Ajax For Java Developers

on the Tomcat Platform

LearningPatterns, Inc. Courseware

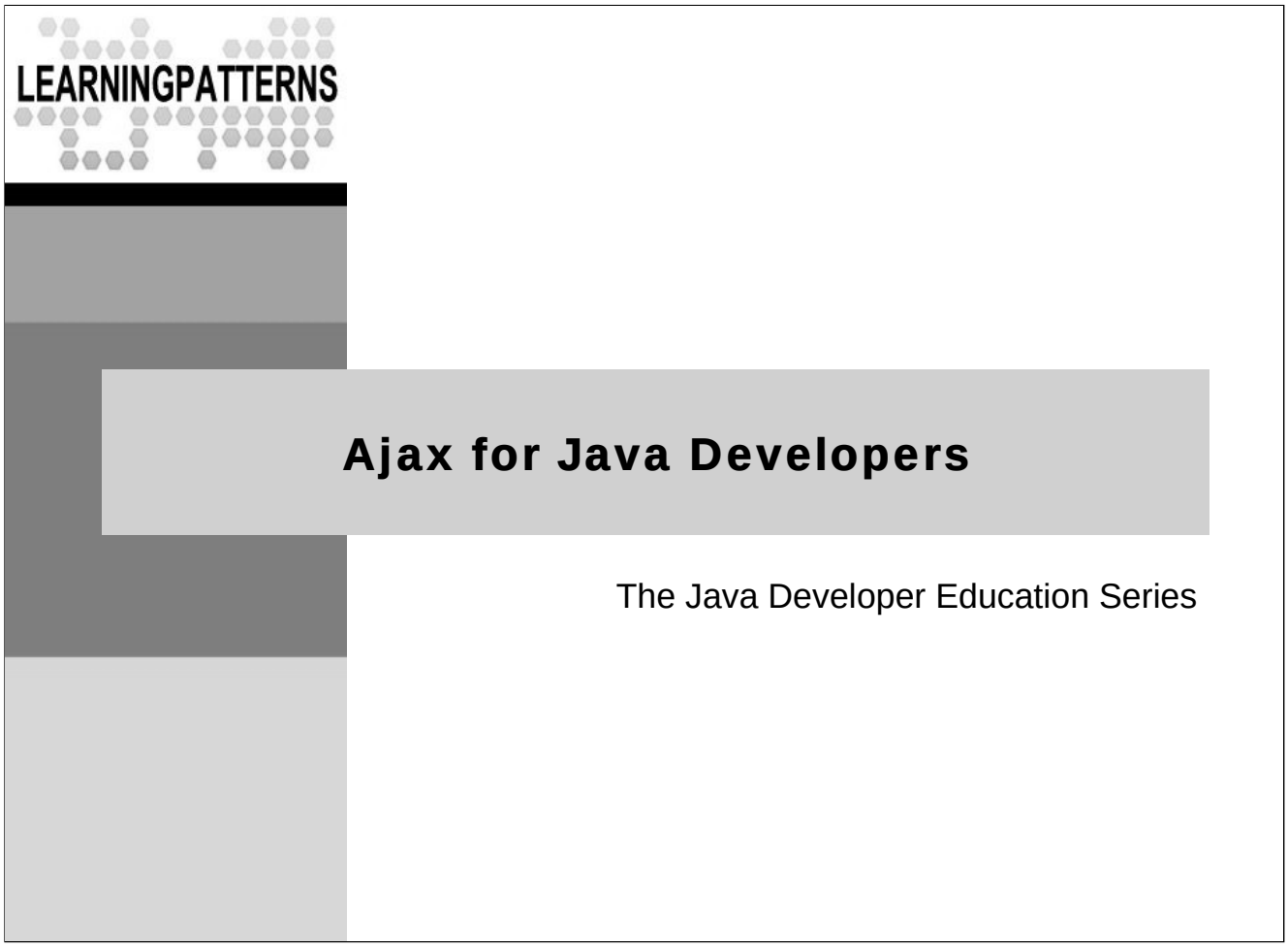
Student Guide

For more information about Java or Enterprise Java, and related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc.

Global Java Courseware Services 55 Wanaque Ave. #188 | Pompton Lakes, NJ 07442 USA
212.487.9064 voice | 201.336.9118 fax

Table of Contents - Ajax for Java Developers



Notes:

Workshop Overview

- This is an in-depth 4-day 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 consists of 50% discussion, 50% 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

Notes:

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

Notes:

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**

Notes:

Course Prerequisites

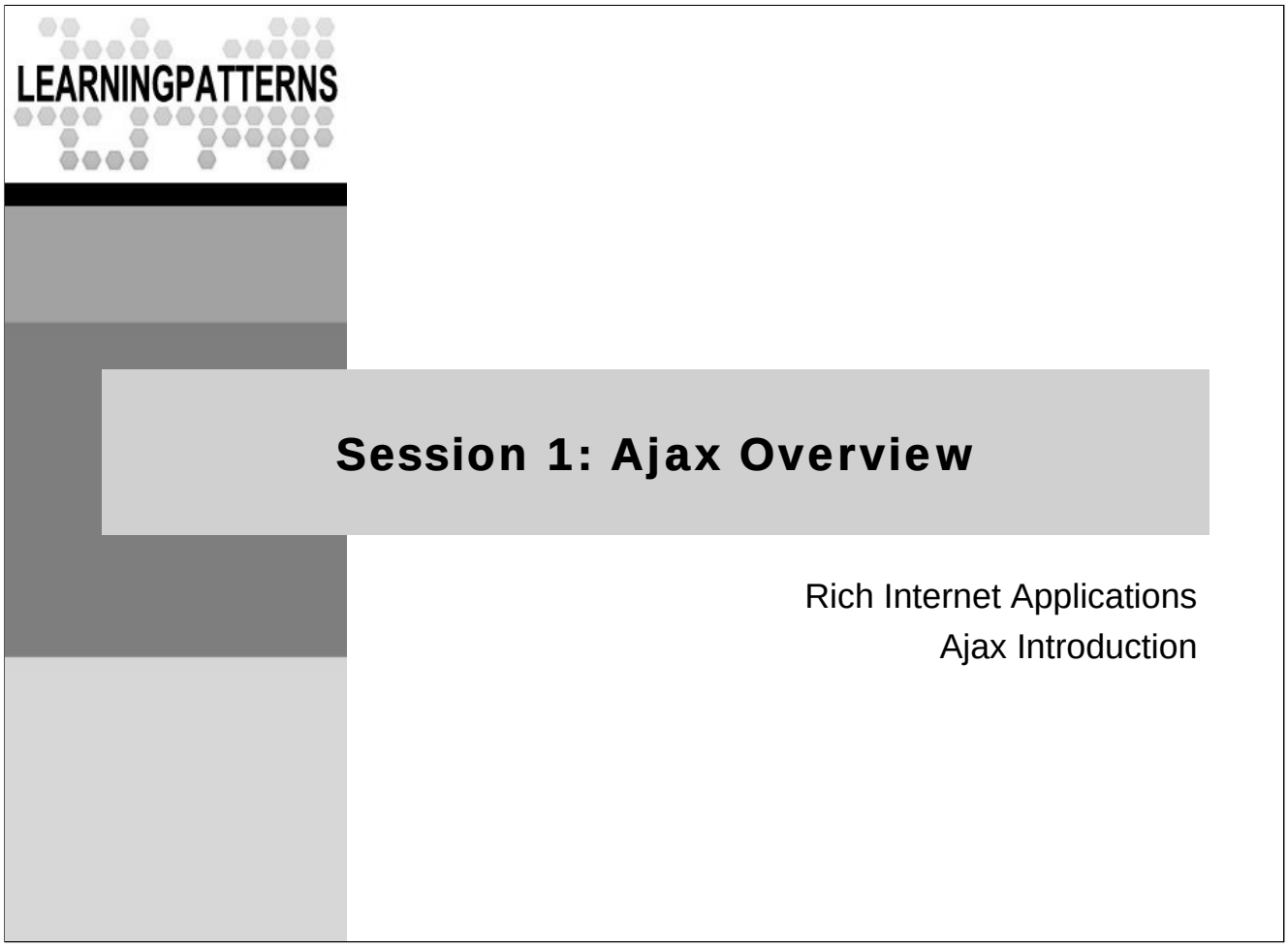
- Basic knowledge of HTML
- Practical Java and Servlet/JSP programming for the Java material
  - We'll review Servlet/JSP programming basics if required
- Some knowledge of JavaScript helpful
  - We'll review JavaScript basics if required

Notes:

Release Level

- This manual has been tested, and contains complete instructions, for running the labs using the following platforms:
  - **Tomcat 6.0.10** (All 6.0 and Tomcat 5.5 versions should work)
  - **Java 5** (Java Development Kit 1.5.0_xx)
    - See notes for running on J2SE 1.4
  - **Eclipse 3.4 Java EE Edition** (Ganymede)
- All labs have been tested on Microsoft Windows XP

- All recent versions of ant should work fine
  - There is nothing fancy in the ant build files
- We are using Tomcat 6.0, with JDK 5.0
  - Earlier versions of Tomcat (e.g. 5.0) should probably work, but they are untested - you'd need to figure out any needed changes yourself
  - If you want to run with J2SE 1.4, it is possible to use Tomcat 5.5 with JDK 1.4, and that should work also – but it is untested by us, but here is what you'd need to do, but you'd need to install the compat package available from http://tomcat.apache.org. See the file RUNNING.txt in the Tomcat home directory for more information on this.

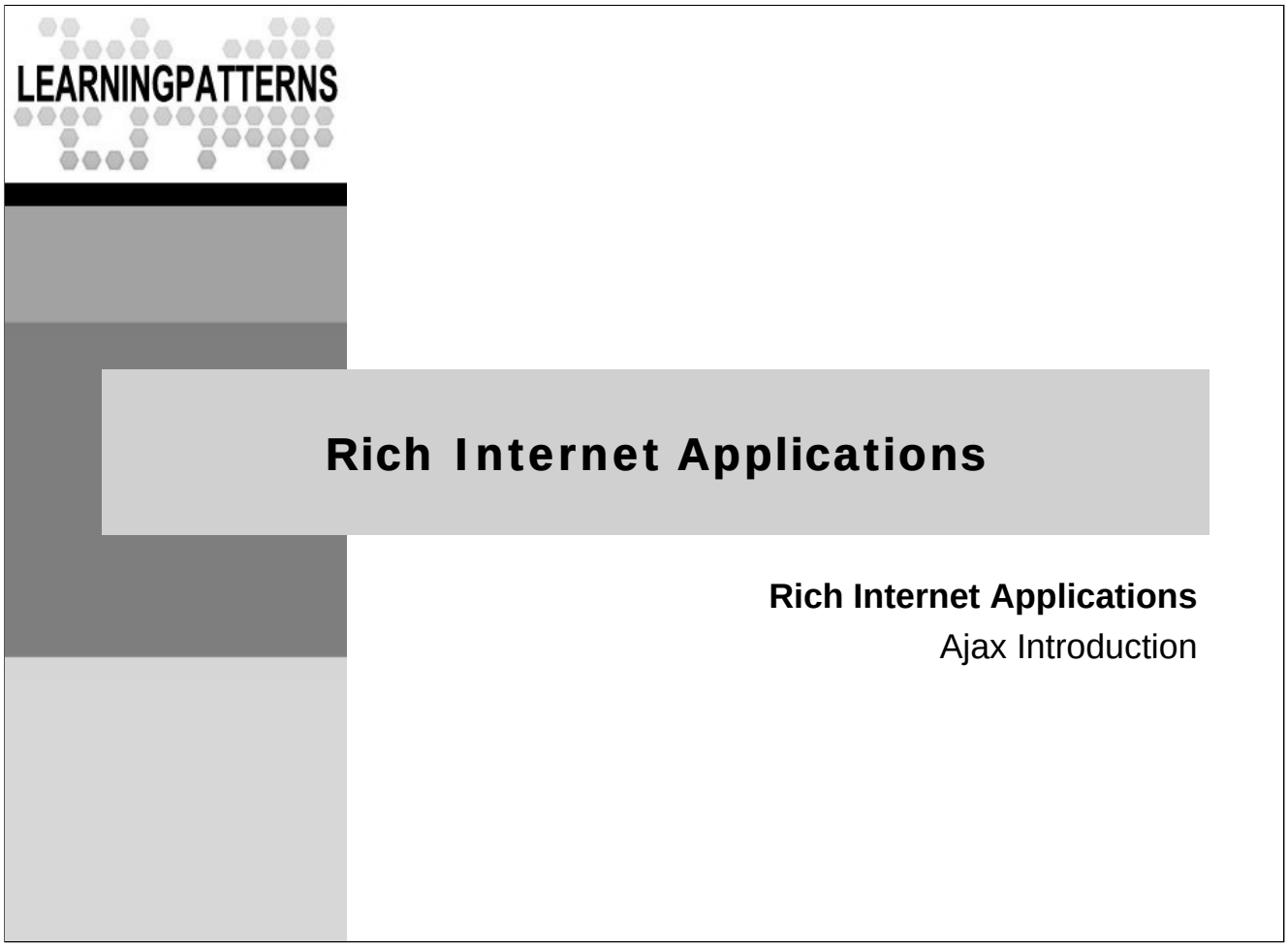


Notes:

Lesson Objectives

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

Notes:



Notes:

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

Notes:

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

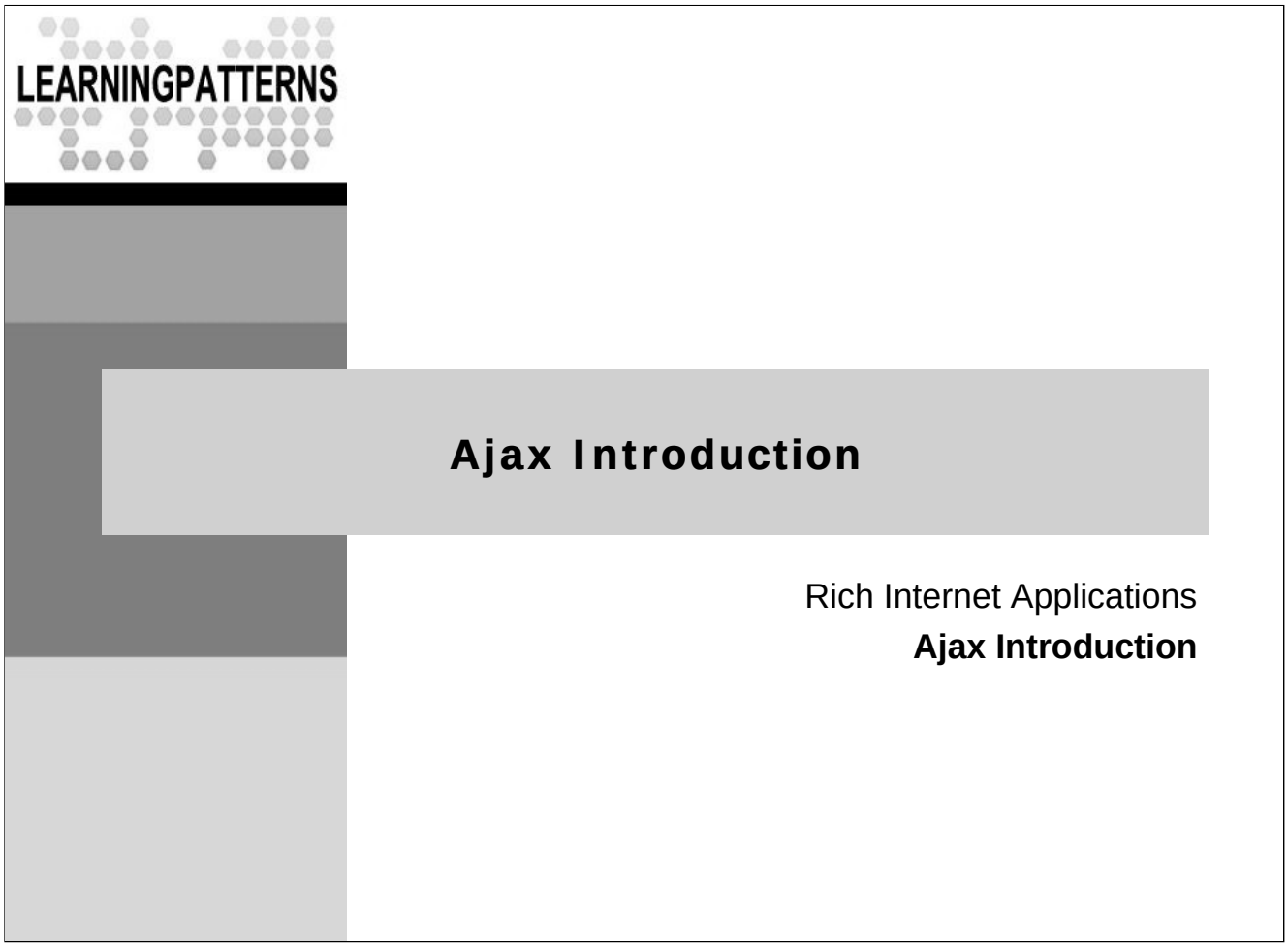
Notes:

- There really is no formal definition of RIA or Web 2.0
  - These are broad definitions that signify a change in the way Web applications are viewed
  - They embody the idea of the "Web as a platform"
- Web 2.0 covers such wide-ranging ideas such as:
  - Rich User Experience: Google applications
  - Radical Trust: Wikipedia
  - Participation, not publishing: Blogs
  - User as contributor: PageRank, eBay reputation, Amazon reviews
  - Radical Decentralization: BitTorrent

Dojo FisheyeList Widget Demo

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

Notes:



Notes:

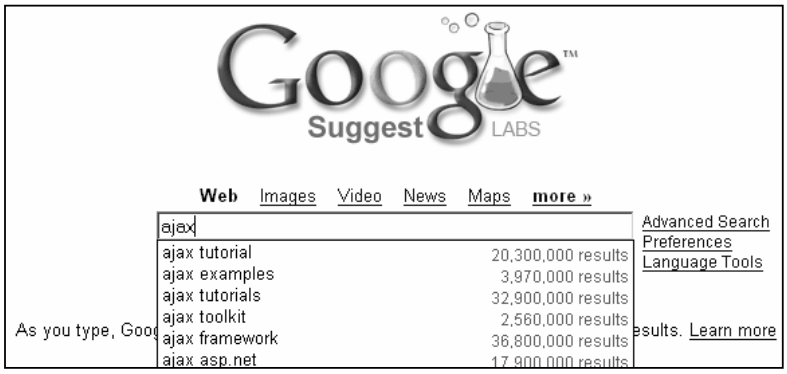
What is Ajax

- **A**synchronous **J**avaScript **a**nd **X**ML
  - 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

Notes:

Ajax Example – Google Suggest

- Uses Ajax to retrieve potential search matches as you type
  - You can try it at **http://www.google.com/webhp?complete=1&hl=en**



- In the Google Suggest page, only the list of suggestions is updated as you type
  - The rest of the Web page remains the same and is not refreshed
- The web page makes requests to the server behind the scenes as you are typing, sending it your current input
  - The server does a search on the current input, and sends the results back to the browser
  - The browser, using JavaScript, then updates the list of suggestions

Notes:

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

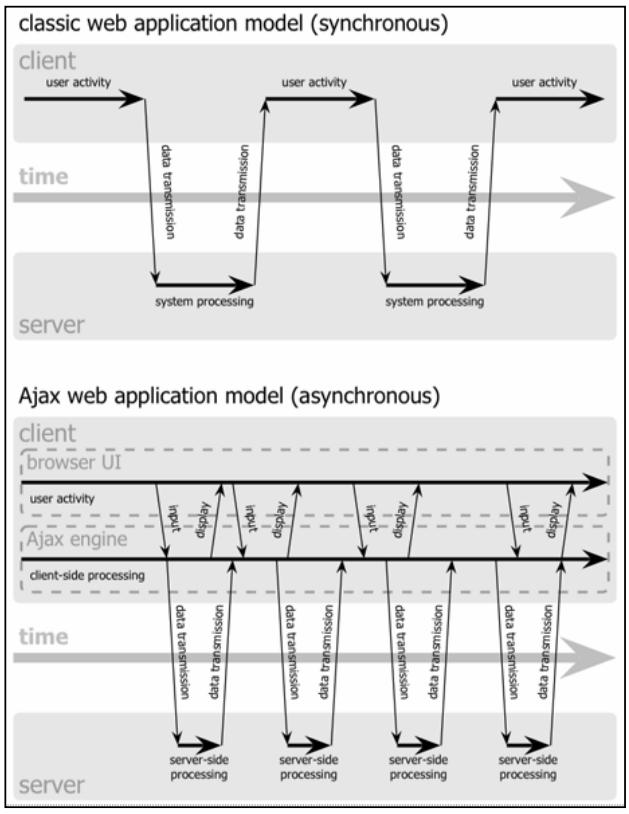
Notes:

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

Notes:

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

Notes:

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

Notes:

- There is no Ajax "specification"
  - It means different things to different people, and is used in many different contexts
- In this course, we will use it to refer to the user interface style we previously mentioned - the updating of small

XMLHttpRequest

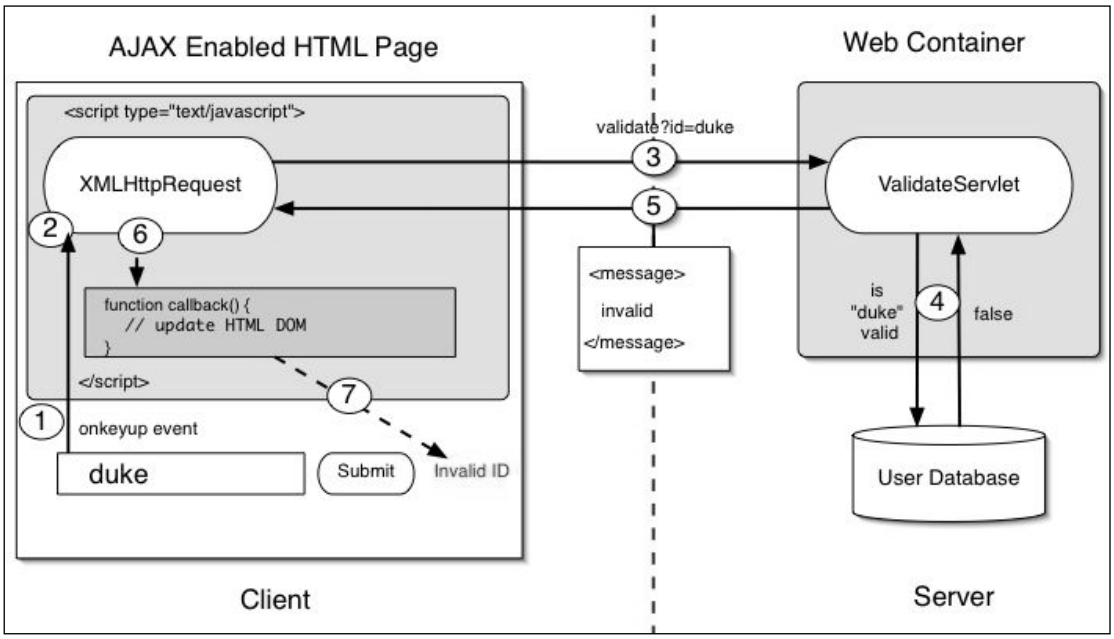
- **XMLHttpRequest** is a JavaScript object that is at the core of Ajax applications
  - It allows HTTP requests to be made to the server using JavaScript
  - It does **not** require a page refresh
  - You can register an event handler (a JavaScript function) that responds to the XMLHttpRequest request 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

Notes:

- In truth, Ajax is not really asynchronous communication
  - In the background, the browser is making an HTTP request and synchronously getting an HTTP response
  - When the browser gets that response, then it triggers an action
  - From the **user** point of view, when that action is triggered by the browser, and the results become visible, it appears asynchronous

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



- The illustration above is from Sun's Technical Article "Asynchronous JavaScript Technology and XML (AJAX) With the Java Platform"

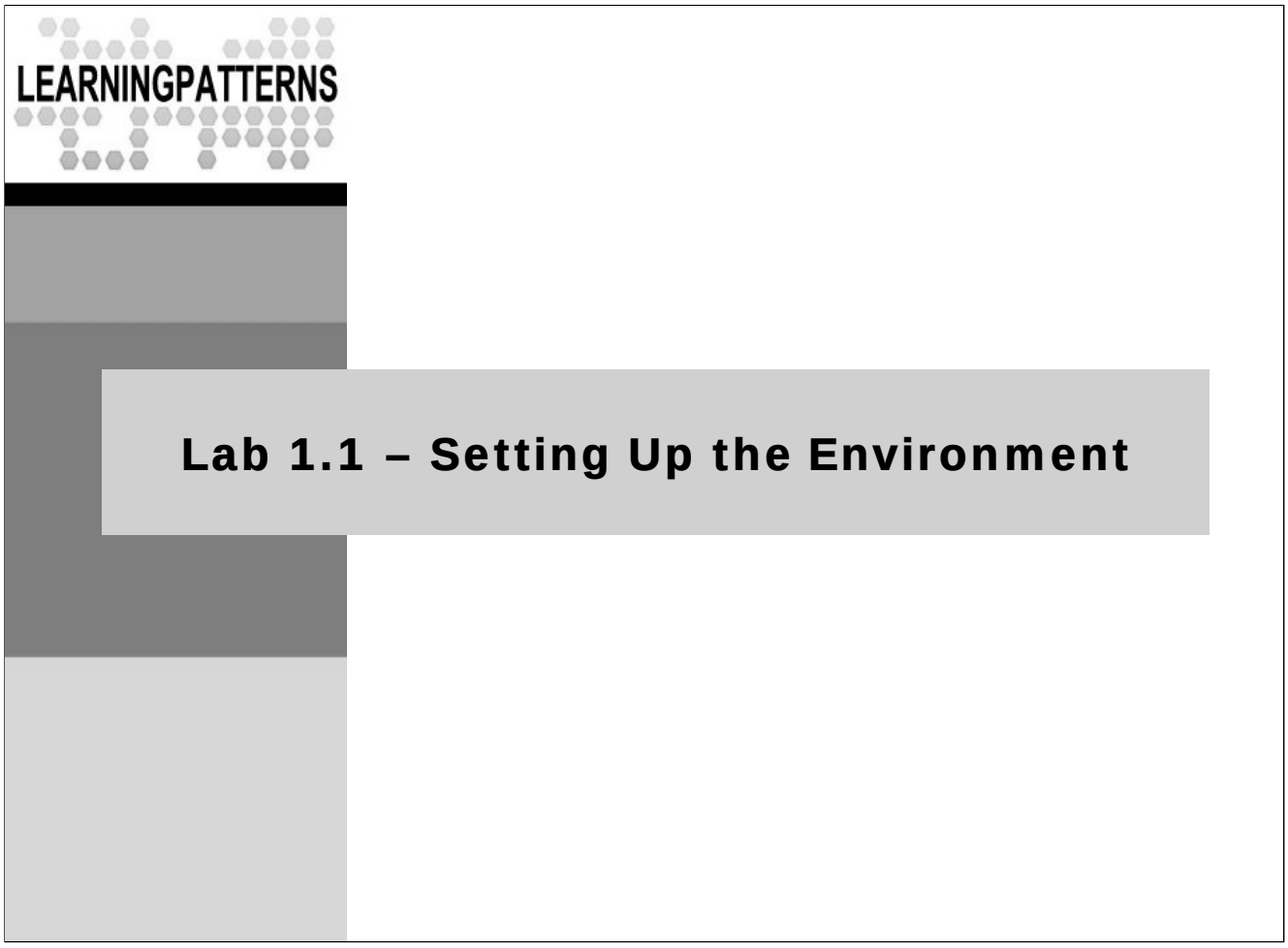
Notes:

- Note that even though this is asynchronous from the user's perspective, it is still initiated by the client (Browser)
  - It is still a "client pull" model
  - There are technologies, such as Comet, that are being explored to create a "server push" model

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

Notes:



Notes:

Lab 1.1 – Set 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

Notes:

Information Content and Task Content

- 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 CD**; we're ready to start working

Notes:

Extract the Lab Setup Zip File

- To set up the labs, you'll need the course setup zip file *
  - It has a name like: ***AjaxJava_LabSetup_Tomcat_20081015.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

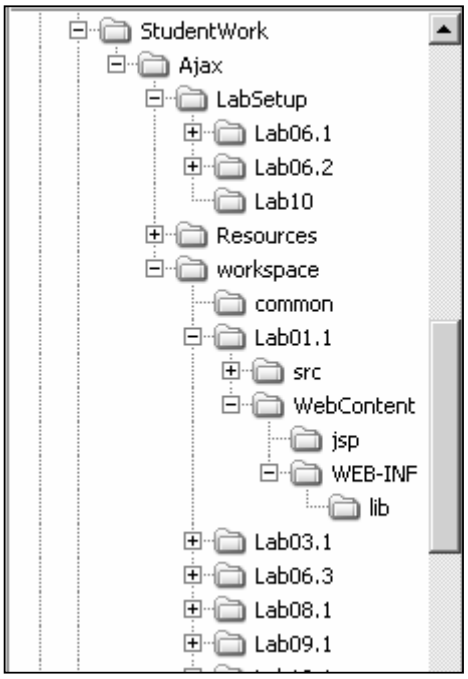
Notes:

- The setup zip will either be given to you on a CD or supplied by your instructor
  - The nnnn stands for the version number of the course
- The CD may also contain additional folders with extra content
  - **InstallFiles**: Extra software install files (e.g. simple editor)
  - **Resources**: Documentation, specifications, etc.
  - Which extra files are supplied will vary based on space limitations

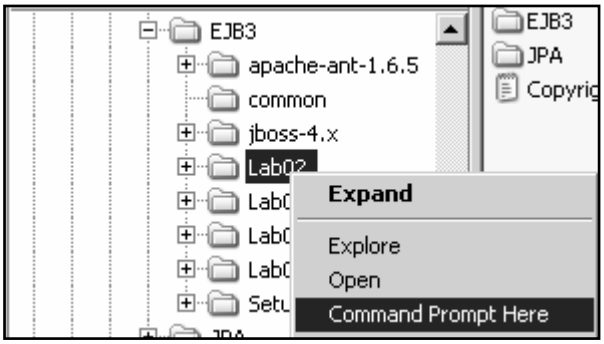
Lab Directory Structure

Lab

- **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



- Some of the labs are Web applications
  - These labs will include the Web related directories
  - Labs that are not Web applications will not include the Web related directories
- To make it easier to open command windows in the lab dirs, we've included a utility in the setup that allows you to right click on a folder to open a command prompt in it – To install this:
  - Go to the Resources/CommandPromptHere folder
  - Right click on the .reg file there, and select Merge
  - Once you've done that, right clicking on a folder will show a Command Prompt Here selection in the context menu - Select that to open a command prompt in the directory

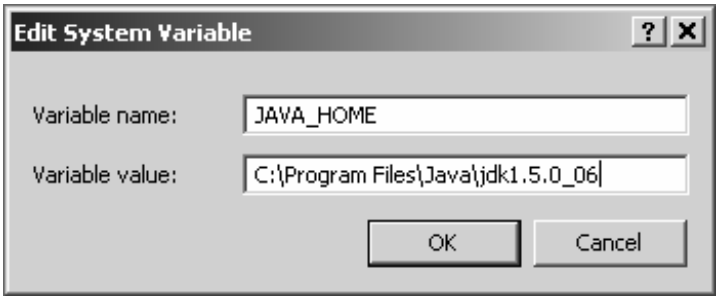


Setup Environment

Tasks to Perform

- Make sure that you have Java 5 installed
  - Likely installed in a directory like *C:\Program Files\Java\jdk1.5.x*
- Make sure the **JAVA_HOME** environment variable is set
  JAVA_HOME=C:\Program Files\Java\jdk1.5.0_06
  (The example shows the setting for Java 5 version 1.5.0_06)
- Add entries to the ***PATH*** environment variable
  - *PATH* should include ***%JAVA_HOME%\bin*** (for the JDK)
- Make sure that Tomcat is installed – likely in a directory such as **C:\apache-tomcat-6.0.10**
  - 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-60.cgi**

- Set the environment variable permanently via the Control Panel by going to:
  - **System | Advanced tab | Environment Variables**



- The value for JAVA_HOME is based on your installing Java 5 in the normal location
  - If you've installed it in a different location, then adjust the value accordingly
  - It is needed to run our client test code and to run Ant

The Eclipse Platform

- **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

Notes:

- The Workbench sits on top of the Workspace and provides visual artifacts that allow you to access and manipulate various aspects of the underlying Workspace resources.

Eclipse and Web Projects

- 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

Notes:

- Eclipse also provides Static Web projects that can be used when you aren't generating dynamic content and don't want any of the overhead associated with dynamic Web content
- When you create the project, you can set many properties for it, including:
  - Build path, project references, default server for deployment, and the application's context root
  - Uses the build path value to resolve references in the compiling code
- The `web.xml` file holds many additional settings
  - Servlets to be run in project
  - Initial (welcome) pages and error pages
  - Environment values and JNDI references to resources made in servlets/JSP's

Web Project Organization

- 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

- The src folder contains all Java source
  - Servlets, supporting classes, JavaBeans
  - When you compile these classes, the compiled output is put in the WEB-INF\classes directory

- The WebContent folder contains all Web resources
  - HTML files, JSP files, image files, etc.
  - Only content in this folder (or a sub-folder) can be accessed in the Web application

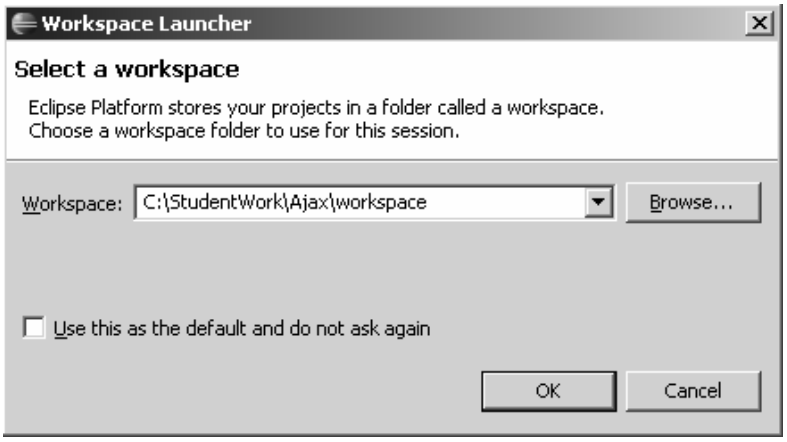
- HTML and JSP pages must be correctly placed relative to the context root
  - Media files, JAR files, loose class files and other resource libraries must also be correctly placed

- Both the Web and Java EE views are useful for managing Web projects
  - Web perspective contains HTML/JSP oriented views not in J2EE perspective

Launch Eclipse

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**



- If Eclipse was installed elsewhere, adjust the paths to the Eclipse executable accordingly
  - You can put a shortcut to this executable on your desktop

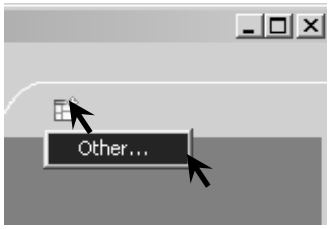
Notes:

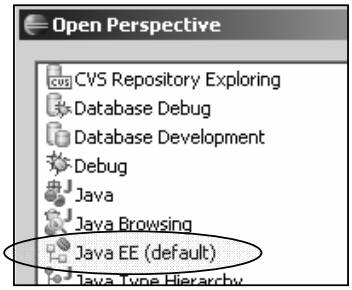
Workbench and Java EE Perspective

Lab

Tasks to Perform

- In the welcome screen, click the **Workbench** icon (see notes)
- 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)





Notes:

Change Warnings Level

- Eclipse gives warnings in many situations that are not an issue in our labs, and can be distracting *
  - We'll change the warnings preferences to remove these
  - This will allow you to see more important errors and warnings more easily

Tasks to Perform

- Go to **Window | Preferences | Java | Compiler | Errors/Warnings**
  - Go to the "**Potential programming problems**" section, and change the settings for "**Serializable class without serialVersionUID**" to "**Ignore**"
  - Go to the "**Generic types**" section, and change the settings for "**Unchecked generic type operation**" and "**Usage of a raw type**"to "**Ignore**"

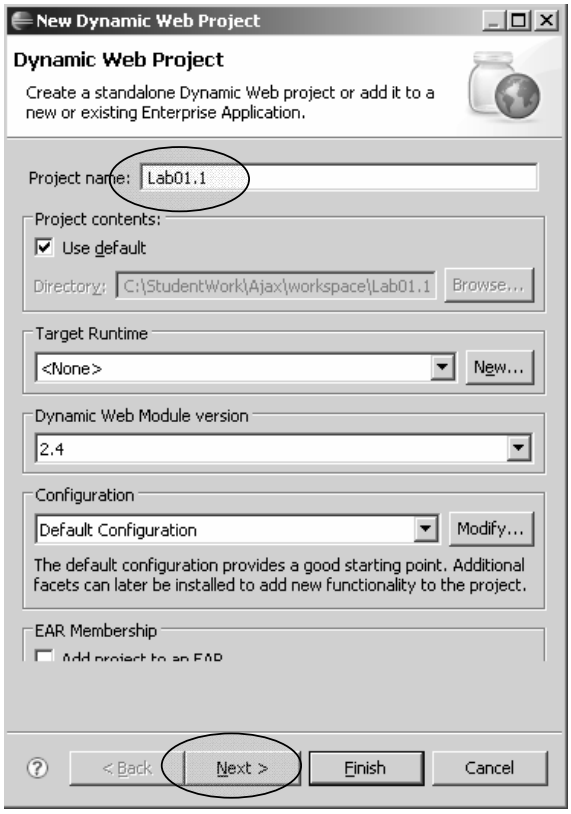
Notes:

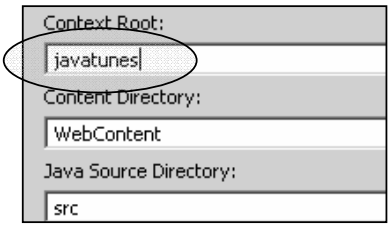
- The following examples are the types of things that Eclipse will warn about that can be more distracting than helpful in our labs
- We use collections that are not type specific (e.g. do not use Java 5 generics)
  - We have written many of our utility classes to be useful in both Java-5 in non-Java 5 environments, so we use the earlier style collections for backwards compatibility
  - This is a situation that is very common
  - Eclipse will warn you of every use of these collections
- We turn off some of these warnings to make important errors and warnings more visible

Create a Web Project

Tasks to Perform

- Create a new Dynamic Web project (as at right)
  - **File | New | Dynamic Web Project** *
  - Call it **Lab01.1 ***
  - Click **Next**
- In the next dialog, change the Context Root to **javatunes** (**all lower case** - as below)
  - Click **Finish**





Notes:

- There are multiple ways to create a new project:
  - Click on the "New Wizard" button in left side of the toolbar
  - Right click in the Package Explorer View, select New → ...
  - etc.

- We supply a skeleton Web project for you in the workspace\Lab01.1 directory
  - When you create a Web project in Eclipse, it just creates the project using the existing files we provide
  - There are a number of files in the project that you'll use in later labs, e.g. Java source files, HTML files, and the *web.xml* deployment descriptor

- **Note** - There may be errors in the project - don't worry about them
  - They are due to the classpath, which we'll deal with shortly

Add the Servlet Jar to the Classpath

- We're compiling servlets so need to set the classpath *
  - **Each project you create** will require you to add the servlet jar to the build path like this

Tasks to Perform

- **Right click** on the Lab01.1 project in Project Explorer and select **Build Path | Configure Build Path**
  - Select the **Libraries Tab**, and click **Add External Jars**
  - In the dialog box that comes up, go to the Tomcat lib directory, which for Tomcat 6.0 * is: **C:\apache-tomcat-6.0.10\lib**
  - Select the **servlet-api.jar** file, click **OPEN**, then **OK**
  - Your project should now be created, and everything should compile cleanly now
  - If you have any compilation errors, check your project classpath, or ask your instructor for help

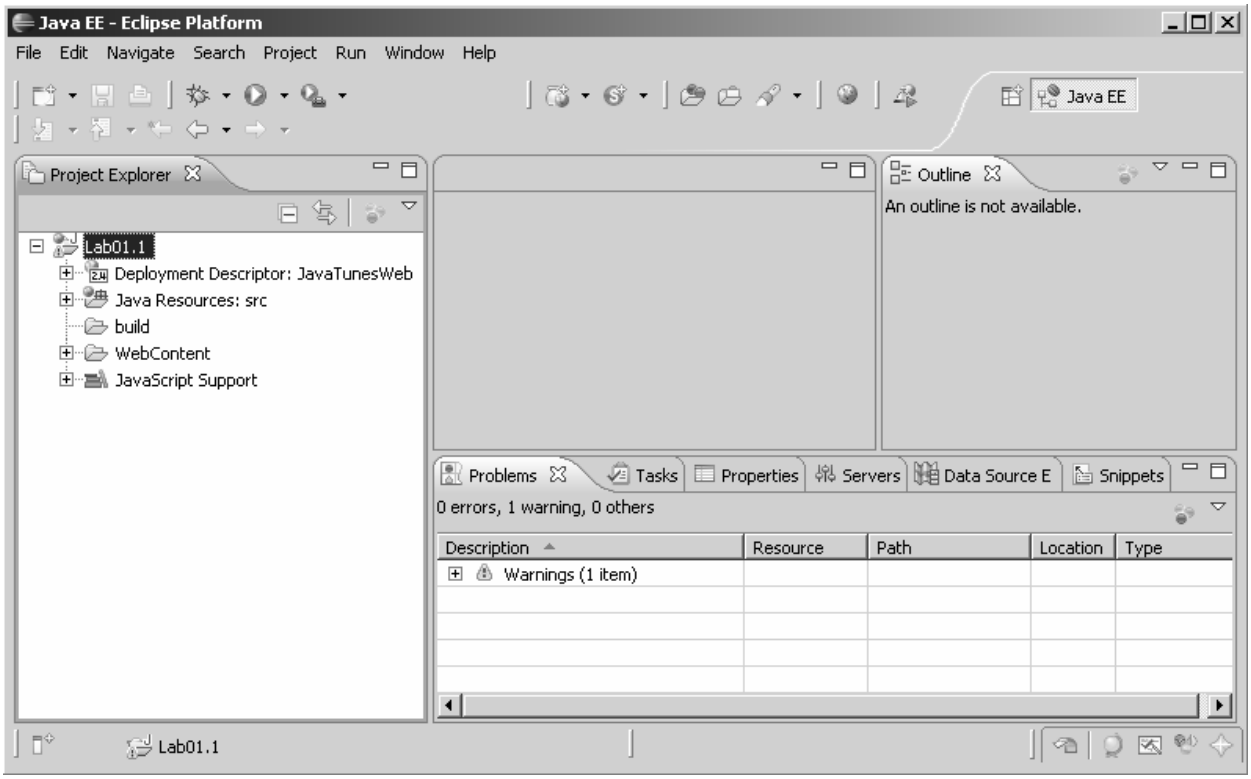
Notes:

- The directory for the Tomcat lib in the slide assumes that Tomcat 6.0 is installed in C:\
  - If you've installed it elsewhere, then adjust this accordingly

- For Tomcat 5.5, theTomcat lib directory is
  - C:\apache-tomcat-5.5.17\common\lib
  - Again, this assumes that Tomcat is installed in C:\
    - If you've installed it elsewhere, then adjust this accordingly

- Eclipse requires you to set the classpath to include any libraries that are used by your Java source code
  - Our Java source uses the servlet API library, which we are adding in this step

The Java EE Perspective

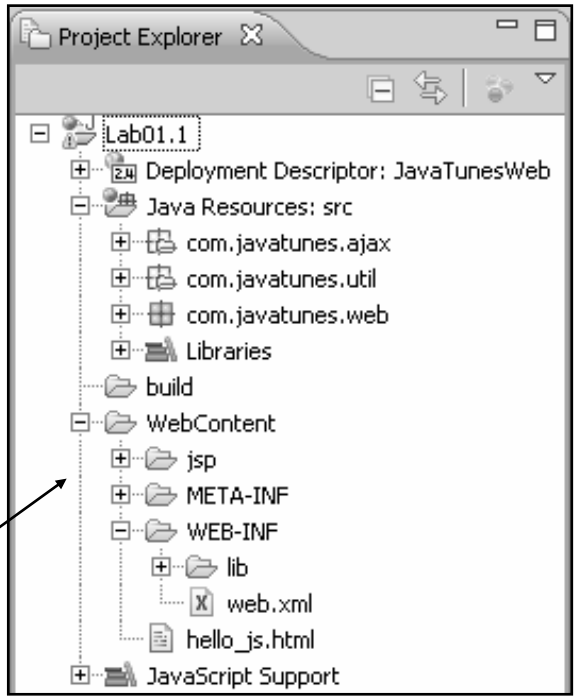


Notes:

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

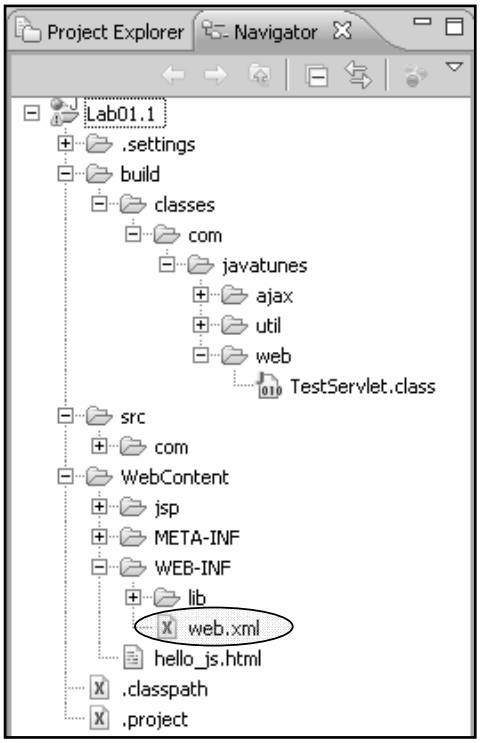


Notes:

Navigator View

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



Notes:

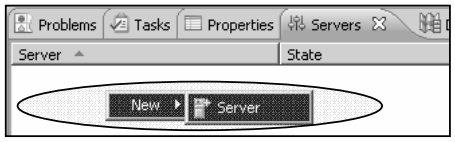
Creating a Server

Lab

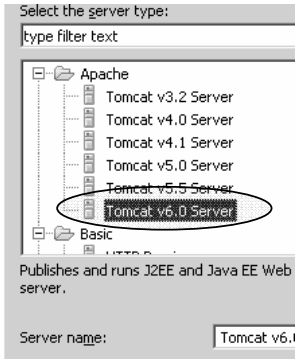
- We will use the Tomcat server to run our Web applications - to do this, we first need to create a server in Eclipse *

Tasks to Perform

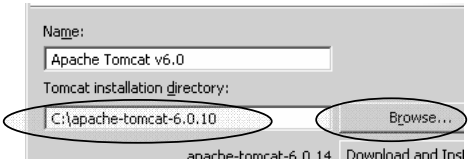
1. Go to the Servers view, right click, and select **New | Server**
2. In the next dialog, select **Apache | Tomcat V6.0** * and click **Next**
3. In the next dialog, browse to your **Tomcat install directory**, and click **Finish** *



1



2



3

- If you use a Tomcat version other than 6.0, then select the appropriate version in the dialog where you choose the server

- Eclipse for Java EE has support for deploying Web applications to a configured server
  - It also has support to start and stop the servers from within Eclipse

- If you click Next instead of Finish in Step 3, you'll come to a dialog that lets you configure the project to run on the server
  - We are going to do this slightly differently

Notes:

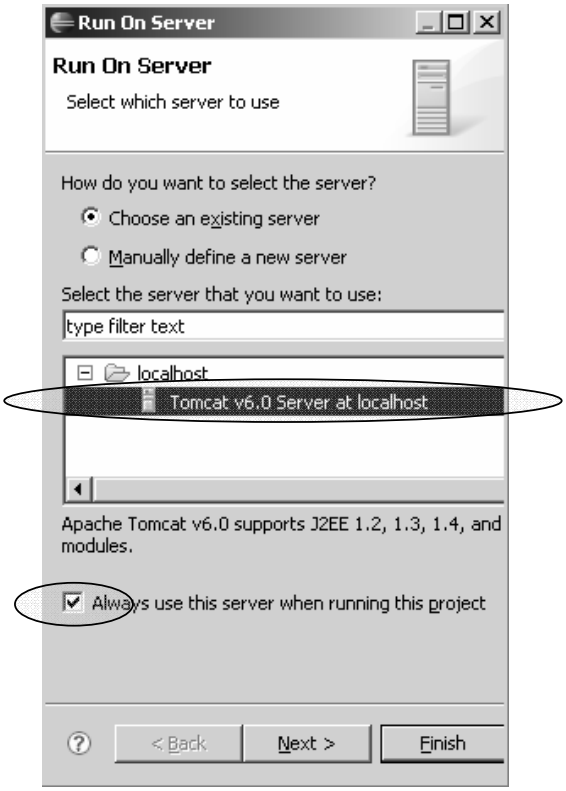
Deploying an Application

Lab

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 v6.0** server
- Also select **Set server as project default**, as we will always use this server
- Click **Finish**

- Note that running a Web app on the server will automatically start the server



Notes:

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

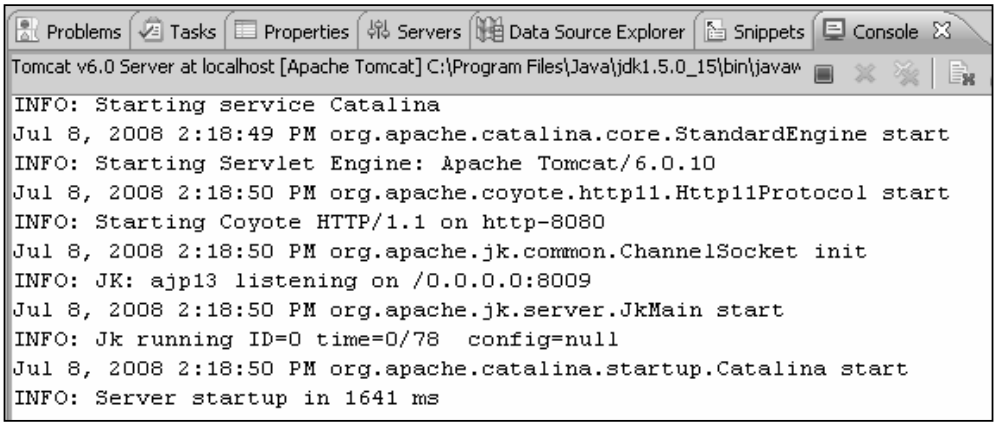
Notes:

- The lab deployed a simple Web application that we provided
  - The Web root context is javatunes – which is why you browse to it using localhost:8080/javatunes
  - It includes a very simple HTML file (*hello_js.html*)
  - The *web.xml* file (found in the WEB-INF dir) specifies *hello_js.html* as the default file for the Web app – which is why you see it when browser to localhost:8080/javatunes in the browser

- Eclipse deployed this web app for you in the Tomcat server

Server Startup

- 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



Notes:

Important Things to Note for Eclipse

- 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
- Each time you **create a new project**, you'll need to **add the servlet API jar** to the project build path so you can compile
  - None of the other configuration we did in this lab needs to be done again – those are global across all projects
- 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

Notes:

- Take note of the Lab instructions

- Any lab that starts out saying that it will be done in a new directory will require you to create a new Java project
  - You'll create the project with the name of the directory specified in the lab
  - You'll do it in the same way as you created this project

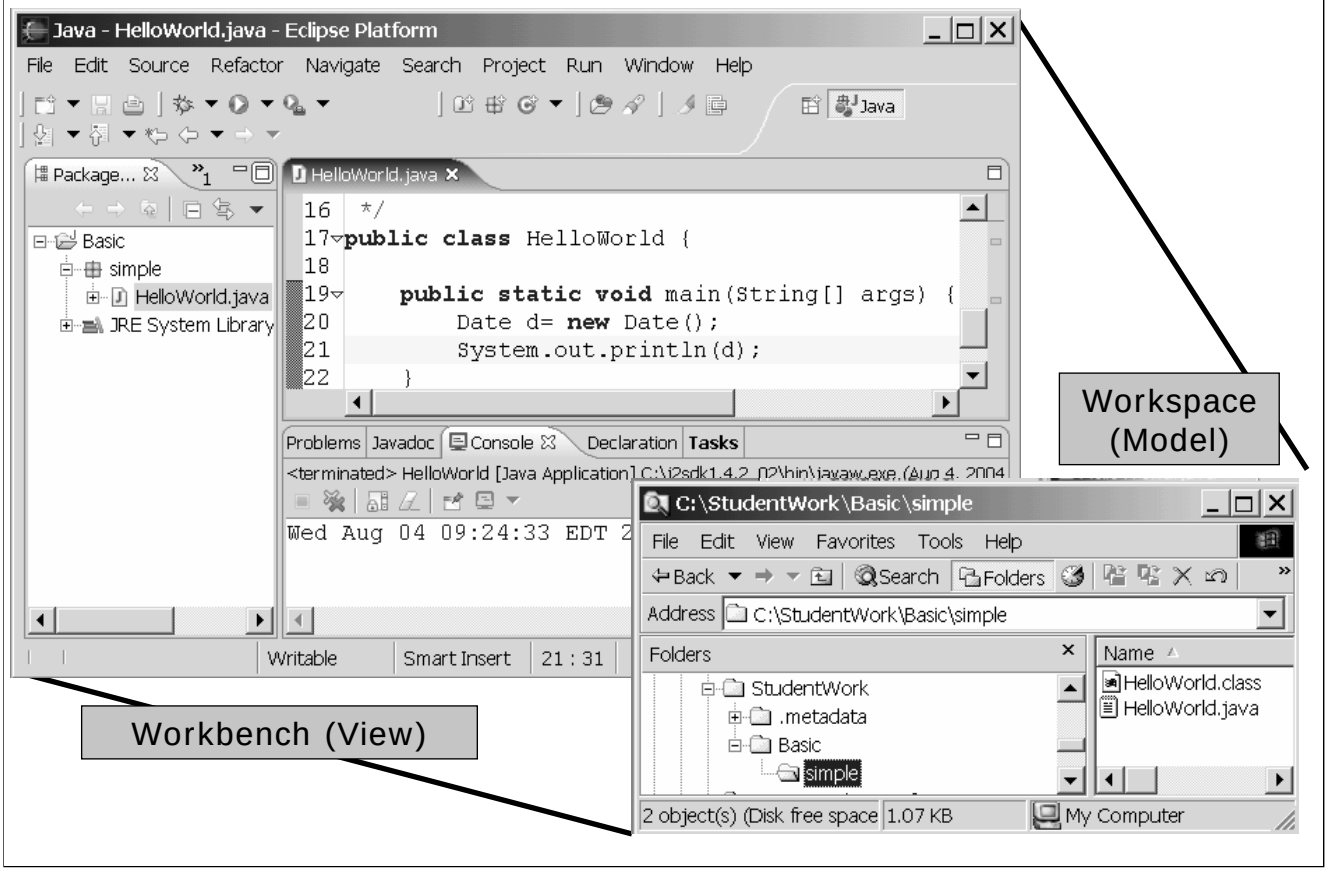
- Any lab that has you copy files from the setup to your working directory will require you to refresh the project as described above
  - This allows Eclipse to become aware of the new files.

The Eclipse Paradigm

- 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

- The physical directory structure for the Workspace can be found in the "workspace" directory
  - The default location is directly under the Eclipse home directory
  - You can specify a different workspace location when you start Eclipse
- It is even possible to set up multiple workspaces (with corresponding Workbenches). Simply create a folder to house the additional workspace, and write a script that uses the Eclipse executable file and supplies the 'data' argument with the location of the workspace directory to load:
  - c:\eclipse\eclipse.exe -data other_workspace_folder
- A Perspective is basically a collection of views that are focused on a given task
  - They provide different tools to work with the resources
  - For example, the debugging perspective has views open for debugging, such as: Active Threads, Variables, Breakpoints, etc.
  - There are perspectives for Java development (Java Perspective), and so on
  - What perspectives are available depends on what version of Eclipse you have, and what plugins you have installed

Workbench and Workspace

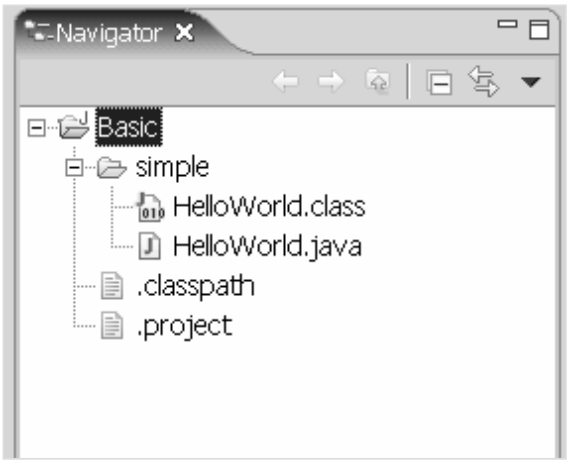


- We use the terms Model and View here in the same sense as when talking about Model-View-Controller (MVC)
  - The Model is the actual data (the files)
  - The View is the Eclipse Workbench

Notes:

Navigator View

- 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.



Notes:

- The Navigator view is a commonly used views
  - It provides a file-based organization of the various resources
  - There are other views that give application-based, and project-based organization of data

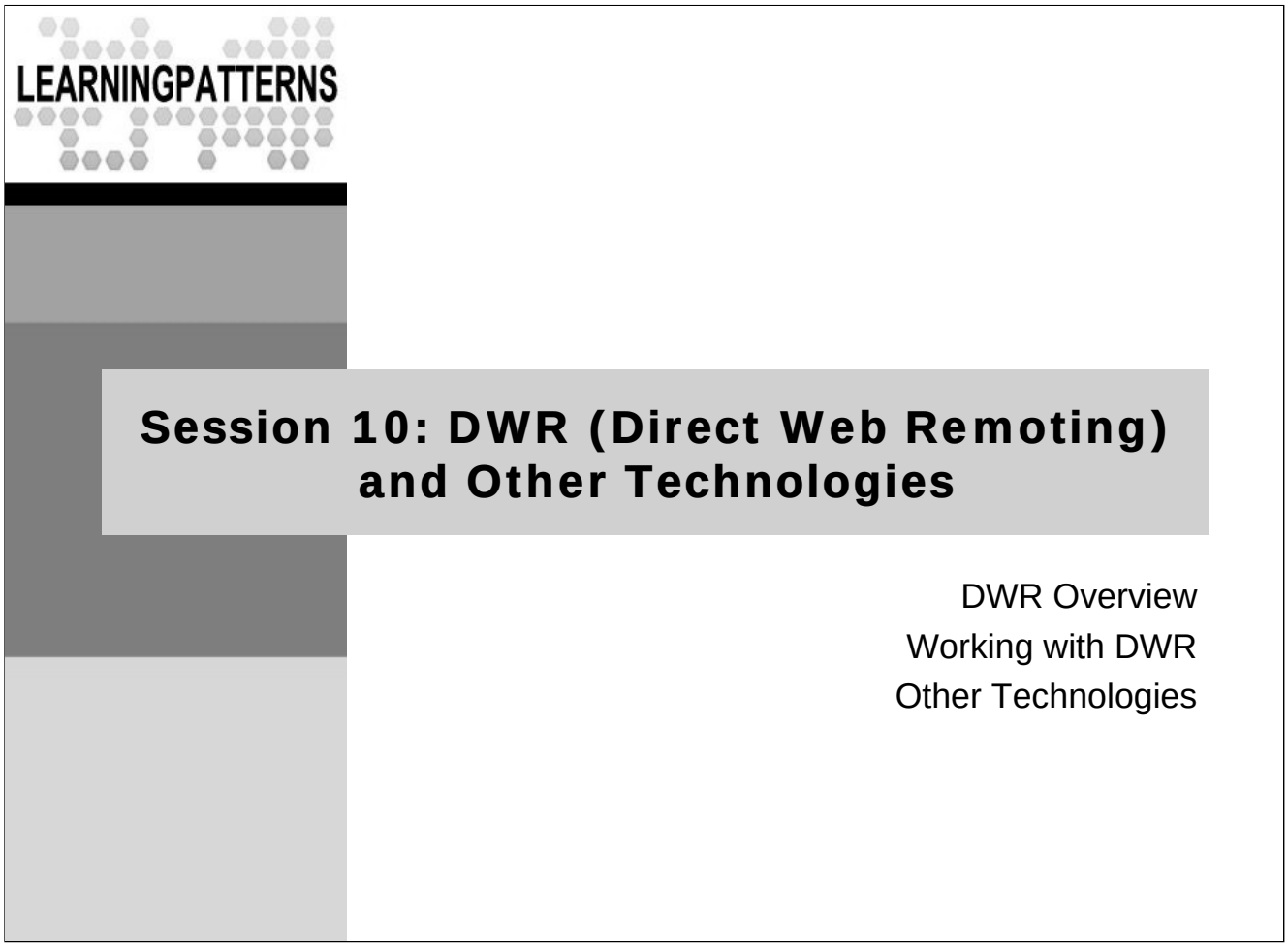
Editors

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

HelloWorld.java

```
 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     }
```

Notes:

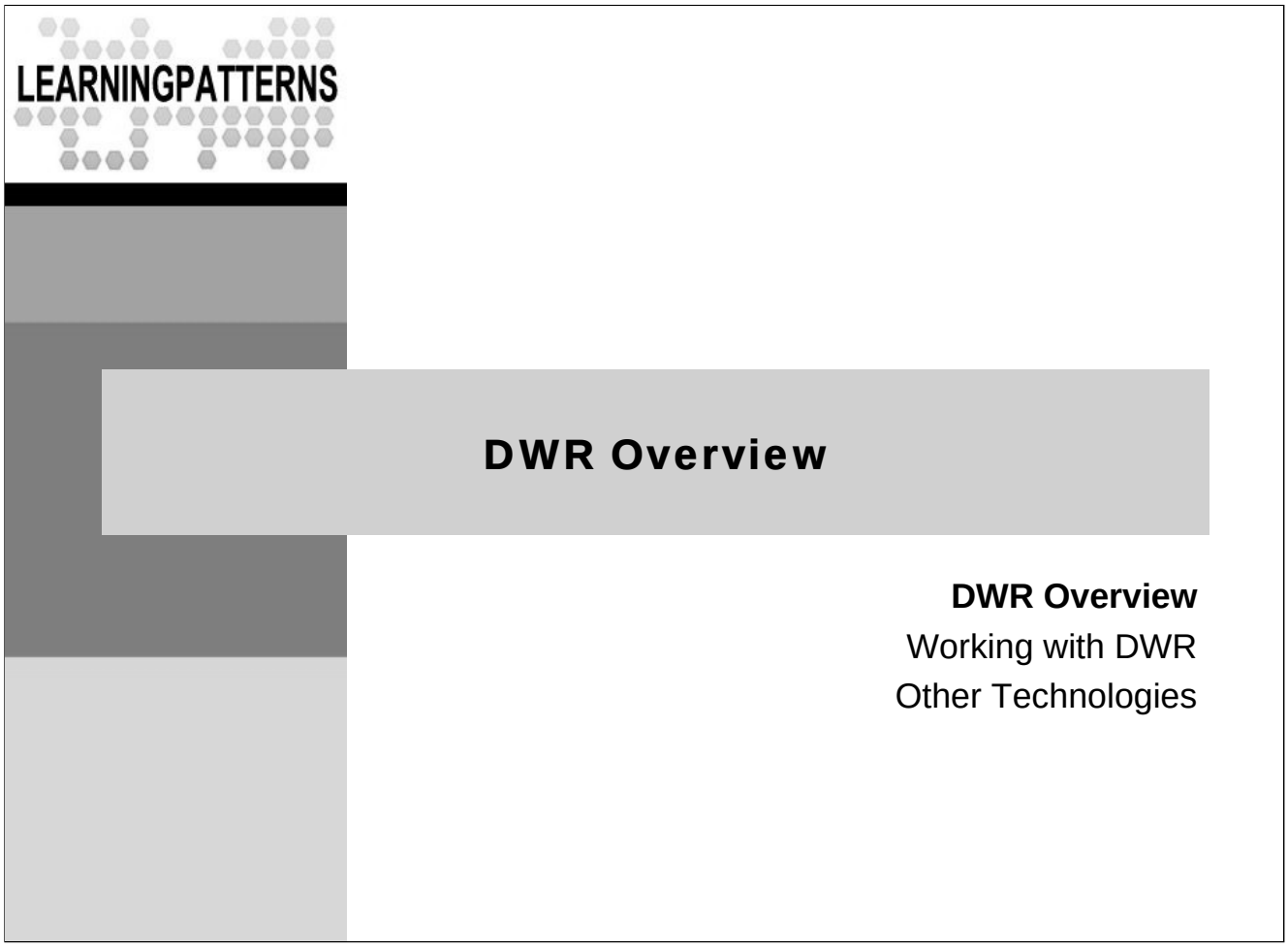


Notes:

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

Notes:



Notes:

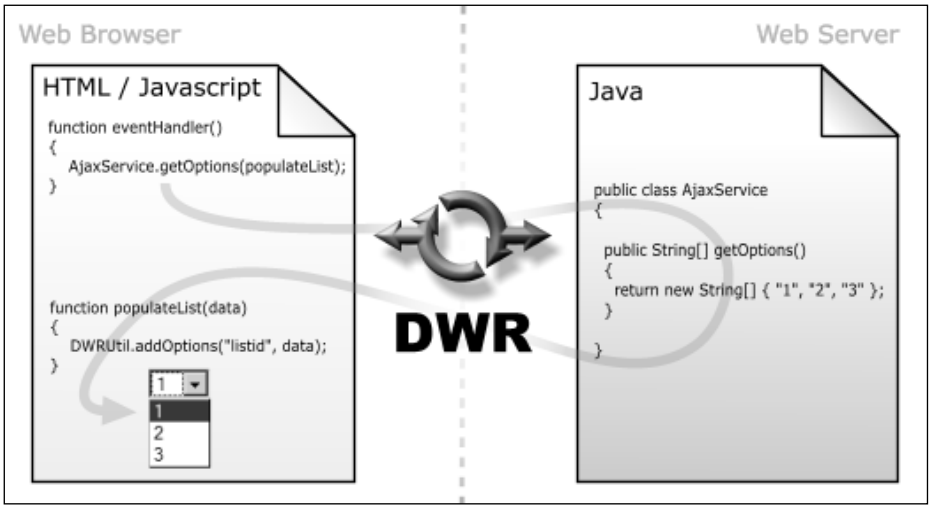
What is DWR?

- **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

- The servlet is provided with the DWR runtime
  - It runs in a normal Web application on your server
- This method of remoting functions from Java to JavaScript gives DWR users a feel much like conventional RPC mechanisms like RMI or SOAP, with the benefit that it runs over the web without requiring web-browser plug-ins.

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()*



- Because Java is fundamentally synchronous, where Ajax is asynchronous, you must provide some way of handling the response
  - We've seen this before when we worked with Ajax directly, and provided a callback function for the XHR object to call when the data was ready to be processed
  - DWR uses a similar solution
  - When you call a remote method with DWR, you provide DWR with a callback function to be called when the data has been returned from the network
  - In the example above, the callback function is *populateList()*

Notes:

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**
  - 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

Notes:

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>
```

- The servlet mapping entry uses what is called prefix mapping
  - It will match any request url that includes /dwr in it, and send that to the DWR servlet
  - For example, http://localhost:8080/javatunes/**dwr** will be forwarded to the DWR servlet, which will bring up the general DWR test page
  - http://localhost:8080/javatunes/**dwr**/test/SearchUtility will also be forwarded to the DWR servlet, and bring up the test page for the SearchUtility class (we'll see the configuration for this next)
  - The example above is for DWR 2.x
    - For DWR 1.x, the class name of the servlet is *uk.ltd.getahead.dwr.DWRServlet*

- The debug init parameter shown above enables test mode for DWR
  - This generates test pages for each of the allowed classes, as we'll see later
  - It should not be used in live deployments for security reasons

dwr.xml Configuration File

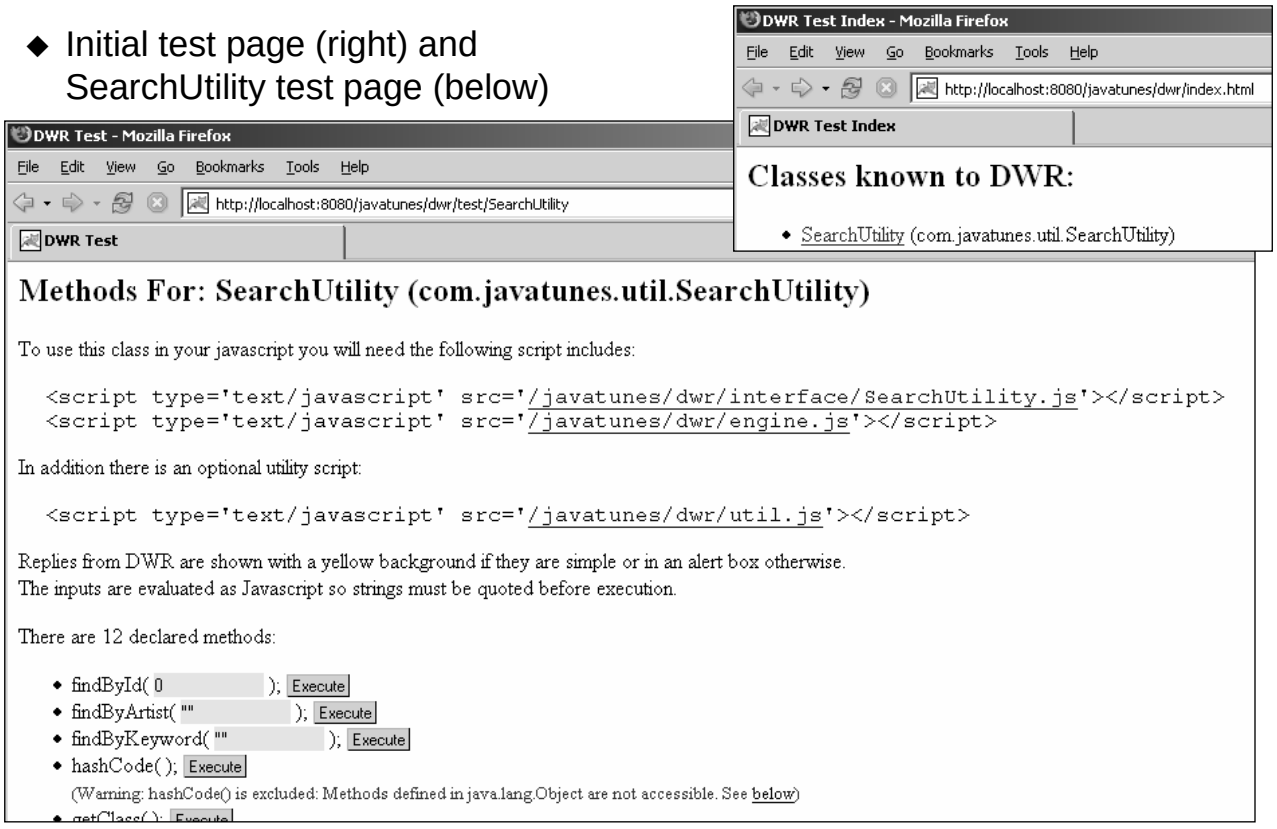
- *dwr.xml* is the configuration file for DWR – by default in WEB-INF
  - The example below includes some of the most common elements
  - **<allow>**: defines which classes DWR can create and convert
  - **<create>**: Needed by each class on which we execute methods
  - **<convert>**: Specifies conversions 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>
```

- DWR is not allowed to create or convert any classes by default
  - You have to enable each specific class that you want to have access to via DWR
- *<create>* specifies how DWR creates the element, with a number of different supported methods
  - **new**: Which uses the Java 'new' operator.
  - **none**: This does not create objects. See below for why. (v1.1+)
  - **scripted**: Uses a scripting language like BeanShell or Groovy via BSF.
  - **spring**: Gives access to beans through the Spring Framework.
  - **jsf**: Uses objects from JSF. (v1.1+)
  - **struts**: Uses struts FormBeans. (v1.1+)
  - **pageflow**: Gives access to a PageFlow from Beehive or Weblogic. (v1.1+)
  - There are different parameters, such as include/exclude to specify which methods are accessible
- <convert> specifies how DWR will convert your own classes (most Java classes are built-in)
- You can also use annotations in your Java file for DWR 2.x
  - Rather than using the dwr.xml file

Running the Test Pages - <webapp>/dwr

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



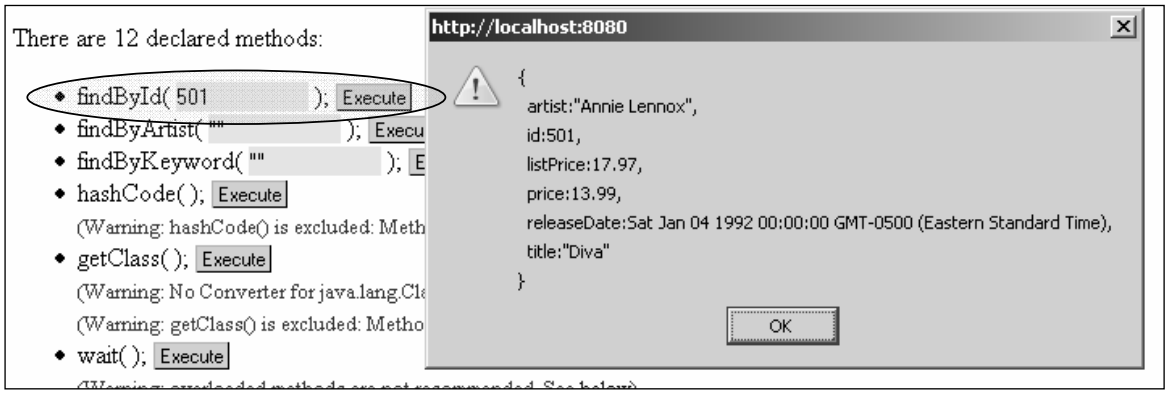
- We've included the DWR functionality in the slides in our javatunes Web app
  - Browsing to **http://localhost:8080/javatunes/dwr** brings up the initial test page
  - Clicking on the SearchUtility link brings up the test page specific for that class which allows you to test all the methods in SearchUtility
  - The test pages are generated from dwr.xml and the Java classes

- The URL above assumes that the server is running on the local host on port 8080
  - Depending on your server setup, this may vary

- SearchUtility is a utility class supplied in the labs that has three methods defined in the class
  - *static List findByArtist(String artist)*
  - *static List findByKeyword(String keyword)*
  - *static MusicItem findById(Long id)*

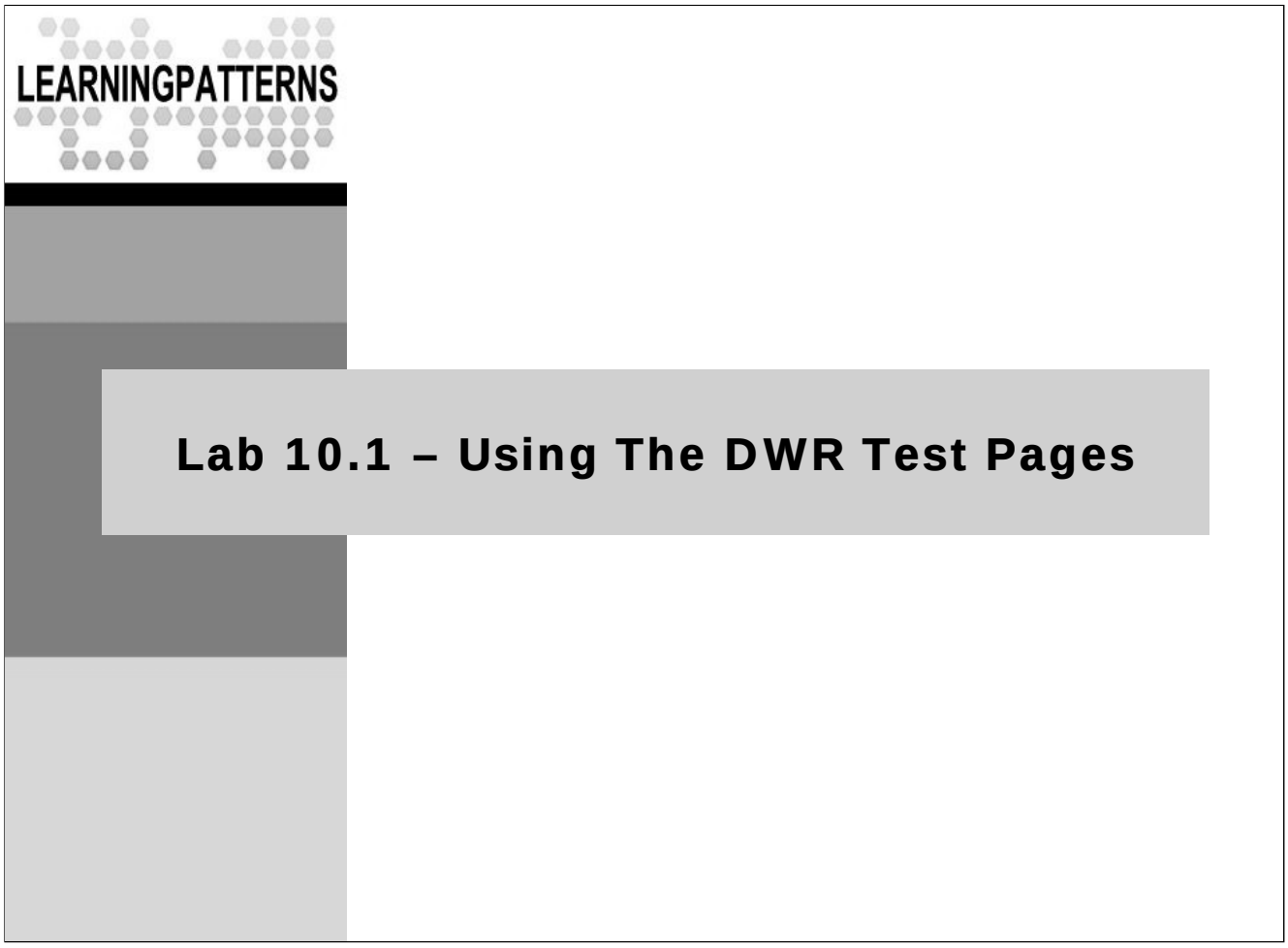
Running The Test Pages

- 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 output shown is the properties of the MusicItem with id=501 that's returned from the call to *findById*
  - The JavaBean converter just reads the properties and gets their values

Notes:



Notes:

Lab 10.1 – 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

Notes:

Lab Preparation

- 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)
  - Remember to set the context root to **javatunes**
  - Remember to add the ***servlet-api.jar*** as a Library
  - You'll see some errors in *dwr.xml* which you can ignore, as we'll fix them as part of the lab

Notes:

- If you are using a project based environment like Eclipse, you'll probably need to do whatever configuration you did for the first project you created
  - For example, modifying the classpath by adding in jars or libraries
  - If you need to refresh your memory, go back to the detailed descriptions for the first project you created

Lab Files Overview, web.xml Config

- 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

Notes:

- You've seen most of the JavaTunes related functionality in previous labs

- *web.xml* is used to configure the DWR invoker servlet
  - Review the example web.xml in the student manual slides
  - It uses prefix mapping, and any URL that has dwr in it will now be handled by the DWR invoker servlet
  - For example, the request to http://localhost:8080/javatunes/**dwr** ends in the dwr prefix, and will be routed to the DWR invoker servlet

Configure DWR

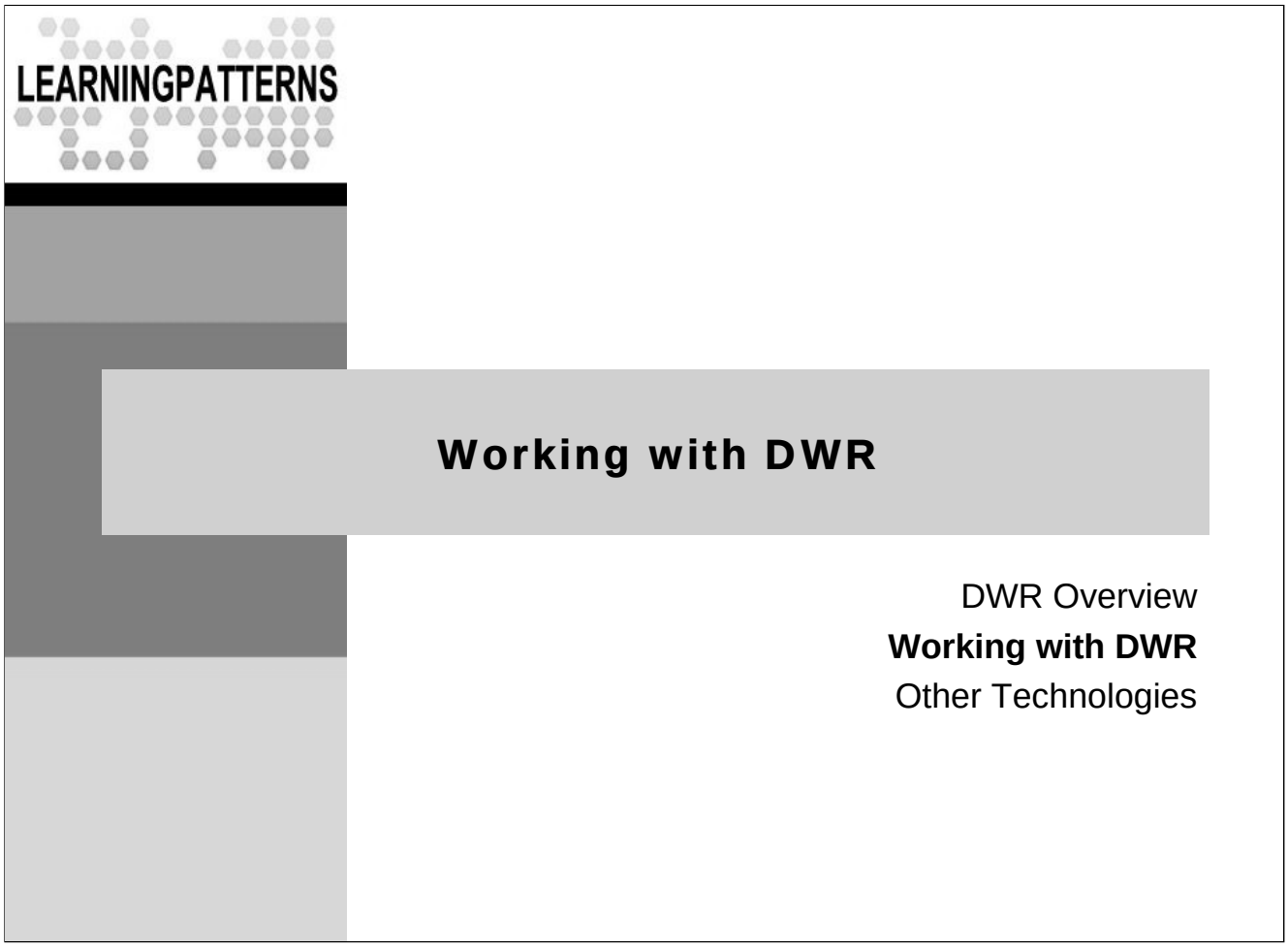
Tasks to Perform

- 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

- *dwr.xml* is used to configure what classes will have methods exposed remotely via DWR
  - Review the example dwr.xml in the student manual slides

- We include the dwr.jar file in the WEB-INF\lib directory
  - This is all you need to do to install DWR
  - Everything else is generated dynamically

- The *SearchUtility* methods are invoked on the client through a JavaScript class called *SearchUtility*
  - This is configured with the *javascript="SearchUtility"* attribute to the *<create>* element
  - You can call the JavaScript class whatever you want, though it makes sense to stay consistent with the Java class name



Notes:

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>
```

- The JavaScript proxies make invoking the Java code on the server almost as easy as invoking a normal JavaScript function
  - You simply invoke the function on the JavaScript proxy, and process the response data, which can be in the form of objects
  - All the marshalling and unmarshalling of data, as well as all the underlying network/XMLHttpRequest programming is taken care of for you
  - There is just a bit of complexity needed to deal with the asynchronous nature of Ajax invocations
- The JavaScript proxies are defined in a JavaScript file with the name
  - *[WEBAPP]*/dwr/interface/[JAVA_CLASSNAME].js"
- Using these proxies also requires the use of the DWR core library
  - *[WEBAPP]*/dwr/engine.js
- We've also included an optional utility library in the example
- We've shown relative URLs in the example, but you can also use absolute URLs for the src attribute, for example

  `src='/javatunes/dwr/interface/SearchUtility.js'`

Using the DWR Proxies

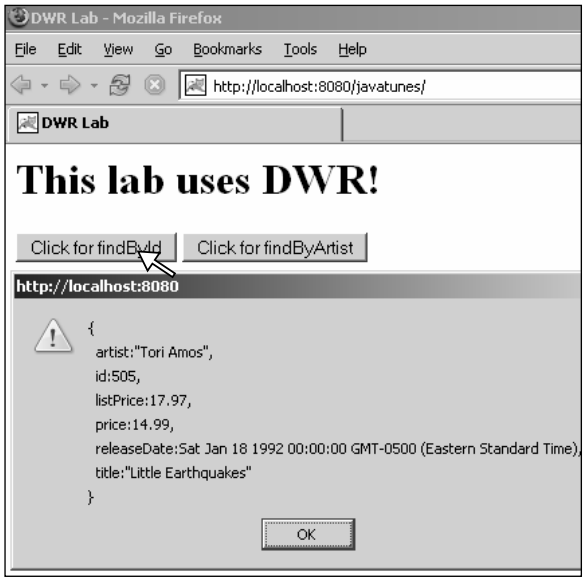
- Below, we show examples calling *SearchUtility* methods
  - Note the second argument to the methods, a callback function
  - This is added to the signature of each method, and is needed, because of the asynchronous nature of Ajax calls, for any function returning a response

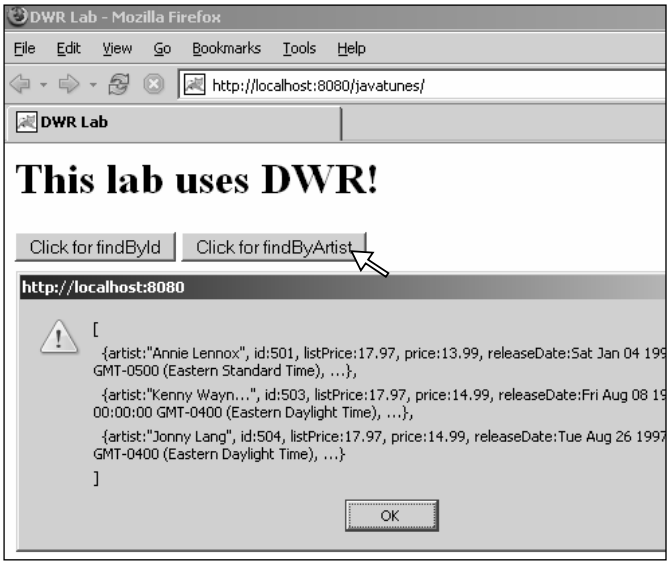
```
<form>
  <input type="button" value="Click for findById"
      onclick='SearchUtility.findById(505, displayDWR)'/>
  <input type="button" value="Click for findByArtist"
     onclick='SearchUtility.findByArtist("nn", displayDWR)'/>
</form>

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

- The methods of the *SearchUtility* class are exposed in the JavaScript proxy class
  - The methods have the same name, and take all the arguments of the original Java class
  - In addition, they all have an additional argument for a callback function
- The callback function should take as an argument the data that is sent as the response to the function call
  - The data is unmarshalled by DWR into the appropriate JavaScript classes and passed to the callback function when the response is completely received
- The *toDescriptiveString()* function is one of the DWR utility functions, and has the signature shown below
  - *dwr.util.toDescriptiveString = function(data, showLevels, options)*
  - It produces a pretty printed output of the data
  - The *showLevels* argument indicates how much detail to show
- DWR has the ability to use a call meta-data object instead of a simple callback function
  - This allows more flexibility and the setting of options such as a timeout and error handler
  - See the DWR documentation for more detail

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"





- DWR comes with a library of utility functions
  - These are not dependent on the rest of DWR
- There are functions in the following areas:
  - List and Table Manipulation
  - HTML Element Utility Functions (e.g. finding an element, setting and getting a value on an element, etc.)
  - Display Methods – which are used in the above example to display a MusicItem, and an array of MusicItems
- See the DWR documentation for more information

Notes:

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);
}
```

Notes:

- *MusicItem* is a simple JavaBean that has the following properties
  - *Long id, String title, String artist, Date releaseDate, BigDecimal listPrice, BigDecimal price*

- Since *updateItem()* returned void, we don't need to bother with a callback function to handle response data
  - We can still pass one in if we want
  - For example, we might pass in a call meta-data object with a timeout and error handler
  - Or we might want to execute some code when the response comes back and we know that the call completed
  - Of course, DWR won't pass in any data into a callback function for *updateItem*, because there is no response data

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
} );'/>
```

- ***DWREngine*** is declared in *dwr/engine.js* which must be included in your file to use DWS
- Some of the other options that DWR supports are:
  - async: Set to false for asynchronous behaviour (not recommended)
  - headers: Extra headers to add to XHR calls.
  - parameters: Meta-data that is made available through request.getParameter()
  - httpMethod: Selects use of GET or POST. Called 'verb' in 1.x
  - rpcType: Selects between xhr, iframe or script-tag remoting. Called 'method' in 1.x
  - timeout: Cancel request after X ms

  Handlers:
  - **errorHandler**: Action when something is broken.
  - **warningHandler**: Action when something breaks which can be triggered by browser bugs
  - **textHtmlHandler**: Action when an unexpected text/html page is received

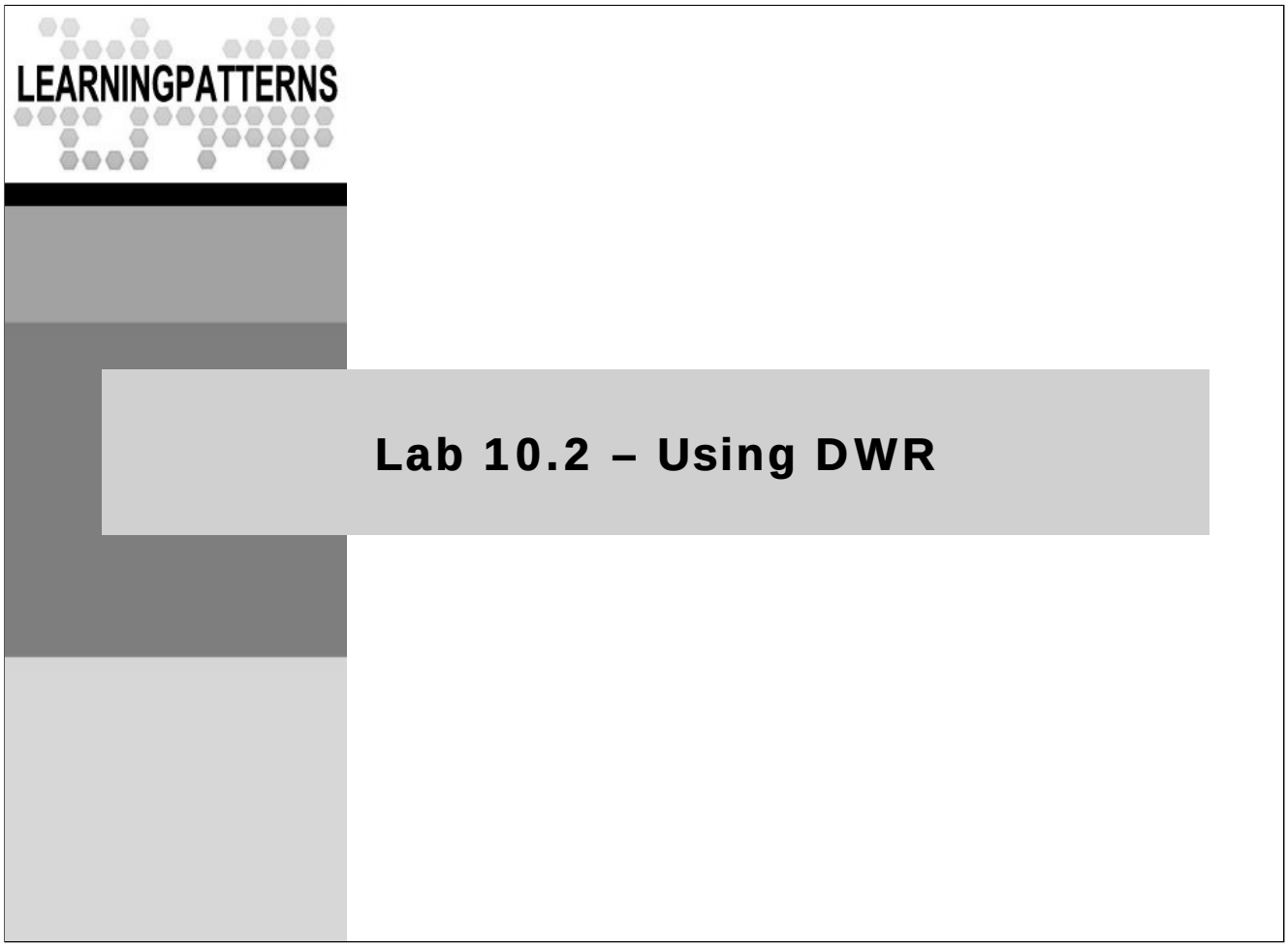
  Callback Handlers:
  - **callback**: Executed on successful call completion with a single parameter; the retrned data.
  - **exceptionHandler**: Executed when a remote call fails either due to a server exception or a data marshalling problem.

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

Notes:

- Reverse Ajax makes it easy to do things like write an Ajax based chat program
  - The server can send out chat messages to all the connected clients
- You need to exercise some caution in using Comet/long lived http
  - It's possible to thread starve a server if it's trying to handle a large number of clients that stay connected
  - The configuration for this functionality allows you to tune this



Notes:

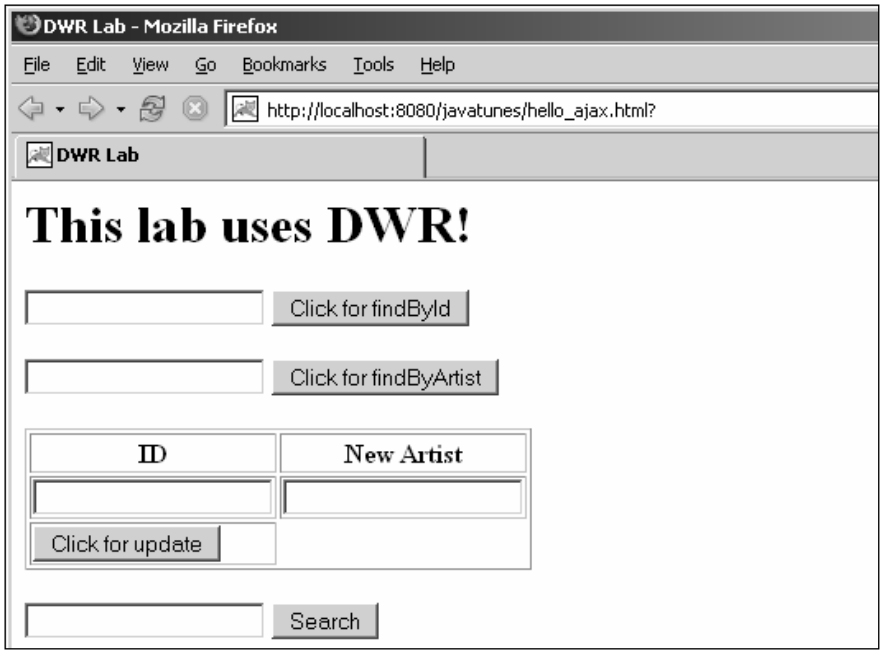
Lab 10.2 – DWR Test Pages

- **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

Notes:

Lab Overview

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



Notes:

Set up DWR on the Client

Lab

- 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 *
- **Publish** the application, browse to *hello_ajax.html*, type in the id 501, click the ***findById*** button, and look at the results

- Two of the included JavaScript files are part of the DWR framework
  - engine.js and util.js
- SearchUtility.js is generated dynamically by DWR based on the dwr.xml file, and your SearchUtility Java class
- When you call *SearchUtility.findById()*, you're calling a client side JavaScript proxy which forwards the call to the actual server side Java class
  - For findById(), you need an ID parameter, and, because of the asynchronous nature of Ajax, a callback function to handle the response data when it is passed back
- To get the value of the input field, you can use the $() utility function provided by DWR
  - If the id of the input field is "findId", then the expression $("findId").value will get the current contents of the input field
- When you enter an id and click the button, you should see the values of the MusicItem for the given id
  - This item is retrieved from the server using DWR
- *displayDWR* uses DWR utility functions to show the response data in a nicely formatted form

Use DWR Functionality

Tasks to Perform

- Next, look for the button with *value="Click for findByArtist"*, and **finish its *onclick* 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 *onclick* 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) *

Notes:

- To get the value of the input field, you can use the $() utility function provided by DWR, as you did for findById
  - Use the id for the artist input field which you can find in the code
- We showed an example of using an object to pass in DWR options in the student manual
  - You use a JavaScript literal with a callback property, as shown below:

```
{ callback: nameOfCallbackFunction }
```

Using Object Arguments - Update

- 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

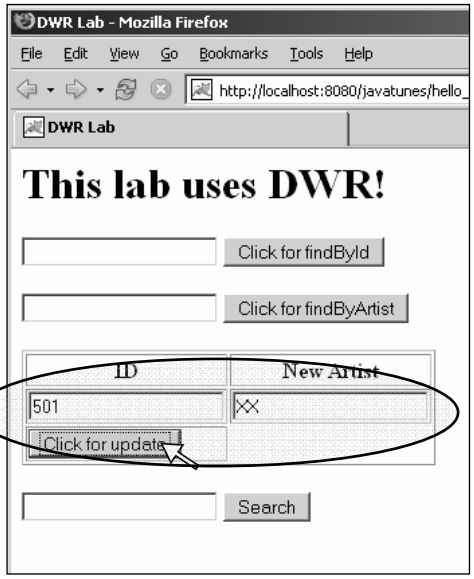
Notes:

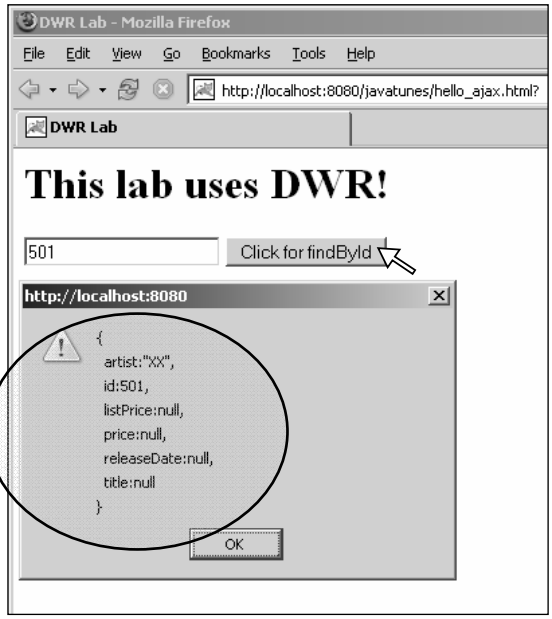
- When passing an object as an argument using DWR, you can simply construct the it win an object literal
  - Any properties that you do not fill in won't be set in the resulting Java object
  - That's OK here, we just want to test out the functionality
- Once you've run the update functionality, you can test it by running the findById functionality and looking at the changed item
  - You should see the new data in this item
  - Most of the data will be null since we didn't bother getting input for them, but you'll see the new artist
  - It's easy enough to add input fields for the additional data if you need to

Seeing Update at Work

Lab

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





Notes:

Autocomplete Suggest with DWR

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

- Article:
- http://today.java.net/pub/a/today/2005/08/25/dwr.html

Notes:

Optional Chat – Reverse Ajax Demo

[Optional] Tasks to Perform

- We will create a Web project with the DWR war that comes with DWR to test some of its features
- In Eclipse, choose the menu item **File | Import…**
  - In the next dialog, choose **Web | WAR file**, and click **Next**
  - In the next dialog, for the WAR file browse to the ***LabSetup\Lab10*** directory, select ***dwr.war*** and click **Finish**
  - **Ignore the compilation errors** - they are not important
  - Remember to add the ***servlet-api.jar*** as a Library
  - Right click on the dwr project, select **Run As | Run on Server** and **restart the server**
- Browse to *http://localhost:8080/dwr*, & click the JavaScript chat link
- Open another browser window to the same chat URL
  - If you can, use IE for one window, and FireFox for another so you're in completely different browser environments

- There are a number of other applications in the WAR
  - You can get to them all from the main page
  - Spend some time and look at them

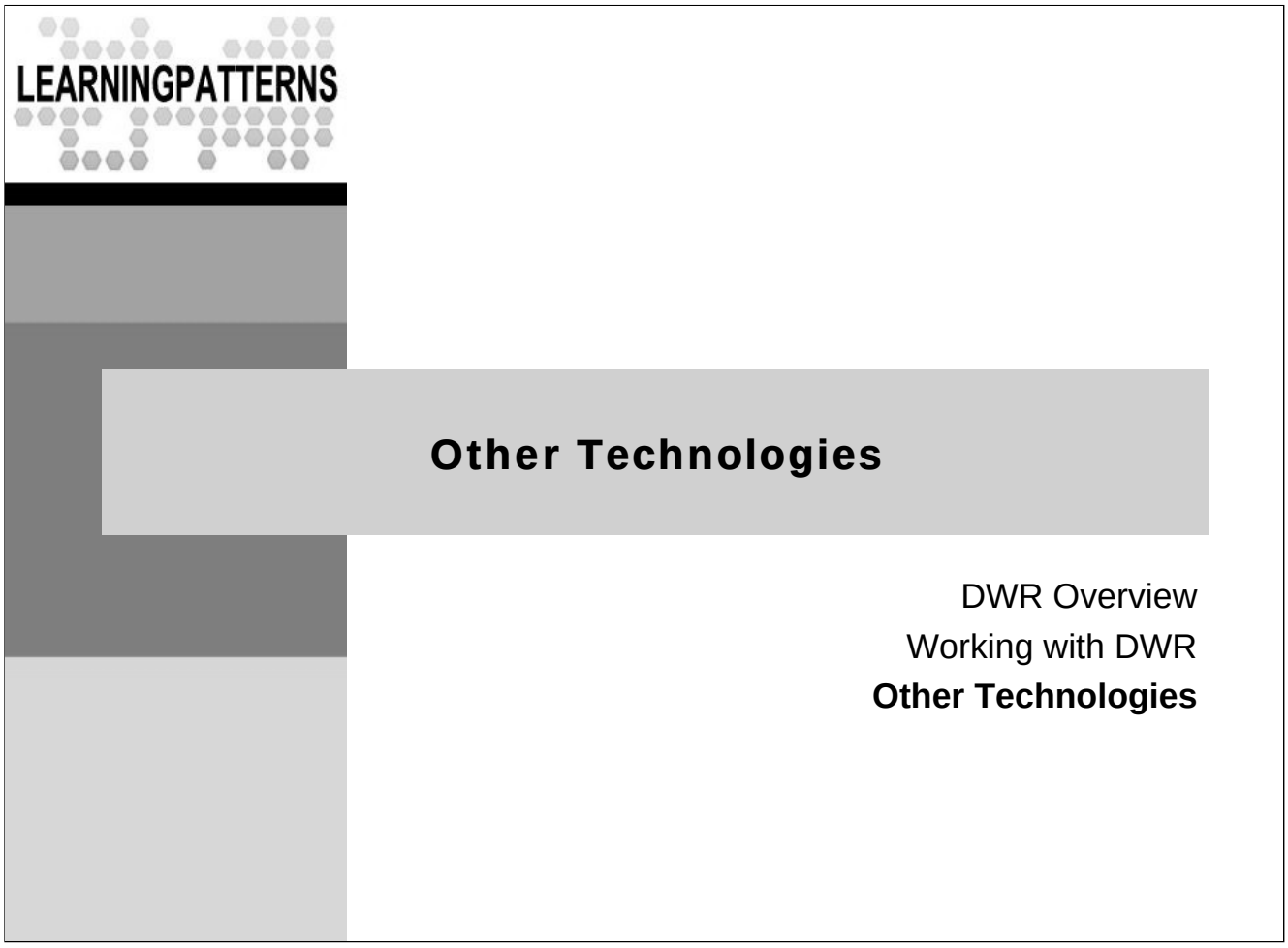
Notes:

Optional - Arguments to DWR

[Optional] Tasks to Perform

- Play with chat for a while – notice how when you send a message in one browser, it shows up in the other
  - This is using Reverse Ajax to push the messages to the clients
  - If you look at the FireBug network traffic, you can see the Reverse Ajax connection that is kept open !
- **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

Notes:



Notes:

JSON-RPC

- 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

Notes:

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 conceivable be done in a servlet also

- The JSONRPCServlet can be added to web.xml as follows:

```
<web-app>
  <servlet>
    <servlet-name>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-
  name>
    <servlet-class>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-
  class>
  </servlet>
  <servlet-mapping>
    <servlet-name>com.metaparadigm.jsonrpc.JSONRPCServlet</servlet-
  name>
    <url-pattern>/JSON-RPC</url-pattern>
  </servlet-mapping>
</web-app>
```

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>
```

- We've shown everything in as simple a way as possible in this example
  - It is possible to make it more sophisticated
  - For example, it is common to create the JSONRpcClient instance in a function called when the page loads
- In the example above, the following occurs:
  - A *JSONRPCBridge* instance is created, and put into the session scope
    - This is a Java object needed to enable JSON-RPC to work
  - The *JSONRPCBridge* is used to register the SearchUtility class
    - This is a Java call needed to export the SearchUtility methods
  - The JSON-RPC JavaScript client library is loaded (jsonrpc.js)
  - An instance of JSONRpcClient is created
    - This is a JavaScript object that is the client side entry point for JSON-RPC
  - The SearchUtility.findById method is invoked through JSON-RPC

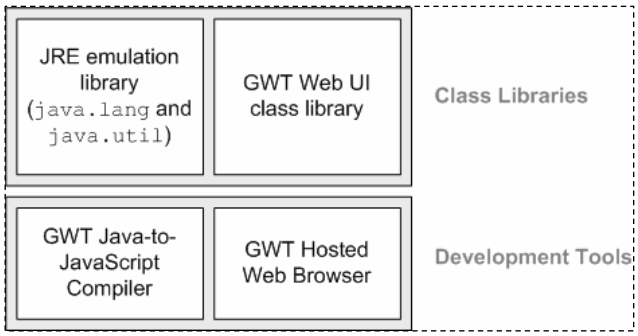
- Google Web Toolkit (GWT) -

- 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

Notes:

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



Notes:

Hello World With GWT

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

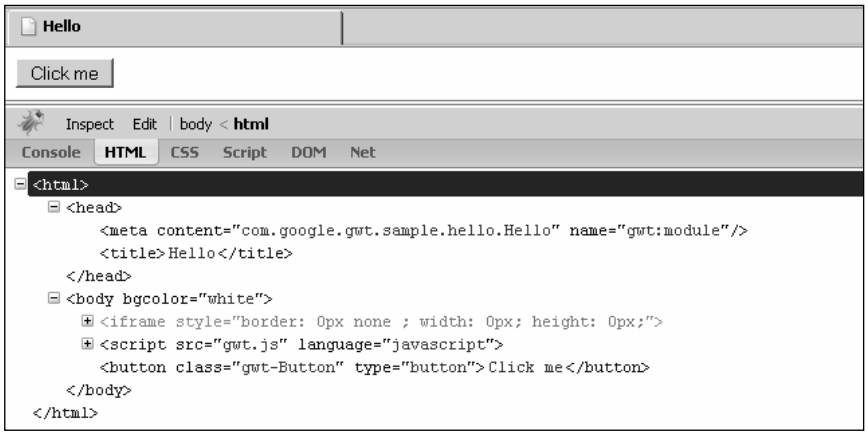
```
package com.google.gwt.sample.hello.client;

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);
  }
}
```

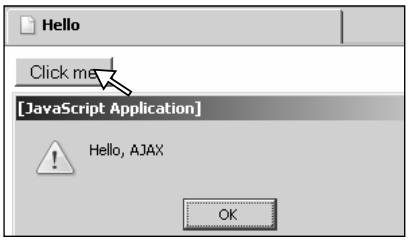
- This simple application generates a button and an event handler that displays "Hello, AJAX" when the button is clicked
  - This class is run through the GWT tools to generate the Web application

Notes:

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

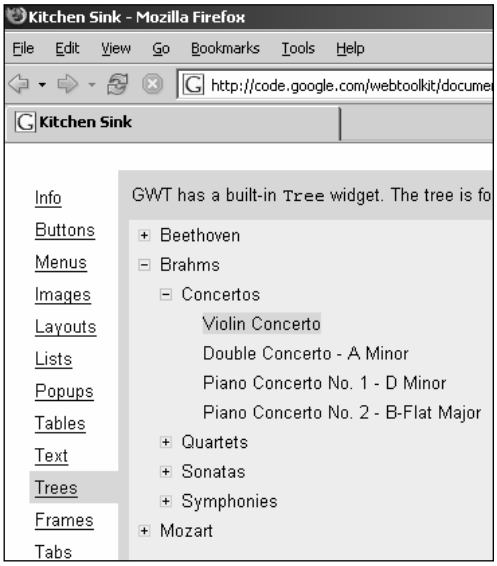


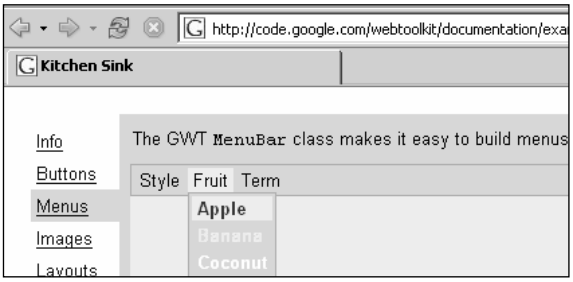


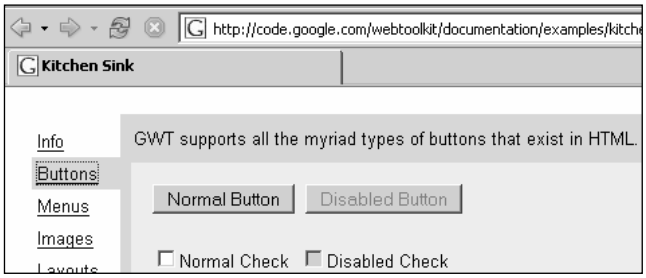
Notes:

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







- These examples are from the KitchenSink sample project
  - You can look at the sample projects at http://code.google.com/webtoolkit/documentation/examples/

Notes:

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

Notes:

Review Questions

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

Notes:

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

Notes: