

Fast Track to Servlets and JSP Developer's Workshop

LearningPatterns, Inc. Courseware

Student Guide



This material is copyrighted by LearningPatterns Inc. This content shall not be reproduced, edited, or distributed, in hard copy or soft copy format, without express written consent of LearningPatterns Inc. Copyright © 2004-8 LearningPatterns Inc.

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

Java, Enterprise JavaBeans and all Java-based trademarks and logo trademarks are registered trademarks of Sun Microsystems, Inc., in the United States and other countries. LearningPatterns and its logos are trademarks of LearningPatterns Inc. All other products referenced herein are trademarks of their respective holders.

Table of Contents: Fast Track to Servlets and JSP with JSTL

WEB APPLICATIONS WITH SERVLETS, JSP AND THE JSTL (JSP STANDARD TAG LIBRARY).....	1
WORKSHOP OVERVIEW	2
WORKSHOP OBJECTIVES.....	3
WORKSHOP AGENDA	4
TYPOGRAPHIC CONVENTIONS.....	5
LABS.....	6
RELEASE LEVEL.....	7
SESSION 1: WEB APPLICATION BASICS	8
LESSON OBJECTIVES.....	9
HOW THE WEB WORKS	10
CLIENTS AND SERVERS	11
HTML - THE <i>LANGUAGE OF THE WEB</i>	12
HTTP, ADDRESSING, REQUESTS, AND RESPONSES	13
BROWSING A SIMPLE WEB PAGE	14
JAVA EE - JAVA ENTERPRISE EDITION BASICS	15
JAVA EE - JAVA PLATFORM ENTERPRISE EDITION	16
SIMPLE WEB-BASED ARCHITECTURE	17
JAVA EE WEB APPLICATIONS.....	18
WEB APPLICATION STRUCTURE	19
LAB 1.1 - CREATE A WEB APPLICATION.....	20
A SIMPLE SERVLET.....	40
SERVLETS AND DYNAMIC CONTENT.....	41
WHAT ARE SERVLETS?	42
HOW A SERVLET RUNS	43
ADVANTAGES OF SERVLETS	44
PACKAGES AND CLASSES	45
CREATING A SERVLET - THE SIMPLEST WAY	46
A SIMPLE HTTP SERVLET	47
DECLARING AND MAPPING SERVLETS	48
LAB 1.2 - CREATE A SIMPLE SERVLET.....	49
REVIEW QUESTIONS	61
LESSON SUMMARY	62
SESSION 2: SERVLET BASICS	63
LESSON OBJECTIVES.....	64
HTML FORMS AND HTTP REVIEW	65
HTML FORMS	66
A SIMPLE SEARCH FORM IN HTML	67
HTTP REQUEST AND RESPONSE DETAILS	68
REQUEST AND RESPONSE EXAMPLE	69
GET AND POST METHODS	70
REQUEST PARAMETERS	71
HOW SERVLETS WORK	72

SERVLET INTERFACE AND SERVLET LIFECYCLE.....	73
SERVLET LIFECYCLE - INITIALIZATION.....	74
SERVLET LIFECYCLE - THE SERVICE() METHOD.....	75
IMPORTANT TYPES FOR SERVLETS	76
CLASS DIAGRAM FOR SERVLET TYPES	77
CLASS GENERICSERVLET	78
REQUESTS AND RESPONSES	79
IMPORTANT SERVLETREQUEST/RESPONSE METHODS.....	80
GETTING DATA / SENDING RESPONSES	81
GETTING DATA AND SENDING RESPONSE	82
GETTING DATA AND SENDING RESPONSE	83
HTTP SERVLETS	84
CLASS DIAGRAM FOR HTTP SERVLET TYPES	85
CLASS HTTPSERVLET	86
CLASS HTTPSERVLET	87
HTTPSERVLETREQUEST INTERFACE.....	88
HTTPSERVLETRESPONSE INTERFACE.....	89
MORE ABOUT WEB.XML.....	90
WEB.XML EXAMPLE	91
WEB.XML ELEMENTS	92
LOOK AT THE JAVADOC !!!	93
LAB 2.1 - USING CLIENT INPUT	94
REVIEW QUESTIONS	100
LESSONS LEARNED	101
SESSION 3: ADDITIONAL SERVLET CAPABILITIES.....	103
LESSON OBJECTIVES.....	104
WORKING WITH HTTPSERVLETRESPONSE.....	105
HTTP RESPONSES - STATUS AND ERRORS	106
HTTPSERVLETRESPONSE STATUS/ERROR METHODS.....	107
HTTP RESPONSE HEADERS	108
SETTING RESPONSE HEADERS	109
MIME TYPES.....	110
SPECIFYING A MIME TYPE	111
INITIALIZATION	112
INITIALIZATION OVERVIEW	113
THE SERVLETCONFIG INTERFACE	114
THE SERVLETCONTEXT INTERFACE	115
DECLARING AND MAPPING SERVLETS.....	116
SERVLET INIT PARAMETERS	117
RETRIEVING INITIALIZATION PARAMETERS.....	118
WEB APP INIT PARAMETERS.....	119
A PROBLEM WITH SEARCHSERVLET	120
NULL AND EMPTY STRING PARAMETERS.....	121
ERROR HANDLING	122
SPECIFYING ERROR PAGES	123
ERROR PAGE CONFIGURATION	124
ERROR HANDLING	125
LAB 3.1 - PERFORMING THE SEARCH.....	126
REVIEW QUESTIONS	136
LESSONS LEARNED	137
SESSION 4: JAVASERVER PAGES (JSP).....	138

LESSON OBJECTIVES.....	139
JAVASERVER PAGES (JSP).....	140
SHORTCOMINGS OF SERVLETS	141
WHAT IS A JSP?.....	142
A VERY SIMPLE JSP - SIMPLE.JSP	143
JSPS LOOK LIKE HTML.....	144
JSP EXPRESSIONS	145
JSP COMMENTS	146
JSPS ARE REALLY SERVLETS	147
THE GENERATED SERVLET	148
LAB 4.1 - CREATING A JSP	149
MODEL VIEW CONTROLLER (MVC) ARCHITECTURE.....	154
SERVLET AND JSP ARCHITECTURE.....	155
MODEL VIEW CONTROLLER (MVC).....	156
SERVLETS AS THE CONTROLLER	157
JSP AS THE VIEW	158
JAVABeans AS THE GLUE AND THE MODEL	159
SERVLETS AS CONTROLLERS.....	160
REQUESTDISPATCHER.....	161
SERVLETS AS CONTROLLER	162
FORWARDING TO A RESOURCE	163
INCLUDING OUTPUT OF ANOTHER RESOURCE	164
REQUESTDISPATCHER PATHS	165
DATA SHARING AMONG SERVLETS AND JSPS	166
OBJECT BUCKETS OR SCOPES	167
USING THE SCOPE OBJECTS	168
JSPs AND JAVABeans	169
PUTTING DATA ON A SCOPE	170
THE OTHER SCOPES	171
CAREFUL - SERVLETS ARE MULTI-THREADED !	172
LAB 4.2 - FORWARDING TO A JSP	174
JSP EXPRESSION LANGUAGE AND DATA ACCESS	179
JSP EXPRESSION LANGUAGE (EL)	180
DATA OBJECTS AND THE EL.....	181
JAVABeans AND THE EL.....	182
IMPLICIT OBJECTS (PREDEFINED EL VARIABLES)	183
EL EXAMPLES	184
JSP MIXES DYNAMIC AND TEMPLATE DATA	185
<JSP:USEBEAN>.....	186
JAVABean PROPERTIES	187
LAB 4.3 – USING THE EXPRESSION LANGUAGE	188
JSP DIRECTIVES	192
THE INCLUDE DIRECTIVE - <% @ INCLUDE %>	193
THE INCLUDE ACTION - <JSP:INCLUDE>	194
<JSP:FORWARD> ACTION.....	195
THE PAGE DIRECTIVE	196
LAB 4.4 - INCLUDING ANOTHER JSP.....	197
REVIEW QUESTIONS	201
LESSONS LEARNED	202
SESSION 5: USING CUSTOM TAGS	204

LESSON OBJECTIVES.....	205
CUSTOM TAG LIBRARIES OVERVIEW.....	206
ISSUES WITH VANILLA JSP	207
SUN'S SOLUTION - CUSTOM TAGS	208
CUSTOM TAG OVERVIEW	209
TAG LIBRARIES.....	210
USING A TAG LIBRARY IN A WEB APPLICATION.....	211
TAGLIB DEFINITION - HOW IT WORKS.....	212
MORE ABOUT URIS AND PREFIXES.....	213
JSP STANDARD TAG LIBRARY (JSTL)	214
THE JSTL	215
EXAMPLE: THE C:OUT TAG	216
NEED FOR ITERATION	217
EXAMPLE: C:FOR EACH TAG FOR ITERATION.....	218
USING THE C:FOR EACH TAG - EXAMPLE.....	219
LAB 5.1 - USING C:FOR EACH.....	220
EXAMPLE: C:URL TAG FOR URLs	225
USING THE C:PARAM TAG FOR URL PARAMETERS	226
USING C:PARAM WITH C:URL	227
LAB 5.2 - WORKING WITH LINKS	228
REVIEW QUESTIONS	237
LESSONS LEARNED	238
SESSION 6: HTTP SESSION TRACKING	240
LESSON OBJECTIVES.....	241
HTTP SESSION OVERVIEW	242
HTTP IS A STATELESS PROTOCOL.....	243
ONLINE SHOPPING WITH A STATELESS PROTOCOL	244
STORING CONTEXTUAL INFORMATION	245
HIDDEN FORM FIELDS	246
COOKIES.....	247
COOKIES	248
COOKIE DETAILS	249
ACCESSING COOKIES WITH THE SERVLET API	250
PERSISTENT COOKIES AND SESSION COOKIES	251
COOKIE USABILITY ISSUES	252
SESSIONS	253
SERVLET/JSP SESSIONS	254
USING SESSIONS	255
USING A SESSION EXAMPLE	256
HOW SESSION TRACKING WORKS	257
SESSIONS AND COOKIES - HOW THEY WORK	258
SESSIONS AND COOKIES - HOW THEY WORK	259
SESSIONS AND COOKIES - HOW THEY WORK	260
SESSIONS AND COOKIES - HOW THEY WORK	261
LAB 6.1 - SESSIONS: IMPLEMENT THE CART	262
REVIEW QUESTIONS	268
LESSONS LEARNED	269
SESSION 7: MORE JSP CAPABILITIES	270

LESSON OBJECTIVES.....	271
ERROR PAGES.....	272
SERVLET EXCEPTIONS AND ERROR PAGES	273
THE IMPLICIT EXCEPTION OBJECT.....	274
JSP 2.0 ERROR HANDLING	275
USING ERRORDATA	276
EXCEPTION HANDLING IN A JSP	277
WEB.XML - DECLARATIVE EXCEPTION HANDLING	278
SERVER LOGGING	279
[OPTIONAL] LAB 7.1 - ERROR PAGES	280
JSPS AS XML FILES.....	283
JSP PAGES AS XML DOCUMENTS	284
JSP PAGES AS XML DOCUMENTS	285
JAVA SCRIPTLETS.....	286
SCRIPTLETS.....	287
SCRIPTLET SYNTAX	288
HOW DO SCRIPTLETS WORK?	289
ANOTHER SIMPLE JSP - SIMPLE.JSP	290
GENERATED CODE FRAGMENT FOR SIMPLE.JSP	291
WHEN TO USE SCRIPTLETS	292
DECLARATIONS	293
DECLARATIONS EXAMPLE	294
[OPTIONAL] LAB 7.2 - SCRIPTLETS	295
REVIEW QUESTIONS	300
LESSONS LEARNED.....	301
SESSION 8 – MORE JSTL AND EL.....	302
SESSION OBJECTIVES - YOU SHOULD BE ABLE TO:.....	303
MORE ABOUT JSTL	304
JSTL TAG LIBRARIES AT A GLANCE	305
CUSTOM TAG ARCHITECTURE	306
MORE ABOUT JSTL URIs AND PREFIXES	307
SOME OTHER COMMON CORE ACTIONS	308
FORMATTING LIBRARY	309
SQL LIBRARY.....	310
XML LIBRARY	311
XML ACTION EXAMPLE.....	312
JSTL 1.1 FUNCTIONS LIBRARY	313
JSTL FUNCTION LIBRARY	314
MORE ABOUT THE JSP EL	315
EL SYNTAX	316
EL IDENTIFIERS	317
EL OPERATORS.....	318
. AND [] OPERATOR USAGE	319
JSP EL IMPLICIT OBJECTS (1 OF 3).....	320
JSP EL IMPLICIT OBJECTS (2 OF 3).....	321
JSP EL IMPLICIT OBJECTS (3 OF 3).....	322
PAGECONTEXT PROPERTIES (1 OF 2)	323
PAGECONTEXT PROPERTIES (2 OF 2)	324
JSP EL LITERAL VALUES	325
CONDITIONAL OPERATOR.....	326

USING COMMON TAGS	327
PUTTING JSTL AND EL IN JSP	328
USING <C:IF>.....	329
USING <C:IF>.....	330
<C:IF> TEST EXPRESSIONS.....	331
<C:CHOOSE>	332
LAB 8.1 - MORE JSTL.....	333
MORE ON <C:FOREACH>	339
<C:SET>	340
<C:SET> EXAMPLES	341
<FMT:FORMATDATE>	342
<FMT:FORMATDATE> EXAMPLES.....	343
<FMT:FORMATNUMBER>.....	344
<FMT:FORMATNUMBER> EXAMPLE	345
LAB 8.2 - JSTL FORMAT TAGS	346
JSTL-EL RESOURCES (1 OF 2).....	350
REVIEW QUESTIONS	351
LESSONS LEARNED	352
LESSONS LEARNED	353
SESSION 9 – SECURITY	354
SESSION OBJECTIVES - YOU SHOULD BE ABLE TO:.....	355
JAVA EE SECURITY OVERVIEW	356
SECURITY REQUIREMENTS	357
JAVA EE SECURITY	358
WEB TIER SECURITY IN JAVA EE	359
TRANSPORT LEVEL SECURITY WITH HTTPS/SSL.....	360
DECLARATIVE SECURITY	361
DECLARATIVE, ROLE-BASED SECURITY	362
EXAMPLE: SECURITY ROLE DECLARATION	363
SPECIFYING SECURITY CONSTRAINTS	364
SECURITY CONSTRAINTS - DEPLOYMENT DESCRIPTOR	365
SECURITY CONSTRAINTS - DEPLOYMENT DESCRIPTOR	366
MAPPING USERS TO ROLES.....	367
SECURITY - THE BIG PICTURE	368
WEB AUTHENTICATION.....	369
AUTHENTICATION.....	370
HTTP BASIC AUTHENTICATION	371
HTTP BASIC AUTHENTICATION ON THE BROWSER.....	372
CONFIGURING BASIC AUTHENTICATION.....	373
BASIC AUTHENTICATION PROS AND CONS	374
FORM BASED AUTHENTICATION.....	375
USING FORM BASED AUTHENTICATION	376
FORM BASED AUTHENTICATION PROS AND CONS	377
DIGEST AUTHENTICATION	378
DIGEST AUTHENTICATION PROS AND CONS	379
HTTPS CLIENT AUTHENTICATION	380
PROGRAMMATIC SECURITY	381
PROGRAMMATIC SECURITY	382
PROGRAMMATIC SECURITY - HTTPSERVLETREQUEST	383
[OPTIONAL] LAB 9.1 - SECURITY	384

[OPTIONAL] LAB 9.1 - SECURITY	385
SESSION 10 – ADDITIONAL TOPICS.....	389
SESSION OBJECTIVES - YOU SHOULD BE ABLE TO:.....	390
DESIGN ISSUES	391
DESIGN ISSUES	392
DIVIDE AND CONQUER	393
MINIMIZE NETWORK OVERHEAD	394
SCALABILITY - CLUSTERING.....	395
SESSIONS AND CLUSTERING	396
SCALABILITY – SEPARATING STATIC CONTENT	397
SCALABILITY – JAVA VIRTUAL MACHINE	398
SCALABILITY –CPUS AND 64 BIT MACHINES	399
MAINTAINABILITY - WRITE YOUR JSPS WELL.....	400
USE THE TECHNOLOGY WISELY	401
USE A FRAMEWORK.....	402
THE JAVA BLUEPRINTS	403
CUSTOM TAGS USING TAG FILES.....	404
CREATING CUSTOM TAGS WITH TAG FILES.....	405
ADDING ATTRIBUTES TO TAGS.....	406
THE TAG DIRECTIVE	407
OTHER TAG FILE CAPABILITIES.....	408
SERVLET FILTERS.....	409
SERVLET FILTER OVERVIEW	410
FILTER API.....	411
FILTER PROCESSING	412
FILTER PROCESSING	413
FILTER CHAIN.....	414
A SIMPLE FILTER	415
A SIMPLE FILTER - CONTINUED.....	416
FILTER IS MAPPED IN THE WEB.XML FILE.....	417
HOW A FILTER CAN MODIFY A REQUEST OR RESPONSE	418
THE WAY A RESPONSE NORMALLY WORKS.....	419
MANIPULATING THE RESPONSE	420
WRAPPING THE RESPONSE.....	421
JAVASERVER FACES (JSF) OVERVIEW	422
VANILLA SERVLET/JSP SHORTCOMINGS.....	423
JAVASERVER FACES	424
JSF ARCHITECTURE.....	425
JSF CAPABILITIES.....	426
JSF COMPONENTS	427
THE UI RUNS ON THE SERVER	428
EVENT HANDLING	429
PROCESS FLOW	430
JSF ADVANTAGES	431
JSF DISADVANTAGES	432
RECAP.....	433
RECAP OF WHAT WE'VE DONE.....	434
WHAT ELSE IS THERE	435
END OF SESSION	436



Web Applications with Servlets, JSP and the JSTL (JSP Standard Tag Library)

The Java Developer Education Series

Notes:

Workshop Overview

- ◆ This hands-on course covers the use of **Servlets**, **JSP** (JavaServer Pages), and **JSTL** (the JSP Standard Tag Library) to build dynamic Web applications
 - You'll also learn the basics of **Java EE** (Java Enterprise Edition)
- ◆ It starts with the basic architecture of the Web and how servlets work, moves on to cover JSP (JavaServer Pages), then custom tags and the JSTL for creating complex actions
 - Key design issues are introduced and illustrated
- ◆ The workshop consists of about 50% discussion, 50% hands-on lab exercises, including a complete Web project
 - Most of the labs follow a common fictional case study - JavaTunes, an online music store – which includes music items (searchable from the Web), and shopping (via a shopping Cart)

Notes:

Workshop Objectives

- ◆ At completion you should be able to
 - Understand the design and development of Web applications using Servlets, JSPs and the JSTL (JSP Standard Tag Library)
 - Develop Servlets to process HTML forms
 - Understand and create JavaServer Pages (JSPs)
 - Understand and use custom tag libraries
 - Understand and use the JSTL
 - Link Servlets and JSPs, and share data between them
 - Store and process session information
 - Deal with concurrency issues
 - Understand and use good design techniques including MVC (Model-View-Controller) and the Model 2 architecture

Notes:

Workshop Agenda

- ◆ Session 1: **Web Application Basics**
- ◆ Session 2: **Servlet Basics**
- ◆ Session 3: **Additional Servlets Capabilities**
- ◆ Session 4: **JSP (JavaServer Pages)**
- ◆ Session 5: **Using Custom Tags**
- ◆ Session 6: **Session Management**
- ◆ Session 7: **More JSP Capabilities**
- ◆ Session 8: **More JSTL and EL**
- ◆ Session 9: **Security**
- ◆ Session 10: **Additional Capabilities**

Notes:

Typographic Conventions

- ◆ Code that is inline in the text will appear in a fixed-width code font, such at this:

```
JavaTeacher teacher = new JavaTeacher()
```

- Any class names, such as `JavaTeacher`, method names, or other code fragments will also appear in the same font
- If we want to emphasize a particular piece of code, we'll also bold it (and in the slide, change it's color) such as **BeanFactory**
- Filenames will be in italics, such as *JavaInstructor.java*
- We sometimes denote more info in the notes with a **star ***
- Lastly, longer code examples will appear in a separate code box as shown below

```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
    public void teach() {
        System.out.println("BeanFactories are way cool");
    }
}
```

Notes:

Labs



- ◆ The workshop consists of approximately 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
 - The labs are contained directly in the course book, and have detailed instructions on what needs to be done
- ◆ The course includes setup zip files that contain skeleton code for the labs
 - Students just need to add code for the particular capabilities that they are working with
 - There is also a solution zip file that contains completed lab code
- ◆ Lab slides contain an icon like the one in the upper right corner of this slide
 - The end of each lab is clearly marked with a stop like this one to the right



Notes:

Release Level



- ◆ This manual contains instructions for creating and running the labs using the following platforms:
 - Java 5 or Java 6
 - Tomcat 6.0
 - Eclipse Java EE Edition (any recent edition should work)
- ◆ All labs have been tested on Tomcat 6.0, Java 5/6, and Eclipse Java EE edition
- ◆ **Complete lab instructions for this platform are included in the student manual**

Notes:

- ◆ The instructions for the labs are geared for Eclipse Java EE edition (Ganymede/3.4 and Galileo/3.5)
 - They should also work fairly well for earlier Java EE / Web versions of Eclipse
 - However, the basic Eclipse version does not have the server deploy support, and these instructions will not be suitable for basic Eclipse
- ◆ Sun Microsystems recently changed the version numbering of the Java Software Development Kit.
 - After version 1.4.2, the next version was called 5.0.
 - However, note that the default home directory path for a 5.0 installation contains references to the fact that it is really version 1.5 renamed, i.e., *C:\Program Files\Java\jdk1.5.0*.



Session 1: Web Application Basics

How the Web Works

Java EE - Java Enterprise Edition Basics

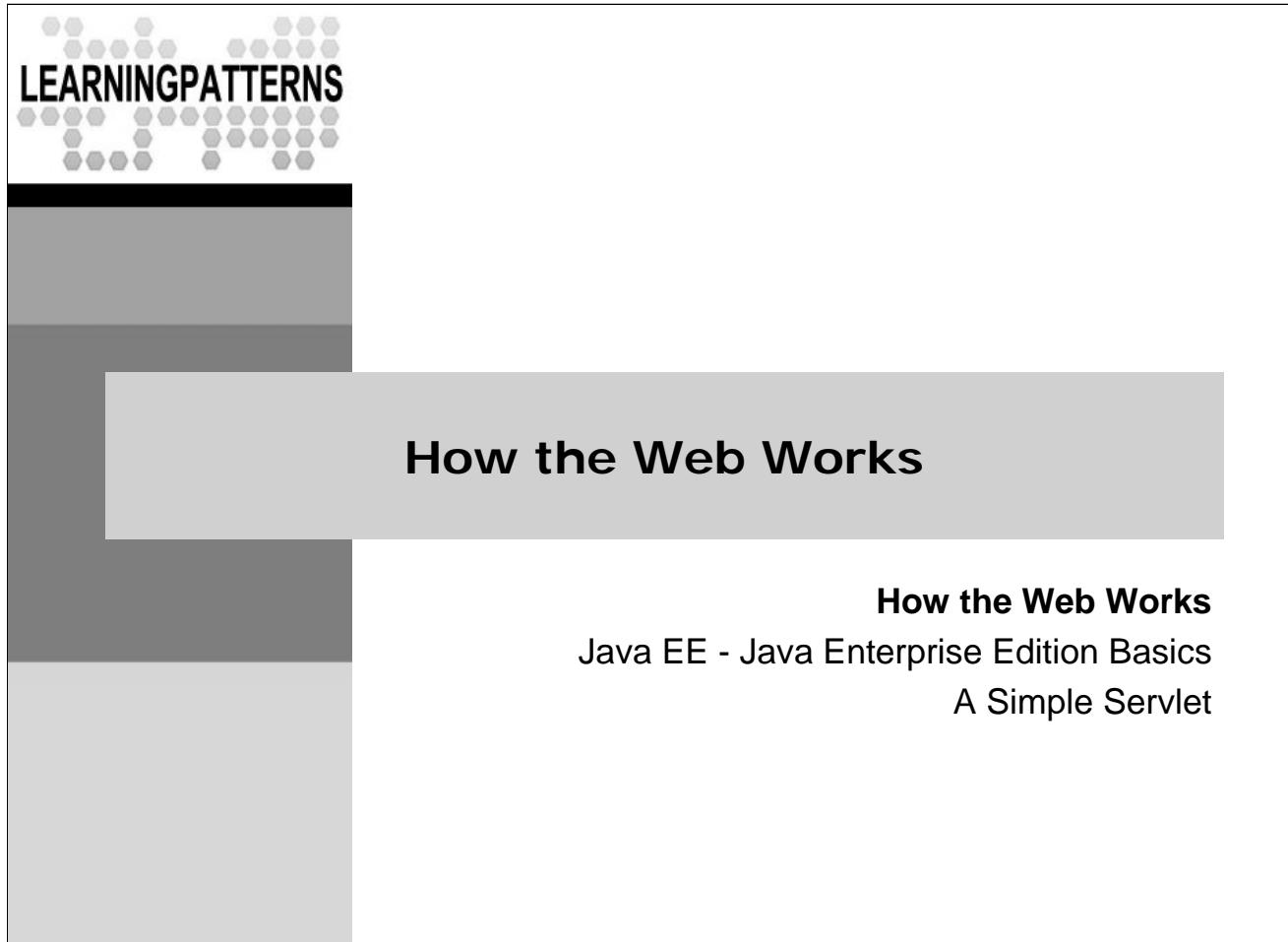
Servlet Basics - Generating HTML Dynamically

Notes:

Lesson Objectives

- ◆ Understand the Basics of HTML and HTTP
 - Understand Clients, Servers and the Internet, HTML tags, and HTTP requests
- ◆ Get an overview of Java EE
 - Understand Java EE Web Applications structure
 - Create a simple Java EE Web Application
- ◆ Gain an understanding of what servlets are
 - Learn the basics of the Servlet API
 - Create a Simple Servlet
 - Learn how to map a servlet in web.xml

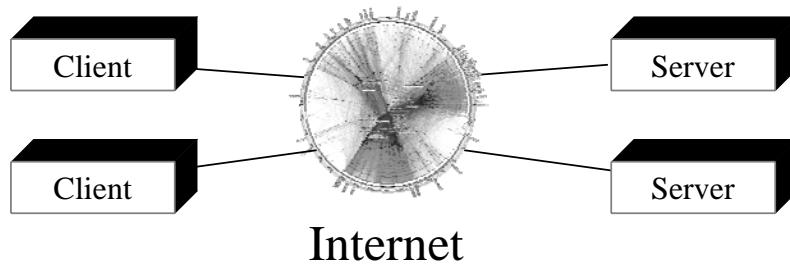
Notes:



Notes:

Clients and Servers

- ◆ The World Wide Web is composed of clients and servers connected by the Internet
- ◆ Clients - Programs that **request** information on the web
 - e.g. browsers like Mozilla Firefox or Microsoft Internet Explorer
- ◆ Servers - Programs that **respond** with information to clients
 - Web servers or application servers – e.g. Apache Tomcat, WebSphere Application Server, GlassFish, etc.

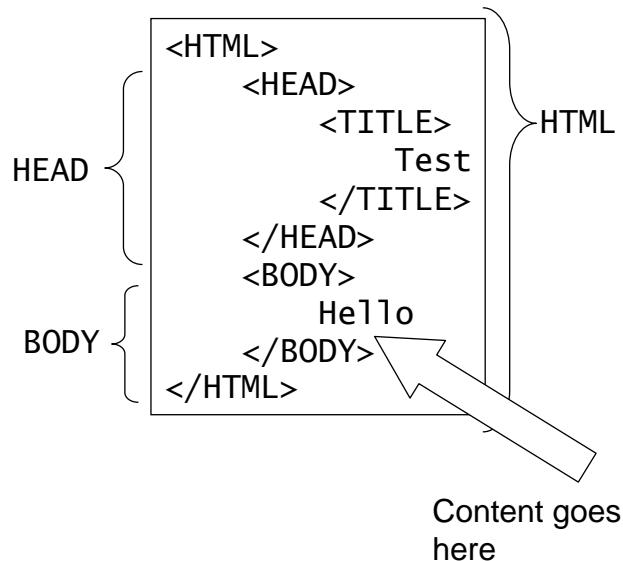


Notes:

- ◆ If you want to see more graphs of the internet, check out <http://www.caida.org>
 - Their skitter tool probes the internet for topology and performance
 - It can visualize network from the data it collects
- ◆ The most common client is a web browser, used by a person
 - Other clients such as Swing applications can also exist
 - Computer programs can be clients also
- ◆ Servers may do many other things than simply serving up web pages
 - It depends on the server, and upon the system that is running on it

HTML - The *Language* of the Web

- ◆ Web servers supply pages to browsers as **Hypertext Markup Language (HTML)**
 - HTML is a plain text format with added **tags**
 - Browsers use HTML tags to interpret and format content
- ◆ HTML uses tags to mark different parts of a document
 - i.e. - To denote the document structure, such as the page title
 - <TITLE>Search page</TITLE>
- ◆ HTML also uses tags for formatting (see notes)
 - This will appear in bold in a browser



Notes:

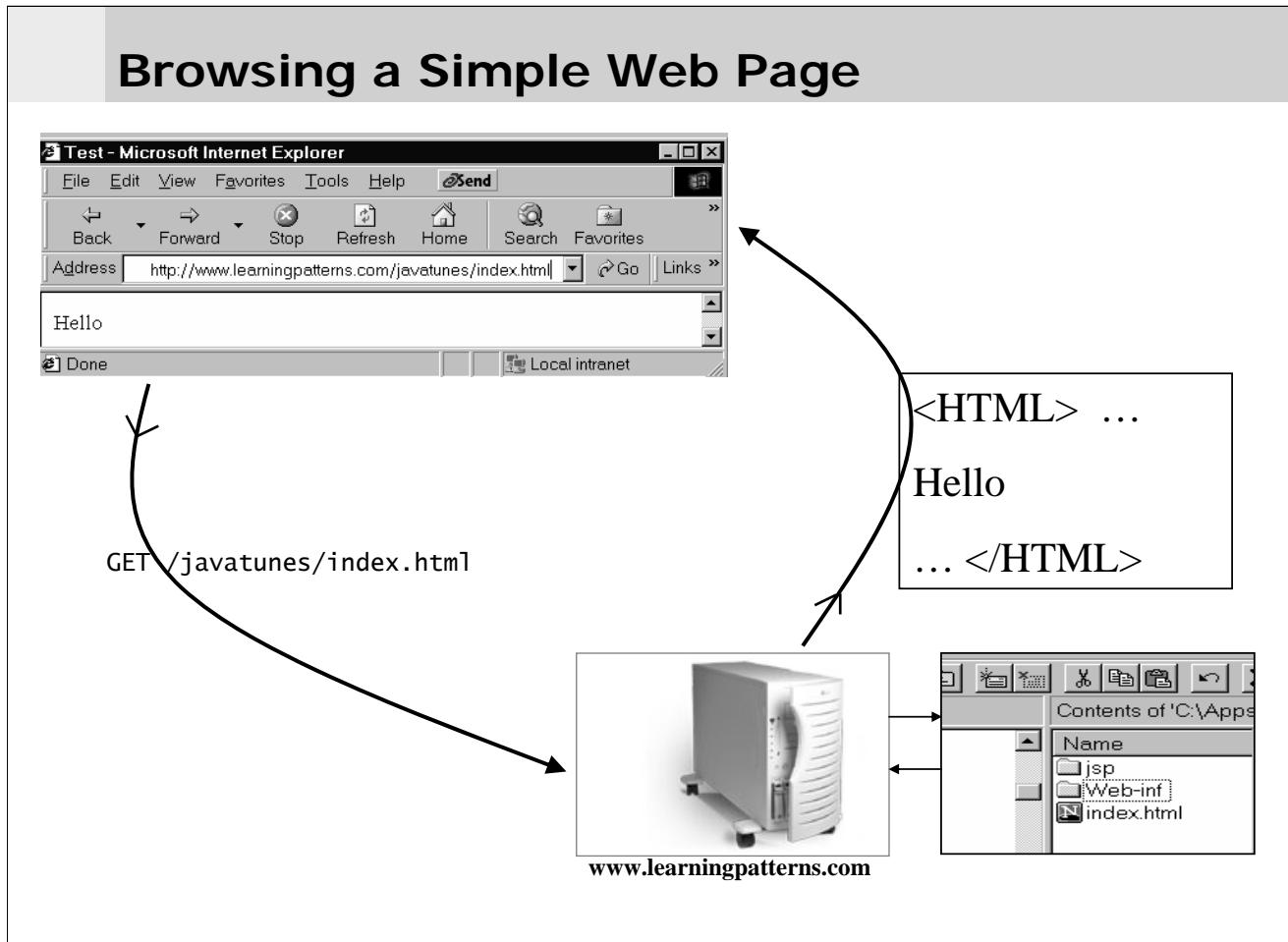
- ◆ Basic HTML Tags
 - <HTML> - Starts an HTML page
 - <HEAD> - The heading part of a page
 - <TITLE> - The title of a page
 - <BODY> - The body (main content) of a page
 - - Bold text
 -
 - “Break”, or start a new line
 - <P> - Start a new paragraph
 - <HR> - Insert a horizontal line
 - <A> - Insert a hypertext link
- Click me!
- ◆ Go to <http://www.w3.org/MarkUp/> for more info on HTML
- ◆ All HTML pages should have at least the following the <HTML> and <BODY> elements
 - Most will have <HEAD> and <TITLE> elements
 - Browsers are very forgiving. Pages with errors often work

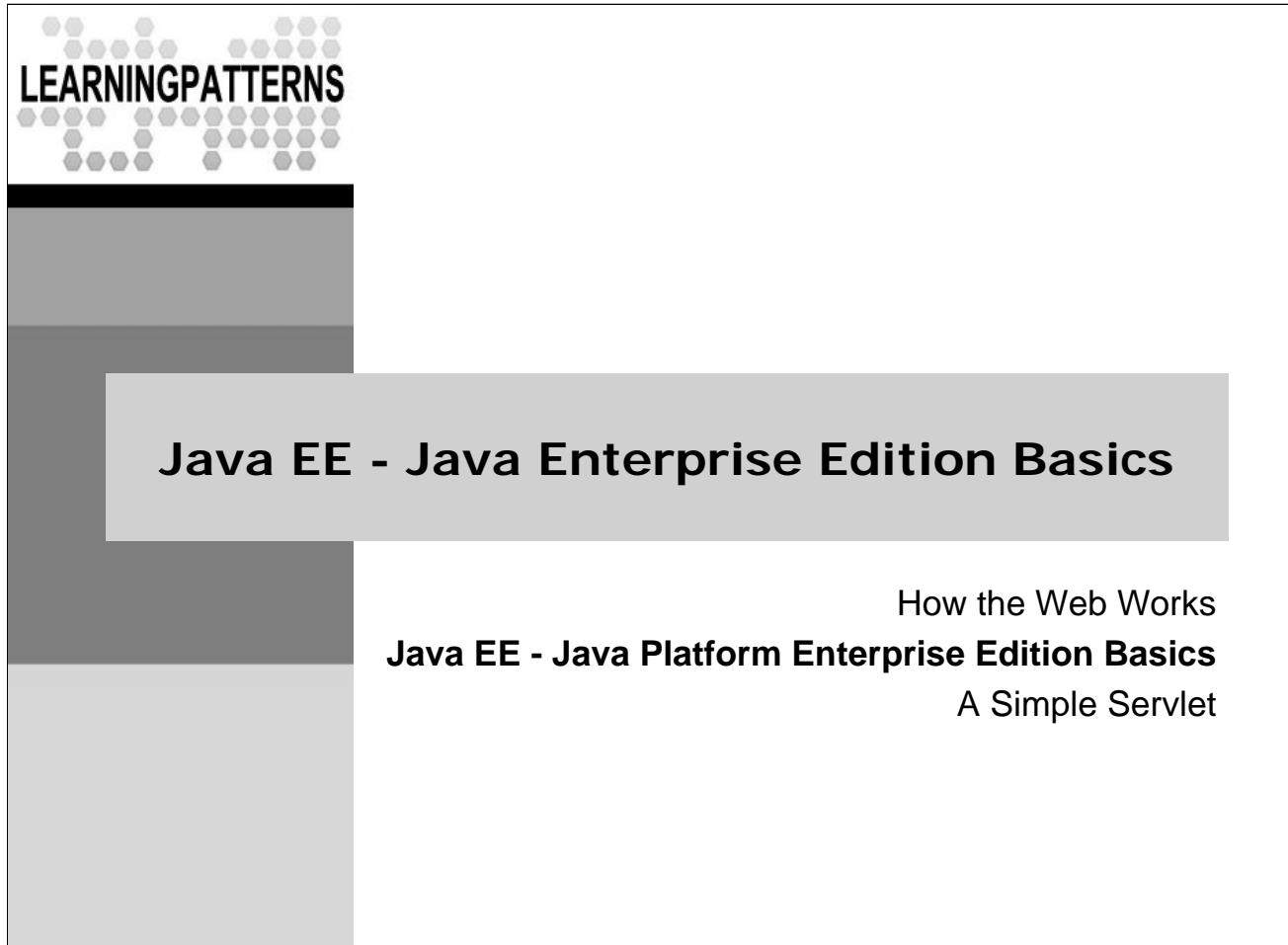
HTTP, Addressing, Requests, and Responses

- ◆ Servers and browsers communicate via the **Hypertext Transfer Protocol (HTTP)**
- ◆ A client specifies a server with an IP address
 - This can be a name, like **LearningPatterns.com**
 - Names translate to numeric addresses, like **206.98.161.38**
 - HTTP servers run on specific ports
 - The default is port **80**, Tomcat runs on port **8080**
- ◆ The browser sends an HTTP **request** to a server for a page
 - With a URL: **http://www.mycompany.com/index.html**
 - The server returns the page in an HTTP **response**
 - Both requests and responses include extra bits of information, called **headers**
 - The request is mostly composed of headers
 - The response includes headers and the HTML itself

Notes:

- ◆ A basic understanding of HTTP is useful in knowing how the web, and servlets, work
 - You'll be interacting with it later through the servlet API
- ◆ You can get more information on HTTP at: <http://www.w3.org/Protocols/>
- ◆ If the server is using a different port, the client should ask for it specifically - e.g.
 - <http://www.learningpatterns.com:8080>
- ◆ URL = Uniform Resource Locator
- ◆ Finding a server is like looking up a phone number and making a phone call
 - The symbolic name, is like the name of the person you are calling.
 - Using the symbolic name, you look up the IP address (using something called DNS)
 - The IP address is like a phone number. It specifies a specific computer on an IP network
 - The port is like an extension
 - There may be a number of applications on the same computer listening for connections
 - The port is what directs incoming connections to a specific program (in this case an HTTP server listening on port (extension) 80

**Notes:**



Notes:

Java EE - Java Platform Enterprise Edition

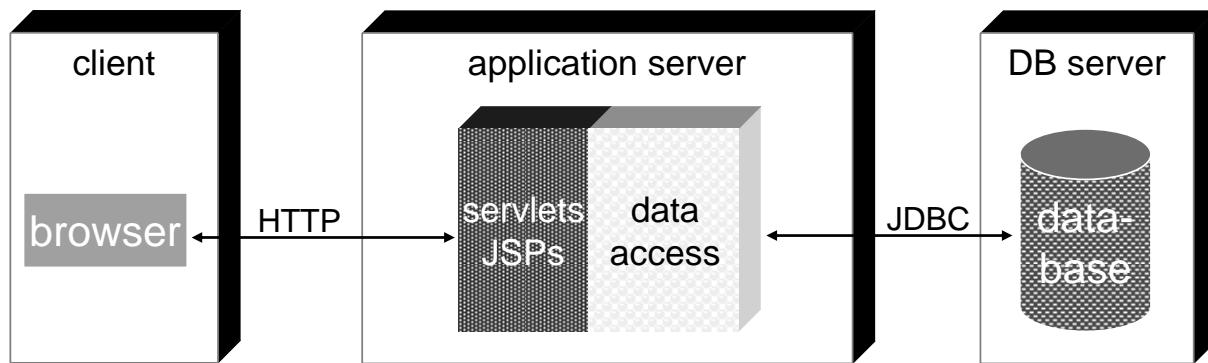
- ◆ Java EE is Java's standard architecture for building scalable, distributed, reliable, Enterprise applications
 - We will use a **Java EE** architecture to study and implement all of these capabilities
 - Formerly (and still widely) called **Java EE**
 - Java EE includes technologies such as Servlets/JavaServer Pages (JSP), JDBC, EJB, JMS, JavaMail, etc.
- ◆ We will focus solely on the **Web tier** here
 - And the base Servlet/JSP technology
 - We will not delve into the other technologies such as JDBC for database access, or EJB for creating transactional, scalable components
 - These are important, but not the area we need to focus on

Notes:

- ◆ Web applications are part of the Java EE standard
 - This gives your applications consistency and portability across implementations

Simple Web-Based Architecture

- ◆ This is a popular Java EE architecture for Web applications
- ◆ It is relatively simple, yet suitable for many types of applications



Notes:

- ◆ The presentation code and business code reside in the same tier.
 - Database access can be implemented directly in the servlets/JSPs, but is often encapsulated into a data access layer.
 - Many types of systems can be built with this relatively simple architecture.

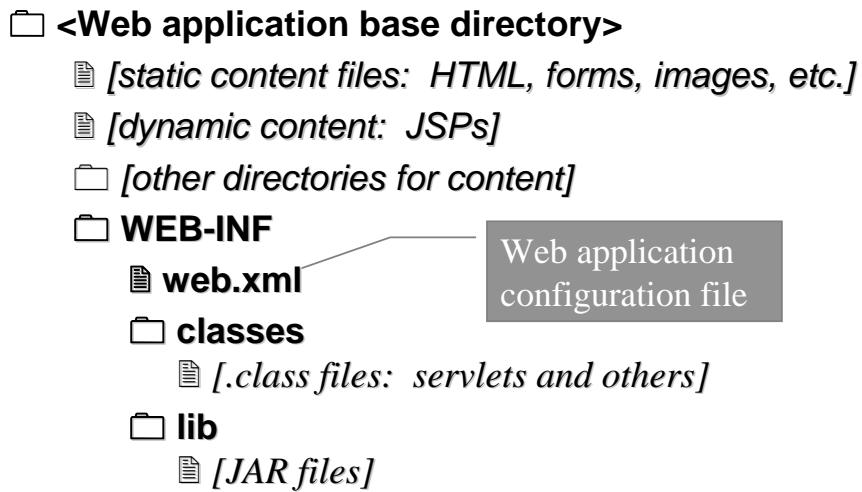
Java EE Web Applications

- ◆ **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
 - This is the standard Java EE way to package up server programs for the Web
- ◆ Every web app has a standard directory structure and includes an XML configuration file - **web.xml**
 - *web.xml* (commonly called the **deployment descriptor** or **DD**) contains configuration information (XML) for the Web app
 - You can declare servlets, servlet mappings URLs and more
- ◆ Often, the Web app is packaged into a Web Archive (WAR)
 - Based on jar, and is the standard packaging for a Web app
 - Can be deployed directly in servers supporting Java Web apps
 - Often servers support "exploded" deployments *

Notes:

- ◆ A jar file is basically a zip file with added information in it
 - It is compatible with the zip format, and jar files can be manipulated using most zip tools
 - There is also a utility program that comes with Java to create and work with jar files
- ◆ A WAR file is just a jar containing a Web app
 - Some servers also support the deployment of the Web app as a directory containing the needed files
 - This directory would have the standard Web app structure, and contain the same contents as in the WAR file
 - This is convenient if you have to change something (e.g. an HTML or JSP page) in the Web app

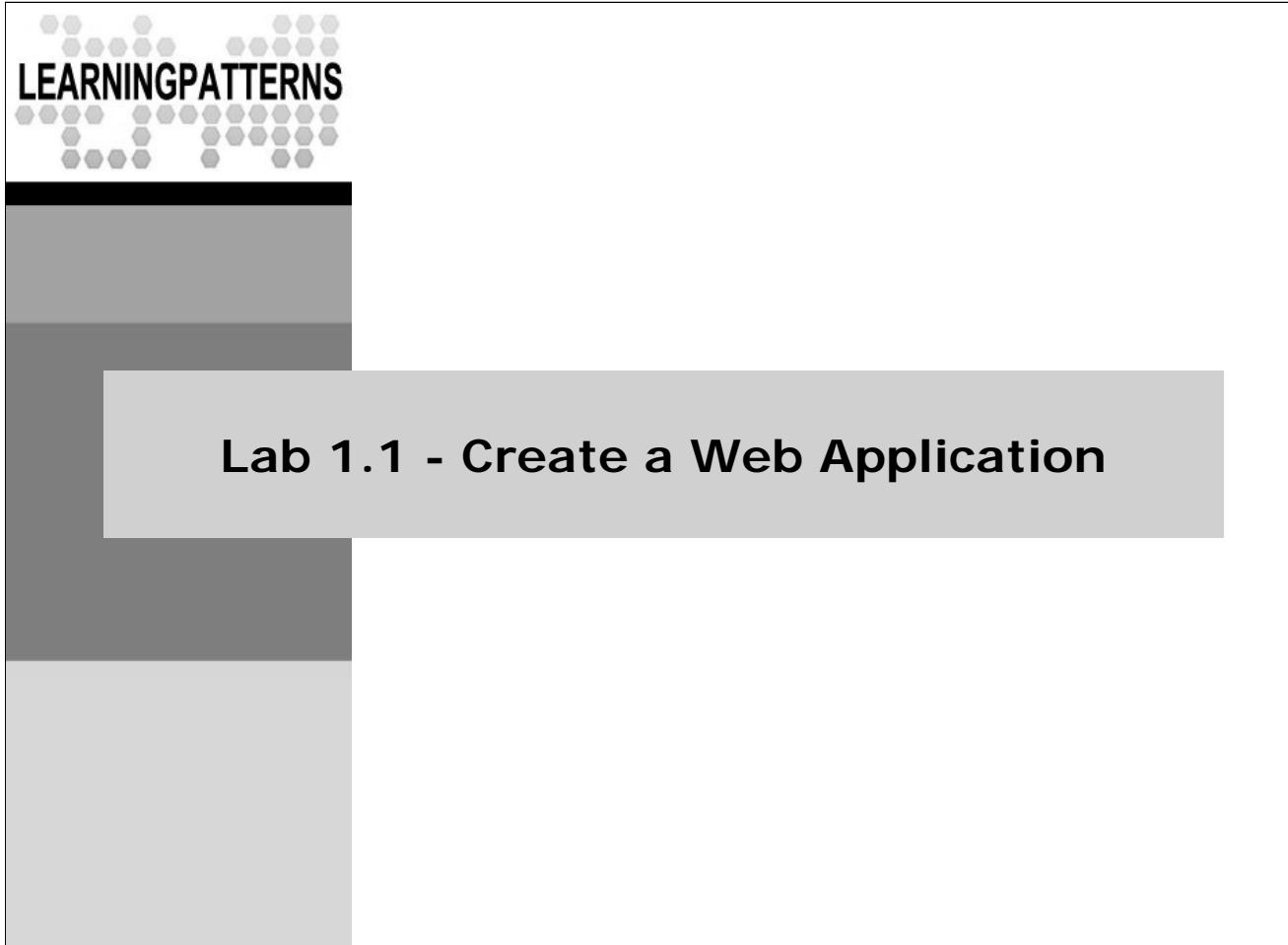
Web Application Structure



- ◆ The root of the directory hierarchy serves as the document root for files that are part of the application
 - The WEB-INF node is not part of the public document tree
 - Nothing in WEB-INF can be served directly to a client

Notes:

- ◆ The WEB-INF directory is just a normal directory of this name
 - It's an unusual name, but don't be confused by that. It's just a directory
- ◆ You can put things in WEB-INF that you don't want visible directly to clients
 - For example application configuration files
 - You can still forward/include them via the RequestDispatcher that we will cover soon
 - Some developers put all the jsp's there that are normally forwarded to by servlets
 - This prevents clients from browsing directly to them



Notes:

Lab 1.1 – Create a Web Application



◆ **Overview:** In this lab, we will setup and become familiar with the lab environment, and create a simple Web application

- The Web app will just have a simple HTML file in it
- We will build, deploy and run it from within Eclipse
- This will take a little time to set up, but future labs will take less setup

◆ **Objectives:**

- Become familiar with the lab structure and Eclipse
- Create a Web application and deploy it using Eclipse

◆ **Builds on previous labs:** None

◆ **Approximate Time:** 30-40 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 - like the one 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
- ◆ **Make sure that you have Eclipse installed**
 - Likely installed in a directory like C:\eclipse *
 - We'll refer to this directory as <eclipse>
 - If it is not installed, or you can't find it, tell your instructor
- ◆ **OK – Now get out your setup CD; we're ready to start working**

Notes:

- ◆ If Eclipse is not installed, then you'll have to download and install it
- ◆ If you need to download Eclipse, go to
<http://www.eclipse.org/downloads>
 - Under the Eclipse Packages section, click on the link for the **Eclipse IDE for Java EE Developers**
 - Save the zip file to your computer
- ◆ To install Eclipse, unzip the zip file (easiest location to unzip it to is C:\, but another location is fine as long as you can get to it to run the eclipse.exe executable)

Extract the Lab Setup Zip File



- ◆ To set up the labs, you'll need the course setup zip file *
 - It has a name like: **SJSP_LabSetup_20100215.zip**
- ◆ Our base working directory for this course will be
C:\StudentWork\SJSP
 - This directory will be created when we extract the Setup zip
 - It includes a directory structure and files (e.g., Java files and other) 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
- ◆ SJSP stands for Servlets-JSP
- ◆ 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

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
 - Released into open source
 - IBM's RAD environment is built on top of Eclipse
- ◆ Eclipse products have two fundamental layers
 - The **Workspace** – files, packages, projects, resource connections, configuration properties
 - The **Workbench** – editors, views, and perspectives
- ◆ We will set up the workspace and workbench, then describe it in more detail at the end of the lab

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

Notes:

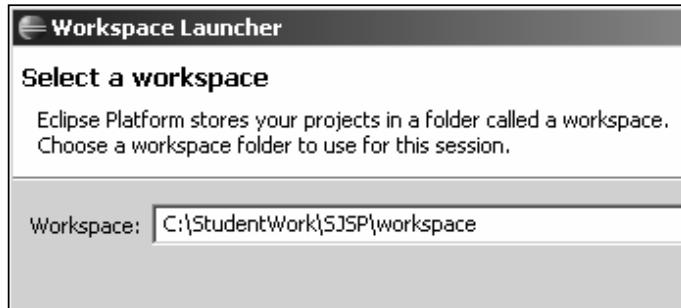
- ◆ The src folder contains all Java source
 - Servlets, supporting classes, JavaBeans
 - When you compile these classes, the compiled output is put in the WEB-INF\classes directory
- ◆ The WebContent folder contains all Web resources
 - HTML files, JSP files, image files, etc.
 - Only content in this 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

Getting Started With 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 workspace location to **C:\StudentWork\SJSP\workspace**
 - If a different default workspace location is set, change it



Notes:

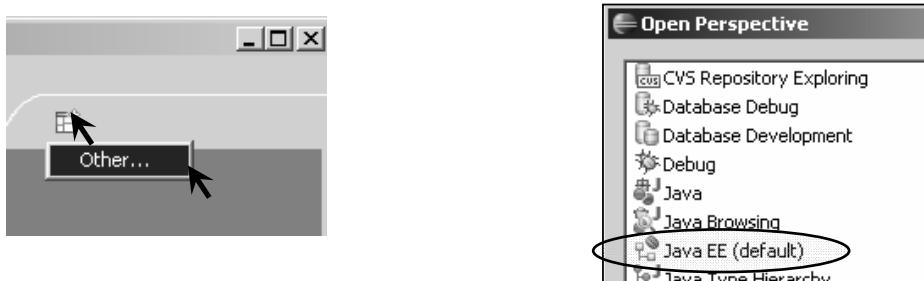
- ◆ You can also put a shortcut on your desktop to start Eclipse

Workbench and Java EE Perspective



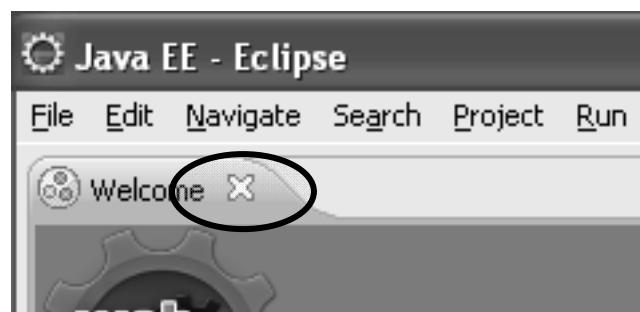
Tasks to Perform

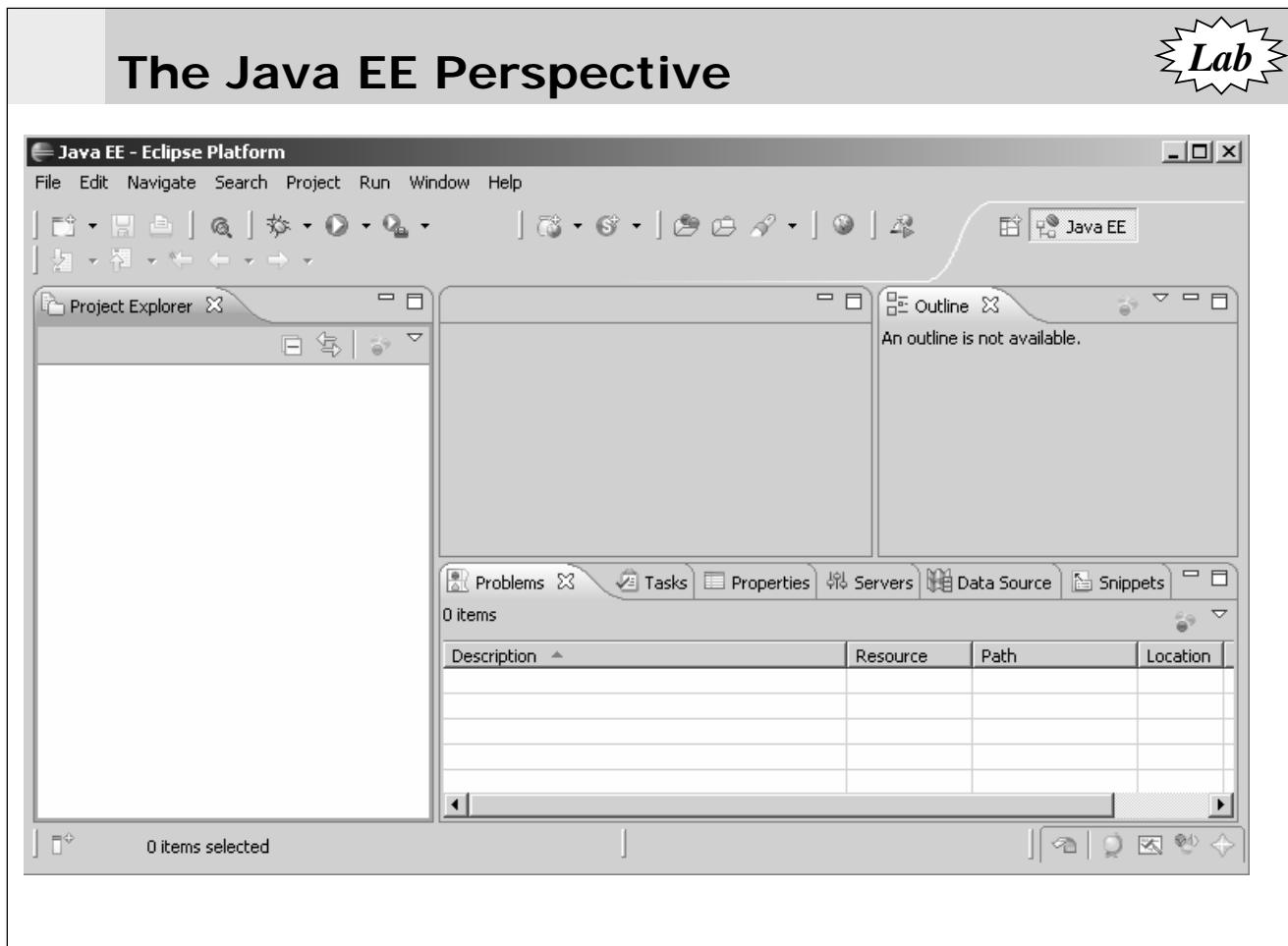
- ◆ In the window that opens, close the **Welcome** screen (see notes)
- ◆ You should be in a Java EE perspective
 - That's fine, this perspective is good for what we do
- ◆ If you need to open the Java EE perspective, (**Shouldn't need to do this now**) you can do so by clicking the Perspective icon at the top right of the Workbench, and select **Other | Java EE** (as shown below)



Notes:

- ◆ Eclipse opens with a Welcome screen which allows you to navigate to different things (such as tutorials)
 - We will not be using this, so we will close it and go directly to the Workbench
 - To close it, just click the X on the Welcome tab



**Notes:**

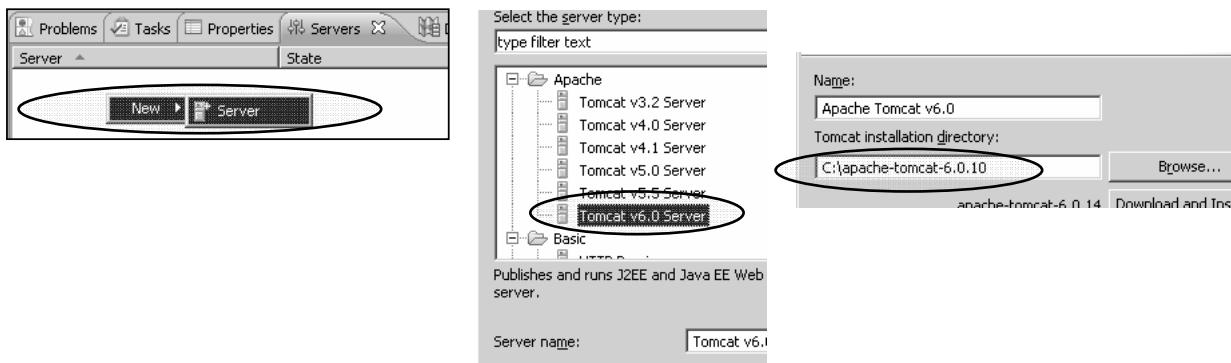
Creating a Server



- ◆ We will use the Tomcat to run our Web applications - first we 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** *



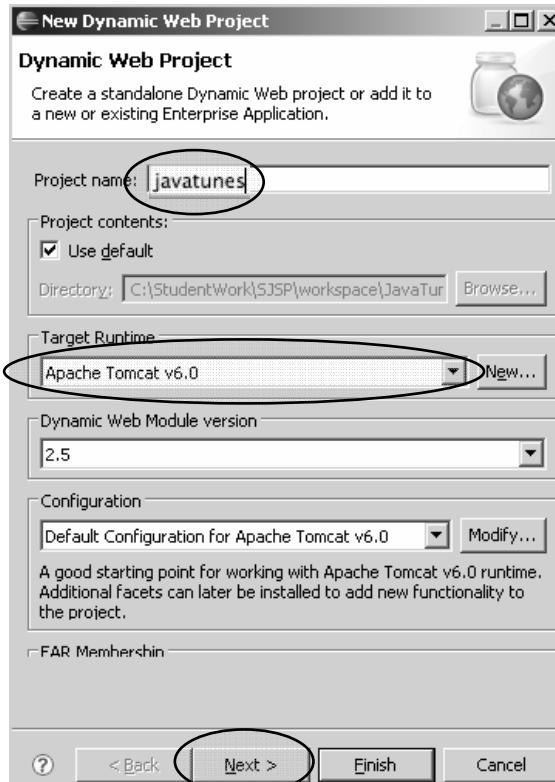
Notes:

- ◆ 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

Create a Web Project

Tasks to Perform

- ◆ Create a Dynamic Web project
 - File | New | Dynamic Web Project *
 - Call it **javatunes** *
 - Use the **Tomcat** runtime *
 - Click **Next** (click **Next** twice if using Eclipse 3.5)
- ◆ In the next dialog, make sure the Context Root is **javatunes**
 - Click **Finish**

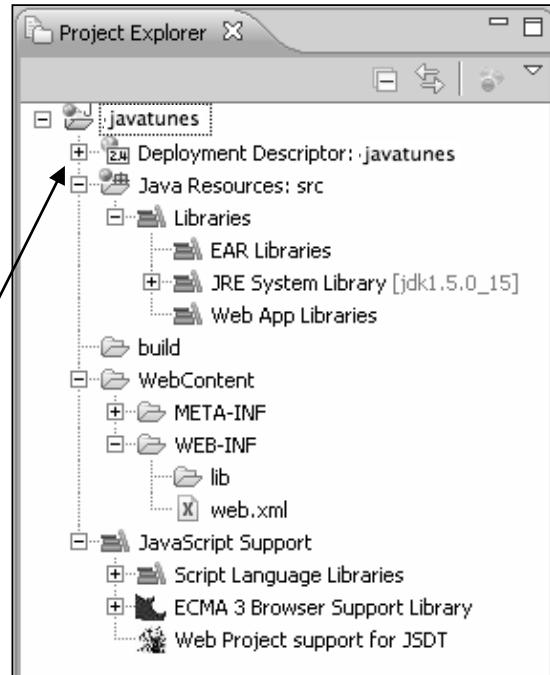

Notes:

- ◆ 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.
- ◆ Note - When you call the project javatunes, it's stored in **workspace\javatunes**
- ◆ The Tomcat runtime should automatically be used, unless you have other servers installed
 - If that's the case, make sure the Tomcat runtime is selected
 - This will also set the Web Module to Servlets 2.5
- ◆ Eclipse 3.4 and 3.5 have slightly different dialogs for creating a Dynamic Web project - but they are very similar
 - You shouldn't have any problems with either

The Project Explorer View



- ◆ 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 that Eclipse has create the deployment descriptor, **web.xml**, for you
- ◆ **Double click on web.xml to open it** for viewing and editing

Notes:

The web.xml file



- ◆ Review the **web.xml** file by clicking the editor **source tab**
 - <**web-app**> is the root element, and contains the schema declaration from the servlet/Java EE specification
 - <**display-name**> is a descriptive name used by tools
 - <**welcome-file-list**> is a list of default files to be displayed if a file is not specified on the request URL

A screenshot of a code editor window titled "web.xml". The window displays XML code for a web application configuration. The code includes a schema declaration, a display name ("javatunes"), and a welcome file list containing multiple entries for "index.html", "index.htm", "index.jsp", and "default.html", "default.htm", "default.jsp".

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="
    <display-name>javatunes </display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.html</welcome-file>
        <welcome-file>default.htm</welcome-file>
        <welcome-file>default.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

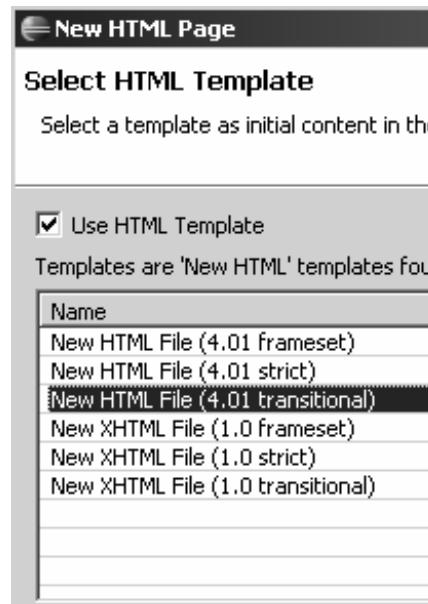
Notes:

Create an HTML file



Tasks to Perform

- ◆ In the **Project Explorer** View, right click on the **WebContent** folder
 - Select **New | HTML**
 - In the dialog that comes up, call the page **index.html**, and click Next
 - Select **New HTML File (4.01 transitional)** in the Template pane
 - Click **Finish**
- ◆ Eclipse will create the web page in the **WebContent** folder, and open it in an editor for you



Notes:

Edit the HTML File



Tasks to Perform

- ◆ Review the HTML file in the editor, and add some simple text or HTML into the body element

A screenshot of a code editor window titled "index.html". The window shows the following HTML code:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.  
<html>  
<head>  
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">  
<title>Insert title here</title>  
</head>  
<body>  
</body>  
</html>
```

An arrow points from the text "Add content" to the opening "<body>" tag.

Notes:

- ◆ You can add the following HTML into the page if you want

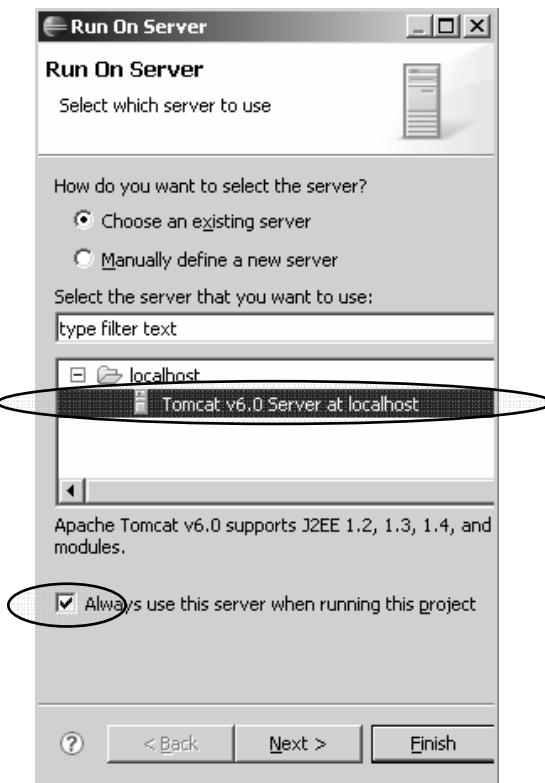
```
<body>  
  <h1>Welcome to JavaTunes</h1>  
  [add whatever else you want]  
</body>
```

Deploying an Application



Tasks to Perform

- ◆ To deploy to the server, right click on the javatunes 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 it (look at the Servers view)



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 your having browser issues, open an external browser viewing the same URL (note that you don't have to specify *index.html* since it's one of the files specified as a welcome file)
 - That's it – your Web app is up and running



Notes:

- ◆ Tomcat listens on port 8080 by default

Server Startup



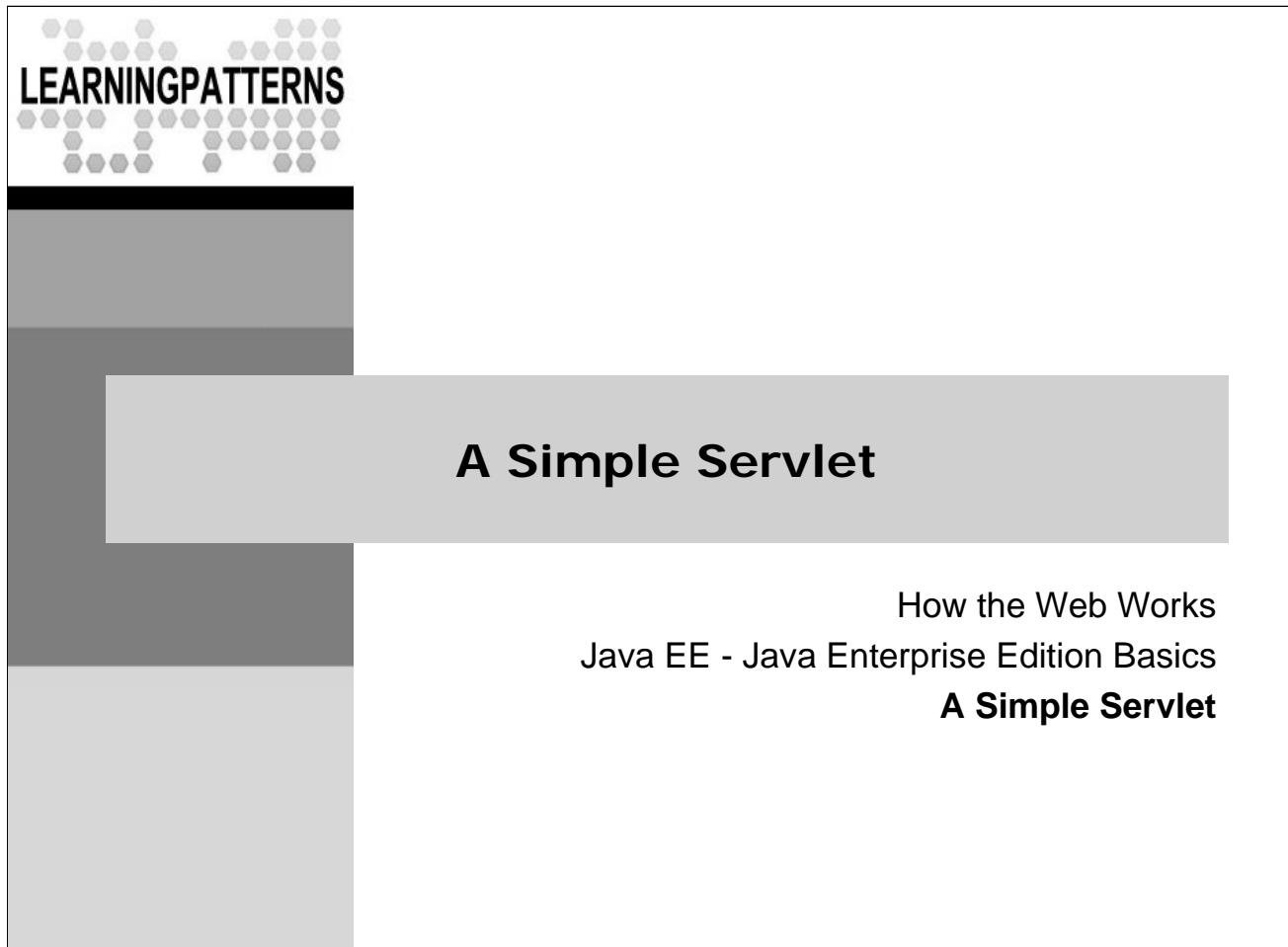
- ◆ You can open the **Console** view to see output from the server startup
 - This is useful to look for exception stack traces in later labs
 - Note that server startup may take some time, especially the first time you start the server
 - You can also look at the server status in the **Servers** view

A screenshot of the Eclipse IDE interface, specifically the Console view. The title bar shows "Tomcat v6.0 Server at localhost [Apache Tomcat] C:\Program Files\Java\jdk1.5.0_15\bin\javaw". The console window displays the following log output:

```
INFO: Starting service Catalina
Jul 8, 2008 2:18:49 PM org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/6.0.10
Jul 8, 2008 2:18:50 PM org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on http-8080
Jul 8, 2008 2:18:50 PM org.apache.jk.common.ChannelSocket init
INFO: JK: ajp13 listening on /0.0.0.0:8009
Jul 8, 2008 2:18:50 PM org.apache.jk.server.JkMain start
INFO: Jk running ID=0 time=0/78 config=null
Jul 8, 2008 2:18:50 PM org.apache.catalina.startup.Catalina start
INFO: Server startup in 1641 ms
```



Notes:



Notes:

Servlets and Dynamic Content

- ◆ Servlets are Java objects that run on the server, **in response to client requests**
 - Servlets are instances of a class whose methods are invoked in response to client requests
 - They can be used to generate dynamic content for the response
- ◆ Some content must be generated **dynamically**
 - For example, the results of a search
 - Servlets can be used to generate this dynamic content
- ◆ Previous approaches to dynamic content had many limitations
 - Common Gateway Interface (CGI)
 - Server extensions like NSAPI and ISAPI

Notes:

- ◆ Some Web pages don't ever change (static content)
 - Static Web content is simply files sitting on a server somewhere
- ◆ Dynamic content is generated on the fly from information
 - For example, if you do web banking, you would want to see your most current account information
- ◆ Early in the evolution of the web, there was no dynamic content
 - People were using it for very simple things
- ◆ Eventually, the Web evolved to where people were doing things that required dynamic content
- ◆ There were two approaches used
 - CGI - A separate process which when invoked generated the dynamic content
 - Proprietary server extensions which allowed you to generate dynamic content
 - The proprietary extensions were basically server side plug-ins

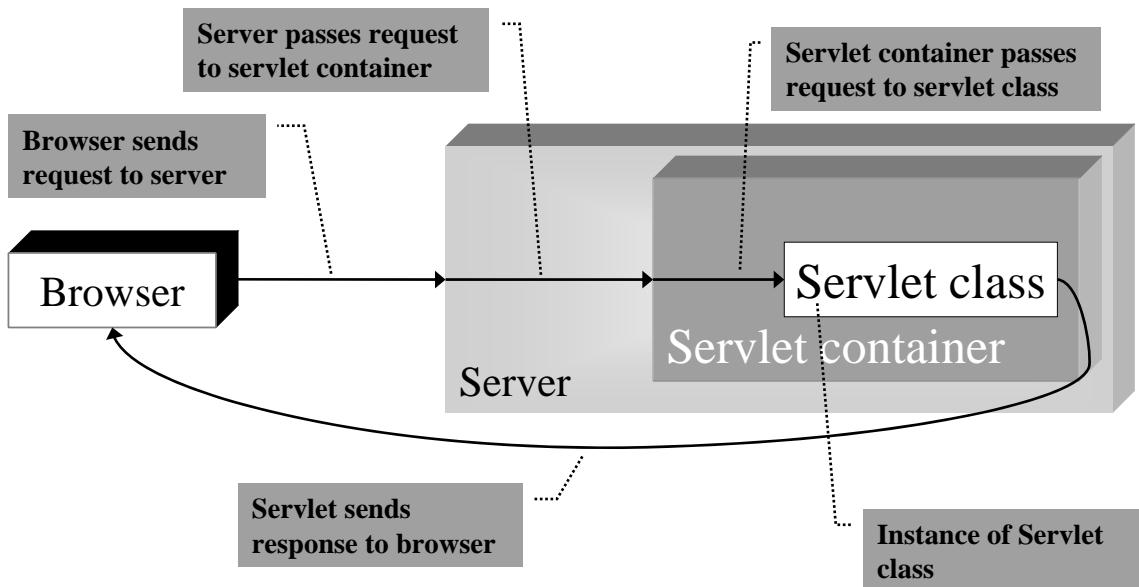
What are Servlets?

- ◆ Servlets run within a **container** that exists as part of a server
 - **Container** is a name for a set of services that must be available to a servlet for it to execute properly
 - It provides network services, handles MIME-based requests/responses, manages servlet lifecycle
- ◆ The servlet container is usually a Web server or server plugin
 - Container must support HTTP(1.0/1.1), and usually HTTPS
 - Servlet support is widespread
- ◆ Most servlet containers are also full-fledged Web servers
 - For example, Tomcat can run as an HTTP server
 - It can also be a plugin for Apache or other servers
- ◆ There are many other servers supporting servlets:
 - JBoss, BEA WebLogic, IBM WebSphere, Sun Application Server, etc.

Notes:

- ◆ Servlets are the Java EE way to respond dynamically to client requests
 - They work alongside JSPs which we talk about later
 - Servlets are really instances of a class whose methods are invoked by the servlet container
- ◆ The servlet container can be provided entirely by the server itself or by a server plug-in
 - Tomcat and other application servers can act as a plug-in servlet container for well known servers like Apache
 - You can architect your system differently depending on your needs

How a Servlet Runs



Notes:

- ◆ A servlet is actually an instance of a Java class
 - It implements a specific interface
 - The methods of the interface are called by the container in a manner that is clearly defined in the servlet spec
 - You write the class to do whatever work you need done to generate your dynamic content
 - For example, you might do a database lookup and then generate HTML from the results of the lookup

Advantages of Servlets

- ◆ Fast performance
 - A servlet call is just a method invocation in a thread
 - Every CGI call is a separate process
- ◆ Rich APIs are available
 - Functionality like sessions that wasn't available in CGI
 - Easy communication between servlets in same JVM
- ◆ Programmer productivity
 - Written in Java and uses a standard API
- ◆ Safety
 - Protects against things like buffer overflows (won't crash server)
 - Allows sandbox-style restrictions on servlets

Notes:

- ◆ CGI was basically a big hack to enable dynamic content
 - When it was created, the systems using it were still relatively small
 - It was never meant to run today's large scale web sites
 - It is slow, even with a later version called Fast CGI
 - It is not very portable, and is vulnerable to hacks/attacks
 - Servlets are much better
- ◆ Proprietary extensions
 - Are just that - proprietary
 - No portability at all
 - They were often hard to work with also
 - They were often written in C, and could crash your server
- ◆ Servlets are much faster than CGI
 - They are also standard
 - It is easier to find developers who can use servlets
 - You are very portable if you want to change platforms

Packages and Classes

- ◆ Servlets are a standard extension library
- ◆ **javax.servlet** is the core servlet package
- ◆ **javax.servlet.http** provides a set of HTTP-specific servlet functionality
- ◆ **javax.servlet.Servlet** is the core interface, implemented in the abstract class **GenericServlet**
- ◆ You will most often extend **javax.servlet.http.HttpServlet**, an abstract subclass of GenericServlet
- ◆ We will go over these types in detail in the next section

Notes:

- ◆ It is one of the key parts of Java EE (Java Enterprise Edition)
 - It is a required part of Java EE
 - Vendors that implement Java EE platforms must support servlets
- ◆ A standard extension library is a standardized Java API that is not part of the core Java implementation
 - Vendors can support it where it makes sense (for example in a web server), and ignore it where they don't need it
 - It will be the same on all implementations

Creating a Servlet - The Simplest Way

- ◆ Subclass the **HttpServlet** class and define a **doGet()** method
 - This overrides the `doGet()` method defined in the `HttpServlet` class
 - For now, we'll show a very simple servlet without thinking about any other details
- ◆ We will actually create a servlet, then install it and invoke it
 - We'll see how to do in the upcoming lab
 - Don't worry about all the details that you see - we'll look at them all in detail later
- ◆ When this servlet is invoked, the container will execute the **doGet()** method
 - The resulting output will go back to the browser

Notes:

A Simple HTTP Servlet

```
package com.javatunes.web;  
import javax.servlet.http.*;  
import javax.servlet.*;  
import java.io.*;  
  
public class SimpleServlet 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("Hello World");  
        out.close();  
    }  
}
```

Notes:

- ◆ There is a lot here we don't understand yet
- ◆ The important point, is that we can write Java code here (`out.println("Hello World")`) and the container takes care of all the mechanics of getting this output from the servlet, back to the client (the browser)
- ◆ In a more complex application, you could first do more complicated things, like a database lookup, and then use it to send output to the client
- ◆ Note: In general, we will NOT generate HTML output this way (print statements in a servlet)
 - This is only illustration
 - We will see how to do this more elegantly when we learn JSP

Declaring and Mapping Servlets

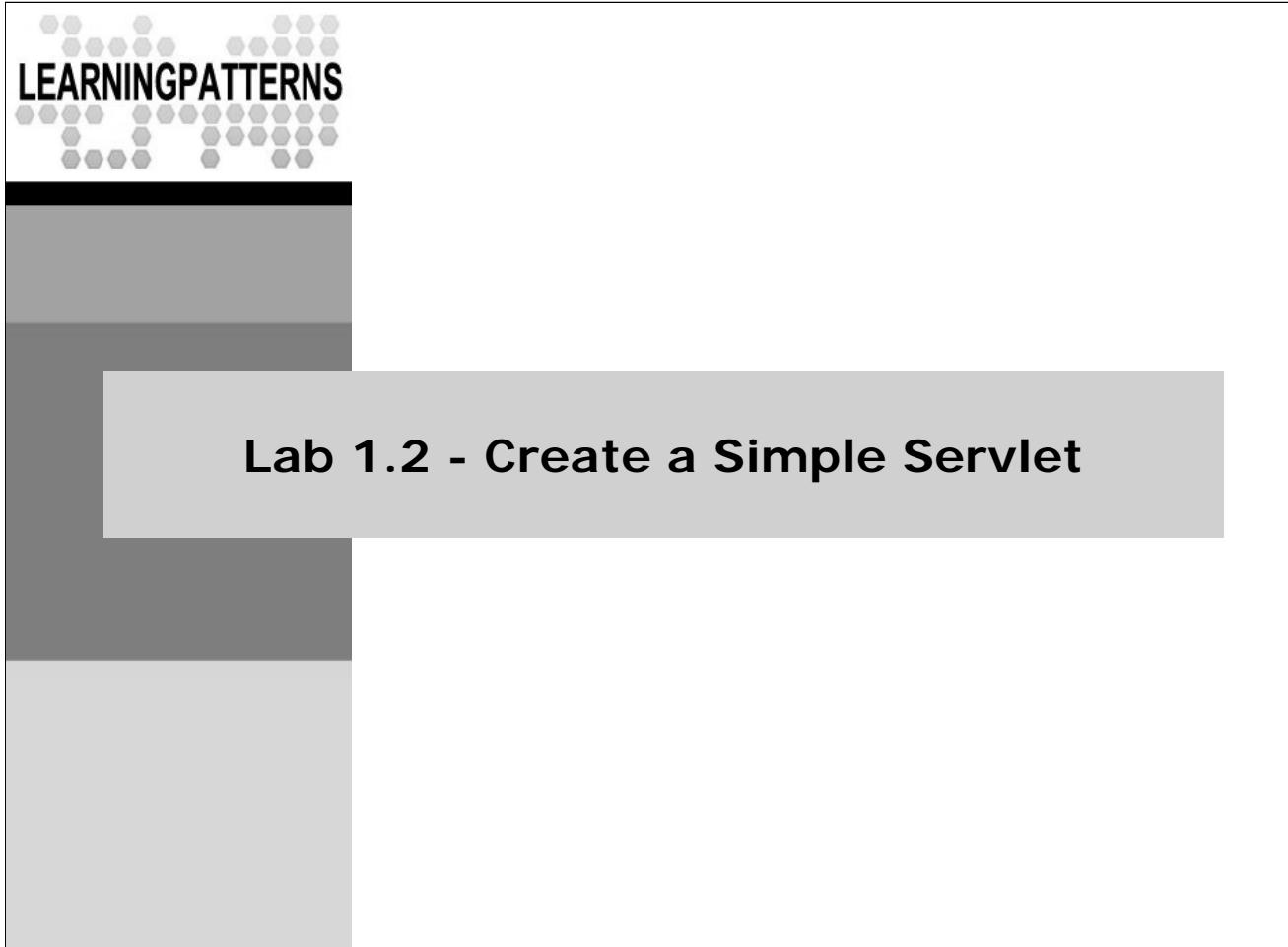
- ◆ A servlet must be declared in *web.xml* so the container recognizes it
 - This includes the servlet class and its URL mappings, as shown below
 - Again - don't worry about the details, we'll cover them later

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

    <display-name>JavaTunes</display-name>
    <servlet>
        <servlet-name>SimpleServlet</servlet-name>
        <servlet-class>com.javatunes.web.SimpleServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SimpleServlet</servlet-name>
        <url-pattern>/simple</url-pattern>
    </servlet-mapping>
</web-app>
```

Notes:

- ◆ The mapping can use wildcards
 - For example /* to map to all URLs
 - Or *.jsp to map to all files ending in JSP
- ◆ The schema elements shown are for the Servlet 2.5 specification
 - Depending on your app server, you might have a schema for a later or earlier specification, such as Servlet 2.4
 - The differences in terms of what you learn in this course are mainly in the schema element in <web-app>
 - Everything else that we cover in this course is the same in Servlet 2.4 and 2.5



Notes:

Lab 1.2 - Create a Simple Servlet



- ◆ **Overview:** In this lab, you'll create a very simple servlet that sends HTML back to the client
 - It will be a subclass of **HttpServlet** that contains a `doGet()` method to handle a request
 - You will also be introduced to the JavaTunes online store which the labs are focused on
- ◆ **Objectives:**
 - Become familiar with servlets, **HttpServlet** and *web.xml*
 - Be introduced to the JavaTunes lab structure
- ◆ **Builds on previous labs:** 1.1
- ◆ **Approximate Time:** 25-35 minutes

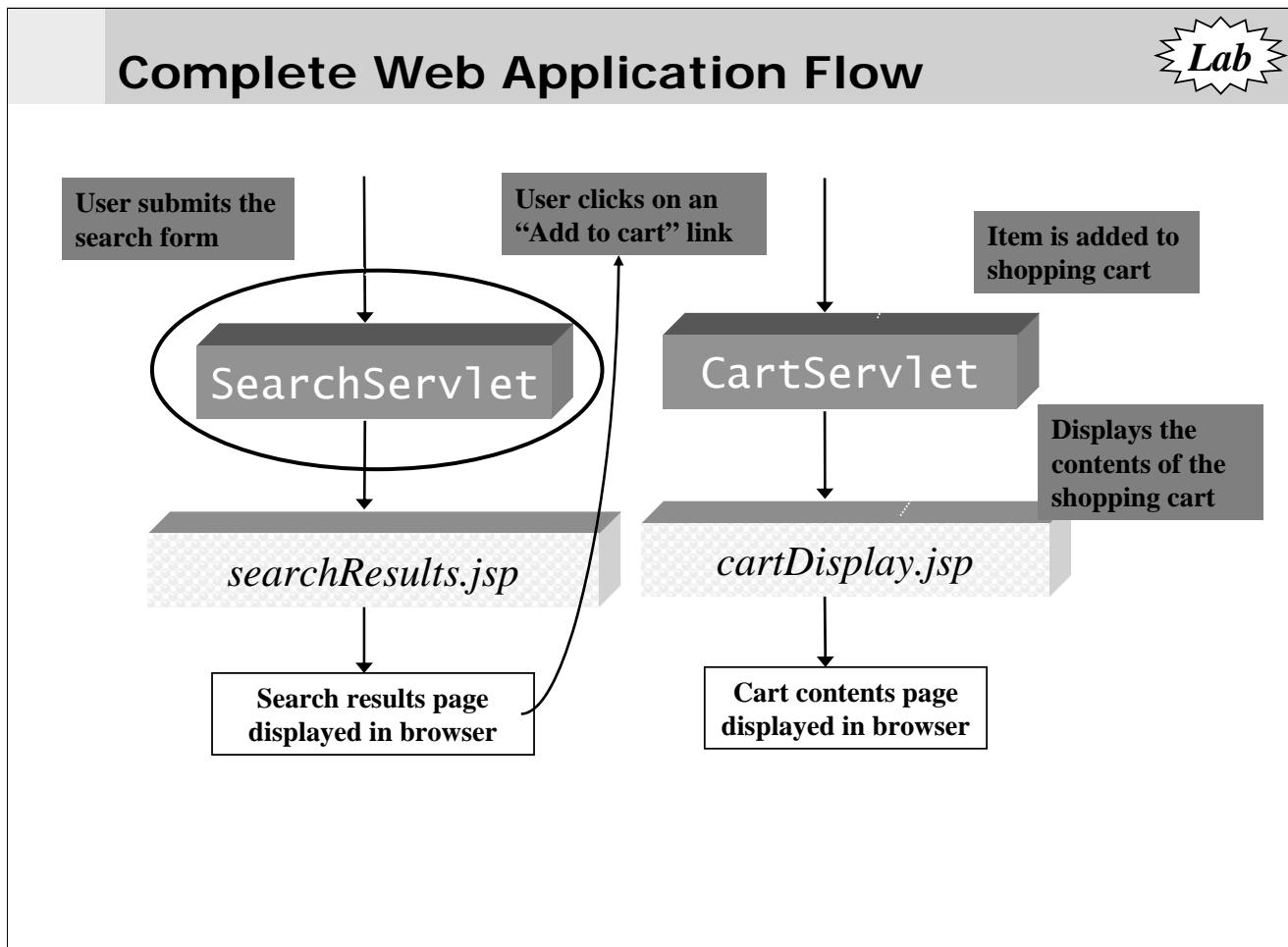
Notes:

The JavaTunes Online Store



- ◆ The Web application we are creating is a small piece of the JavaTunes online music store
 - It displays a search form
 - The search form sends a request to a servlet that does a search and forwards to a results page that displays the results
 - The results page allows you to add items to a shopping cart, which are then displayed on a shopping cart display page
 - We are creating the servlet that processes the search request
 - The complete flow for JavaTunes appears following this slide, but we'll go into more detail on it later

Notes:



Notes:

Creating the Servlet



- ◆ In this lab, we will create a class called **SearchServlet**, a subclass of `javax.servlet.http.HttpServlet`
 - It should be in the `com.javatunes.web` package
- ◆ You will define a `doGet()` method like that in the manual examples that sends back simple HTML
 - You can send back different HTML in the `println` if you want
 - For example: "`<html><body>Searching</body></html>`"
- ◆ Eclipse does most of the work of creating the servlet
 - See the next slides

Notes:

- ◆ We will be putting all our code in packages.
 - This adds a little complication, but it is important to understand.
 - It also reflects what you'll see in the real world.

Declaring and Mapping Servlets



- ◆ All servlets need to be declared in *web.xml*
 - Including their implementing class and their associated URL mappings
 - You associate a name with the servlet, which is used in other parts of the *web.xml* file to refer to the servlet
 - This will be **done by Eclipse** in this lab

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee" ... >
  <display-name>JavaTunes</display-name>
  <servlet>
    <description>Servlet to execute Search</description>
    <display-name>SearchServlet</display-name>
    <servlet-name>SearchServlet</servlet-name>
    <servlet-class>com.javatunes.web.SearchServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>SearchServlet</servlet-name>
    <url-pattern>/search</url-pattern>
  </servlet-mapping>
</web-app>
```

Notes:

- ◆ The mapping can use wildcards
 - For example /* to map to all URLs
 - Or *.jsp to map to all files ending in JSP

Creating a Servlet

Tasks to Perform

- ◆ Right click on the **javatunes** project
 - Select **New | Servlet**
- ◆ In the first dialog box (there are several):
 - In Java package enter **com.javatunes.web**
 - For Class name enter **SearchServlet**
 - Click **Next**


Notes:

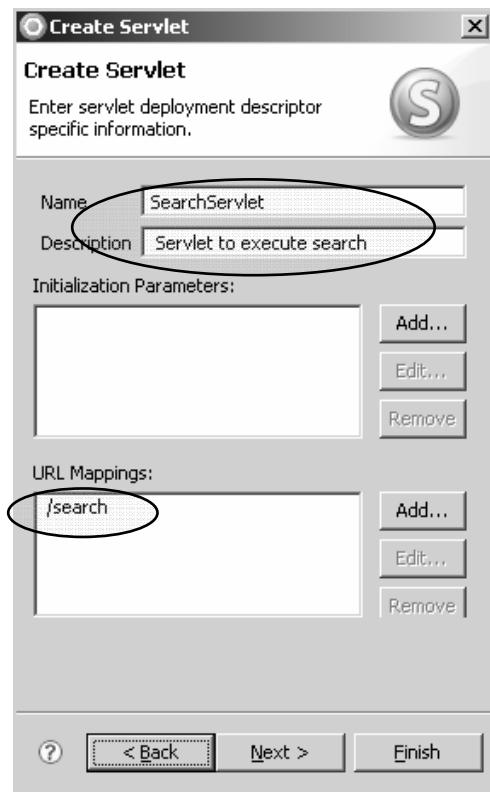
- ◆ Note that the project needs the servlet jar on its classpath to compile
 - Because we associated this project with the Tomcat runtime, Eclipse automatically puts the Tomcat servlet jar on the project's classpath
 - If we had not associated the project with the Tomcat runtime (for example, if we had created the project before we created the server), we would need to manually add the servlet jar to the classpath
 - For Tomcat 6.x, the jar is:
`<Tomcat-Install>\lib\servlet-api.jar`
- ◆ To add the jar to the project manually (you should NOT normally need to do this) you would:
 - Right click on the JavaTunes project in Project Explorer and select **Build Path | Configure Build Path**
 - Select the **Libraries Tab**, and click **Add External Jars**
 - Browse to the appropriate directory (above) and add in the *servlet-api.jar* file
 - Click **OK**

Creating a Servlet



Tasks to Perform

- ◆ In the next dialog
 - In Servlet Name enter **SearchServlet**
 - Under description, enter "**Servlet to execute search**"
- ◆ Select the default /SearchServlet URL mapping and edit it to be **/search**
- ◆ Click **Finish**



Notes:



Edit the Generated Servlet

Tasks to Perform

- ◆ Eclipse generates a servlet for you, and should open it for editing

```

package com.javatunes.web;

import java.io.IOException;

/*
 * Servlet implementation class SearchServlet
 */
public class SearchServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    /**
     * Default constructor.
     */
    public SearchServlet() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse)
     */
    protected void doGet(HttpServletRequest request, HttpServletResponse response) {
        super.doGet(request, response);
    }
}

```

Notes:

- ◆ If it is not open for editing, open it by double clicking on it in the Project Explorer view
- ◆ You may see a warning in the Problems view about servlet-api.jar not being exported, and that possibly causing problems
 - This isn't a problem, because the Web app will be running in a servlet container (Tomcat in this case) that has its own copy of servlet-api.jar
 - You can right click on this problem, and select Quick Fix, then select "Exclude the associated raw classpath entry ... "
 - This will remove the warning

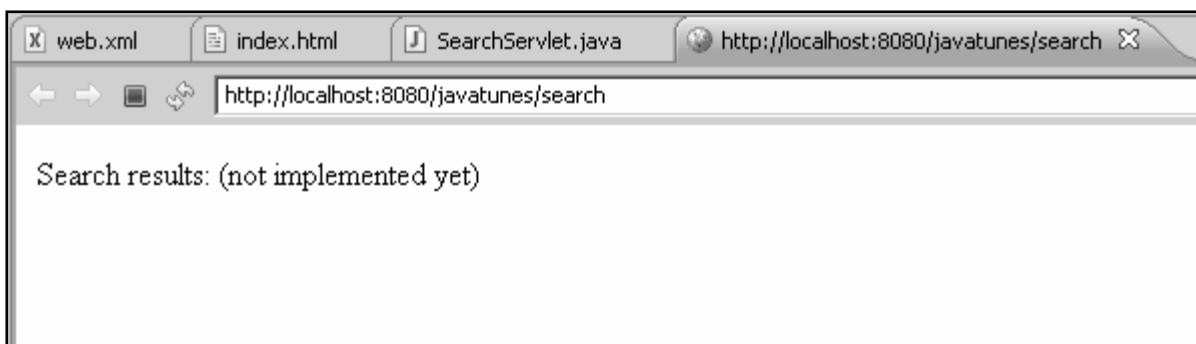
Finish the doGet() method



Tasks to Perform

- ◆ Finish the **doGet()** method so that it sends output to the browser
 - **Delete** the automatically generated `super.doGet()` line
 - See the manual examples that get a `PrintWriter` from the response object, and use that to send back simple HTML
- ◆ You can send back different HTML in the `println` if you want
 - For example: "`<html><body>Searching</body></html>`"
- ◆ In Project Explorer, right click on **SearchServlet.java**, and select **Run As | Run on Server** (If prompted to restart the server, click OK)
 - This will open a browser pointing to **http://localhost:8080/javatunes/search**
 - The container will go to the **javatunes** web app
 - In the Web app, it will see that **/search** is mapped to a servlet
 - It will invoke the **doGet()** method on the servlet, and send the results back to the browser
 - Your browser will display the HTML (see notes)

Notes:



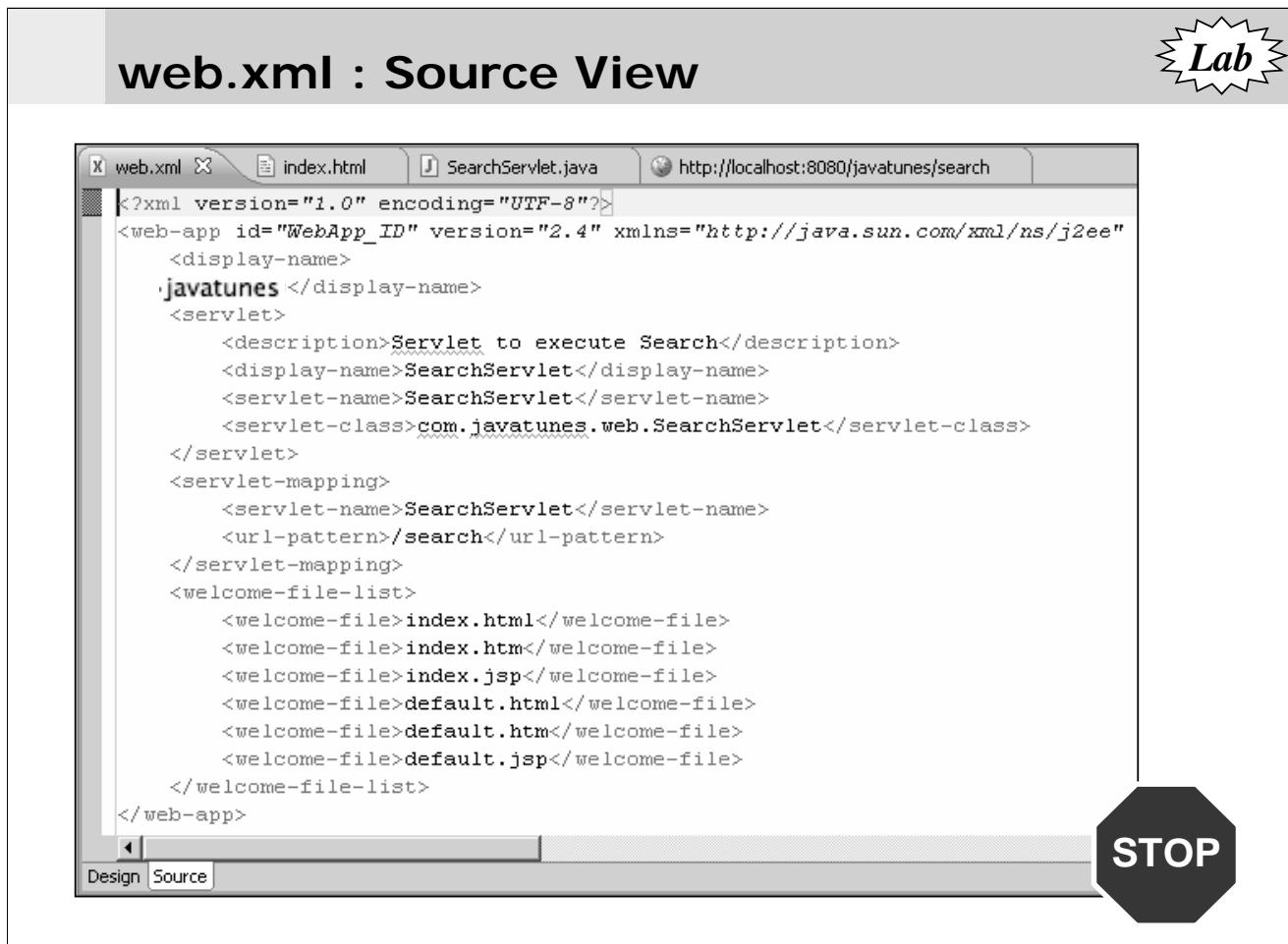
View the **web.xml**



- ◆ The **web.xml** file contains information about the servlets and other resources in a Web application
 - Eclipse created a web.xml for us
 - Open it in source view and look at the elements listed below
 - You don't need to do anything with these elements as Eclipse filled them in with the information you gave when creating the servlet
- ◆ For a servlet, we can declare :
 - **<servlet-name>** - An **internal name** used within the XML file
 - **<display-name>** - A **descriptive name**, used in tools
 - **<servlet-class>** - The actual **class name** of the servlet
 - **<url-pattern>** - A **URL pattern**, which tells the server which URL requests map to the servlet

Notes:

- ◆ There are other entries in the web.xml file that we will get to later

**Notes:**

- ◆ The screen shot shows a Servlet 2.4 version web.xml
 - The 2.5 version would be exactly the same, but would have the schema elements for 2.5, as shown earlier

Review Questions

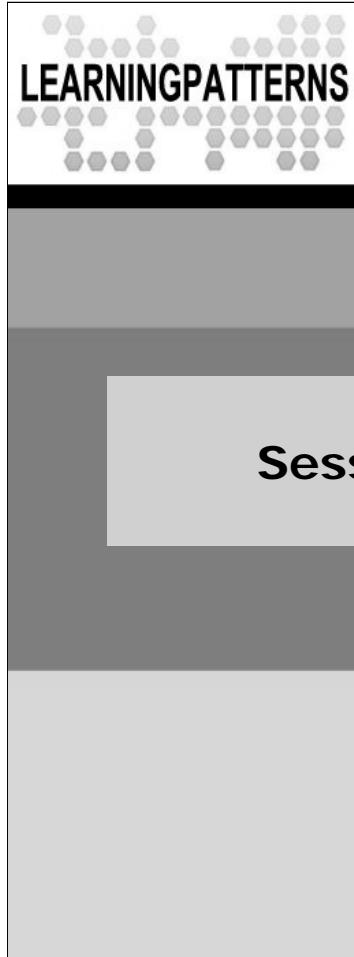
- ◆ What is a servlet?
- ◆ Why do we bother with servlets?
- ◆ How do you write a servlet?

Notes:

Lesson Summary

- ◆ Servlets are Java classes that can be used by servers to respond to Web client requests
 - Providing a standard efficient mechanism to build Web apps
- ◆ Servlets are very easy to create
 - For example, you can just extend HttpServlet and override its doGet() or doPost() method
- ◆ HTML is a text based markup language that uses tags to format content that is displayed in a browser
- ◆ HTML is used by marking up the information you want to display with appropriate HTML tags
- ◆ HTTP is the protocol that governs how information is exchanged between a Web client and a Web server
 - Information is **requested** by a client, and supplied to the client in a **response** from the server

Notes:



The logo for LearningPatterns features the word "LEARNINGPATTERNS" in a bold, sans-serif font. Above the text is a decorative pattern of grey hexagons arranged in a grid-like shape.

Session 7: More JSP Capabilities

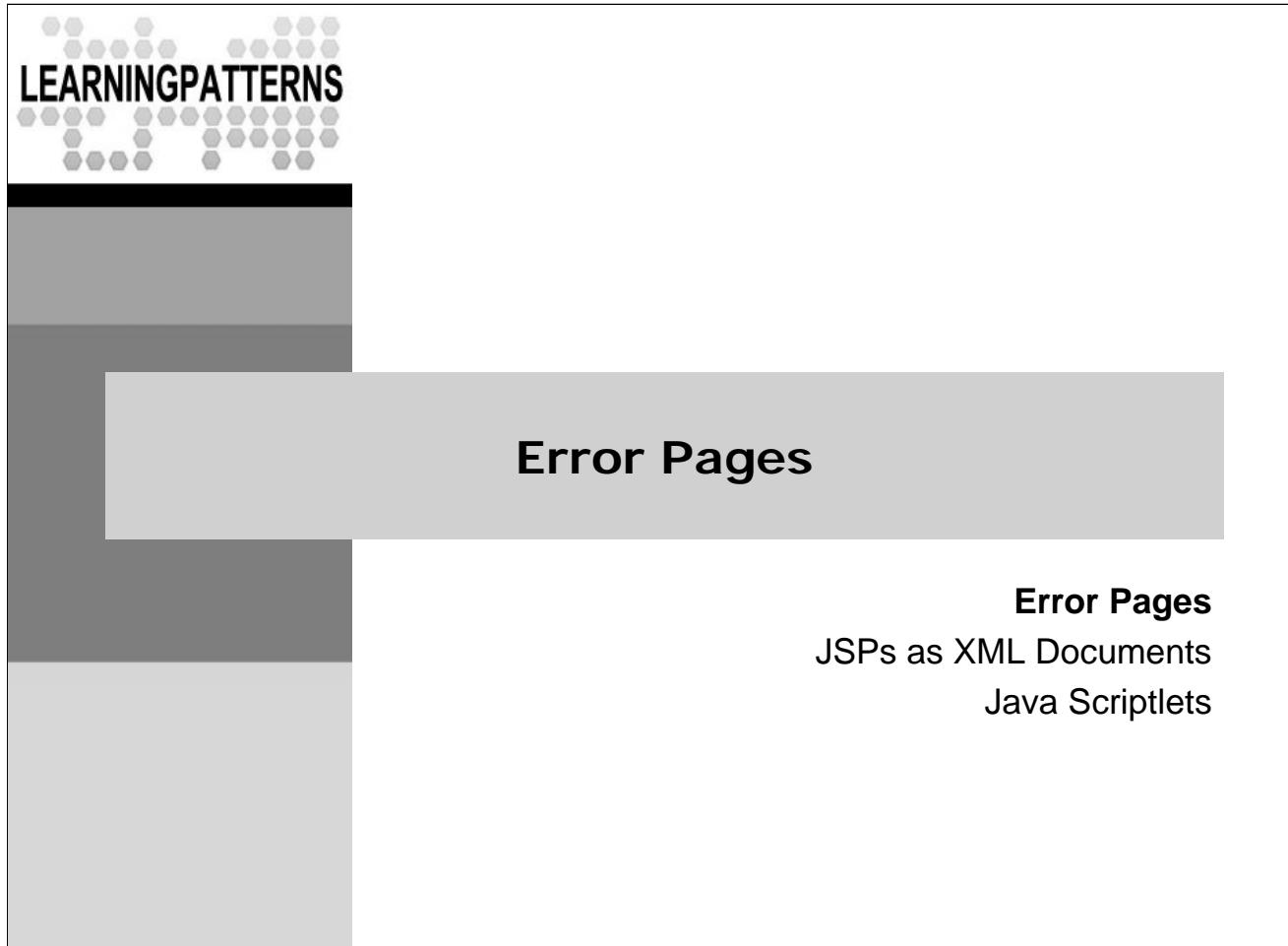
Error Pages
JSPs as XML Documents
Java Scriptlets

Notes:

Lesson Objectives

- ◆ Learn and Use the Error Handling Capabilities of JSP
- ◆ Learn about Java Scriptlets in JSP (but don't use them)

Notes:



Notes:

Servlet Exceptions and Error Pages

- ◆ Servlets generated from JSPs may throw exceptions
 - The `service()` method may throw `IOException` and `ServletException`, & other unchecked exceptions may occur
 - Raw exception reports from the server are likely to be very confusing to the user
- ◆ You can create a JSP specifically for handling exceptions
 - It is a normal JSP, but includes a page directive at the top

```
<%-- error_page.jsp --%>
<%@ page isErrorPage="true" %>
```
- ◆ JSPs that wish to use an error page refer to it like this:

```
<%-- Some other JSP --%>
<%@ page errorPage="error_page.jsp" %>
```

Notes:

- ◆ The **page** directives specifies information about the generated servlet class
 - Allows you to import packages, customize the servlet superclass, set the content type, etc.
 - You can use it multiple times anywhere in a JSP
 - You use attributes to specify what the directive does

The Implicit exception Object

- ◆ Error page JSPs have an additional implicit object, **exception**, that represents the exception that was thrown
 - The exception object is available in the EL through the `pageContext`
- ◆ You can use this object to provide information about the exception, using the following properties
 - `message`: The exception message (may be null)
 - `class`: The exception class object
 - `class.name`: The classname of the exception

```
The exception is:${pageContext.exception.class.name}
```

```
The message is: ${pageContext.exception.message}
```

Notes:

- ◆ Because the exception object is made available to the error page, you can access it through the implicit JSTL `pageContext` variable
- ◆ It is also an implicit object in the JSP itself, so it can be accessed via JSP expressions

```
The error was <%= exception.getMessage() %>  
getClass().getName() will get the exception  
type
```

JSP 2.0 Error Handling

- ◆ JSP 2.0 added improvements to the error handling
- ◆ New implicit EL pageContext variable called **errorData**
 - Contains the properties shown at bottom
- ◆ An error attribute (containing the exception object causing the error) is put on request with same name as for servlets
(javax.servlet.error.exception)
 - Compatible with Servlet error handling

Property	Java Type	Description
requestURI	String	URI for request that failed
ServletName	String	name of servlet or JSP page that failed
statusCode	int	failure status code
throwable	Throwable	Exception that caused error page to be invoked

Notes:

Using **errorData**

- ◆ The example below shows how you can give more information about an error that occurred

```
<%@ page isErrorPage="true" contentType="text/html" %>
```

```
There's been a problem processing your request. Please call  
customer service if you need assistance:
```

```
Request that generated the error:  
${pageContext.errorData.requestURI}
```

```
Status code: ${pageContext.errorData.statusCode}
```

```
Exception information: ${pageContext.errorData.throwable}
```

Notes:

Exception Handling in a JSP

- ◆ Exceptions may be caught and handled in the body of a JSP page
 - JSTL has a catch tag
- ◆ For exceptions that are *not* caught:
 - The client request is *forwarded* to the `errorPage` specified in the JSP's `page` directive
 - The implicit exception object in the specified `errorPage` is initialized to this uncaught exception
- ◆ If no `errorPage` is specified, the server's default behavior for unhandled exceptions will be performed
 - Often, this is a raw exception report
 - **Ugly**, and **scary** to a user!

Notes:

web.xml - Declarative Exception Handling

- ◆ You can also specify a mapping between an error code or exception type to the path of a resource (servlet/JSP/HTML)
 - Done with an **<error-page>** element in your web.xml file
 - Lets you define your own error display for an error or exception
 - Here we send all uncaught exceptions to our error page

```
<error-page>
    <exception-type>java.lang.Exception</exception-type>
    <location>/error_page.jsp</location>
</error-page>
```

- Here we send 404 errors to a special error page

```
<error-page>
    <error-code>404</error-code>
    <location>/not_found.jsp</location>
</error-page>
```

Notes:

- ◆ For exceptions, the container matches the exception type by comparing the exception thrown with the list of error-page definitions that use the exception-type element
 - A match results in the container returning the resource indicated in the location entry
 - The closest match in the class hierarchy wins
- ◆ For error codes, if the sendError method is called on the response, the container consults the list of error page declarations for the Web application that use the error-code syntax and attempts a match
 - If there is a match, the container returns the resource as indicated by the location entry.

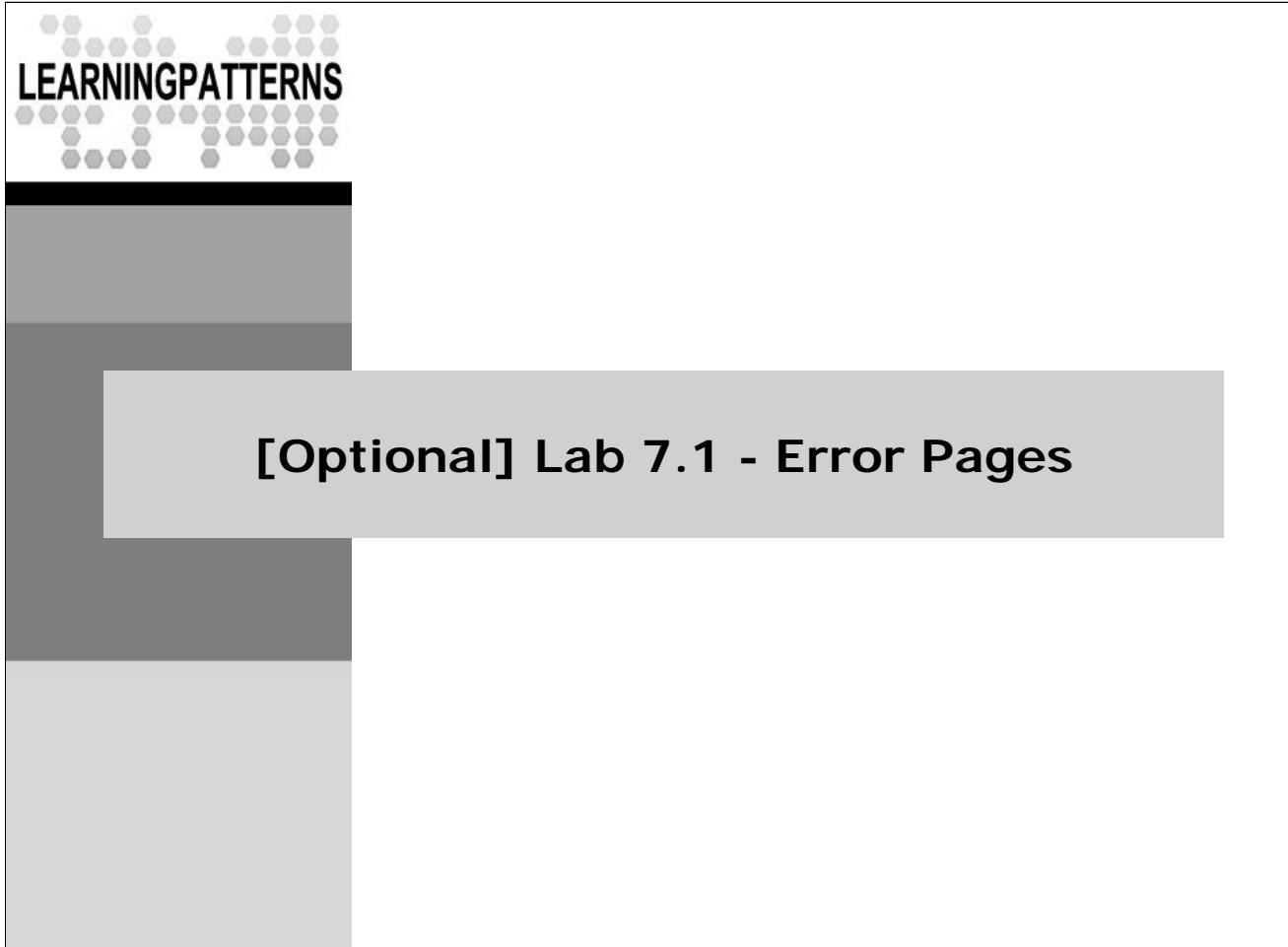
Server Logging

- ◆ Servlets can also write to the server log to log their errors
 - The log is not visible to clients
 - The log is a more permanent record than just writing messages to `System.out`
- ◆ The relevant methods are in `ServletContext` - also implemented in `GenericServlet`
 - `void log(String msg)`
 - `void log(String msg, Throwable t)` - writes out the message and a stack trace for the supplied `Throwable`

```
// Call it directly in a method if you subclass  
// GenericServlet or HttpServlet  
log("Oh oh");
```

Notes:

- ◆ Very often applications will use a more sophisticated framework for logging
 - For example, the log4j or commons logging packages
- ◆ However, the log method is useable for simple logging purposes



Notes:

[Optional] Lab 7.1 - Error Pages



- ◆ **Overview:** In this (optional) lab, you will add an error pages for error handling
- ◆ **Objectives:**
 - Use Error Pages
- ◆ **Builds on previous labs:** 6.1
- ◆ **Approximate Time:** 20-30 minutes

Notes:

Create the Error Page



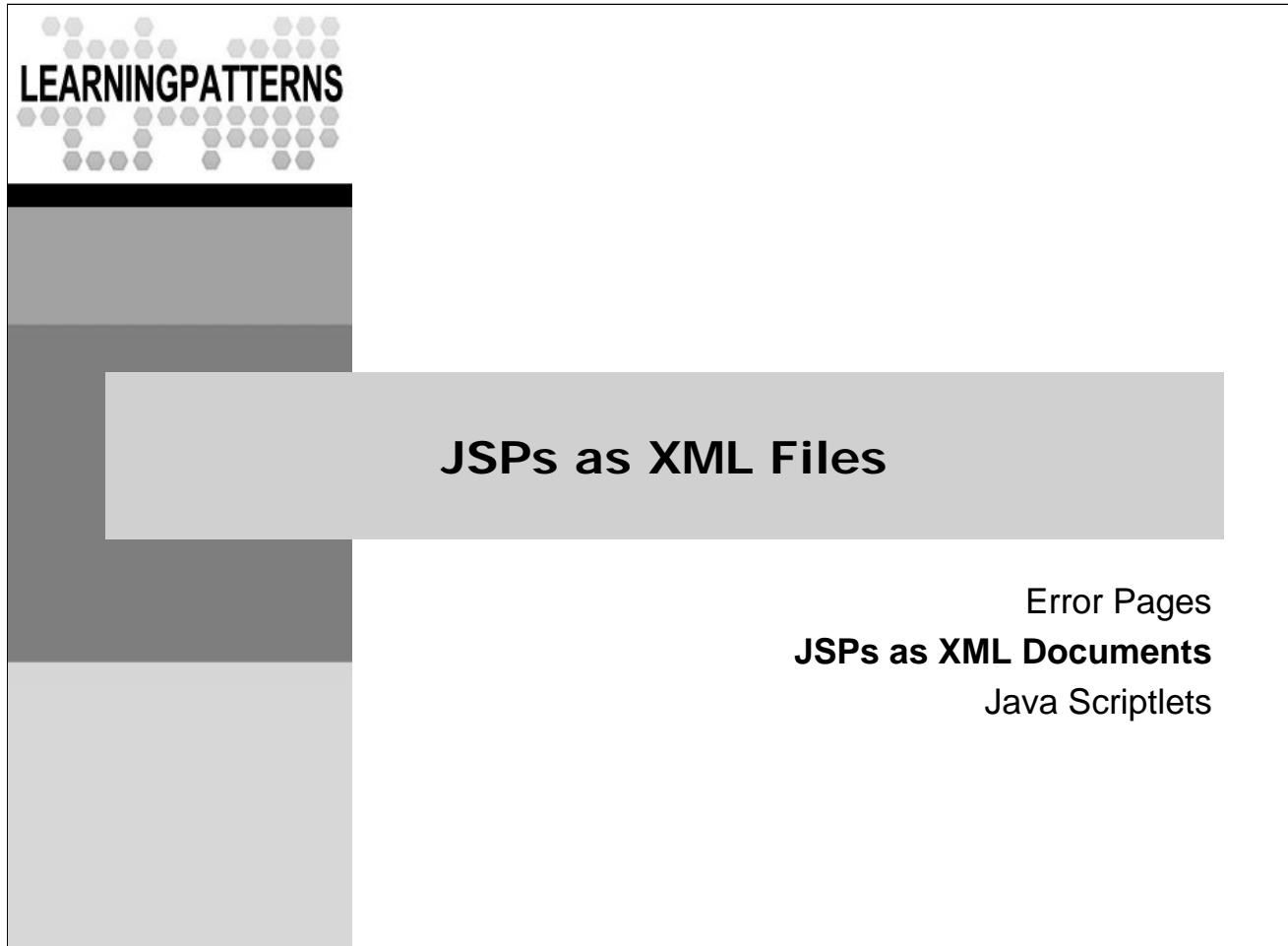
Tasks to Perform

- ◆ Create an error JSP * called **exception.jsp**
 - Use the page directive for this
 - Include *header.jsp* for a common look and feel if you like
 - Output data from the **exception** and **errorData** variables
- ◆ Modify **searchResults.jsp** to use the error JSP
 - Add a page directive specifying *exception.jsp* as the error page
 - Add something that will blow up in *searchResults.jsp*
 - Like a friendly divide by 0 expression
`<%= 5/0 %>`
 - When you access the search results page, you should now see your error page output, not the server's raw exception report
 - Once you're done, **remove the expression** causing the error



Notes:

- ◆ Follow the same procedure to create the JSP page as in earlier labs
 - Put it in the *WebContent\jsp* folder
 - If you need review, go back to the earlier lab



Notes:

JSP Pages as XML Documents

- ◆ JSP 2.0 added improved support for creating **JSP Documents** that are valid XML documents
 - These contain XHTML and JSP elements
- ◆ Replacements for JSP directives and scripting elements
 - To allow JSP elements that are valid XML elements
 - For example:
 - `<% page attribute list %>` becomes
`<jsp:directive.page attribute list />`
 - `<% scriptlet %>` becomes
`<jsp:scriptlet>scriptlet</jsp:scriptlet>`

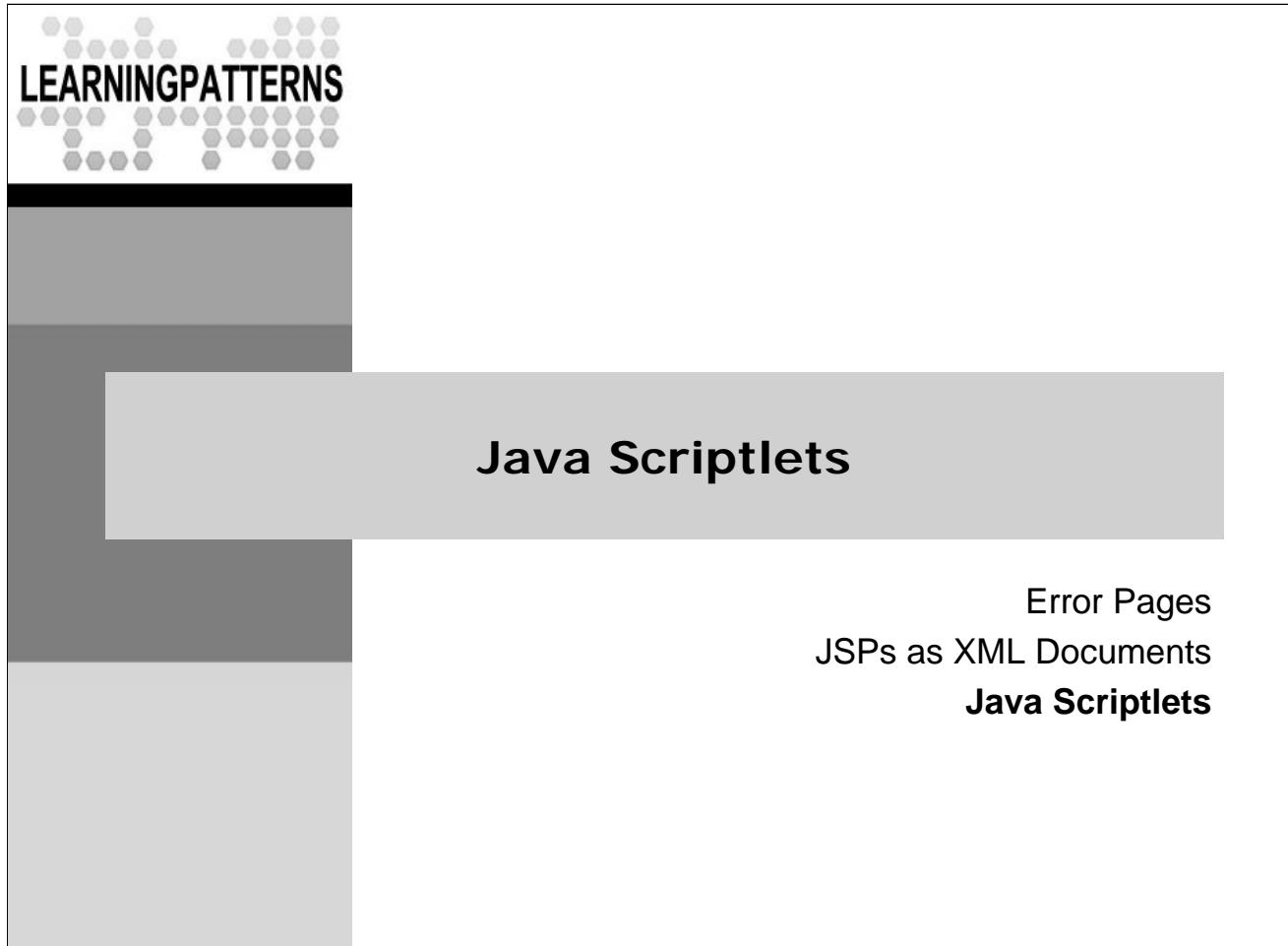
Notes:

JSP Pages as XML Documents

- ◆ Tag libraries are declared as namespaces, instead of with taglib directives
 - They should appear in the <html> root element
 - The following declaration

- ```
<html xmlns="http://www.w3c.org/1999/xhtml"
 xmlns:jsp="http://java.sun.com/JSP/Page"
 xmlns:c="http://java.sun.com/jsp/jstl/core"
 xml:lang="en" lang="en">
```
  
- Other details which we don't go into here

Notes:



**Notes:**

## Scriptlets

- ◆ Scriptlets are pieces of **Java code embedded in a JSP**
  - You can think of this code as being placed directly in the `service()` method of the JSP's corresponding servlet
- ◆ They allow you to use Java code and generate dynamic content directly in a JSP
  - For example, you could use Java for flow of control within a JSP page
- ◆ You may declare and use variables
  - They are equivalent to local variables in the `service()` method

### Notes:

- ◆ They allow you to do more complex things than inserting expressions
  - You can put any Java code that's legal in a scriptlet
  - They can access all the implicit objects of the JSP page
- ◆ The code is actually put in the `_jspService()` method of the servlet
  - This is called by the `service()` method
- ◆ Note that variables that you declare in a scriptlet can be considered, in a way, to be page scope since they are only available to code on that page
  - They are not exactly the same, however, as those created with `useBean`
  - The ones created with `useBean` are added to the `PageContext` (the "bucket" holding the page scope)
  - The ones created in scriptlets are just declared as local variables in the `service` method

## Scriptlet Syntax

- ◆ Scriptlets are marked with <% and %>

```
<% java.text.DateFormat df =
 java.text.DateFormat.getInstance(); %>
A nicely formatted date:
<%= df.format(new java.util.Date()) %>
```

- ◆ Loops are possible, too

```
<% for (int i = 0; i < 10; i++) { %>
Busy <%= String.valueOf(i) %>
<% } %>
– Note that you can weave in Java with HTML
```

- ◆ It gets ugly fast - Use Java sparingly!

### Notes:

- ◆ You want to be judicious in your use of scriptlets
  - It's tempting to put lots of scriptlets in a JSP page
  - The more intermixing of Java and HTML, the harder to maintain
- ◆ You don't want to use JSPs to contain lots of controlling Java code
  - A JSP is for presentation
  - Keep as much logic in servlets and in separate classes as is possible
  - Only use Java in a JSP when it is really necessary, and is useful to have there
- ◆ The way the interleaving of Java and HTML code works is
  - The Java code gets inserted into the `_jspService()` method exactly as written
  - The HTML gets converted to print statements
  - If your scriptlet opens a block (e.g. as part of an if), then the generated print statements are enclosed in that block

## How do Scriptlets Work?

- ◆ Any code you put in scriptlets gets translated into Java code in the generated servlet
  - The code appears in the service() method
- ◆ You can put anything in a scriptlet that you would put in your doGet/doPost method in a servlet
  - Code, comments, etc.
- ◆ You can import packages for scriptlets with the page directive
  - As shown below to use DateFormat and Date

```
<%@ page import="java.text.*, java.util.*" %>
<% DateFormat df = DateFormat.getInstance(); %>
A nicely formatted date:
<%= df.format(new Date()) %>
```

### Notes:

- ◆ The packages to import are a comma separated list
  - Note that this is different from the Java syntax
  - You can also have multiple page/import directives
- ◆ Note: You still need to use the fully qualified package name with jsp:useBean - even if you've done an import
- ◆ When using JSTL, you usually don't need to do an import
  - The EL can access bean properties without needing an import statement

## Another Simple JSP - simple.jsp

```
<HTML>
 <HEAD><TITLE>The time of day</TITLE></HEAD>
 <BODY>
 <CENTER>
 <%
 // Let's create a nicely formatted date:
 // Hey this is a Java comment !
 java.text.DateFormat df =
 java.text.DateFormat.getInstance();
 %>
 Current Time:
 <%= df.format(new java.util.Date()) %>
 </CENTER>
 </BODY>
</HTML>
```

### Notes:

## Generated Code Fragment For simple.jsp

```
// Various declarations and initializations ...
public void _jspService(// ...

// begin [file="/jsp/simple.jsp";from=(4,5);to=(9,3)]

// Let's create a nicely formatted date:
// Hey this is a Java comment !
java.text.DateFormat df = java.text.DateFormat.getInstance();

// end
// HTML // begin [file="/jsp/simple.jsp";from=(9,5);to=(11,3)]
out.write("\r\n\t\t\tCurrent Time:\r\n\t\t\t");

// end
// begin [file="/jsp/simple.jsp";from=(11,6);to=(11,39)]
out.print(df.format(new java.util.Date()));
```

### Notes:

## When to Use Scriptlets

- ◆ As little as possible
- ◆ Look at the custom tag tags available first
  - The JSTL has a good selection
  - There are many other tag libraries around
- ◆ You can often do your work in a servlet
  - For instance, you might be tempted to put some Java code that calculates your shopping cart totals in a scriptlet
  - This can happen in the servlet, with the result passed to the JSP
- ◆ If you can't find a custom tag, consider writing your own
  - It's not that hard, it's cleaner, and it can be reused

Notes:

## Declarations

- ◆ *Declarations* allow you to add member variables and methods to a JSP's generated servlet class
  - Though you probably don't really want to do this
  - These kinds of declarations are not used very much
- ◆ Declarations are marked with `<%!` and `%>`
- ◆ Essentially, declarations are pieces of code placed **inside the braces of the generated servlet class definition**
  - Variables declared in declarations become instance variables in the generated servlet class
- ◆ Compare this with scriptlets, which are pieces of code placed inside the braces of the `service()` method

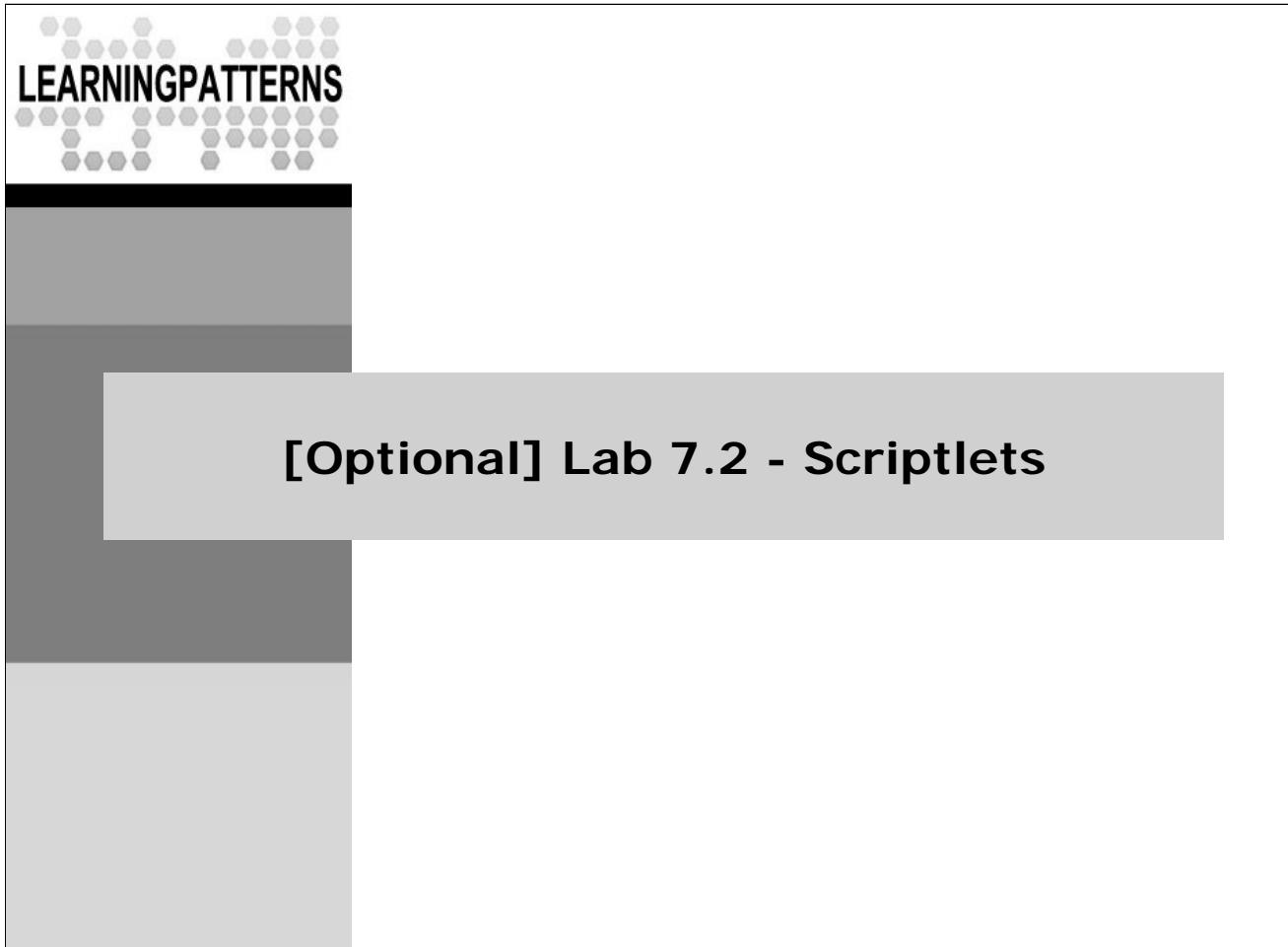
### Notes:

- ◆ Variables declared in declarations become instance variables of the generated servlet
- ◆ Methods become regular methods of the servlet
  - This is not used that often

## Declarations Example

```
<HTML>
 <HEAD>
 <TITLE>
 JSP Counter
 </TITLE>
 </HEAD>
 <BODY>
 <%! private int m_counter = 0; %>
 <%! public synchronized int getCounter() {
 return ++m_counter;
 }
 %>
 The counter's value is <%= getCounter() %>.
 Have a nice day.
 </BODY>
</HTML>
```

Notes:



Notes:

## [Optional] Lab 7.2 - Scriptlets



◆ **Overview:** In this (optional) lab, you will work with scriptlets

- You probably don't need to do this lab unless you either:
  - Know you're using scriptlets to build your system (**why?**)
  - Are supporting an existing system that uses scriptlets and want to practice with them
  - Have lots of extra time on your hands

◆ **Objectives:**

- Use Java Scriptlets in a JSP page

◆ **Builds on previous labs:** 6.1 or 7.1

◆ **Approximate Time:** 30-40 minutes

### Notes:

## Imports and jsp:useBean



### Tasks to Perform

- ◆ We will implement the search results page using scriptlets
  - Copy searchResults.jsp to **searchResults-2.jsp** \*
  - Modify *searchResults-2.jsp*, and keep the original unchanged
  - Change SearchServlet to forward to the *searchResults-2.jsp*
- ◆ First: import collections classes and MusicItem

```
<%@ page import="java.util.*,
 com.javatunes.util.MusicItem" %>
```
- ◆ You'll need to bring the results object into the page with a jsp:useBean tag (shown using Java 5 generic collections)

```
<jsp:useBean id="results"
 type="java.util.Collection<com.javatunes.util.MusicItem>"
 scope="request"/>
```
- ◆ **Delete** the **<c:forEach>** and **</c:forEach>** tags

### Notes:

- ◆ You can copy the Web page from within Eclipse
  - Simply go to it in Project Explorer view, and copy and paste it
- ◆ When using scriptlets, you need to follow all the regular Java coding rules.
  - For example, to use MusicItem, you would import it as shown in the slide.

## Iterating



### Tasks to Perform

- ◆ Continuing in `searchResults-2.jsp`, and loop over the results and print the `MusicItem` data
  - Display the data in some tabular form, as before
  - Keep it simple, just display enough to know it works
  - See the code on the **next slide**
- ◆ **Deploy/Run** the application, and see that it works
  - Once you are done, **change SearchServlet to point back to your normal JSTL version**

### Notes:

## Scriptlets Code



```

<%
for (MusicItem item : results) {
%

<%=item.getId()%> <%=item.getTitle()%> <%=item.getPrice()%>

<%
}
%

```



### Notes:

- ◆ The HTML you generate within the iteration can be whatever you want
  - If you can reuse some of the code from the JSTL version, that's fine
  - Keep the HTML simple, and don't spend much time on it
  - The point is to work with the scriptlets

## Review Questions

- ◆ What are JSP directives for?
- ◆ What are some things you can do with the page directive?
- ◆ How do you declare and use a JSP error page?
- ◆ What are JSP scriptlets?
- ◆ True or False - One should use scriptlets as much as possible in a JSP page.

Notes:

## Lessons Learned

- ◆ JSP directives affect how a JSP page will be translated
- ◆ For example, the page directive can include:
  - Import statements, Error page declarations, Threading directives
- ◆ Error pages can be declared using the **isErrorPage=true** attribute to the page directive
  - Error pages can be used via the **errorCode="..."** attribute to the page directive
- ◆ Scriptlets are sections of Java code embedded in a JSP page
  - Marked by the tags `<% ... %>`
  - They are very powerful, but also very cumbersome to use
- ◆ Use scriptlets with care
  - And consider all other options first

### Notes: