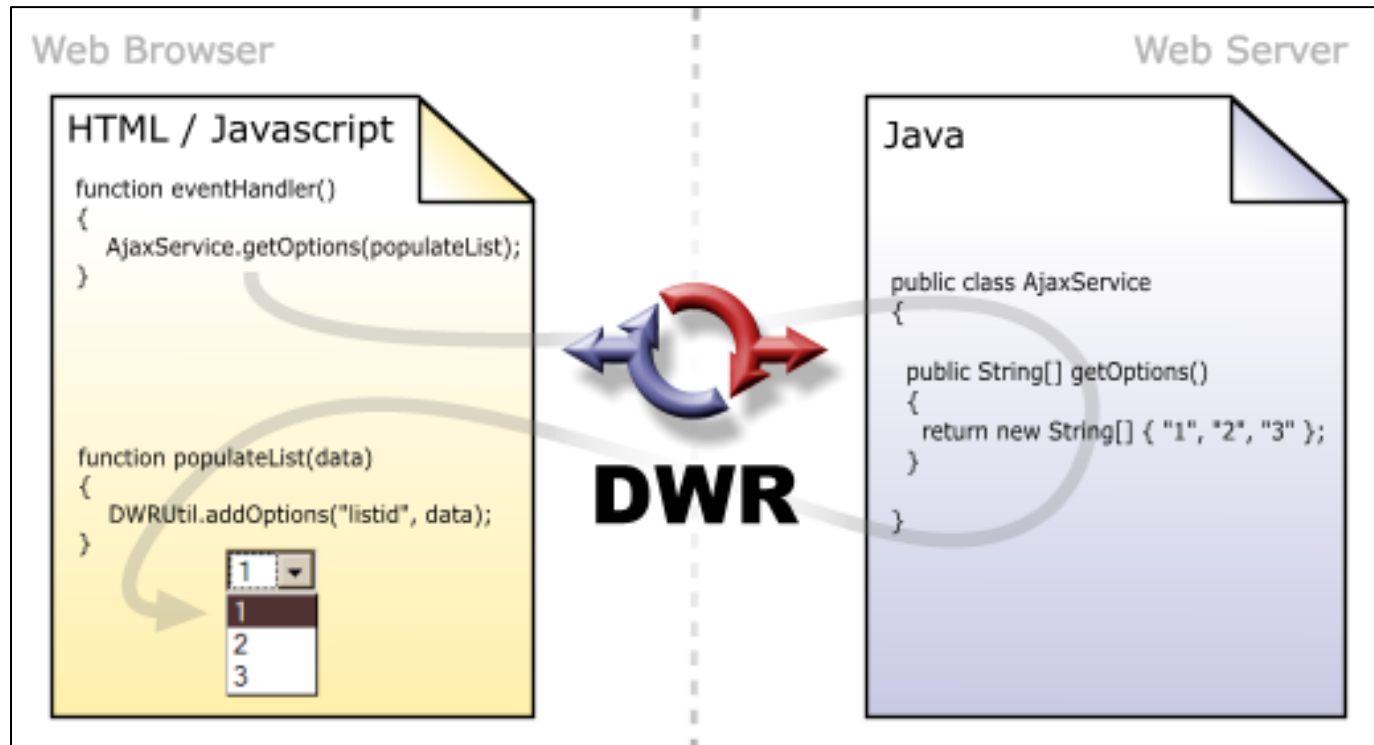
Working with DWR
Other Technologies

What is DWR?

- ◆ **DWR** (Direct Web Remoting) - open source technology that lets you invoke server side Java functions from JavaScript in the browser - as if the Java code was running in the browser
 - Available from <http://getahead.org/dwr/>
- ◆ DWR consists of two core parts
 - A **Java servlet** that processes requests and returns responses
 - **JavaScript** in the browser that sends requests and can dynamically update web pages
- ◆ DWR **dynamically** generates JavaScript proxies for your Java classes, allowing you to invoke them on the browser
 - **dwr.xml**, the DWR configuration file, is used to configure the generation of these proxies
 - Your Java code is executed on the server, and DWR transparently marshals the request/response back and forth

How DWR Works

- ◆ In the diagram below, we see how DWR is used to get an array of Strings from the server, which is used to populate a selection list
 - The *AjaxService* JavaScript class is generated by DWR from your Java class, and all the remoting details are handled by DWR
 - You just call the method as if it was native JavaScript, and supply a callback function, in this example the function *populateList()*



Getting Started with DWR

- ◆ Very few steps are needed to start working with DWR
- ◆ Place *dwr.jar* into the WEB-INF/lib dir of your Web app
 - This is required for the server side functionality
- ◆ Edit *web.xml* to register the DWR servlet
- ◆ Create a *WEB-INF\dwr.xml* file to configure the classes that will be exposed remotely
- ◆ Test the functionality by browsing to [http://\[YOUR-WEBSERVER\]/\[YOUR-WEBAPP\]/dwr](http://[YOUR-WEBSERVER]/[YOUR-WEBAPP]/dwr)
 - This will bring up a web page that lists all the classes you configured in *dwr.xml*
 - If you follow the link to a specific class, you will come to a page that allows you to invoke the methods of that class
 - These example pages are dynamically generated by DWR

web.xml Configuration for DWR

◆ Sample web.xml for using DWR

- It configures your web app to forward any urls with **/dwr** in them to the DWR servlet

```
<!-- Servlet entry for DWR servlet -->
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <display-name>DWR Servlet</display-name>
  <servlet-class>
    org.directwebremoting.servlet.DwrServlet
  </servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>

<!-- Servlet mapping entry for DWR servlet -->
<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/dwr/*</url-pattern>
</servlet-mapping>
```

dwr.xml Configuration File

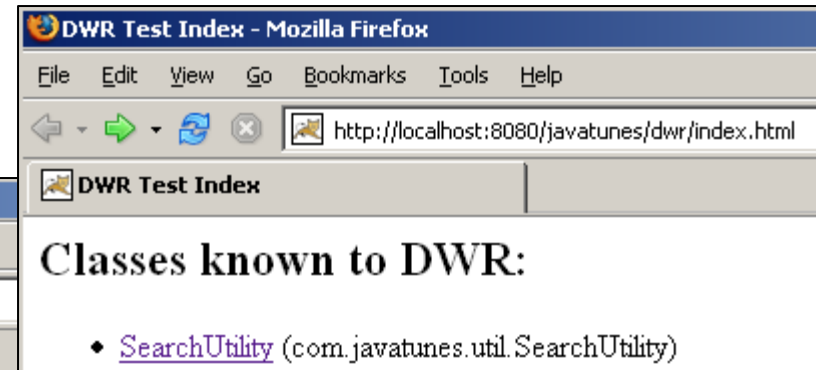
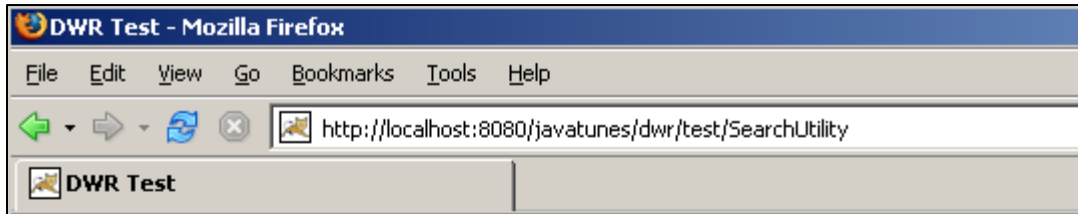
- ◆ *dwr.xml* is the configuration file for DWR – by default in WEB-INF
 - The example below includes some of the most common elements
 - **<allow>**: defines which classes DWR can create and convert
 - **<create>**: Needed by each class on which we execute methods
 - **<convert>**: Specifies conversions for request/response data
 - We've **allowed** the *SearchUtility* class and specified a standard JavaBean **converter** for the *MusicItem* class (classes from our labs)
- ◆ DWR automatically generates test pages for the allowed classes

```
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 1.0//EN"
    "http://www.getahead.ltd.uk/dwr/dwr10.dtd">

<dwr>
    <allow>
        <create creator="new" javascript="SearchUtility">
            <param name="class" value="com.javatunes.util.SearchUtility"/>
        </create>
        <convert converter="bean" match="com.javatunes.util.MusicItem"/>
    </allow>
</dwr>
```

Running the Test Pages - <webapp>/dwr

- ◆ Initial test page (right) and SearchUtility test page (below)



Methods For: SearchUtility (com.javatunes.util.SearchUtility)

To use this class in your javascript you will need the following script includes:

```
<script type='text/javascript' src='/javatunes/dwr/interface/SearchUtility.js'></script>
<script type='text/javascript' src='/javatunes/dwr/engine.js'></script>
```

In addition there is an optional utility script:

```
<script type='text/javascript' src='/javatunes/dwr/util.js'></script>
```

Replies from DWR are shown with a yellow background if they are simple or in an alert box otherwise.

The inputs are evaluated as Javascript so strings must be quoted before execution.

There are 12 declared methods:

- `findById(0);`
- `findByArtist("");`
- `findByKeyword("");`
- `hashCode();`

(Warning: hashCode() is excluded: Methods defined in java.lang.Object are not accessible. See [below](#))

- `getClass();`

Running The Test Pages

- ◆ For our javatunes web app, the DWR main test page is accessed at `http://localhost:8080/javatunes/dwr`
 - The main test page has a list of all the allowed classes
 - Clicking on one of them, brings up the test page for that class
 - You can then execute the methods and see the results, as shown below in invoking *`findById(501)`* on *`SearchUtility`*

The screenshot displays the DWR test page interface. On the left, a list of 12 declared methods is shown. The method `findById(501);` is selected and circled in red. Below the list, several warning messages are visible: `(Warning: hashCode() is excluded: Metho`, `(Warning: No Converter for java.lang.Cle`, and `(Warning: getClass() is excluded: Metho`. On the right, a modal window titled `http://localhost:8080` shows the JSON response for the `findById(501)` call. The response is a JSON object with the following fields: `artist` ("Annie Lennox"), `id` (501), `listPrice` (17.97), `price` (13.99), `releaseDate` (Sat Jan 04 1992 00:00:00 GMT-0500 (Eastern Standard Time)), and `title` ("Diva"). An "OK" button is located at the bottom of the modal window.

There are 12 declared methods:

- ◆ `findById(501);`
- ◆ `findByArtist("");`
- ◆ `findByKeyword("");`
- ◆ `hashCode();`
(Warning: hashCode() is excluded: Metho
- ◆ `getClass();`
(Warning: No Converter for java.lang.Cle
(Warning: getClass() is excluded: Metho
- ◆ `wait();`
(Warning: excluded methods are not recommended. See below)

Modal Window Title: `http://localhost:8080`

```
{
  artist: "Annie Lennox",
  id: 501,
  listPrice: 17.97,
  price: 13.99,
  releaseDate: Sat Jan 04 1992 00:00:00 GMT-0500 (Eastern Standard Time),
  title: "Diva"
}
```



Lab 10.1 – Using The DWR Test Pages

- ◆ **Overview:** In this lab you will install DWR and use the test pages
 - We will use our *SearchUtility* and *MusicItem* classes, and try invoking some of the SearchUtility functionality from the test page
- ◆ **Objectives:**
 - Gain experience using DWR
 - Use the DWR test pages to invoke server side Java code
- ◆ **Builds on previous labs:** none
- ◆ **Approximate Time:** 20-30 minutes

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab10.1

- This is a new lab directory

Tasks to Perform

- ◆ Right click on the server in Servers View
 - Select **Add and Remove Projects**, and **remove** Lab09.1
- ◆ Create a new **Dynamic Web Project** called **Lab10.1** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server is selected
 - Make sure **2.5 module version** is selected
 - Remember to set the context root to **javatunes**
 - You'll see some errors in *dwr.xml* which you can ignore, as we'll fix them as part of the lab

- ◆ You'll work with two files in **Lab10.1\WebContent\web.xml**
 - **web.xml**: To configure the DWR servlet
 - **dwr.xml**: To configure the DWR functionality that is available
- ◆ The lab is based on our JavaTunes application, and accesses some of the JavaTunes utility classes using DWR
 - **SearchUtility** functionality is exposed remotely
 - The **MusicItem** class is registered as a bean in *dwr.xml* so it can be used as the return result in *SearchUtility* remote calls

Tasks to Perform

- ◆ Open **web.xml**, and look for the TODO comments
 - Find the `<servlet>` entry for the *dwr-invoker* servlet, and fill in the **`<servlet-class>`** element with the DWR servlet name:
`org.directwebremoting.servlet.DwrServlet`
 - Find the `<servlet-mapping>` element for the *dwr-invoker* servlet, and fill in the **`<url-pattern>`** element with **`/dwr/*`**
 - /dwr is the URL that will signify all dwr calls

Tasks to Perform

- ◆ Open *dwr.xml* and look for the TODO comments
 - Finish the **<create>** element to declare a "new" creator, that defines a JavaScript class called *SearchUtility*
 - Finish the nested **<param>** element to specify that this will be applicable to the *com.javatunes.util.SearchUtility* class
 - Finish the **<convert>** element to specify a bean converter for class *com.javatunes.util.MusicItem*
- ◆ **Deploy** (right click on project, **Run As | Run on Server**, restart server)
- ◆ Browse to ***http://localhost:8080/javatunes/dwr*** - the general DWR test page which lists the classes that are allowed access via DWR
 - The only class should be *SearchUtility*, so click on its link to bring you to the *SearchUtility* DWR test page
- ◆ Try the *SearchUtility* test page, executing some of the methods
 - You can also view the source of this test page to see how they invoked the *SearchUtility* methods

STOP

Working with DWR

DWR Overview
Working with DWR
Other Technologies

Including the DWR JavaScript Code

- ◆ DWR generates JavaScript proxies for the classes that you configure in *dwr.xml*
 - You can use these classes in your HTML pages to invoke the Java code
- ◆ You need to include the dynamically generated JavaScript files to use the classes, as shown below for SearchUtility
 - **SearchUtility.js** is generated from your class based on *dwr.xml*
 - **engine.js** contains the underlying DWR library
 - **util.js** contains a set of JavaScript utility functions

```
<script type="text/javascript"
        src='dwr/interface/SearchUtility.js'></script>
<script type="text/javascript" src='dwr/engine.js'></script>
<script type="text/javascript" src='dwr/util.js'></script>
```

Using the DWR Proxies

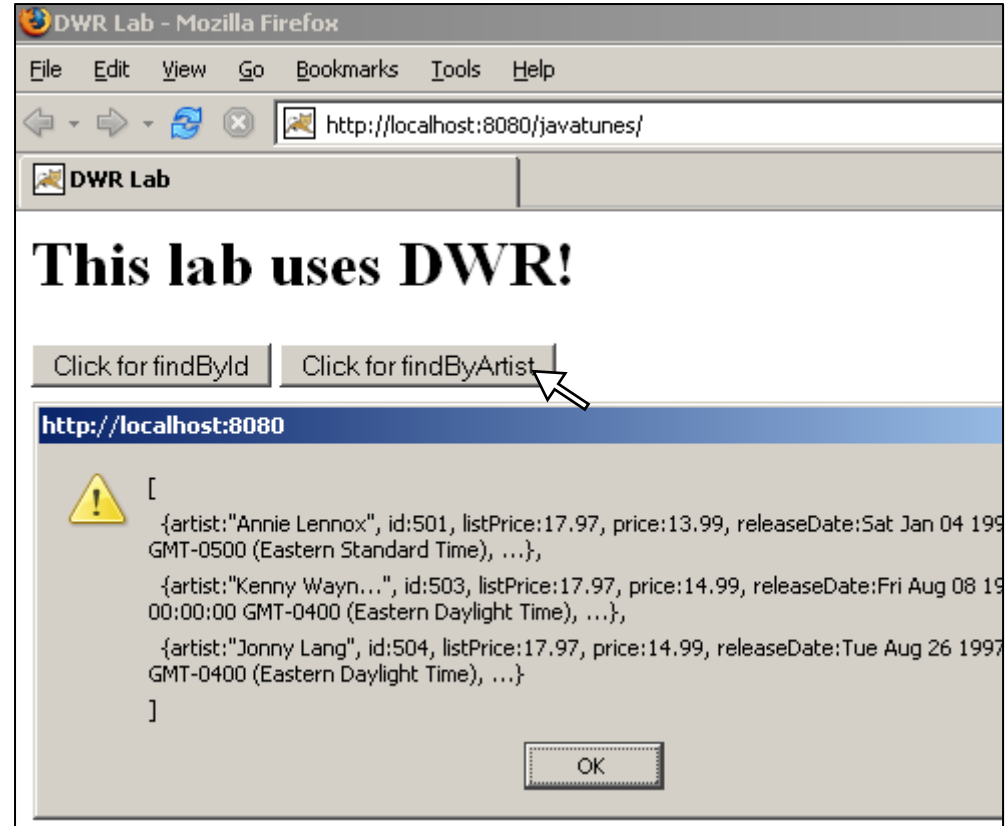
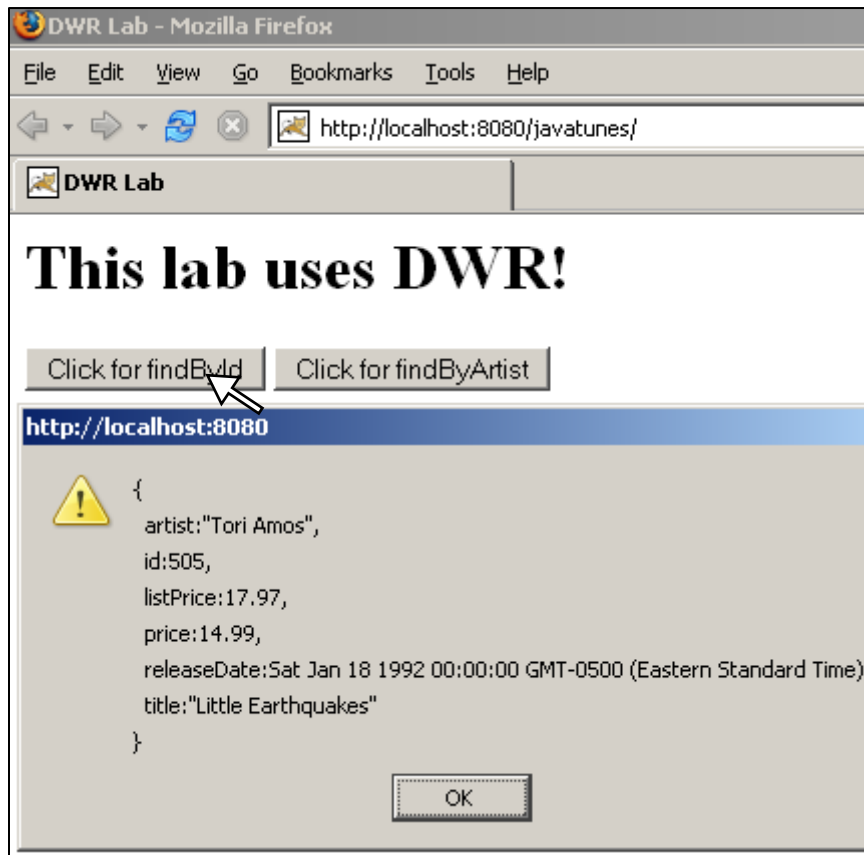
- ◆ Below, we show examples calling *SearchUtility* methods
 - Note the second argument to the methods, a callback function
 - This is added to the signature of each method, and is needed, because of the asynchronous nature of Ajax calls, for any function returning a response

```
<form>
  <input type="button" value="Click for findById"
    onclick='SearchUtility.findById(505, displayDWR)' />
  <input type="button" value="Click for findByArtist"
    onclick='SearchUtility.findByArtist("nn", displayDWR)' />
</form>

<script type="text/javascript">
  var displayDWR = function(data) {
    if (data != null && typeof data == 'object')
      alert(dwr.util.toDescriptiveString(data, 2));
  }
</script>
```

Using the DWR Proxies

- ◆ Below, we show the results of the two invocations on the previous slide
 - Searching for id=505, and for artist="aa"



Functions with Java Object Arguments

- ◆ Suppose you had a SearchUtility method (in Java) as follows:
void updateItem(MusicItem i)
 - It's simple to call this method from JavaScript – you just create a JavaScript object with the needed properties, as shown below
 - DWR will convert the object literal into your Java type
 - Fields missing in the JavaScript object will be unset in the Java

```
function updateItem() {  
    var personObj = {  
        id: 501,  
        title: "New Title",  
        artist: "Great artist",  
        releaseDate: new Date("1 January 2008"),  
        listPrice: 29.99,  
        price: 19.99  
    }  
    SearchUtility.update(personObj);  
}
```

DWR Options

- ◆ DWR has many options that you can set
 - These can be set globally, or on a per call basis as shown below
 - The options are set globally using the *DWREngine* object
 - They are set on a per call basis using a call meta-data object
 - In the per call example, we set the callback function, a timeout, and specify that the call should use IFrame, rather than XHR
- ◆ DWR also lets you batch calls together for efficiency

```
// Global Example
DWREngine.setTimeout(2000);

// Per call example
<input type="button" value="Click for findByArtist"
  onclick='SearchUtility.findByArtist("nn", {
    callback: displayDWR,
    timeout: 2000,
    rpcType: DWREngine.IFrame
  } );'/>
```

Reverse Ajax

- ◆ Allows you to send data from the server to the browser
 - New in DWR 2.0
- ◆ Three supported methods of pushing data to the browser
 - **Polling**: Browser makes request to server at fixed intervals
 - Simplest, most obvious
 - **Comet** (long lived http): Server answers the client request very, very slowly – in effect keeping the communications channel open
 - Closest to real server push
 - Can configure how long the channel is kept open
 - Uses connection resources on both client and server
 - **PiggyBack**: When a server has data to send, it waits for a new browser request, and includes additional data in the response
 - Also called passive mode
 - Uses least additional resources, but you don't control when you get response

Lab 10.2 – Using DWR

- ◆ **Overview:** In this lab you will invoke various *SearchUtility* methods using DWR and view the results
 - We will also look at a DWR implementation of the autocomplete functionality which uses the results of a remote call to *SearchUtility.findByArtist*, to populate the suggestion list
 - it's similar to the JSON example, except the Ajax call is simpler
 - We'll also look at the a sample reverse Ajax chat application
- ◆ **Objectives:**
 - Gain experience scripting with DWR
 - Look at a reverse Ajax chat application
- ◆ **Builds on previous labs:** 10.1
 - Continue working in your **Lab10.1** directory
- ◆ **Approximate Time:** 30-40 minutes

- ◆ In this lab, you'll be using DWR to fill in the functionality of the page pictured below

The screenshot shows a Mozilla Firefox browser window titled "DWR Lab - Mozilla Firefox". The address bar displays the URL "http://localhost:8080/javatunes/hello_ajax.html?". The browser's menu bar includes File, Edit, View, Go, Bookmarks, Tools, and Help. The page content features a header "DWR Lab" and a main heading "This lab uses DWR!". Below the heading, there are two input fields with corresponding buttons: "Click for findById" and "Click for findByArtist". Further down, there is a table with two columns: "ID" and "New Artist". The "ID" column contains an input field and a "Click for update" button. The "New Artist" column contains an input field. At the bottom, there is a search bar with a "Search" button.

ID	New Artist
<input type="text"/>	<input type="text"/>
<input type="button" value="Click for update"/>	

- ◆ Continue working in the **Lab10.1** directory, and open the file **WebContent\hello_ajax.html** which contains all our DWR code

Tasks to Perform

- ◆ Look for the TODOs, and finish the `<script>` tags to include the DWR client side JavaScript files containing the DWR functionality
 - **dwr/interface/SearchUtility.js, dwr/engine.js, dwr/util.js**
- ◆ In the `<body>` section, look for the button with `value="Click for findById"`, and **finish its `onclick` event handler**
 - This button appears after an input field for entering the id to find
 - The event handler should call `SearchUtility.findById`, and pass in two arguments, the first is the text typed into the id input field *, and the second is the callback function to handle the data
 - For the callback, use the `displayDWR` function we provide *
- ◆ **Restart** Tomcat, browse to `hello_ajax.html`, type in the id 501, click the **`findById`** button, and look at the results

Tasks to Perform

- ◆ Next, look for the button with *value="Click for findByArtist"*, and **finish its *onlick* event handler**
 - This button appears after an input field for entering the artist
 - The event handler should call *SearchUtility.findByArtist*, and pass in 2 arguments, the first is the text typed into the input field *
 - For the second argument, use the *displayDWR* function as before, but pass it in within a **JavaScript object that has a callback property** instead of passing the callback function directly *
- ◆ **Publish** the application, type in something to search by e.g. "nn", and click the button to see the result list of music items
- ◆ Next, look for the button with *value="Click for update"*, and **finish its *onlick* event handler**
 - This button appears after a table with two input fields for entering an id, and a new artist value (we don't bother with the other properties) *

- ◆ The update button calls *SearchUtility.update(MusicItem)*, and requires a *MusicItem* as an argument
 - The *MusicItem* should be constructed as an object literal, using the data from the input fields in the table *
 - We only fill in the artist property to save time in the lab, but you could make input fields for all the properties and use them

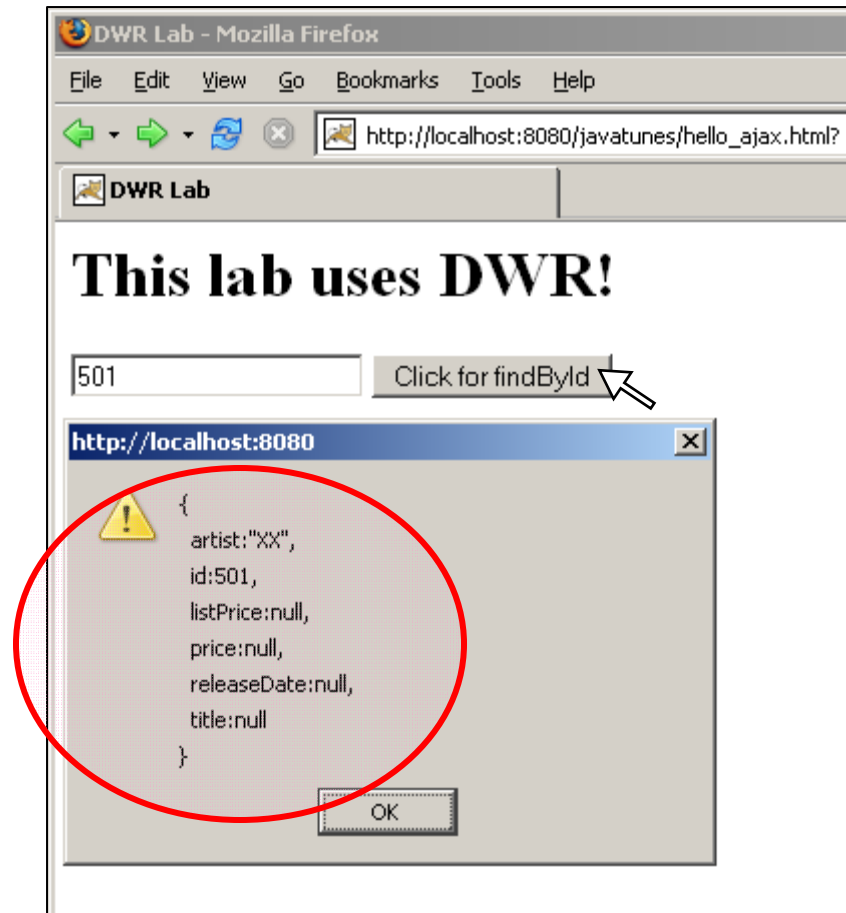
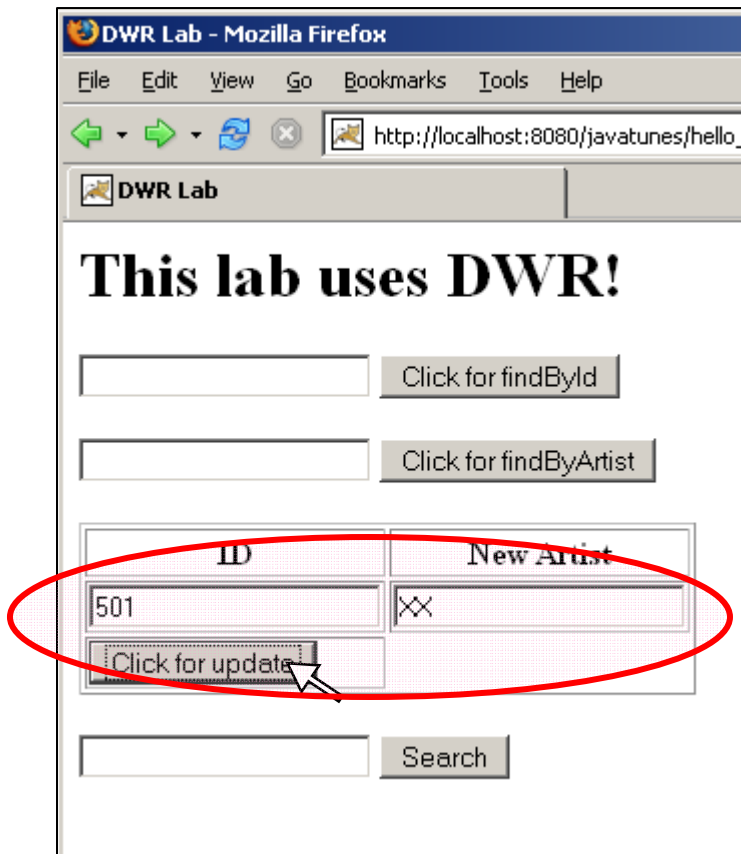
Tasks to Perform

- ◆ Finish the call to *SearchUtility.update()* by adding in its argument – an object literal you create representing a *MusicItem*
 - The literal should have **only** two properties
 - **id**: Initialized from the value of the id input field
 - **artist**: Initialized from the value of the artist input field
- ◆ **Publish**, then update the item with id=501 to have an artist of XX. Run *findById* with id=501 to see the new data

Seeing Update at Work

Lab

- ◆ On the right, you can see the results of running the update functionality shown on the left



Tasks to Perform

- ◆ Browse to *hello_ajax.html*, & type a letter into the search field
 - You **don't need any coding here** - this is coded for you already
 - This field has completion suggestions generated using a DWR call to *SearchUtility.findByArtist()*
 - Look at the *<form>* element where this code is triggered, noting the call to *SearchUtility.findByArtist*, and the *doCompletions* function which works with a JavaScript array returned by DWR
 - This code is almost exactly like the JSON example, except the Ajax call looks almost like a normal method call
- ◆ In FireBug look at the XHR calls to see a little bit of how DWR works
 - You can do this for all the DWR functionality you wrote

[Optional] Tasks to Perform

- ◆ **Optional:** In the *findByArtist* button's *onclick* event handler, add properties to the object literal passed as the second argument to the *findByArtist()* call
 - You can pass a number of arguments here, along with the callback handler – try the ones below:
 - ***timeout: 5000*** and/or ***rpcType: DWREngine.IFrame***
 - Before you use the IFrames rpc type, first run the app normally, and look at the network traffic in FireBug, then run it using IFrames and look at the network traffic
 - You'll see that XHR is no longer used after you specify IFrames as the transport



Other Technologies

DWR Overview
Working with DWR
Other Technologies

- ◆ JSON-RPC is a specification for a lightweight remote procedure protocol
 - The requests and responses are encoded using JSON text
 - It is fairly simple and easy to use
 - There are implementations in multiple languages, such as Python, Java, C, C#, and many more
- ◆ There is a JSON-RPC to Java bridge that is similar to DWR
 - Lets you transparently call server-side code from JavaScript
 - Somewhat more cumbersome than DWR
 - You need to write Java code to initialize the system, and then write Java code to register classes
 - This can be done in a JSP if you want, but DWR's configuration file approach is much easier

Using JSON-RPC-Java

- ◆ There are a number of steps needed to use the JSON-RPC Java bridge – both Java and JavaScript steps
- ◆ The Java steps include:
 - Add the JSONRPCServlet to your web.xml
 - This servlet handles JSON-RPC requests over HTTP, and dispatches them to a JSONRPCBridge instance in the session
 - Create an instance of JSONRPCBridge in the session
 - Register objects or classes you want to call messages on
- ◆ The JavaScript steps include:
 - Include the jsonrpc JavaScript code
 - Create the JSONRPCClient instance
 - Invoke remote methods
- ◆ The example following shows everything done in a JSP
 - The Java steps could conceivably be done in a servlet also

Using JSON-RPC-Java

```
<!-- Create the JSONRPCBridge and put it in the session --%>
<jsp:useBean id="JSONRPCBridge" scope="session"
    class="com.metaparadigm.jsonrpc.JSONRPCBridge" />
<!-- Register the SearchUtility classes static methods --%>
<% JSONRPCBridge.registerClass("search",
    com.javatunes.util.SearchUtility.class); %>
<html>
    <head> <!-- Include the JSON-RPC JavaScript library -->
        <script type="text/javascript" src="jsonrpc.js"></script>
    </head>

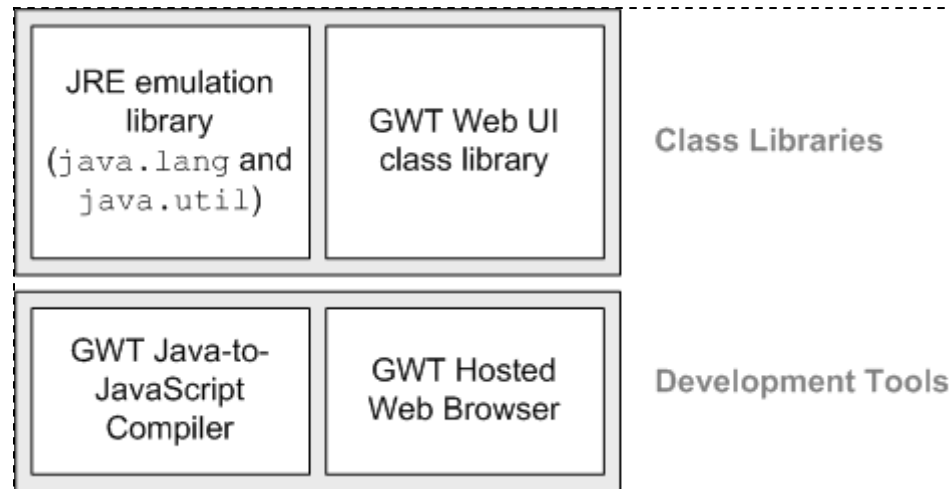
    <body>
        <script>
            // Initialize the JSON-RPC client
            var jsonrpc = new JSONRpcClient("/javatunes/JSON-RPC");
            // Call a remote method - synchronous version shown
            var result = jsonrpc.search.findById(501);
            alert (result);
        </script>
    </body>
</html>
```

- Google Web Toolkit (GWT) -

- ◆ GWT is an open source Java framework for creating Web applications
 - You write your front end in Java, and the GWT compiler generates HTML/JavaScript
- ◆ It has support for an extensive selection of widgets
 - It also has support for making Ajax calls
- ◆ The development cycle for creating a Web app with GWT is very different from a traditional one
 - Create and debug an application in Java, using GWT libraries
 - Use the GWT Java-to-JavaScript compiler to distill the application into a set of JavaScript and HTML files
 - Test the application in the browser

GWT Architecture

- ◆ GWT has four major components
 - **GWT Java-to-JavaScript Compiler**: Translates the Java code to JavaScript / HTML for running in the browser
 - **GWT Hosted Web Browser**: Let's you run GWT programs in a Java VM for testing
 - **JRE emulation library**: JavaScript implementations of all java.lang and some java.util classes
 - **GWT Web UI class library**: Set of interfaces and classes for building Web applications



Hello World With GWT

- ◆ Below is the GWT "Hello World" application
 - It contains a button and an event handler

```
package com.google.gwt.sample.hello.client;

import com.google.gwt.core.client.EntryPoint;
import com.google.gwt.user.client.Window;
import com.google.gwt.user.client.ui.Button;
import com.google.gwt.user.client.ui.ClickListener;
import com.google.gwt.user.client.ui.RootPanel;
import com.google.gwt.user.client.ui.Widget;

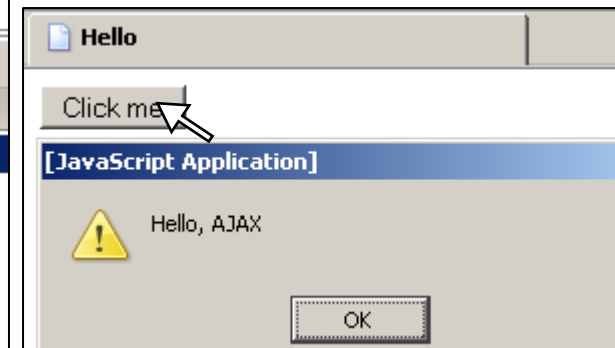
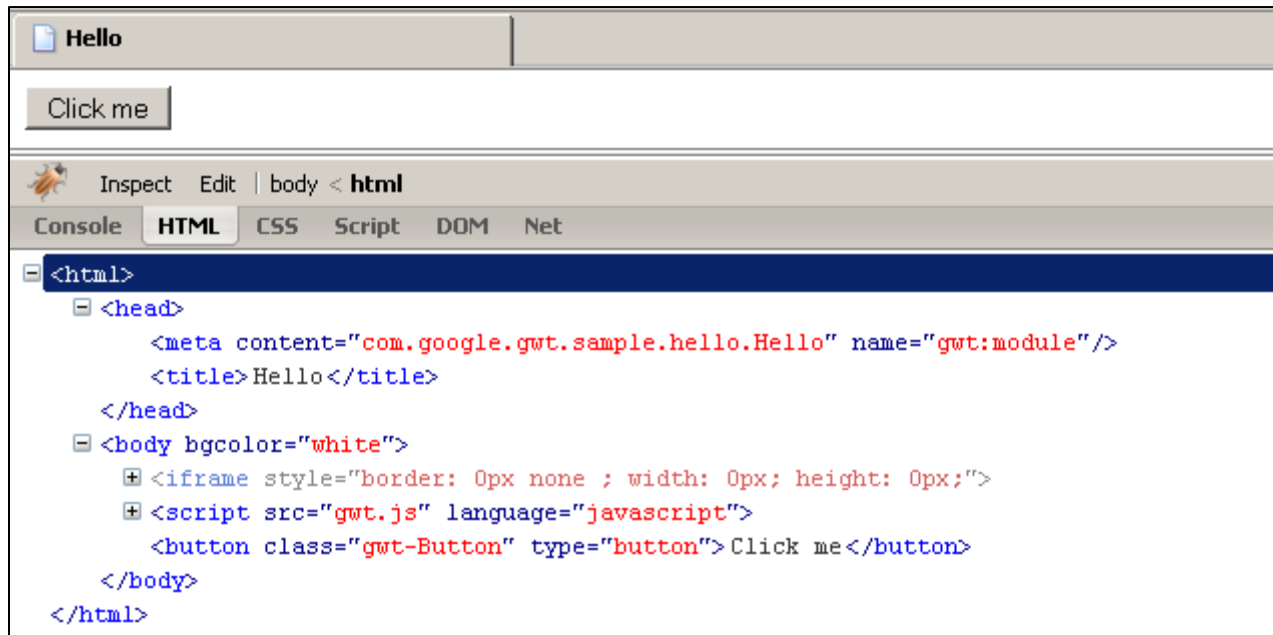
/** HelloWorld application. */
public class Hello implements EntryPoint {

    public void onModuleLoad() {
        Button b = new Button("Click me", new ClickListener() {
            public void onClick(Widget sender) {
                Window.alert("Hello, AJAX"); // There really isn't any Ajax in the app
            }
        });

        RootPanel.get().add(b);
    }
}
```

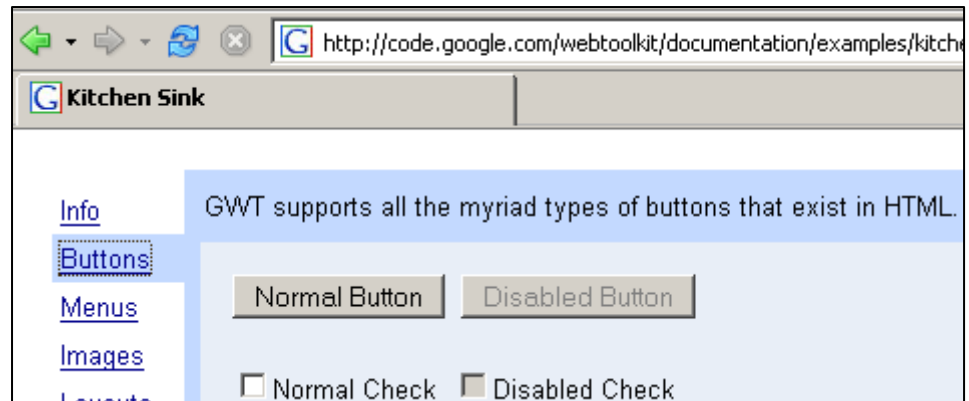
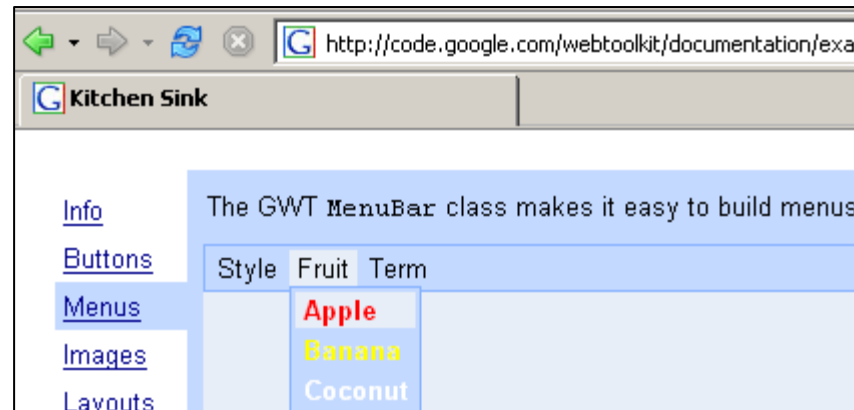
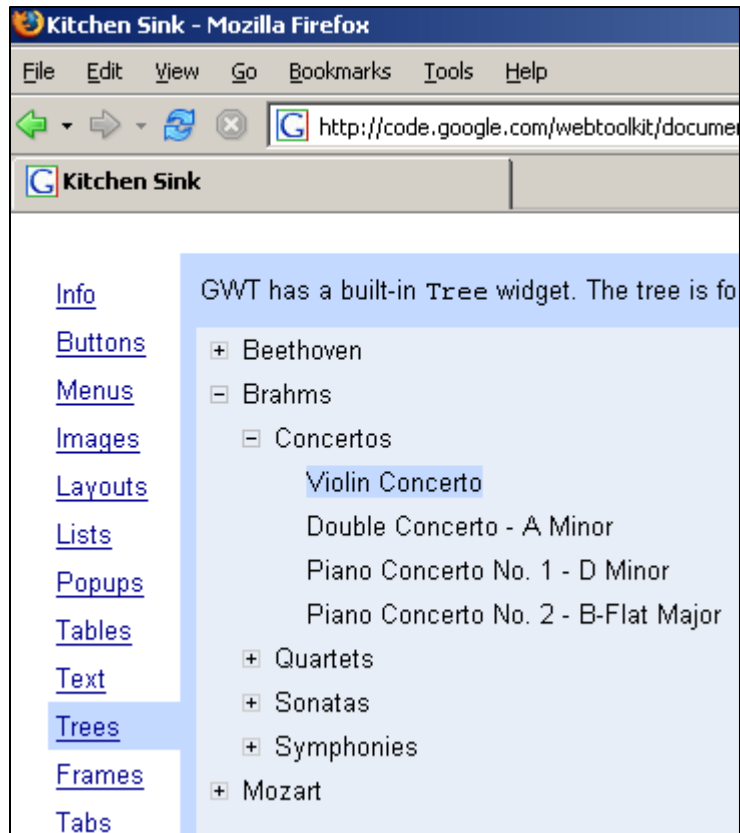

The Generated Application

- ◆ Below, you can see the generated HTML/JavaScript
 - You can also see the results of clicking the button
 - It's a simple application, but shows the workings of GWT



More About GWT

- ◆ GWT has an extensive array of UI Widgets
 - For example, trees, tables, Menus and so on
 - They are all created using Java code, and the Web apps are then generated using the GWT tools



Pros / Cons of GWT

◆ Pros:

- Takes care of all browser differences
- Easy to integrate Ajax-style operations into the application
- Compile time type checking in Java catches many otherwise common JavaScript errors
- Ajax applications are typically complex, and tools to manage them in Java are more mature than in JavaScript

◆ Cons:

- Must learn a whole new set of skills to create Web applications
- You don't really have complete control over the resulting HTML
- Open source, but basically you're tied to Google technology
- If you decide you don't want to use the technology, you need to rewrite any applications completely

Review Questions

- ◆ What is DWR?
- ◆ How do you configure DWR?
- ◆ How do you invoke the server code from JavaScript?

Lesson Summary

- ◆ **DWR** (Direct Web Remoting) lets you invoke server side Java methods from JavaScript in the browser
 - As if the Java code was running in the browser
- ◆ DWR requires a **Java servlet** that processes requests and returns responses be configured in `web.xml`
 - It also requires you to declare the remotely accessible classes, and other things, in a configuration file – *dwr.xml*
- ◆ DWR generates JavaScript proxies for the classes that you configure in *dwr.xml*
 - You can use these classes in your HTML pages to invoke the Java code

Session 11: Ajax and JSF

JSF Overview
Ajax and JSF

Lesson Objectives

- ◆ Understand what JSF is
- ◆ Become familiar with and see JSF and Ajax working together

JSF Overview

JSF Overview Ajax and JSF

JSF Purpose and Goals

- ◆ **JavaServer Faces (JSF)** is a user interface framework for rapid development of Java Web applications
 - It is built on top of Servlets, JSP and Custom Tags
 - Developed after careful study of existing frameworks like Struts
- ◆ It provides ease-of-use for developers in the following ways:
 - Makes it easy to construct a UI from a set of **reusable UI components** (i.e. a form is represented by a component on the server)
 - **Simplifies migration of application data** to and from the UI (from the HTTP request to and from JSF components)
 - **Helps manage UI state** across server requests
 - Provides a simple model for **wiring client-generated events to server-side application code**
 - **Allows custom UI components** to be built and re-used

- ◆ This overview based on JSF 2.0, which requires the following:
 - Servlet 2.5 / JSP 2.1 (Java EE 5 standard), and JSTL 1.2
 - Many users today are using this version
- ◆ JSF is an API extension (*javax.faces.**) containing:
 - Sixteen **packages** total
 - Hundreds of classes total (many are Tag Handler classes)
 - Many **JSF custom tags** (with "render kits")
 - An Expression language
 - Merged with JSTL EL / JSP 2.1 in JSF 1.2 and beyond
 - Earlier versions of JSF EL were very similar to the JSTL EL
- ◆ The main JSF site is
<http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html>

JSF as MVC

- ◆ As with other MVC frameworks (like Struts), use of JSF forces the developer to follow an MVC pattern
 - The pattern is enforced by the JSF structure and components
 - **Managed Beans** for the Model (holding the state of the view)
 - Facelets, and comprehensive **tag libraries** for the View
 - ***javax.faces.webapp.FacesServlet*** for the Controller
 - **Managed Beans** with action methods as part of the controller
- ◆ JSF is much more than just a user interface framework
 - It defines a very broad scope of components

JSF Views - Facelets

- ◆ Generally, the views are generated by **facelet pages**
 - Earlier JSF versions used JSP as the view technology, but this is considered deprecated starting in JSF 2.0
- ◆ A **facelet** – An xhtml page supporting these JSF capabilities:
 - **Tag libraries** to add JSF components to the page
 - Allowing, for example, dynamic data in a page's output
 - Easy creation of **custom and composition components**
 - **Templating** abilities for sharing page layout (ala Tiles)
 - The example below shows a very simple facelet *
- ◆ Tag libraries include:
 - **core**: non-HTML specific tags shared by various "render kits"
 - **html**: used to render HTML views
 - **ui**: Template library for creating templates
 - **composite**: Composite library for to composite components

Facelets Defined

- ◆ The example below shows a very simple facelet file
 - It includes declarations of the ui, f, and h tag libraries
 - This is done using standard XML namespaces
 - It includes usage of a small number of components from the tag libraries (e.g. h:outputText)
 - We'll cover this all in much more detail later

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head> ... </h:head>
  <h:body> <h:outputText value="Hello JSF World" /> </h:body>
</html>
```

JavaBean Defined

- ◆ A class that exposes properties to its environment
 - Properties are recognized from the get/set methods in the bean
 - May also have behavior (methods)
 - The *MyBean* class below has a single property, called *title*
 - A JavaBean is a **POJO** (Plain Old Java Object), as opposed to something more sophisticated like an EJB (Enterprise JavaBean)
 - True JavaBeans need to implement *Serializable*, but for JSF, this isn't strictly required
 - Though may be needed for other reasons

```
package com.mycompany.beans;  
public class MyBean {  
    private String title = null;  
    public String getTitle()    { return title; }  
    public void setTitle(String t) { title = t; }  
}
```

Managed Beans as JSF Model

- ◆ **Managed Beans** are POJO JavaBeans that are "managed" by the JSF environment at runtime
 - This means they are configured via *faces-config.xml*
 - Created and managed by the JSF controller components
 - They capture the state of the view as properties of a bean instance ¹
 - Used to refresh the view data when the page is requested again for any reason
- ◆ Note that these JavaBeans are "model" components in a limited sense only
 - They hold data and behavior relating to a JSF view
 - Other non-JSF model components, relating to the business model, are responsible for larger needs like business application logic and persistence

JSF Controller Components

- ◆ The main controller components are a controller servlet (***FacesServlet***) and a configuration file (***faces-config.xml***)
- ◆ ***faces-config.xml*** has entries to declare **managed beans** and **navigation rules** (among other things)
 - Navigation rules tell JSF which page should generate the view for the browser after a request has been processed
- ◆ The ***FacesServlet*** handles the requests for all JSF page views, and **must be configured in web.xml**
 - It works with *faces-config.xml* and managed beans to handle the request and determine the response
 - Navigation is determined by choosing the next best view based on *faces-config.xml* and the outcomes from methods in managed bean(s) (we'll cover this later in depth)

Managed Beans - Part of JSF Controller

- ◆ Managed Beans are also active, insofar as they have methods that can be invoked by the view (this is a little unusual)
 - These managed bean methods can be seen as part of the controller
 - The return value of these methods (the **outcome**) is used by the JSF controller components, as we'll see later in the class
 - Managed Beans can also implement the *EventListener* interface(s) for more dynamic work
- ◆ There is an entire session on Managed Beans coming shortly

faces-config.xml details

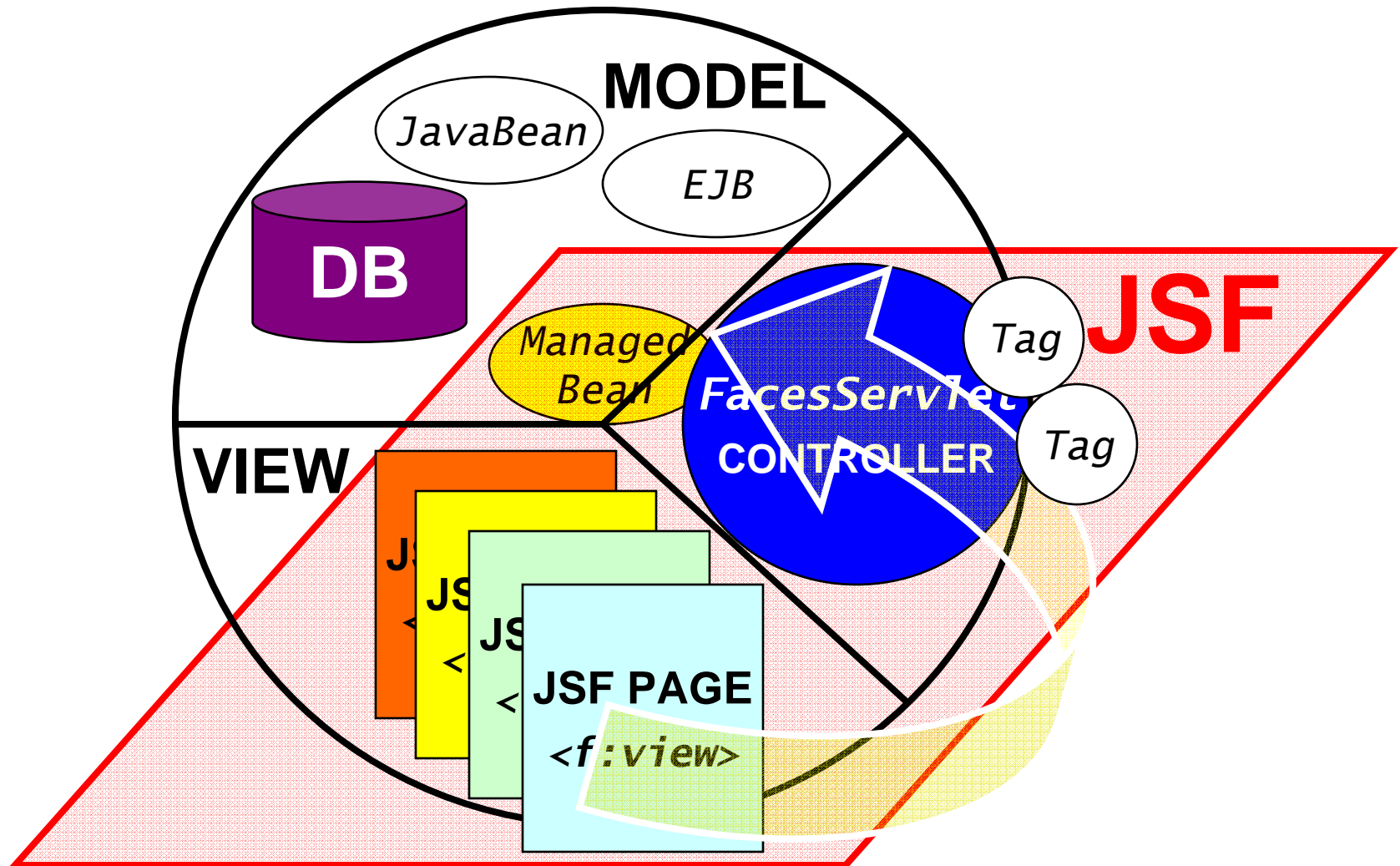
- ◆ You use the *faces-config.xml* file to configure a JSF web application
 - This is the standard name for the configuration file, but it can be changed via init parameters to the faces servlet
- ◆ Within *faces-config.xml*, surprisingly, all of the children of the parent tag *<faces-config>* are OPTIONAL
- ◆ The *<navigation-rule>* elements is one of the more frequently used sections
 - Previously, *<managed-bean>* was also commonly used, but in JSF 2.0+, **annotations** are more likely to be used to declare a managed bean

faces-config.xml structure

```
<faces-config>
  <application></application>
  <managed-bean> <!-- Not generally used in JSF 2.0+ -->
    <managed-bean-name></managed-bean-name>
    <managed-bean-class></managed-bean-class>
    <managed-bean-scope></managed-bean-scope>
  </managed-bean>
  <referenced-bean></referenced-bean>
  <navigation-rule>
    <from-view-id></from-view-id>
    <navigation-case>
      <from-outcome></from-outcome>
      <to-view-id></to-view-id>
    </navigation-case>
  </navigation-rule>
  <component></component>
  <render-kit></render-kit>
  <validator></validator>
  <converter></converter>
  <lifecycle></lifecycle>
  <factory></factory>
</faces-config>
```

Architecture Diagram

- ◆ The JSF architecture is built upon the MVC design pattern and covers the shaded area below:



Your first JSF application

- ◆ To write your first JSF application is easy
- ◆ You need all the JSF pieces (which should be available after downloading JSF)
- ◆ The web application structure is identical to plain Servlets and JSP, with some specific additions
 - You need to configure the *FacesServlet* in *web.xml*
 - You need a *faces-config.xml* file
 - You need the *jars* for the tag libraries and the JSF API

Configuring FacesServlet in web.xml

- ◆ JSF comes with a controller servlet which must be registered in the *web.xml* for your web application to function
 - *javax.faces.webapp.FacesServlet*
 - Requests are mapped to the *FacesServlet* based on standard URL mapping
 - You must define the *FacesServlet* in the *web.xml*
 - You must define the *<url-pattern>* element based on the type of mapping you prefer
 - You cannot use another controller to handle the JSF requests*

JSF Controller

- ◆ Two kinds of mapping are generally used:
 - **Prefix mapping:** */faces/** : Means that any request URL that begins with */faces/* will invoke the FacesServlet controller
 - **Extension mapping:** **.faces* : Means any URL ending with *.faces* will be mapped to the JSF controller
 - Which you use is mainly a style issue, we use extension mapping here
- ◆ A request for *hello.faces* to an application using extension mapping would invoke the JSF controller
 - By default, the controller would then translate this as a request for *hello.xhtml*

JSF Controller

- ◆ Sample *web.xml* controller definition for JSF
 - The example shows the use of extension mapping
 - There are other configuration parameters available

```
<servlet>
  <servlet-name>FacesServlet</servlet-name>
  <servlet-class>javax.faces.webapp.FacesServlet
</servlet-class>
  <load-on-startup>1</load-on-startup>
  <!-- Other init parameters omitted -->
</servlet>

<servlet-mapping>
  <servlet-name>FacesServlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```


JSF HelloWorld - Facelet

- ◆ Below is a very simple JSF file (*hello.xhtml*)
 - It's a xhtml page using the **core** and **html** JSF tag libraries
 - The tag libraries are declared as namespaces

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://java.sun.com/jsf/facelets"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">
  <h:head>
    <title>hello</title>
    <meta http-equiv="Content-Type"
          content="application/xhtml+xml; charset=UTF-8" />
  </h:head>
  <h:body>
    <h:outputText value="Hello JSF World" />
  </h:body>
</html>
```

JSF HelloWorld – Viewing the Facelet

- ◆ *index.html* (at bottom) contains a link to the JSF page (*hello.faces*)
 - Note the link in *index.html* specifies *hello.faces* and NOT *hello.xhtml*
 - This is required because we need the JSF controller servlet to be involved in the request
 - Because of the **.faces* mapping for the faces servlet, the controller will receive the request for *hello.faces*
 - It will initiate the JSF lifecycle, and then generate the view using *hello.xhtml* based on the default mapping of page requests to *.xhtml* files

```
<!-- index.html - detail omitted -->
```

```
This is a starting page for JSF<p>
```

```
A link <a href="pages/hello.faces">to a JSF page</a>
```

faces-config.xml

- ◆ Since we don't have any managed beans or navigation rules, the *faces-config.xml* file can be very simple
 - Most of the contents of the file are the XML schema declarations required by the specification

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

<faces-config version="2.0"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd">

</faces-config>
```

Review of Managed Beans

- ◆ **Managed beans** are POJOs that hold data for JSF requests
 - They are simple POJO classes that follows a few rules
 - Must have a **no-argument constructor**, & must have **accessor methods** (get/set) for each property accessed with the EL
- ◆ JSF Managed Beans can have behavior initiated by a request
 - Action methods that return a string representing the "outcome"
- ◆ Managed bean's **lifecycle is handled by the JSF controller**
 - Which creates an instance, populates it with data, invokes it
- ◆ The developer must write the actual managed bean class
 - With appropriate properties and methods

A simple Managed Bean

- ◆ The following *LogonBean* class defines a simple JavaBean with two properties (name and password):

```
package com.javatunes.beans;
public class LogonBean {
    private String userName = null;
    private String password = null;

    public String getUserName() {
        return name;
    }
    public void setUserName(String s) {
        userName = s;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String s) {
        password = s;
    }
}
```

Managed Beans Lifecycle

- ◆ With JSF, we don't create the managed bean directly
- ◆ The managed bean lifecycle is handled by the JSF controller
 - The managed bean instance is created, populated with data, and invoked upon, by the JSF controller
- ◆ Instead, of writing code, we let the the *FacesServlet* (JSF Controller) know that a bean is to be used
 - The controller instantiates and makes the bean available on a specific scope for use by the entire JSF application
 - The controller is managing the bean's lifecycle
 - That's why it's called a Managed Bean
- ◆ This behavior is **configured**, either with annotations, or in the *faces-config.xml* file
 - See example on next slide

Declaring a Managed Bean

- ◆ In JSF 2, the managed bean is easiest declared with annotations, as shown below
 - We declare the bean as managed, its scope as session, and the name used to refer to it as logon (via the *name* element)
 - The default name is the name of the bean with first letter lower case
 - The default scope is request scope

```
package com.javatunes.beans;
import javax.faces.bean.ManagedBean;
import javax.faces.bean.SessionScoped;

@ManagedBean(name="logon")
@SessionScoped
public class LogonBean {
    // Remaining detail omitted
}
```

Sample Logon Form

- ◆ Let's examine a form that might be used for users to logon
 - It uses the same core and html tag libraries as our earlier example
 - It has fields to input logon information, and submit the form

```
<html ... > <-- Much detail omitted - see notes -->
<h:head> ... </h:head>

<h:form>
    Enter Name: <h:inputText value="#{logon.userName}"/>
    Enter Password: <h:inputSecret
                        value="#{logon.password}"/>
    <h:commandButton value="Go"
                        action="#{logon.checkLogon}"/>
</h:form>

<h:head>
</html>
```


Examining the Logon Form

- ◆ We already know that the namespaces declare the JSF tag libraries we are using.
- ◆ The rest of the elements make up a JSF form
 - **`<h:form>`** is the top level element for declaring a form
 - **`<h:inputText>`** declares a standard text field input element
 - **`<h:inputSecret>`** is a text field input element that doesn't display its input
 - **`<h:commandButton>`** submits the form

Linking Input Fields to Bean Properties

- ◆ Let's examine now, how the form is linked to the managed bean
- ◆ In the element below, we link the name input field to the name property of the logon bean

Enter Name: <h:inputText value="#{logon.userName}"/>

- This is done through the attribute *value="#{logon.userName}"*
- This links the input field to the name property of *LogonBean*
- Similarly the password field is linked to the password property
- This is called a **value binding expression** and we will look at it more later

Submitting the Form

- ◆ When the *commandButton* is clicked, the form is submitted to the Web application
 - JSF will retrieve an instance of *LogonBean* and populate it with the submitted form values
 - How the form is processed depends on the *commandButton* element in the form

```
<h:commandButton value="Go"  
                 action="#{logon.checkLogon}"/>
```

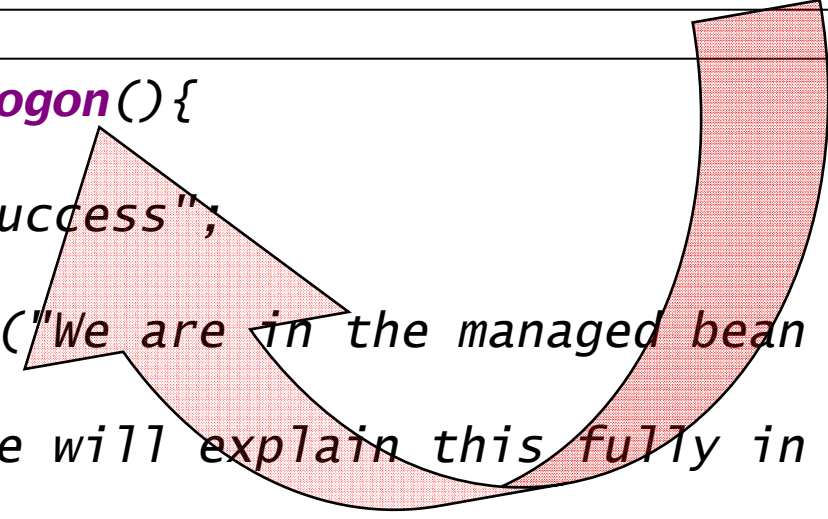
- The *action="#{logon.checkLogon}"* attribute tells the controller how to handle the submission
 - It is a method, and the return value is a logical outcome string that tells the application which page to access next

Method Binding Expressions

- ◆ We've bound a user interface component to a method
 - It basically describes a method invocation that will be executed
 - The JSF HTML tag `<h:commandButton>` is "bound" to the managed bean `checkLogon()` method
 - When the expression is evaluated, (e.g. the button pressed) the action (the `checkLogon()` method) is executed

```
<h:commandButton value="Go" action="#{logon.checkLogon}"/>
```

```
public String checkLogon(){  
    String result = "success";  
    System.out.println("We are in the managed bean method");  
    return result; //we will explain this fully in Navigation  
}
```



Dynamic Navigation Rule

- ◆ *checkLogon*'s return value is used to lookup a navigation rule, which determines what the next view displayed will be
 - The rule below works with the code on the previous page
 - *checkLogon()* returns **success**: go to *pages/searchForm.xhtml*,
 - *checkLogon()* returns **failure**: go to *pages/logonForm.xhtml*

```
<...>
</managed-bean>
<navigation-rule>
  <from-view-id>/pages/logonForm.xhtml</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/pages/searchForm.xhtml</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/pages/logonForm.xhtml</to-view-id>
  </navigation-case>
</navigation-rule>
</faces-config>
```

Creating / Deploying a JSF Application

- ◆ Creating the actual Web app is exactly the same as creating any Java EE Web application
 - You simply package everything up in a WAR file
 - JSP files, jars in the lib directory, faces-config.xml file, etc.
 - You need all the supporting jars for JSF
 - JSF is a standard part of Java EE 5+, so you don't need to include the JSF jars if using a compliant JEE server
- ◆ Once you have the war file, you deploy it as you would any other war file
- ◆ That's it – this example shows the core parts of a basic JSF application

Ajax and JSF

JSF Overview
Ajax and JSF

JSF and Ajax

- ◆ JSF 2 introduced Ajax directly into the specification
 - Draws on (and replaces) existing non-standard JSF Ajax libraries
- ◆ **<f:ajax>** used to easily "Ajaxify" standard JSF components
 - Can be used with any regular JSF component
 - Transparently handles all the (messy) JavaScript
 - Generates Ajax requests, processes responses, and renders components
- ◆ Uses a **JavaScript resource library** under the hood
 - Part of JSF 2
 - This resource library can be accessed independently for low-level Ajax-type programming
 - We will focus on the higher level (and much simpler) **<f:ajax>**

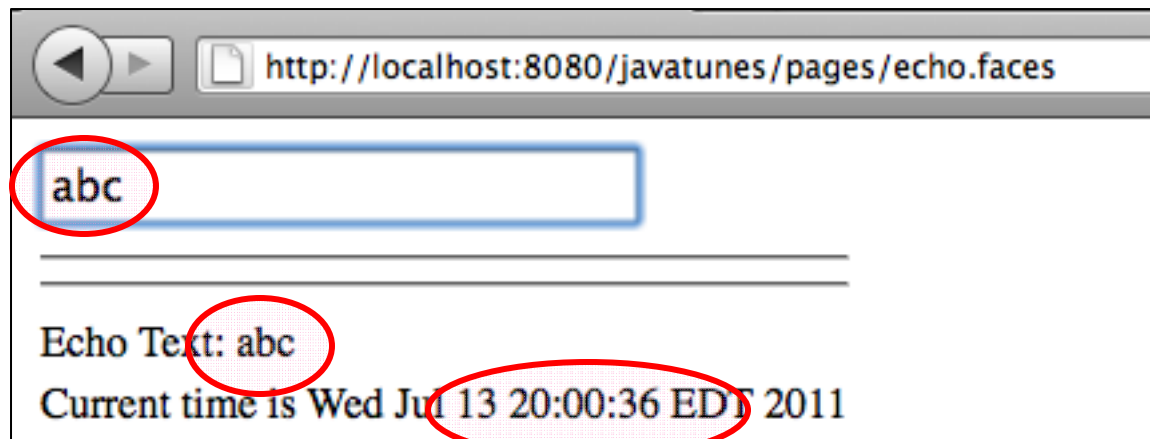
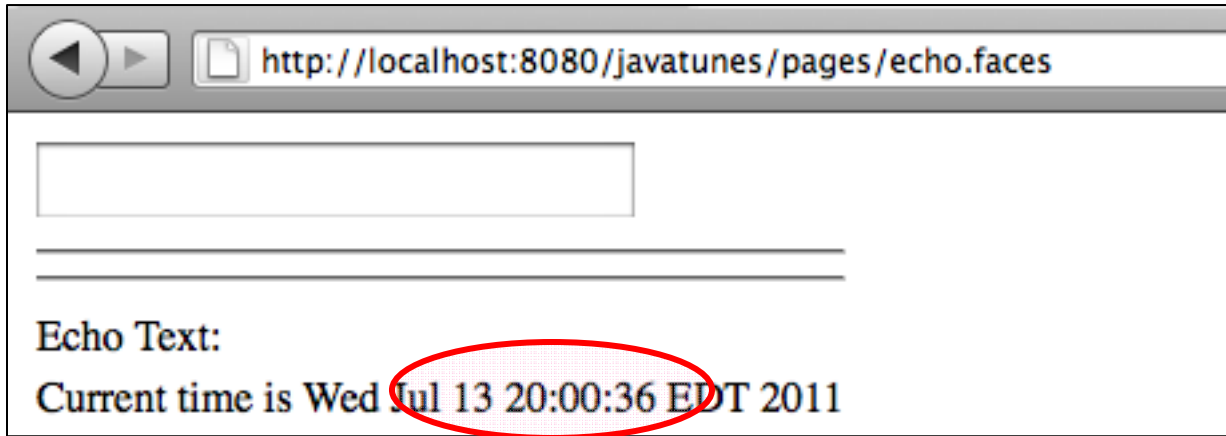
Simple Ajax Example

- ◆ In the example at bottom, the h:inputText is Ajax enabled
 - **<f:ajax>** enables Ajax
 - **event="keyup"** specifies that a keyup triggers the Ajax request
 - **render="echoTextFieId"** specifies the component with id=echoTextFieId (an outputText) as the component to render
 - The echo bean is a simple bean with an echoText property (a string), and a currentTime property (a date with the current time)
 - Each time a key is typed in the input field, it will be echoed in the output field – the rest of the page is not re-rendered

```
<h:form> <!-- Much detail omitted -->
  <h:inputText value="#{echo.echoText}" >
    <b>f:ajax event="keyup" render="echoTextFieId"/>
  </h:inputText>
  <hr/> <hr/>
  Echo Text:
    <h:outputText id="echoTextFieId" value="#{echo.echoText}" />
    <h:outputText id="theTime" value="Time is #{echo.currentTime}"/>
</h:form>
```

Simple Ajax Example

- ◆ Below are screen shots from the previous example
 - Note how the echo text is updated, but the current time is not



f:ajax Details

- ◆ *f:ajax* can be used to add Ajax functionality to:
 - A single component by nesting it within the component's tag (as seen in the previous example)
 - A group of components, by wrapping *f:ajax* around the component tags
- ◆ It allows the following attributes (and others covered later)
 - **event**: The event(s) that triggers the Ajax action
 - List of space-separated event names (derived from DOM events, not JavaScript – e.g. "click" rather than "onClick")
 - Must be compatible with events supported by the component
 - If not specified, the default event for the component is used
 - **render**: The component(s) executed on the server
 - List of space-separated ids

Ajax Example - Grouping

- ◆ In the example at bottom, the complete form is Ajax enabled
 - Note how f:ajax now encloses the whole form
 - The behavior will be exactly the same in this case, as we specify the event (keyup) and the rendered component (echoTextField)
 - However, if you, for example change render to the below
render="echoTextField theTime"
 - then the time will change also, since you are specifying that you render the outputText that displays the time

```
<f:ajax event="keyup" render="echoTextField"/>
<h:form> <!-- Much detail omitted -->
  <h:inputText value="#{echo.echoText}" />
  <hr/> <hr/>
  Echo Text:
    <h:outputText id="echoTextField" value="#{echo.echoText}" />
    <h:outputText id="theTime" value="Time is #{echo.currentTime}"/>
</h:form>
```

Lab 11.1 – Using JSF and Ajax

- ◆ **Overview:** In this lab you will use some of the JSF/Ajax functionality
 - We'll first show a simple echo page, as seen in the slides
 - You'll then look at an Ajax-enabled search box
- ◆ **Objectives:**
 - Review the JSF/Ajax capabilities
- ◆ **Builds on previous labs:** None
- ◆ **Approximate Time:** 20-30 minutes

- ◆ The root lab directory where you will do all your work is:

C:\StudentWork\Ajax\workspace\Lab11.1

- This is a new lab directory

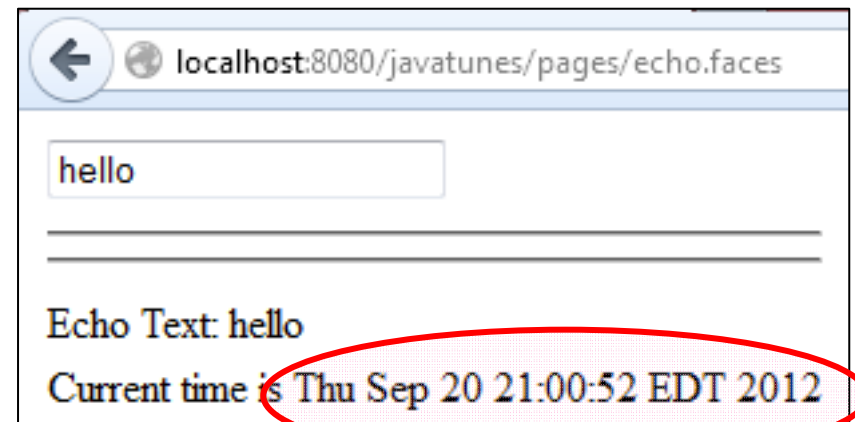
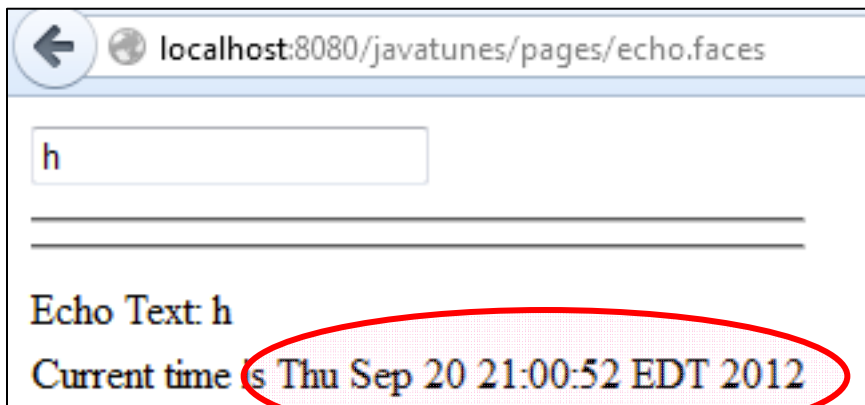
Tasks to Perform

- ◆ Right click on the server in Servers View
 - Select **Add and Remove Projects**, and **remove** Lab10.1
- ◆ Create a new **Dynamic Web Project** called **Lab11.1** (see Lab01.1 instructions if necessary)
 - Make sure the Tomcat server is selected
 - Make sure **2.5 module version** is selected
 - Remember to set the context root to **javatunes**

- ◆ If you look in the *WEB-INF\lib* directory, you'll see the JSF jar files
 - The Ajax functionality is included in JSF 2
 - The JSTL jars are standard supporting libraries for JSTL
 - *WEB-INF\faces-config.xml* is a standard Faces config file
 - *WEB-INF\web.xml* configures the Faces servlet
 - The configuration files are configured as usual for JSF, and there is nothing special needed in them to use Ajax
- ◆ You'll first view one file with Ajax functionality - the echo page
 - *WebContent\pages\echo.xhtml*
 - There is also a Java source file - EchoBean - the simple managed bean for the Ajax echo functionality
 - It's needed because Ajax behavior is full JSF behavior, so you need the same kinds of managed beans for the data model

Tasks to Perform

- ◆ Right click on project, **Run As | Run on Server**, restart server
 - Browse to ***http://localhost:8080/javatunes/pages/echo.faces***
 - Type something in the input field, you should see it echoed
 - Notice that the date displayed lower in the page doesn't change
 - The page is NOT re-rendered because it is an Ajax request that specifies only the outputText be rendered
 - **Open** pages/echo.xhtml for review - note the ***f:ajax*** tag
 - That's **all** that's needed for this Ajax functionality



Tasks to Perform

- ◆ We've also included Ajax functionality in a search page
 - It will use Ajax to show and update search results as you type
 - This functionality is in *WebContent\pages\ajaxSearch.xhtml*
- ◆ Open *ajaxSearch.xhtml* for review
 - Note the *f:ajax* tag - this adds Ajax behavior to the search input field
- ◆ Open *SearchBean.java* for review
 - It has an **Ajax listener** method named *ajaxSearch*
- ◆ **Browse** to *localhost:8080/javatunes/pages/ajaxSearch.faces*
 - Try typing something in the search field (like the letter "a")
 - You'll see search results show up immediately below the search field
 - This is done with an Ajax call
- ◆ You can see how how easy JSF makes working with Ajax

Results Showing Ajax Search

JavaTunes Search Form

localhost:8080/javatunes/pages/ajaxSearch.faces

This is the JSF JavaTunes Search Form

JavaTunes:

Title	Artist	Price	Buy Now
Ian Moore	Ian Moore	9.97	Add to Cart
So Much for the Afterglow	Everclear	13.99	Add to Cart
Surfacing	Sarah McLachlan	13.99	Add to Cart
Hysteria	Def Leppard	14.99	Add to Cart
A Life of Saturdays	Dexter Freebish	12.99	Add to Cart
Human Clay	Creed	13.28	Add to Cart
My, I'm Large	Bobs	11.97	Add to Cart
So	Peter Gabriel	13.99	Add to Cart
Big Ones	Aerosmith	14.99	Add to Cart
1984	Van Halen	11.97	Add to Cart
Escape	Journey	11.97	Add to Cart



Session 12: Design and Best Practices

JavaScript Best Practices

Ajax Design

Ajax Security

Lesson Objectives

- ◆ Think about the issues involved in creating good Ajax applications
- ◆ Learn best practices and techniques for creating good Ajax applications
- ◆ Become aware of Ajax security issues

JavaScript Best Practices

JavaScript Best Practices

Ajax Design

Ajax Security

JavaScript is a Key Ajax Technology

- ◆ Creating Ajax applications involves JavaScript
 - Depending on how you work with Ajax, how much you use JavaScript directly will vary
 - In general, there will be some usage
 - It's important that you create good JavaScript, so we'll first look at some of the techniques and practices that help you do that
- ◆ Programming with JavaScript and Ajax is **closer to writing regular software** applications than it is to Web design
 - You have a great deal of active code to deal with – much more than in standard Web applications
 - You need to use techniques that are appropriate for this
 - It's especially important to create **modular, well organized code**
 - It's likely that you'll want to use the Object-Oriented capabilities of JavaScript

Object-Oriented, Modular JavaScript

◆ Create object-oriented JavaScript code

- We've seen that JavaScript has the ability to create types that have a lot of similarities to Java classes – with constructors, properties and methods defined in a type
- The OO model has proven to be very strong in complex GUI applications – which is what Ajax applications are
- Use these capabilities to create JavaScript types for your application – we've seen examples of that in the course

◆ Use good modeling and design techniques

- It's important to go through as rigorous a process with your JavaScript applications as you do with any other
- Good Ajax applications can't be written without good design

◆ Create modules to organize your code

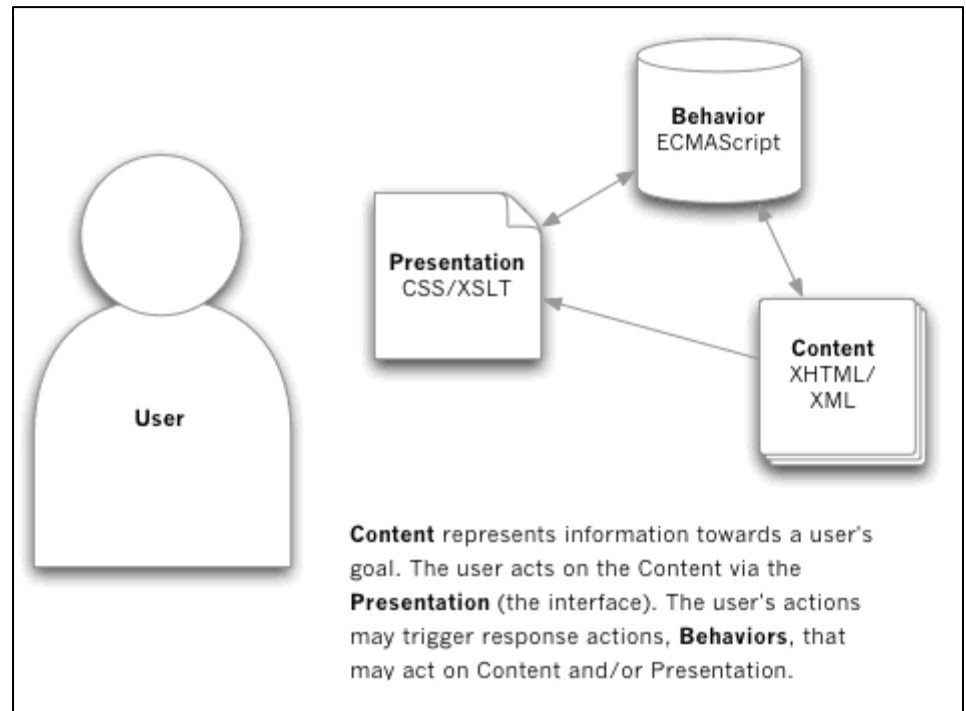
- We've seen that JavaScript has the capability to create modules that can serve the same purpose as Java packages

Dealing With Browsers

- ◆ Unfortunately, we often **don't own the client environment**
 - There may be many different browsers that run our JavaScript applications – and will break them if we're not careful
 - There is incompatibility in browser JavaScript support
- ◆ **Keep browser issues in mind**
 - There is a level of base functionality that almost all browsers will support, and you can keep your code in this base
 - There will still be differences which you should encapsulate
- ◆ There are many libraries which deal with these issues for you
 - We've looked at some of them – such as Prototype and GWT
 - It makes sense to use some of these libraries
- ◆ **Users may turn off JavaScript in their browser**
 - This will BREAK your Ajax application, so be prepared for it
 - Decide on a policy for this, and then implement it

Separate Content, Behavior, & Presentation

- ◆ Using CSS to separate Presentation from Content is standard
- ◆ We should also **separate out our Behavior**
 - Which is coded in JavaScript
 - At the least, this means putting your scripts into separate files
- ◆ You can also follow a policy called "**unobtrusive JavaScript**"
 - Instead of hard coding the behavior in the content, you dynamically add it to elements using DOM
 - The content document should only contain HTML (or XHTML, XML ...)



JavaScript Tips and Techniques

- ◆ **Provide progressive enhancement**
 - JavaScript should enhance behavior, but not be required
 - The content should still be usable with JavaScript
- ◆ **Alter content as little as possible**
 - For example, if you're changing presentation, then it's simpler and cleaner to do it by just changing a style attribute
 - If you do change content, then your JavaScript should output HTML that is as simple as possible
- ◆ **Document your output, parameters, and dependencies**
 - If your JavaScript changes a class or id, then document it
 - If it changes content, document what the content looks like
 - Make your fellow programmers happy !

Don't Reinvent the Wheel

- ◆ There is a lot of information and tool support available
 - Spend some time looking at this
 - Choose what works for you
- ◆ It is beyond the scope of this course to talk more about general JavaScript Best Practices
- ◆ Some good resources are:
 - **Ten good practices for writing JavaScript in 2005** by Bobby van der Sluis – Excellent article about working with JavaScript
 - **<http://www.w3schools.com>** has extensive HTML and JavaScript references
 - **<http://developer.mozilla.org/en/docs/JavaScript>** contains a wealth of information about JavaScript

Ajax Design

JavaScript Best Practices

Ajax Design

Ajax Security

Ajax is Still Evolving and Maturing

- ◆ Ajax is a young technology that is rapidly changing
 - In many cases the infrastructure and tools are still immature
 - How to best use the technologies is still not always clear
- ◆ There is a great deal of innovation going on
 - New capabilities are continually being introduced
 - There are many different frameworks available that offer different functionality and have different strengths and weaknesses
- ◆ It's hard to get a handle on all that is going on
 - Developers are still ramping up on the technology
 - There are many ways to use the technology
 - It is changing rapidly
- ◆ We'll try to present some **core guidelines** in this section
 - These are general guidelines that will help create good Ajax applications no matter what technology you use

Basic Ajax Design Principles and Patterns

- ◆ We've tried to use good practices throughout the course, so you are already familiar with many basic design principles
- ◆ Many of these are representative of general Ajax patterns
 - These can be viewed in a more formal, in-depth style at the **Ajax Patterns page** - <http://ajaxpatterns.org>
 - We'll summarize the patterns we've covered in the course, then go on to cover other basic principles
- ◆ **Ajax App with XMLHttpRequest** – Creating a rich Web app with Ajax - the basic building blocks of Ajax
 - Can also use other transports, such as **IFrame**
- ◆ **RPC Service and Ajax Stub** – Expose server-side services on the web, and access them with JavaScript objects
 - DWR is an example of this style of programming
- ◆ **HTML/XML/JSON message** – Pass messages between the server using different formats – the usage depends on your needs

Basic Ajax Design Patterns

- ◆ **On-Demand JavaScript** – Download code when needed
 - Instead of all at once, for example the way Dojo does this
- ◆ **Server-Side Code Generation** – Generate the JavaScript/HTML
 - For example, GWT where you code in Java
- ◆ **Suggestion** – Suggest words or phrases for completion
 - Many of our labs used this technique
- ◆ We've covered many patterns for building good Ajax apps
 - There is a lot of information available for further refinement
 - For example, there are many more patterns available
 - Browser-server communication patterns such as **Submission Throttling** where you batch Ajax requests, or **Periodic Refresh** where you refresh volatile data periodically with a timer
 - XML patterns if you're using XML as a data representation
- ◆ It's important that you continue to explore Ajax Patterns
 - When you have a problem, often there will be a related pattern

Use Ajax Where Appropriate

- ◆ Use Ajax **where appropriate** – and **don't when it isn't**
 - Ajax is a wonderful tool, but don't make it the only tool
 - It should be used when you need the interactivity that it makes possible
- ◆ Some places where Ajax is useful
 - **Form input** – We've seen an autosuggest text box, but there are other uses, for example linked Select Menus where options are generated via Ajax based on input from the previous select
 - **Deep hierarchical data** – Can lazy load the data with Ajax
 - **Rapid User to User Communication** – e.g. chat. Really feasible with an HTML/Ajax client, especially with something like reverse Ajax
 - **Computationally expensive ops** – offload to server with Ajax
 - **Paging** – Can load incremental data with Ajax

Network Usage Considerations

- ◆ There are a number of Ajax related **network performance** issues
 - You must keep in mind that every Ajax call is a network request
 - Even though a typical Ajax call is usually smaller than a page refresh, there may be many more of them
 - You must also take into account the time it takes you load your "application" – which is written in JavaScript
 - Some libraries, such as Dojo, are quite large, and loading them may incur significant overhead
- ◆ **Server side resources** are also used to service Ajax requests
 - Each request requires server side resources to handle
 - You may need to tune how the server handles connections (for example, connection timeouts, and threading/request limits)
 - There are also connection limits from browser to server, which must be taken into account – especially with Reverse Ajax, which may try to keep connections open for long periods

Ajax and the Back Button – The Problem

- ◆ The **back button** is always an issue when working with Ajax
- ◆ Simple use of Ajax breaks the expected behavior of the back button – which is that one URL refers to one document
 - With vanilla Ajax, there are problems in three areas – as follows:
- ◆ **Clicking back from an Ajax page:** Users naturally expect this to undo the last Ajax change
 - This actually will display the previous page in the browser's history – which may be a completely different view
- ◆ **Clicking forward to an Ajax page:** This has similar issues
 - e.g. If you've mistakenly taken yourself out of an Ajax app by clicking the back button, & you then click the forward button, you go to the first page of the Ajax app, not where you came from
- ◆ **Bookmarking an Ajax page**, and then revisiting it, brings you to the start page – no matter where you bookmarked from

Ajax and the Back Button - Solutions

- ◆ Some solutions are available that deal with the back button
 - Most of them use a combination of **hidden IFRAMEs**, and/or adding a unique value to the **fragment identifier** of the URL
 - The **DOJO** toolkit and **Really Simple History Framework** are two frameworks that provide this functionality
- ◆ These solutions still require you to manage the state yourself
 - They provide the framework for your web application to respond to the back/forward buttons, and potentially allow bookmarks
 - You need to record the changing state of the application and provide meaningful fragment identifiers that allow you to identify and restore previous states
 - You also need to decide what state to save and when
- ◆ This helps the situation, but it still takes work to implement
 - You may want to balance how important this behavior is, with the work required for a solution

User Interface Design Considerations

- ◆ Because Ajax Web applications have much more flexibility in the user interface, you need to design them carefully
- ◆ **Give visual cues** that something is going on
 - For example, if the user takes an action, and is waiting for an asynchronous Ajax response, display something to show this
 - For example, GMail displays a small red "loading" icon when it is loading something in the background
- ◆ Be careful when you **change the page display**
 - If you update some unexpected area, users may not notice it
 - Conversely, if an update happens in some obscure area, and you draw attention to it (**blink, blink**) you may annoy the user
 - Your page design should take into account what data might be updated, and make it a natural part of the page flow and usage

User Interface Design Considerations

- ◆ Be careful how you **affect reading and scrolling**
 - If someone is reading a page, and an Ajax response comes and changes the text, it may interrupt the user flow
- ◆ Make sure to **keep your page consistent**
 - Because it's easy to change small parts of a page with Ajax, you can end up with inconsistent information in the page
 - Be careful to update all parts of a page related to a change
- ◆ **Don't change expected behavior**
 - For example, it's easy to change state with a link, but you probably shouldn't - users are used to these driving navigation, and may get confused
 - Adding many exotic bells and whistles to a page may also make it complicated for users to learn the application
 - This is a problem with Web 2.0 applications in general – there is a lot more choice in how Web applications behave

Other Ajax Design Considerations

- ◆ **Search engines** can have a difficult time with Ajax apps
 - If a lot of your content is delivered via Ajax, and there are no links to it, then search engine spiders may not be able to reach it
- ◆ JavaScript can contribute to **performance issues**
 - It is an interpreted language, and not highly performant
 - If you do a lot with JavaScript, the client may run out of CPU
 - Desktop machines generally have powerful CPUs these days, but for JavaScript intensive websites, you may have issues
- ◆ Enough is enough, but **too much is too much**
 - Too much Ajax can be a problem – for instance if you're making an Ajax request each time a user clicks one of many checkboxes, it might be better to just do it once upon submission
 - If the user has to wait for each request to finish, then you can end up with a the web app that seems slower instead of faster

Ajax Security Ideas

JavaScript Best Practices
Ajax Design
Ajax Security

General Security Issues for Ajax

- ◆ Ajax adds new security issues to traditional Web applications
 - We'll look at some general issues, then move onto specific ones
- ◆ **JavaScript is required** to run Ajax and Web 2.0 apps
 - Enabling JavaScript opens up a whole class of security holes
- ◆ Many of the new technologies are not as well known
 - Because of unfamiliarity, developers may create security holes
- ◆ There is a larger **attack surface** exposed
 - Use of Ajax is more fine grained than traditional Web apps, and also requires the server to expose more functionality
 - This allows traditional attacks, for example SQL injection, more possibilities to work with
- ◆ The combination of technologies makes it harder to control
 - The interactions of all the technologies makes for many possible combinations that may be attacked

Basic Security Guidelines

- ◆ Basic security practices are just as important as always
 - We'll summarize some of the important ones
- ◆ Always **secure sensitive applications** to unauthorized users
 - **Ajax supports user authentication**
- ◆ **Form-based authentication** (session/cookie based) works
 - Ajax requests pass along the session cookie, and can use the standard server-side authentication mechanism
 - There is some added complexity, for example how do you handle a session timeout in response to an Ajax request
- ◆ Ajax can also work with **Basic Authentication**
 - The *XHR.open()* method accepts optional user/password arguments that are used for basic authentication
 - Basic authentication is not as flexible as form-based, and has all the usual shortcomings, such as not being able to log out

Basic Security Guidelines

- ◆ **Don't send data in the clear** is as true for Ajax as other apps
 - Ajax supports HTTPS without any additional work
 - Since this is handled at the network layer, using a URL with the https: protocol will use SSL for your Ajax requests
- ◆ **SQL injection** is also an issue – you should take common safeguards to protect against malicious user input
 - This is really a scripting attack, with SQL as the language
 - We'll talk about scripting attacks more
- ◆ If you're using **XML data**, it has its own class of problems
 - Denial of service with large messages
 - Entity expansion attacks or injection of malicious code
 - Because the data generally has no schema associated with it, it's very hard to guarantee its safety

Scripting Vulnerabilities – Malicious Code

- ◆ Allowing JavaScript to execute in a browser opens up a whole class of security holes based on injecting malicious code into a web page
 - Can occur where dynamically generated pages contain HTML tags/scripts based on unvalidated input from untrusted sources
 - **Ajax requires JavaScript**, so it is vulnerable to these attacks
- ◆ e.g., on a discussion group site, a user might post a message like:

Hello all **<script>Malicious code</script>** See y'all later.

- When a victim reads this message, and the script executes, the malicious code is executed
- ◆ **Cross Site Scripting (XSS)** is another example of this
 - In addition to code being injected, the malicious code then exploits security vulnerabilities to gain access to other websites via the malicious code, potentially with the security access of the compromised user

The Dangers of Code Injection

- ◆ Users can encounter malicious code when they follow insecure links, in mail messages, web pages, bulletin boards
 - The malicious code may be encountered on a site that the user is interested in and trusts
 - The malicious code could gain access to any information that the user enters – such as credit card or social security numbers
- ◆ The famous **MySpace Samy worm** used these techniques
 - MySpace allows users to put their own content on their profile pages – and has safeguards such as stripping out JavaScript
 - Samy found a subtle way **to bypass the JavaScript safeguards**
 - He put a JavaScript worm on his profile so when anyone visited it, their browser would load his code, which used XMLHttpRequest (yes, **Ajax** !) interactions to make Samy the visitors "friend and hero"
 - It also modified the visitors profile to replicate and include the worm
 - Within a day, **the worm infected 1 million users and MySpace had to be shut down**

XSS - Same Origin Policy

- ◆ One of the techniques used to limit the damage of XSS is called the **Same Origin Policy**
 - Content should only be allowed to access resources from the same website, but not resources from other websites
 - Origin includes domain name, protocol and port
 - This means that a script can only access data and web pages loaded from the same origin – which means malicious code can't grab information from your banking webpage
- ◆ All modern browsers enforce this restriction
- ◆ Web browsers also restrict XMLHttpRequest to requests to **only the server that the containing web page came from**
 - So if a web page was loaded from `http://www.javatunes.com` then it could ONLY make XHR requests to `www.javatunes.com`
 - This is also to help prevent Cross-Site attacks

Same Origin Policy – The Good and the Bad

- ◆ The Same Origin Policy is a good thing
 - It helps prevent XSS attacks from doing damage
 - Even if malicious code does make it to the browser, it is usually encapsulated to working with the site it was loaded from
- ◆ Sometimes you need to get around the policy, and below are some ways you can do this, but be careful if you do
 - Doing so means that any code downloaded will run with the full authority of the user on your site
 - **Application Proxies**: An application on your server takes your XHR request, and forwards it to another domain
 - The proxy should be very smart, and selective in what it forwards
 - **Server Proxies**: You set up your server so the XHR requests are invisibly rerouted to the target Web domain
 - **Script tag hack**: Dynamically created script tags with appropriate src attributes download and eval JavaScript

Preventing Malicious Content

- ◆ One of the most important ways to prevent scripting attacks is to prevent the generation of malicious content in your apps
- ◆ One common guideline for this is **never trust your client**
 - You must prevent untrusted input from creating malicious code
- ◆ Untrusted input can come from
 - URL Parameters, Form elements, Cookies, Database Queries
- ◆ To mitigate vulnerability from these inputs, there are a number of things that can be done
 - These try to prevent the input or generation of special content – such as tags, scripts – basically any dangerous markup
 - This in turn depends on identifying the **special characters** that are used to create markup – such as <
- ◆ We'll cover some of the core principles involved in this

Preventing Malicious Content

- ◆ Set the **character encoding** for server generated pages
 - If the encoding is unknown, you can't identify special characters
- ```
<META http-equiv="Content-Type" content="text/html;
charset=ISO-8859-1">
```
- ◆ **Identify special characters** such as <, &, > and others
    - Text, attributes, URLs, <script> and more all can contain special characters – the details are beyond the scope of this course
    - You may want to restrict these
  - ◆ **Filter input** - You should always validate your input to make sure that it contains only what it is supposed to
    - For example, an input field for age should only accept the characters 0-9, and special characters should be filtered from block level input (e.g. a free form text input field)

# Preventing Malicious Content

- ◆ **Encode and filter dynamic output**
- ◆ If special characters are permissible in dynamic output, you should encode them
  - e.g. use the entity representation &lt; instead of <
- ◆ You should also consider **filtering dynamic output**
  - Filtering on output makes sure that all output is examined for malicious code
  - You can create a set of characters that are allowed, and make sure that all dynamically generated output consists only of that set
  - Filtering the input is still useful, but may not get all potential means of entering malicious code
  - For example, there might be database data that can be entered by means other than HTTP

# JSON Issues

- ◆ At the very least, you need to sanitize **JSON**,
  - If you `eval()` JSON text without taking precautions, you will execute any arbitrary JavaScript code that might be present
  - Perhaps inserted into your data somehow by a malicious attacker
  - Parse JSON before you `eval()` it to make sure there is no executable JavaScript – **`JSON.parse()`** in `json2.js` does this
- ◆ JSON is vulnerable to attacks just because it's JavaScript
  - **JavaScript hijacking** is a new form of attack that targets JSON
  - Put simply, malicious code overrides some of the building blocks of JavaScript (such as the array or Object constructors)
  - It then uses a `<script>` to access another website, and peeks at confidential data that was transported as JavaScript (JSON) via the overridden building blocks – bypassing Same-Origin
  - Google's GMail was subject to this kind of vulnerability because user contact lists were stored unprotected in JavaScript objects

# Security Summary

- ◆ We've seen that just using JavaScript has significant security implications
  - Once a Web page becomes "active" rather than just being display code, there are more vulnerabilities
  - These vulnerabilities are constantly being exploited
  - For example, the Super Bowl 2007 website was compromised\*
- ◆ Ajax uses JavaScript, so has all the associated vulnerabilities
  - Since Ajax usually involves dynamic generation of web pages, often based on user input, they are vulnerable to additional areas of attack such as injection attacks
- ◆ Developers **must** guard against these kinds of attacks
  - We've seen some strategies for improving security
  - Security is ongoing – threats evolve, and so must the response
  - Security should be included in your design from the beginning

## - Final Summary -

- ◆ As you can see, Ajax is a complex field, with many different facets, and many different ways to use it
  - How you use it will depend on your needs and existing systems
  - e.g., if you're already using JSF, Ajax4sf is a natural choice
  - Otherwise, one of the other frameworks might be more suitable
- ◆ Ajax will continue to grow and mature
  - Within the next year or so you can expect things to become more standardized and stable
  - Until then, you'll need to evaluate your needs carefully with respect to what is available
- ◆ This course should give you a good understanding of Ajax
  - It is an excellent start for on the tools available for using Ajax
  - You'll need to decide which of these tools is best for you

# Resources

- ◆ **Ajax Patterns**: Extensive collection of Ajax related patterns
  - <http://ajaxpatterns.org>
- ◆ Books:
  - **JavaScript: The Definitive Guide** by David Flanagan
- ◆ Other Websites
  - **DWR Home Page** - <http://directwebremoting.org>
  - **JSON Home Page** - <http://www.json.org>
  - **Prototype Home Page** - <http://www.prototypejs.org>