



**JSP EL**

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Contents

- About EL
- Using EL Expressions
- EL Variables
- EL Implicit Objects
- EL Literals
- EL Operators
- EL Reserved Words

A decorative header featuring a close-up, slightly blurred image of several colored pencils (yellow, orange, and red) lying diagonally across the top of the slide. The word "Objectives" is written in a bold, black, sans-serif font in the upper right corner of this header area.

# Objectives

- Give background on how EL is used in JSP
- Discuss how to use EL expressions
- Discuss how variables are treated in EL
- Identify the implicit objects readily available for EL
- Identify EL literals
- Identify EL operators
- Identify EL reserved words
- Illustrate how to define a function and use it in EL expressions



# About EL

- The Expression Language (EL) provides a way to simplify expressions in JSP.
- It is a simple language that is based on:
  - available namespace (the `PageContext` attributes)
  - nested properties and accessors to collections
  - operators (arithmetic, relational and logical)
  - extensible functions mapping into static methods in Java classes
  - a set of implicit objects
- EL is invoked via the construct “`${ exp }`”
- EL expressions can be used in any static text or any tag attribute that accepts an expression



# Using EL Expressions

- In attribute values of actions,
  - Single expression - result is converted to expected type  
`<some:tag attr="\${expr}" />`
  - Multiple expression - all expressions casted to String then converted to expected type  
`<some:tag attr="\${expr1}sometext\${expr2}" />`
- In template text,  
`<html><body>`  
...  
`\${expr}`  
...  
`</body></html>`



# Enabling EL Expressions

- In a `page` directive,
  - Set `isELIgnored` attribute to `false`
- In `web.xml`,
  - In `jsp-config` element and `jsp-property-group` sub-element,
    - Set the `url-pattern` element to include the JSP page(s) using the EL expressions
    - set `el-ignored` element to `false`
  - Ensure that the schema being used by the deployment descriptor is version 2.4 or higher.



# Accessing Nested Properties

- Properties of variables are accessed by the dot “.” operator or the “[]” operator
- For the expression `exp1.exp2`,
  - if `exp1` or `exp2` is null, returns `null`
  - If `exp1` is a map, returns `exp1.get(exp2)`
  - If `exp1` is a list, returns `exp1.get((int)exp2)`
  - If `exp1` is a `JavaBean`, returns `exp1.get((String)exp2)`



# EL Implicit Objects

- `PageContext` – provides access to `servletContext`, `session`, `request`, and `response` objects
- `param`, `paramValues` – map a request parameter to a single value or an array of values
- `header`, `headerValues` – map a request header to a single value or an array of values
- `cookie` – maps a cookie to a single value
- `initParam` – maps a context initialization parameter to a single value
- Scope objects – map values to the following scoped variables:
  - `pageScope`, `requestScope`, `sessionScope`, `applicationScope`





# EL Literals

- Boolean: `true`, `false`
- Integer: same in Java
- Floating point: same in Java
- String: in single (') or double (")quotes
- Object: `null`



# EL Operators

Type	Operators
Arithmetic	<code>* / div % mod + -</code>
Relational	<code>&lt; lt &lt;= le &gt; gt &gt;= ge == eq != ne ? :</code>
Logical	<code>! not &amp;&amp; and    or</code>
Object	<code>empty . [] isinstanceof</code>



# EL Reserved Words

<code>and</code>	<code>div</code>	<code>empty</code>
<code>eq</code>	<code>false</code>	<code>ge</code>
<code>gt</code>	<code>instanceof</code>	<code>le</code>
<code>lt</code>	<code>mod</code>	<code>ne</code>
<code>not</code>	<code>null</code>	<code>or</code>
<code>true</code>		

# Sample Expressions

Expression	Result
<code>\${1 &gt; (4/2)}</code>	false
<code>\${4.0 &gt;= 3}</code>	true
<code>\${(10*10) ne 100}</code>	false
<code>\${'a' &lt; 'b'}</code>	true
<code>\${1.2E4 + 1.4}</code>	12001.4
<code>\${10 mod 4}</code>	2
<code>\${param['mycom.productId']}</code>	(value of parameter)
<code>\${header["host"]}</code>	(host name)
<code>\${pageContext.request.contextPath}</code>	(context path)



# **JSP Standard Tag Library (JSTL)**



# Contents

- About JSTL
- Defining Tag Libraries
- Configuring Tag Libraries
- Using Tag Libraries
- JSTL Tag Libraries
  - Core Tag Library
  - Format Tag Library
  - Functions Tag Library



# About JSTL

- The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications.
- Instead of mixing tags from numerous vendors in your JSP applications, JSTL allows you to employ a single, standard set of tags.
- JSTL has tags such as iterators and conditionals for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

# The Tag Library Descriptor

- A Tag Library Descriptor (TLD) is an XML document that describes a tag library. It contains information about a library as a whole and about each tag contained in the library.
- TLD file names must have the extension “.tld”

Element	Description
<code>tlib-version</code>	The library's version number
<code>jsp-version</code>	The JSP specification version required by the current tag library to function properly
<code>description</code>	A text string describing the library's purpose
<code>uri</code>	An address that uniquely identifies this taglib.
<code>tag</code>	Provides information for each tag. Its sub-elements: <code>name</code> - the name defined after the prefix <code>tag-class</code> - class name for the custom action <code>body-content</code> – values: JSP, Tagdependent, Empty <code>attribute</code> – parameter values used by the tag



A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Deploying Tag Libraries

- When deployed inside a JAR file, the tag library descriptor files must be in the `META-INF` directory, or a subdirectory of it. The JAR file is then dropped into `WEB-INF/lib` directory.
- When deployed directly into a web application, the tag library descriptor files must always be in the `WEB-INF` directory, or some subdirectory of it.

# Configuring web.xml

- A tag library is defined within the `web.xml` by using `jsp-config` element and `taglib` subelement.
- The URI and exact directory location are defined by using both the `taglib-uri` and `taglib-location` tags, respectively.

```
<web-app>
```

```
...
```

```
  <jsp-config>
```

```
    <taglib>
```

```
      <taglib-uri>/Utilities</taglib-uri>
```

```
      <taglib-location>/WEB-INF/tld/Utilities.tld
```

```
    </taglib-location>
```

```
  </taglib>
```

```
  </jsp-config>
```

```
...
```

```
</web-app>
```

# Using Tag Libraries

- Identify the use of custom tag by using the `taglib` directive

Syntax: `<%@ taglib uri="URI" prefix="prefix"%>`

- Mapping options to `uri`:

1. *Absolute URI* - passing an absolute value in the `uri` attribute that represents the location of the tag library's descriptor file


Example: `<%@ taglib uri="/WEB-INF/tld/Utilities.tld" prefix="datef" %>`

2. *Relative URI* - A relative mapping must be configured in the `web.xml` using the `<taglib>` element. Any JSP pages could use the tag library by using this relative URI value in the `taglib` directive.

Example: `<%@ taglib uri="/Utilities" prefix="datef" %>`

3. *Packaged JAR* - providing an absolute path to an external JAR file as the value of the `uri` attribute

Example: `<%@ taglib uri="="/WEB-INF/lib/Utilities.jar" prefix="datef" %>`



# JSTL Tag Libraries

- JSTL is a collection of several tag libraries:
  1. **Core** – contains general-purpose actions such as variable support, flow control, and URL processing
  2. **Format** – a set of actions to format display based on various languages and regions
  3. **Functions** – actions to get collections length and various string manipulation functions



# Core Tag Library

- URI: `http://java.sun.com/jsp/jstl/core`
- Prefix: `c`
- Actions by functional areas:
  1. Variable manipulation and error-handling
    - `<c:out>` `<c:set>` `<c:remove>` `<c:catch>`
  2. Conditionals
    - `<c:if>` `<c:choose>` `<c:when>` `<c:otherwise>`
  3. Iterations
    - `<c:forEach>` `<c:forEachTokens>`
  4. URL actions
    - `<c:url>` `<c:redirect>` `<c:import>`  
`<c:param>`

# Core Tag Library

## Variable Manipulation & Error Handling

- `<c:out>` evaluates an expression and outputs it to the current `JspWriter`

```
<c:out value="A Lonely String" />
```

```
<c:out value="${book.title}" default="Unknown" />
```

- `<c:set>` sets a variable in a particular web application scope

```
<c:set var="code" value="A567" scope="session" />
```

- Your code is:

```
<c:out value="${code}" />
```

# Core Tag Library

## Variable Manipulation & Error Handling

- `<c:remove>` removes a variable from a specific application scope

```
<c:set var="code" value="A567" scope="session" />
```

Your initial code is:

```
<c:out value="${code}" />
```

```
<c:remove var="code" scope="session" />
```

Your code now is:

```
<c:out value="${code}" />
```

# Core Tag Library

## Variable Manipulation & Error Handling

- `<c:catch>` catches exceptions thrown by any nested action

```
<c:catch var="exception">
  (use another tag here)
</c:catch>
<c:if test="${exception != null}">
  Error found!
</c:if>
```



# Core Tag Library Conditionals

- **<c:if>** evaluates a single condition

```
<c:set var="hour" value="10" scope="request" />
<c:if test="${hour <=12}">
  Good Morning!
</c:if>
<c:if test="${hour > 12}">
  Good Afternoon!
</c:if>
```

- **<c:choose>** **<c:when>** **<c:otherwise>** evaluates a set of multiple exclusive conditions

```
<c:set var="hour" value="10" scope="request" />
<c:choose>
  <c:when test="${hour>0 && hour<12}">
    Good Morning!
  </c:when>
  <c:when test="${hour > 12 && hour<18}">
    Good Afternoon!
  </c:when>
  <c:otherwise>
    Good Evening!
  </c:otherwise>
</c:choose>
```

# Core Tag Library Iterations

- `<c:forEach>` repeats a process over a collection of objects or for a fixed no. of iterations

```
<c:forEach var="product" items="${sessionScope.productlist}"/>  
  <c:out value="${product}"/>  
</c:forEach>
```

- `<c:forTokens>` used to iterate over a string of tokens delimited by some character

```
<c:set var="empData"  
  value="Jermaine,Male,26,Java Developer"  
  scope="request" />  
<c:forTokens items="${empData}" delims="," var="field"/>  
  <c:out value="${field}"/>  
</c:forTokens>
```

# Core Tag Library

## URL Manipulation

- `<c:import>` includes URL-based resources and processed within a JSP page

```
<c:import url="http://mysite.com/catalog.xml" var="catxml" />
<x:parse xml="${catxml}" var="catalog" scope="session" />
```

- `<c:url>` constructs a URL based on correct URL rewriting rules

```
<c:url value="http://www.mysite.com/DiplayItems.jsp"/>
```

- `<c:redirect>` used to redirect client to another url

```
<c:redirect value="http://www.newsite.com/index.jsp"/>
```

- `<c:param>` used to pass request parameters to URL by appending them to the query string

```
<c:url value="http://www.mysite.com/DiplayItems.jsp">
  <c:param name="category" value="electronics"/>
</c:url>
```



# Format Tag Library

- URI: `http://java.sun.com/jsp/jstl/fmt`
- Prefix: `fmt`
- Actions by functional areas:
  - Setting locale
    - `<fmt:setLocale>`
    - `<fmt:requestEncoding>`
  - Messaging
    - `<fmt:bundle>` `<fmt:setBundle>`
    - `<fmt:message>`
  - Formatting
    - `<fmt:timeZone>` `<fmt:setTimeZone>`
    - `<fmt:formatDate>` `<fmt:parseDate>`
    - `<fmt:formatNumber>` `<fmt:parseNumber>`

# Format Tag Library

## Set Locale & Messaging

<code>&lt;fmt:setLocale&gt;</code>	Overrides the client locale for processing of a JSP page
<code>&lt;fmt:requestEncoding&gt;</code>	Used to set the request's character encoding to decode parameters whose encoding is different from ISO-8859-1
<code>&lt;fmt:bundle&gt;</code> <code>&lt;fmt:setBundle&gt;</code>	Specifies the required resource bundle that provides the localized messages
<code>&lt;fmt:setBundle&gt;</code>	Used to redirect client to another url
<code>&lt;fmt:message&gt;</code>	Used to retrieve message from a resource bundle

### •Example:

```
<fmt:setLocale value="en_US" />
<fmt:bundle basename="labels">
    <fmt:message key="labels.name"/>
    <fmt:message key="labels.age"/>
    <fmt:message key="labels.sex"/>
</fmt:bundle>
```

# Format Tag Library

## Formatting

<code>&lt;fmt:timezone&gt;</code> <code>&lt;fmt:setTimeZone&gt;</code>	Displays date & time according to preferred time zone of client
	<pre>&lt;fmt:timeZone value="GMT"&gt; (date &amp; time actions here) &lt;/fmt:timeZone&gt;</pre>
<code>&lt;fmt:formatDate&gt;</code>	Provides time zone-aware formatting of java.util.Date objects
	<pre>&lt;fmt:formatDate value="\${today}"/&gt; &lt;fmt:formatDate value="\${today}" type="DATE" pattern="dd/MM/yyyy"/&gt; &lt;fmt:formatDate value="\${today}" type="TIME" dateStyle="MEDIUM"/&gt;</pre>
<code>&lt;fmt:parseDate&gt;</code>	Parses and converts the string representation of dates and times into java.util.Date objects
	<pre>&lt;fmt:parseDate type="date" pattern="dd/MM/yyyy" var="parsedDate"&gt; 04/03/2005 &lt;/fmt:parseDate&gt;</pre>
<code>&lt;fmt:formatNumber&gt;</code>	Formats numeric value in a locale-sensitive or custom format as a number, currency, or percentage
	<pre>&lt;fmt:formatNumber&gt; &lt;fmt:formatNumber type="CURRENCY" value="\${discount}" /&gt;</pre>
<code>&lt;fmt:parseNumber&gt;</code>	The reverse formatting tag, to convert a formatted string into appropriate java.lang.Number



# Functions Tag Library

- URI: <http://java.sun.com/jsp/jstl/functions>
- Prefix: `fn`
- Actions list:
  - `<fn:length>`
  - `<fn:toUpperCase>` `<fn:toLowerCase>`
  - `<fn:substring>` `<fn:substringAfter>`  
`<fn:substringBefore>`
  - `<fn:trim>` `<replace>`
  - `<fn:indexOf>` `<fn:startsWith>`  
`<fn:endsWith>`
  - `<fn:contains>` `<fn:containsIgnoreCase>`
  - `<fn:split>` `<fn:join>`
  - `<fn:escapeXml>`





# **JEE Best Practices (Developing Web Component)**





# Developing Web Components

- Use a generic MVC implementation
  - The MVC architecture pattern should be used to isolate and modularize screen logic, control logic, and business logic.
  - The tight integration between the controller architecture and the tag library is used to rapidly create JSP components
  - This automates much of the forms processing, keeps the JSP code relatively clean, and allows the control logic to be isolated in action classes

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Developing Web Components

- Use a JSP template mechanism
  - It's an excellent way to apply a common look and feel to your web pages
  - It's easy to make global changes to common aspects of pages and reduces the amount of duplicated code



# Developing Web Components

- Don't hard-code urls
  - Abstract physical URLs out of the application into metadata to enable automation, avoid broken links, and create easy to maintain pages
  - The controller architecture can use the metadata to resolve logical page names and forward control to the corresponding physical URL, typically implemented by a JSP



# Developing Web Components

- Keep the session size to a minimum
  - A large session size can quickly degrade the scalability and performance of high throughput applications
  - In general, use the session to store a minimal amount of state needed to maintain future operations
  - Use the `request` scope wherever possible to pass data from the controller architecture to JSP components



# Developing Web Components

- Minimize the amount of Java code in a JSP
  - Use custom tags wherever possible to encapsulate presentation logic
  - If you can't avoid scriptlets, use a few large scriptlets as opposed to interspersing Java code with HTML content



# **JEE Best Practices**

## Best Practices in:

1. Developing Web Components
2. Developing Business Components
3. Implementing Data Services
4. J2EE Development
5. Designing for Performance



# Developing Web Components

- Use a generic MVC implementation
  - The MVC architecture pattern should be used to isolate and modularize screen logic, control logic, and business logic.
  - The tight integration between the controller architecture and the tag library is used to rapidly create JSP components
  - This automates much of the forms processing, keeps the JSP code relatively clean, and allows the control logic to be isolated in action classes



A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across a light-colored surface.

# Developing Web Components

- Use a JSP template mechanism
  - It's an excellent way to apply a common look and feel to your web pages
  - It's easy to make global changes to common aspects of pages and reduces the amount of duplicated code



# Developing Web Components

- Don't hard-code urls
  - Abstract physical URLs out of the application into metadata to enable automation, avoid broken links, and create easy to maintain pages
  - The controller architecture can use the metadata to resolve logical page names and forward control to the corresponding physical URL, typically implemented by a JSP



# Developing Web Components

- Keep the session size to a minimum
  - A large session size can quickly degrade the scalability and performance of high throughput applications
  - In general, use the session to store a minimal amount of state needed to maintain future operations
  - Use the `request` scope wherever possible to pass data from the controller architecture to JSP components

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across a light-colored surface.

# Developing Web Components

- Minimize the amount of Java code in a JSP
  - Use custom tags wherever possible to encapsulate presentation logic
  - If you can't avoid scriptlets, use a few large scriptlets as opposed to interspersing Java code with HTML content



# Developing Business Components

- Use stateless session beans over stateful session beans
  - In J2EE Application Servers, requests for stateless session beans are load-balanced across all of the members of a cluster, while requests for stateful session beans cannot be load-balanced
  - Any user-specific state necessary for processing should either be passed in as an argument to the EJB methods (and stored outside the EJB through a mechanism like the HttpSession) or be retrieved as part of the EJB transaction from a persistent back-end store (for instance, through the use of Entity beans)

A decorative header image showing several colored pencils (yellow, orange, red, blue) lying diagonally across the top of the slide.

# Developing Business Components

- Use Session Facades when accessing EJB components
  - Never expose Entity beans directly to any client type.
  - Create very large-grained facade objects that wrap logical subsystems and that can accomplish useful business functions in a single method call.
  - Not only will this reduce network overhead, but within EJBs, it also reduces the number of database calls by creating a single transaction context for the entire business function.



# Developing Business Components

- Use the session bean as a component wrapper
  - Implement the actual workflow and transaction logic of the service in a regular Java class.
  - This class can then be wrapped by a Session Bean to engage the container-managed services of transaction management, distribution, and so on.
  - This allows services to be reused within other services more efficiently because you can avoid another EJB method invocation.

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Developing Business Components

- Build generic, reusable components
  - Use generic services to realize the benefits of automation, rapid application development, and increased code quality and maintainability
  - Look for data retrieval patterns and other common business logic in your application for other possibilities.





# Developing Business Components

- Deciding between entity beans and regular java objects
  - Applications with many fine-grained objects or a large number of business objects in a given transaction should use Entity Beans.
  - Java classes provide a more lightweight alternative, but you lose the standard component model and you need to implement the equivalent component services on your own.
  - Combination of the two, use Entity Beans where they provide the most value in terms of optimized persistence, standard component deployment, and distribution.

A decorative header image showing several colored pencils (yellow, orange, red, blue) lying diagonally across a light-colored surface.

# Developing Business Components

- Using CMP entity beans instead of BMP where possible
  - CMP Entity Beans should typically be used to take advantage of container optimizations and avoid two database hits (ie, one for finder method and one for ejbLoad operation).
  - BMP can be used to support object-relational mapping strategies not supported by the container or to manage the persistence of dependent Java business objects.



# Developing Business Components

- Managing aggregated Business objects
  - Create and delete operations of business objects should also encapsulate the corresponding creation and deletion of aggregated objects that share the same lifecycle.
  - This logic can be placed in template methods of your own Java business object implementation or the equivalent Entity Bean hook methods.
  - In the case of cascading deletes, Entity Beans can be configured so the logic is accomplished by the container automatically when the parent object is deleted.

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Developing Business Components

- Using EJB local interfaces wherever possible
  - In many cases, EJB components do not need to be distributed. Thus, they are typically deployed uniformly throughout an application server environment.
  - Local interfaces should be used in these cases to avoid RMI and serialization overhead on method invocations.



# Implementing Data Services

- Isolate and encapsulate JDBC logic
  - Isolate any JDBC logic to execute a SQL statement in a common utility class. This prevents every application developer from having to write this common logic and ensures that all JDBC resources are closed properly.
  - The implementation of regular Java business objects and BMP Entity Beans can use the JDBC utility for object data persistence in the database.
  - In the case of CMP Entity Beans, this utility might still be used for read-only operations such as executing join queries.



# Implementing Data Services

- Externalize SQL from the application code
  - If JDBC is used, externalize the SQL from the Java code to minimize impacts to the application if the database schema changes.
  - The SQL strings could be stored in a resource file or in the XML metadata and then referenced from the application. This approach also makes it fairly easy to determine impacts to the application if the database schema changes, because the SQL is all in one searchable repository.



# Implementing Data Services

- **Minimize use of database names in code**
  - One of the primary risks of using queries directly in your application is that the database column names can start to appear all over the code if not managed well.
  - This can be a maintenance nightmare if the database schema changes or even if a few column names change.
  - Map the result set rows to some kind of value object with logical property names in order to isolate or eliminate references to database names in application code.





# Implementing Data Services

- Use one-to-one object-relational mapping
  - Keep the object-relational mapping approach simple by using a one-to-one table mapping.
  - This allows for a standard code-generation process to create the data-access objects. EJB 2.0 local interfaces provide an efficient way to access related Entity Beans co-located in the same JVM.



A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Implementing Data Services

- Implement complex persistence options
  - Generate data access objects that map one-to-one for each table and then implement business object logic that uses the appropriate set of data objects.



# J2EE Development

- Apply automated Unit Tests and test every layer
  - Before you can test your code, you need to isolate it into testable fragments. A "big ball of mud" is hard to test because it does not do a single, easily identifiable function.
  - Don't just test your GUI. The GUI is only one way of affecting the system. There may be background jobs, scripts, and various other access points that also need to be tested
  - Distributed, component based development with EJBs and Web services makes testing your individual components absolutely necessary.



# J2EE Development

- Map EJB-level exceptions to application-defined errors
  - Map EJB with defined business errors for user-friendly error messages.
  - This simplifies the client code because there is only one high-level application exception that needs to be handled.
  - The messages can also be used as templates so that run-time values can be added to provide additional information.



# J2EE Development

- Develop to the specifications, not to the application server
  - Avoid relying on built-in application server mechanisms for authentication and authorization, instead use J2EE security through JAAS
  - Likewise, rely on the authorization mechanisms provided by the servlet and EJB specs
  - Avoid using persistence mechanisms that are not tied into the J2EE spec (making transaction management difficult)
  - Doing otherwise can cause no end of difficulties when moving from one J2EE compliant server to another, or even when moving to new versions of the same server

A decorative header image showing several colored pencils (yellow, orange, red, blue) lying diagonally across the top of the slide.

# J2EE Development

- **Build iteratively**
  - Iterative development allows you to gradually master all the moving pieces of J2EE.
  - Build small, vertical slices through your application rather than doing everything at once.
  - J2EE is big. If a development team is just starting with J2EE, it is far too difficult to try to learn it all at once. There are simply too many concepts and APIs to master. The key to success in this environment is to take J2EE on in small, controlled steps.



# J2EE Development

- Use container-managed security
  - Use J2EE security whenever possible to safely protect application resources.
  - Container-managed security is portable across J2EE environments and integrates well with enterprise security architectures.
  - Lock down all your EJBs and URLs to at least all authenticated users
  - Leverage the J2EE authentication model and J2EE roles in conjunction with your specific extended rules.



# J2EE Development

- Use Container-Managed Transactions
  - Learn how 2-phase commit transactions work in J2EE and rely on them rather than developing your own transaction management. The container will almost always be better at transaction optimization.
  - If you use CMT and access to 2-phase commit capable resources (like JMS and most databases) in a single CMT, J2EE Application Server will take care of the dirty work. It will make sure that the transaction is entirely done or entirely not done, including failure cases such as a system crash, database crash, or whatever.





# J2EE Development

- Use a Business Object Factory abstraction
  - Use a factory method abstraction to create and discover instances of business objects. This simplifies the client code and provides a hook for optimizations such as caching EJB Home interfaces.
  - A common interface or base class can be used to create implementations for each type of business object. The find method on the factory interface encapsulates the logic necessary to look up an object.
  - In the case of Entity Beans, this prevents the developer from having to use JNDI and the EJB Home interface every time an Entity Bean is needed.



A decorative header image showing several colored pencils (yellow, orange, red, blue) lying diagonally across the top of the slide.

# J2EE Development

- **Use a Collection Service Object**
  - Managing a list of objects for data retrieval or for selective updates is a common operation in business applications. Consider the use of a utility class that consistently and effectively manages collections of objects for you.



# J2EE Development

- Use of persistence tools
  - For both automation and complex database mapping, it is best to use an object-relational mapping tool or vendor-specific persistence mechanism if you can afford the additional overhead cost in terms of performance.
  - Entity Beans and Java objects using persistence frameworks are still largely portable even if a vendor-specific mechanism is used at deployment.



# Designing Performance

- **Minimizing Object Instantiation whenever possible**
  - Use lazy instantiation to delay object creation until necessary. Pay particular attention to objects that are serialized and sent over RMI. If you are invoking a remote Session Bean, try to send only the object data that is required for the component method.



# Designing Performance

- Use an extensible caching mechanism
  - Cache frequently used data that is fairly static to speed application performance. Use a consistent, extensible cache mechanism that can be implemented either as a stateless Session Bean or as a Java singleton class.
  - If the cached data can be updated and the application needs the latest data, consider the use of a JMS solution for notifying caches across a cluster to refresh their data.
  - If there are large volumes of data or if the data is updated frequently, using the application database is likely to be just as efficient as a caching solution.

A decorative header image featuring several colored pencils (yellow, orange, and purple) lying diagonally across the top of the slide.

# Designing Performance

- Removing stateful session beans when finished
  - Be sure to remove instances of stateful Session Beans when you are done with them to avoid unnecessary container overhead and processing.

A decorative header image showing several colored pencils (yellow, orange, red, purple) lying diagonally across the top of the slide.

# Designing Performance

- Use asynchronous processing
  - Asynchronous processing is an option that can be used to alleviate performance concerns in applications with semi-real-time updates, multiple external applications that can be invoked in parallel, or work that can be partitioned into segments.
  - Use Message-Driven Beans and JMS to implement parallel processing in a J2EE container. Asynchronous processing can also be used to increase the perceived performance of an application.

# Questions and Comments

