JavaBeans and JSPs

- Calling a JavaBean from a JSP
- Architecture and Design

Separating Content from Presentation

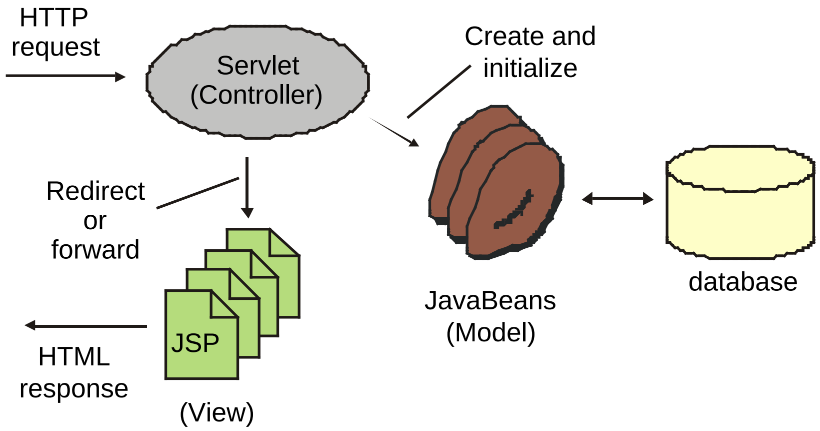
- If you are not careful, you can end up with JSPs that are "cluttered" with too much code in scriptlets, obscuring the templated text
- That makes pages hard to understand and maintain -- in addition, if the business logic is in the JSP itself, you need to update the JSP if the logic changes
- A better design is to move the bulk of the application's data and logic to an external entity: JavaBeans, EJBs or custom actions

Remember that JSPs are designed to be written and maintained by non-programmers -- if you have too much scriptlet code, you lose this advantage. In fact, the ideal JSP has exactly zero lines of scriptlet code -- you can use the techniques in this chapter to approach this ideal.

We will cover custom actions in more detail in a later chapter.

The Model-View-Controller Architecture

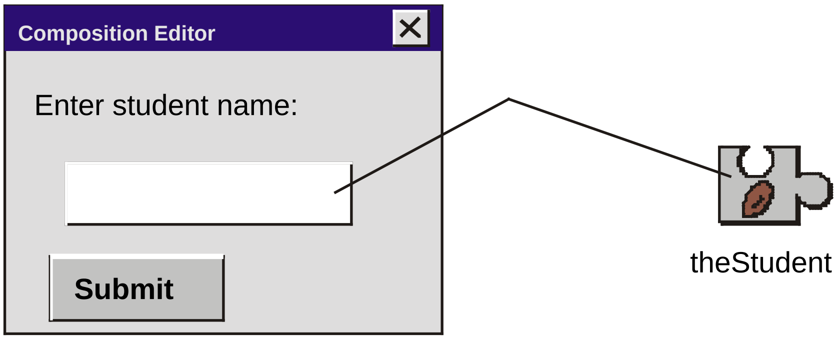
- One very useful pattern you can use to combine servlets and JSPs is referred to as MVC
- Until we cover MVC in detail, keep this in mind as we cover JSP fundamentals



To reduce code bulk in your JSPs, you can separate out the business logic and data into external JavaBeans. This will map nicely to the MVC architecture that we will cover in detail later.

What is a JavaBean?

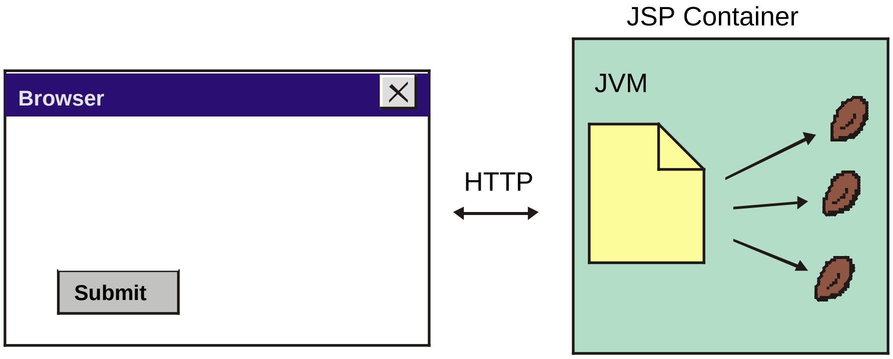
- A JavaBean is a Java class that is designed to be manipulated by tools
- Sun created JavaBeans mainly to act as GUI widgets, but has always supported the notion of "invisible" beans



Sun invented the JavaBean specification originally so programmers could use visual design tools to layout Java graphical user interfaces. However, the notion of Beans has proven to be more widely applicable than just GUIs. Beans are nicely packaged units of business logic and data.

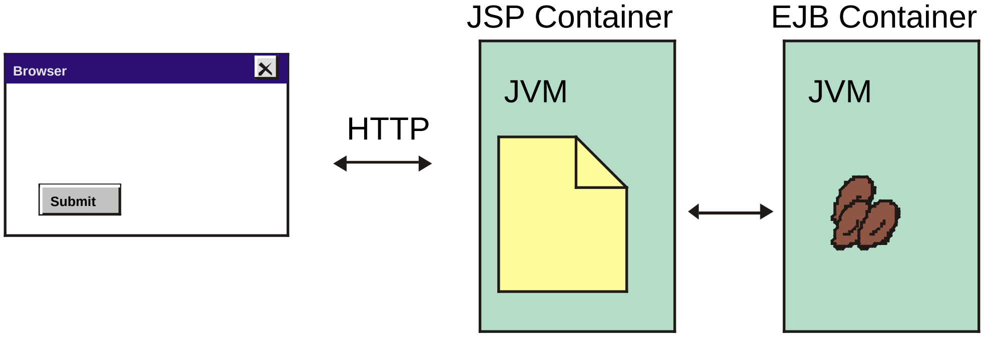
Using JavaBeans with JSPs

- JavaBeans run within the same Java Virtual Machine (JVM) as the JSP's generated servlet
- The JSP specification provides special syntax to make it easier for non-programmers to create and access JavaBeans



JavaBeans vs Enterprise JavaBeans

- Enterprise JavaBeans implement specific interfaces so that they can be installed into a container running on a server computer
- EJBs are designed to be accessed remotely
- EJB containers provide advanced services such as transactions, security and persistence

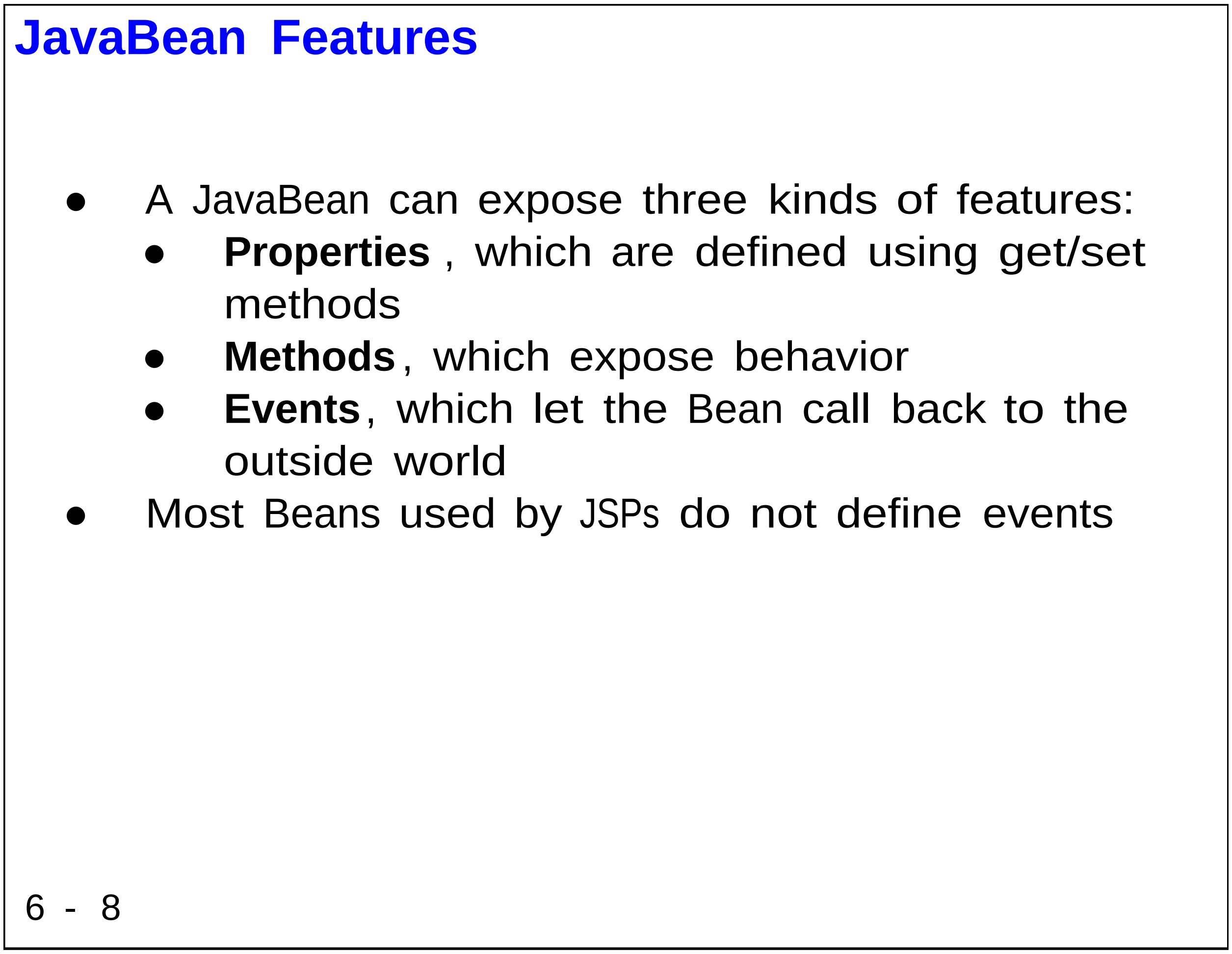


The EJB specification is an integral part of J2EE and allows for components that execute remotely. We will not cover EJBs in this class, but you should be aware that it's very possible to design web applications that include JSPs, servlets and EJBs. Such designs are complex but scalable, since they let you separate the business logic onto remote servers. In addition, since the EJB container provides powerful services, applications that use EJBs can have truly enterprise-class robustness.

JavaBean Requirements

- A JavaBean is a Java class that follows a few simple conventions
- To be a proper JavaBean, a Java class should:
  - Implement the Serializable interface
  - Be in a named package
  - Expose properties, methods and events as necessary
  - Define a no-argument constructor
  - Optionally packaged into a JAR file

This page lists some of the conventions you should follow so your JavaBeans are accessible from JSPs. We will see a complete example in a few pages.



JavaBean Features

- A JavaBean can expose three kinds of features:
  - **Properties** , which are defined using get/set methods
  - **Methods**, which expose behavior
  - **Events**, which let the Bean call back to the outside world
- Most Beans used by JSPs do not define events

The JavaBean specification provides precise definitions of how to code a JavaBean that provides these three kinds of features. For JSPs, you really only need to worry about properties and methods, especially properties.

What are Reflection and Introspection?

- The Java API includes a technique known as **reflection** that lets a program enumerate the public features of any Java class
- If the Java class follows some simple naming conventions, the examining program can make assumptions about the class using a process referred to as **introspection**
- For example, if a JavaBean has a two public methods, one named *getAddress* and one named *setAddress*, we can infer that the Bean defines a property named address

The JSP container uses reflection and introspection to determine a Bean's features and thus check that the JSP author accesses the features correctly.

What is a Property?

- A **property** is defined by the existence of "getter" and "setter" methods
- Note that the property name is case sensitive!
- A **readonly property** defines only a "getter" method

```
public String getAddress() {...}
public void setAddress ( String s ) {...}
public int getSerialNumber () {...}
```

Property Name	Property Type
address	String
serialNumber	int

This code snippet shows a fragment of a Bean class that defines three methods. The combination of the "getAddress" and "setAddress" methods forms a property named "address", while the existence of the "getSerialNumber" method implies a readonly property named "serialNumber".

Note that property names always begin with a lower-case character.

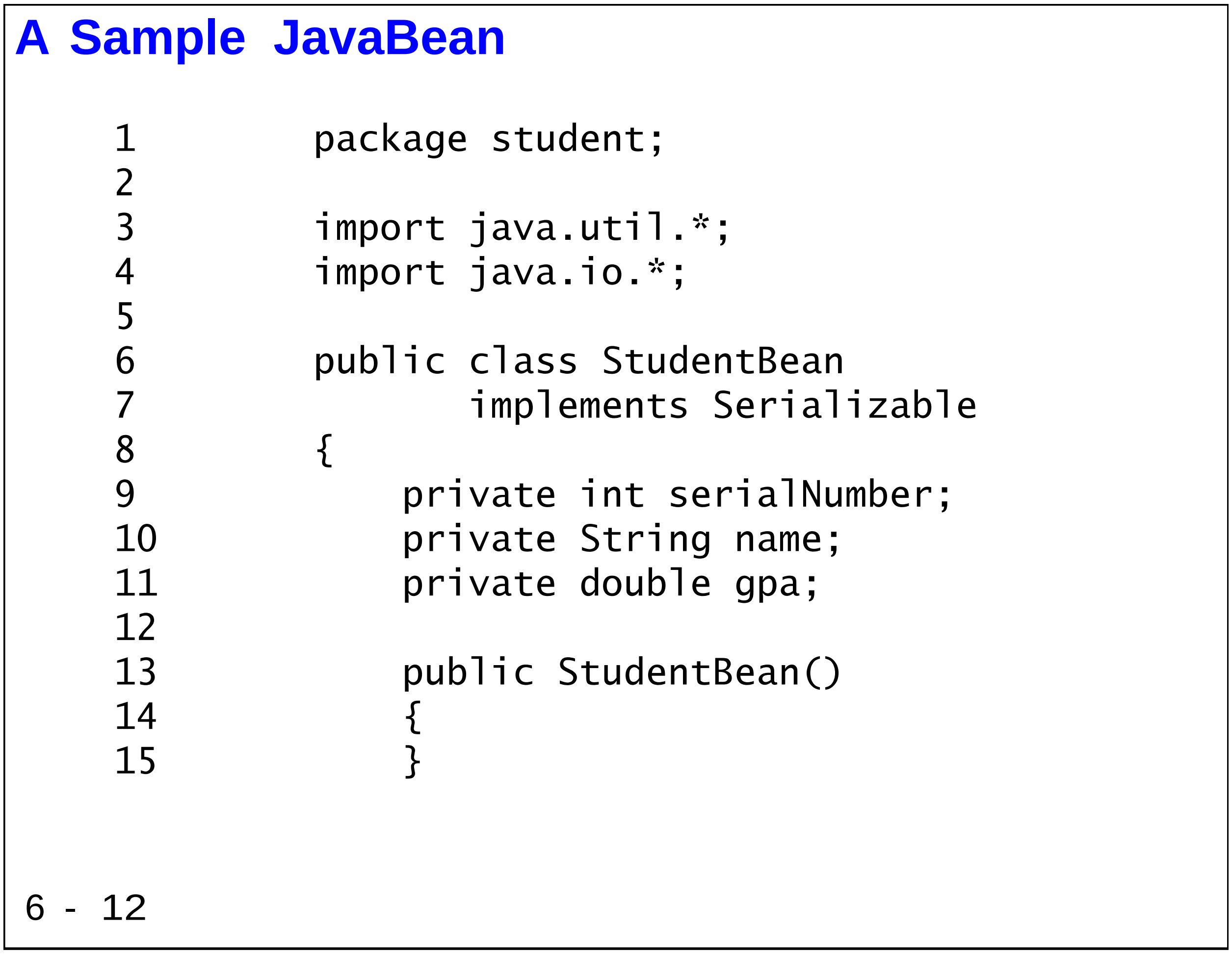
What is a JavaBean Method?

- A JavaBean method is simply a public method in the class that comprises the JavaBean

```
public boolean isHonorRoll()
{
    if ( getGpa() > 3.5 )
        return true;
    else
        return false;
}
```

Here we show a method (behavior) from a Student JavaBean that implements a bit of business logic (a student must have a GPA above 3.5 to be on the honor roll). Note that the method must be public to be accessible from the JSP.

Later in the course, we will cover custom actions, which can be alternative to Beans that have a lot of business logic. Custom actions should be easier for non-programmers to work with.

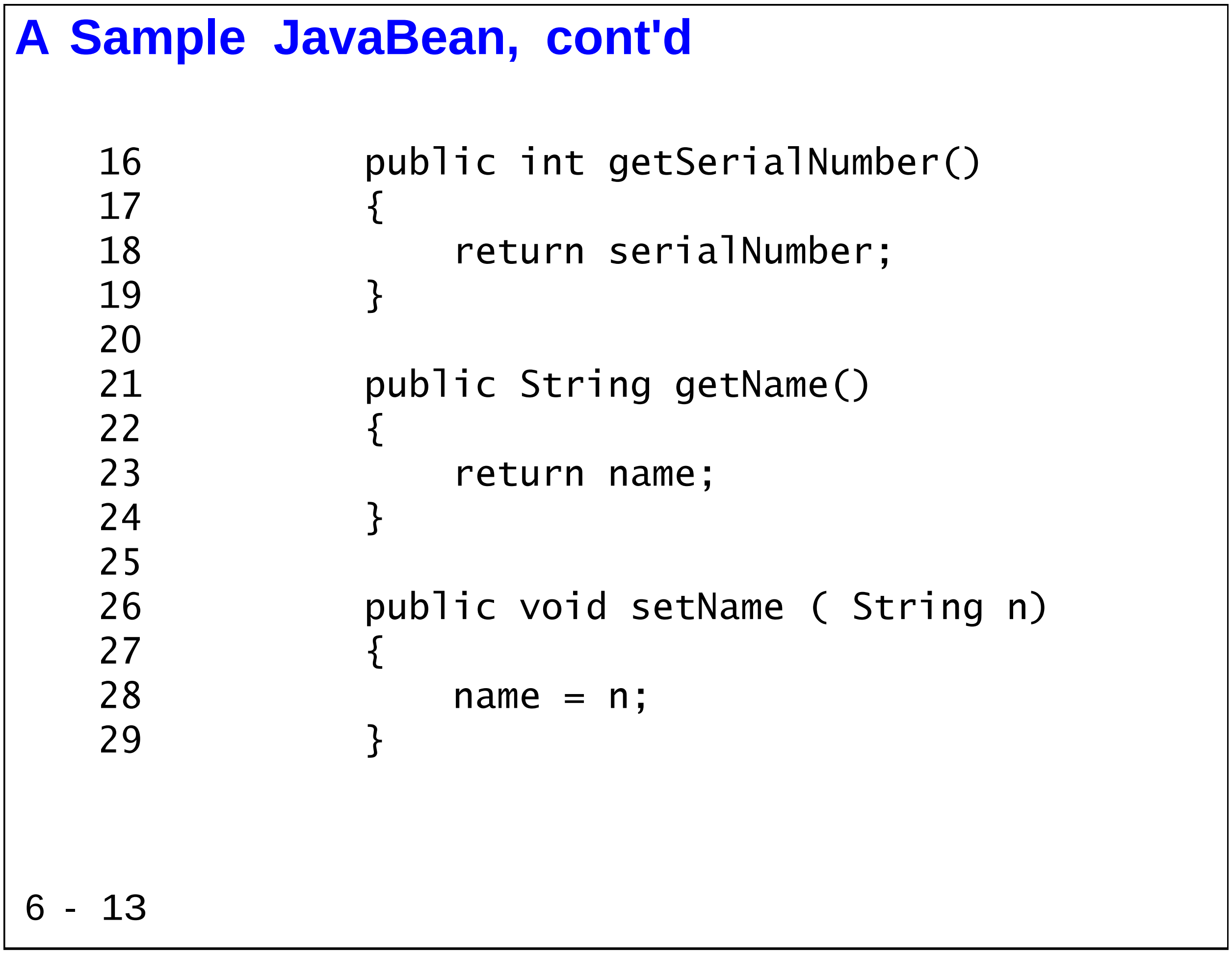


A Sample JavaBean

```
1        package student;
2
3        import java.util.*;
4        import java.io.*;
5
6        public class StudentBean
7               implements Serializable
8        {
9            private int serialNumber;
10           private String name;
11           private double gpa;
12
13           public StudentBean()
14           {
15           }
```

Here we show part of a simple JavaBean that represents a student. Line 1 defines the named package in which the class resides. In line 7, the class implements the Serializable interface, which has no methods but indicates that the object's data could be converted to a stream and saved in a file or transmitted to another JVM.

Lines 9 to 11 define variables to hold the Bean's properties. It's important to note that these are not the properties -- recall that properties are implemented by get/set methods. We will see those on the next page.



A Sample JavaBean, cont'd

```
public int getSerialNumber()
{
    return serialNumber;
}

public String getName()
{
    return name;
}

public void setName ( String n)
{
    name = n;
}
```

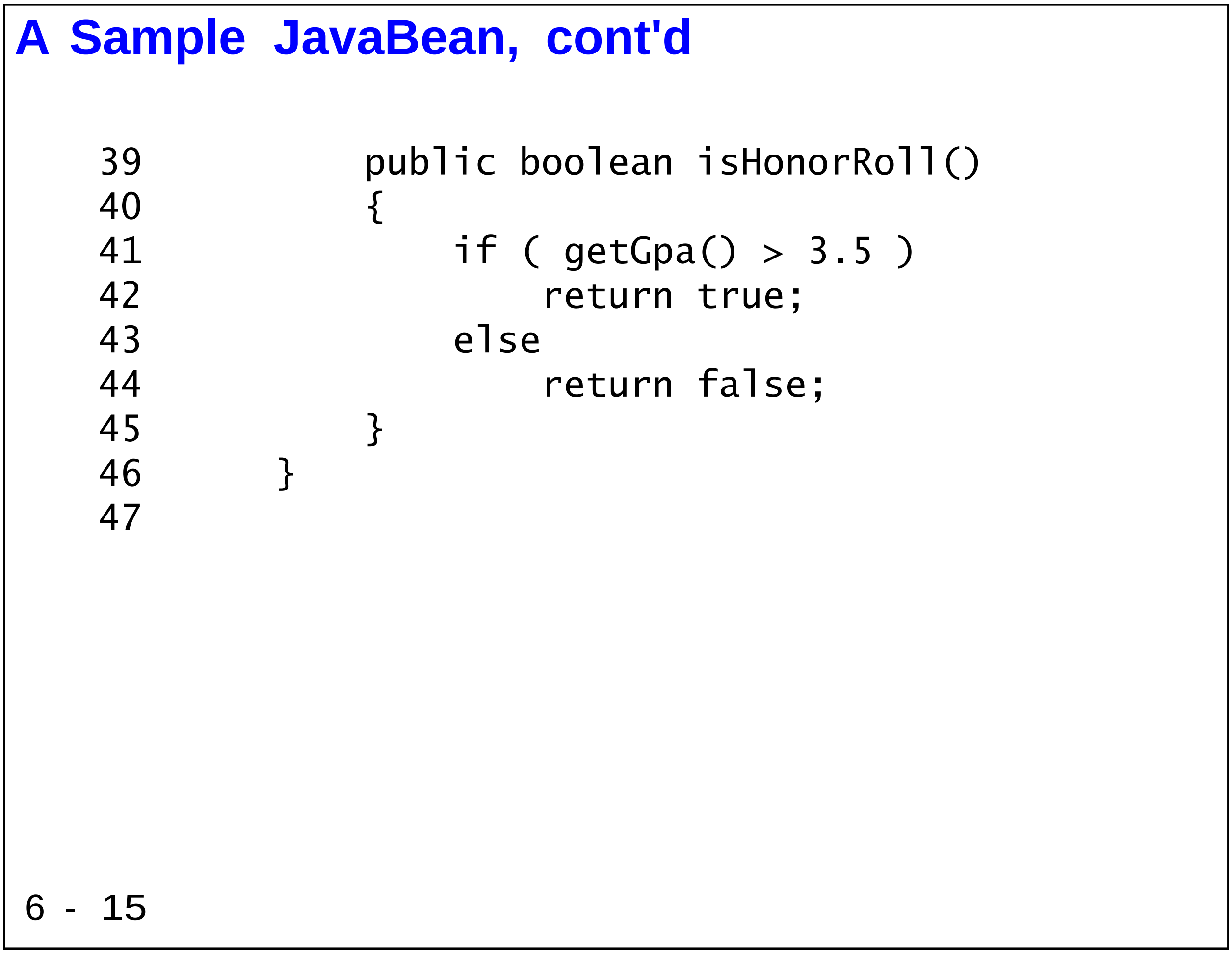
Continuing the JavaBean listing, here we show the read-only "serialNumber" property and the "name" property. Note how these get/set methods are trivial -- they simply read or write the instance variables shown on the last page.

A Sample JavaBean, cont'd

```
        public double getGpa()
        {
            return gpa;
        }

        public void setGpa ( double g )
        {
            gpa = g;
        }
```

Next we show this Bean's "gpa" property.



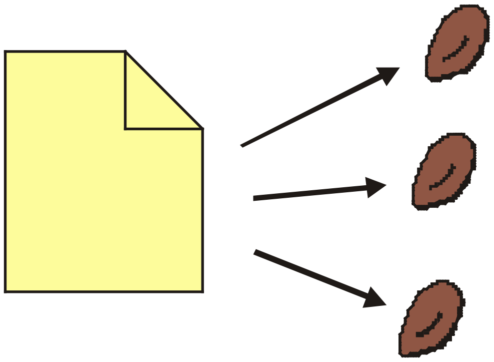
A Sample JavaBean, cont'd

```
  public boolean isHonorRoll()
  {
    if ( getGpa() > 3.5 )
      return true;
    else
      return false;
  }
}
```

Finally, this simple Bean defines a single behavior -- the "isHonorRoll" method that returns true if the student represented by the Bean has a high enough grade-point average.

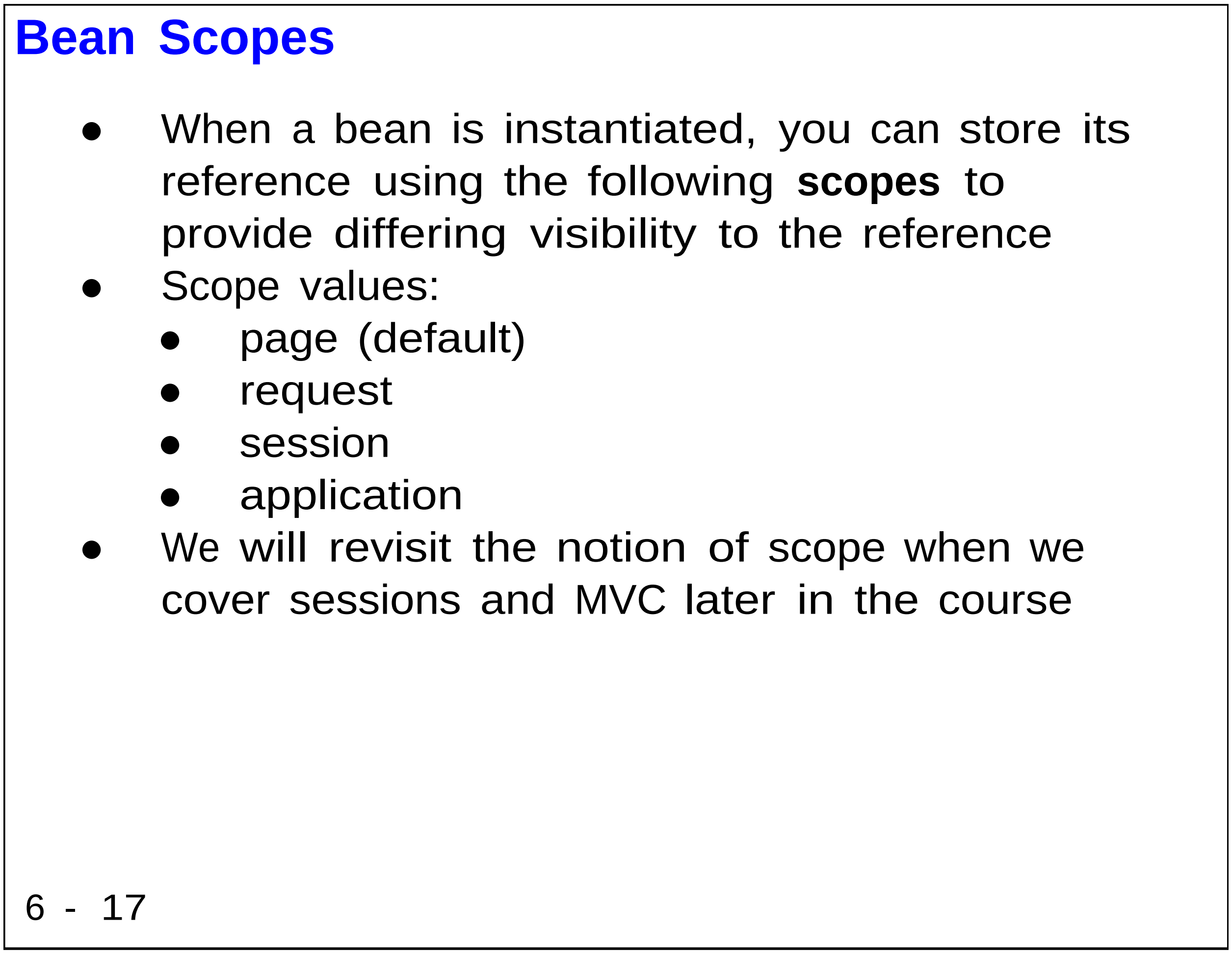
Using a JavaBean from a JSP

- To use a JavaBean from a JSP, you must obtain a reference to the Bean and then typically access its properties and call its methods
- The JSP specification provides special syntax for these operations



Now that you know how to write a Bean, let's turn our attention on how to write a JSP that uses the Bean.

Those of you taking the class that are Java programmers know that to access any Java object, you must first call the "new" operator to instantiate the object and then call the object's methods. To understand that process, you first need to learn quite a few fundamental Java concepts such as references, garbage collection and the like. Fortunately, non-programmers writing JSPs can avoid that learning curve by using the special syntax that we will cover next.



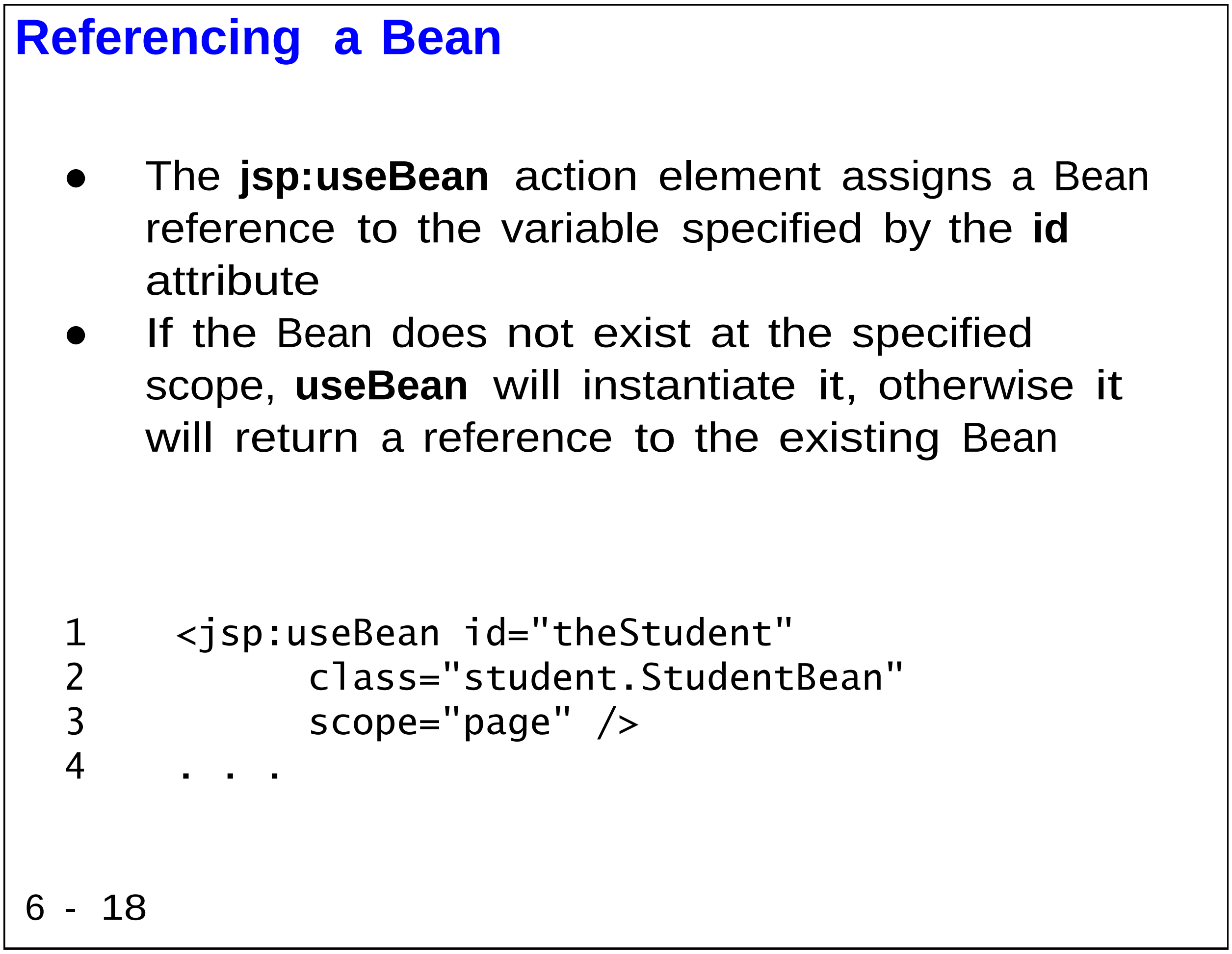
Bean Scopes

- When a bean is instantiated, you can store its reference using the following **scopes** to provide differing visibility to the reference
- Scope values:
  - page (default)
  - request
  - session
  - application
- We will revisit the notion of scope when we cover sessions and MVC later in the course

Instantiation is the process of creating an object and retrieving a reference to the new object. In a web application, you can store the reference in some of the predefined implicit objects covered earlier in the course. By choosing the scope (i.e. which implicit object), you can control how widely the reference is available.

For example, if you store the reference at "page" scope, only the JSP that created the Bean can access it, but if you store the reference at "application" scope, any JSP or servlet that's in the same web application could retrieve the reference and access the Bean.

In a model-view-controller (MVC) web application, the controller often stores Bean references at request or session scope so that the JSPs that comprise the view can display them.



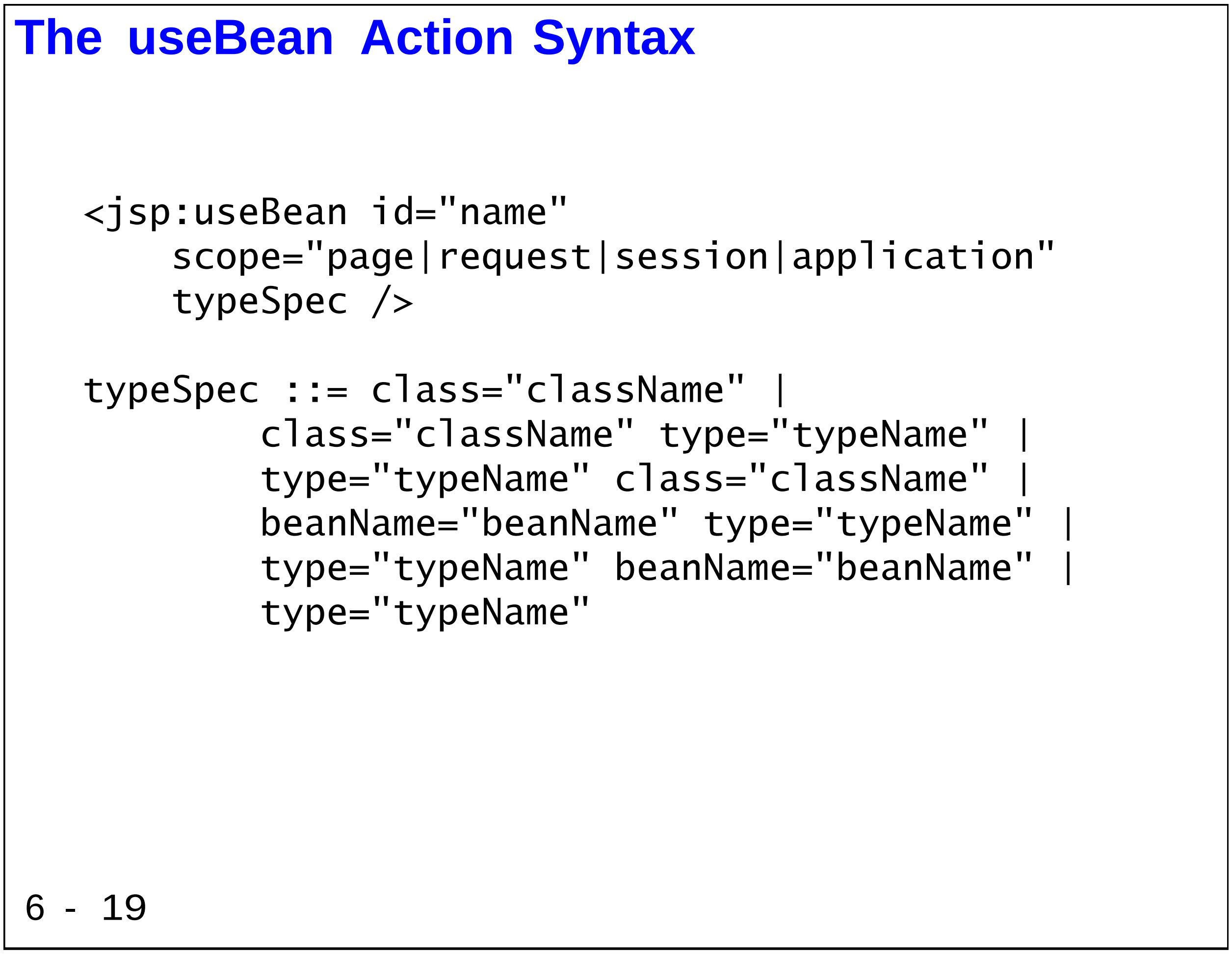
Referencing a Bean

- The **jsp:useBean** action element assigns a Bean reference to the variable specified by the **id** attribute
- If the Bean does not exist at the specified scope, **useBean** will instantiate it, otherwise **it** will return a reference to the existing Bean

```
1    <jsp:useBean id="theStudent"
2          class="student.StudentBean"
3          scope="page" />
4    . . .
```

Here we show a fragment of a JSP that accesses a Bean of the class we saw earlier. In line 1, the JSP invokes the jsp:useBean action, specifying an "id" and a "class" (here we are using the default "page" scope). The JSP could then access the Bean's properties and methods.

Here's a place where it's illuminating to examine the generated servlet -- to implement useBean, the translator creates a Java variable using the "id" as the variable's name, and initializes the variable either by calling the "new" operator or by retrieving a reference from an implicit object (e.g. the session).



The useBean Action Syntax

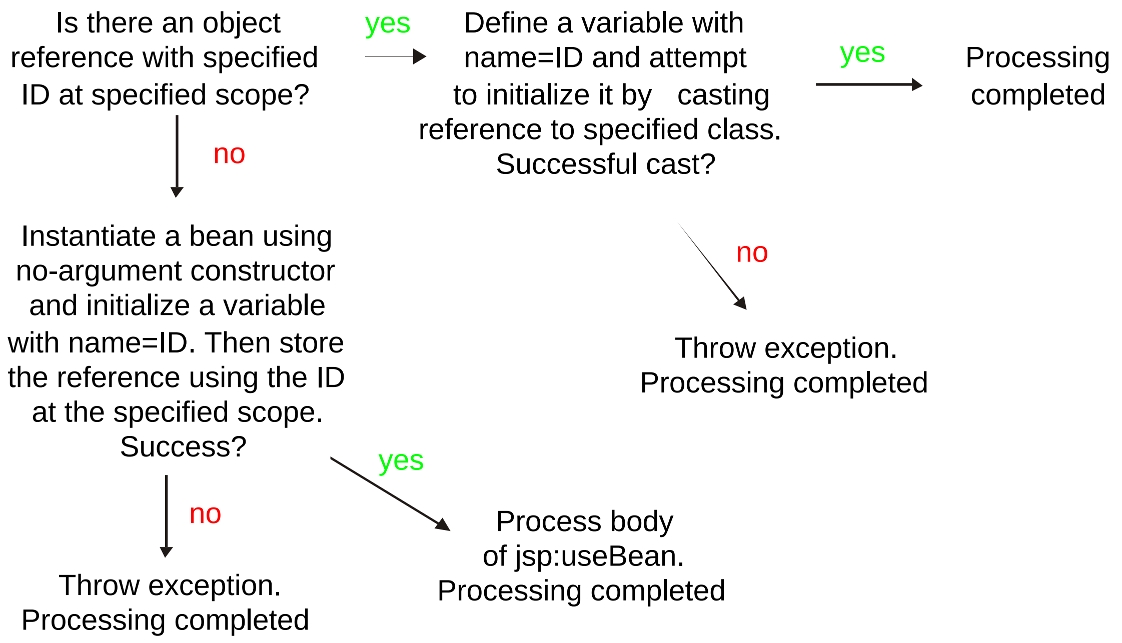
```
<jsp:useBean id="name"
    scope="page|request|session|application"
    typeSpec />

typeSpec ::= class="className" |
        class="className" type="typeName" |
        type="typeName" class="className" |
        beanName="beanName" type="typeName" |
        type="typeName" beanName="beanName" |
        type="typeName"
```

This syntax diagram from the JSP specification shows the complete syntax for the useBean action element. Note that the 'id" attribute is required, but there are various allowed combinations of "class", "type" and "beanName" attributes. If you use "class", you specify a Java class name. If you use "type", you can specify either a Java class or interface. If you use "beanName", then the container will attempt to reconstitute a JavaBean from a serialized file (see the "java.beans.Beans.instantiate method for details).

In this course, we will concentrate on the syntax that uses the "class" attribute, since it's the normal way to create a new Bean.

The useBean Action Flowchart



Is there an object reference with specified ID at specified scope?

yes

Define a variable with name=ID and attempt to initialize it by casting reference to specified class. Successful cast?

yes

Processing completed

no

Throw exception. Processing completed

no

Instantiate a bean using no-argument constructor and initialize a variable with name=ID. Then store the reference using the ID at the specified scope. Success?

yes

Process body of jsp:useBean. Processing completed

no

Throw exception. Processing completed

This chart shows a simplified flowchart for how the container processes useBean (for brevity, the chart omits processing for "type" and "beanName" attributes). The key point here is that the useBean action first searches for an existing Bean and only instantiates a new one if the container can't find an existing Bean at the specified scope.

Note that exceptions can occur -- for example, if the Bean doesn't have a no-argument constructor, the "create object" step will throw an InstantiationException. Another exception could occur during casting if the found object's actual type is incompatible with the specified "class" attribute.

Finally, make special note that if the container finds an existing Bean, it does not process any body of the useBean action element. More on this in a moment.

Setting Properties

- The **jsp:setProperty** action element assigns the specified value to the the specified property (calls the "setter" method)
- The container converts to the property's type from String as described later

```
1    <jsp:useBean id="theStudent"
2                 class="student.StudentBean" />
3
4    <jsp:setProperty name="theStudent"
5                     property="gpa"
6                     value="3.54"/>
```

Remember that Beans can define properties, which are implemented as get/set methods. Instead of writing a scriptlet and calling the "set" method, a JSP author can instead use the setProperty action element. This helps reduce the number of lines of Java code in the JSP, which is a good thing. It also obviates the need for the JSP author to understand the notion of references and Java types.

Here we show getting a reference to a student Bean in lines 1 and 2, then setting the Beans "gpa" property in lines 4 to 6. Note that property name is lower case! Also note that even though the Bean defined the "gpa" property as a "double", the JSP author simply provides a value as a quoted attribute. We will see the rules for conversion later, but the idea is that the JSP author just uses intuitive values for the properties.

In MVC-architected applications, JSPs don't often set properties, since that task is handled by servlets.

Setting Properties from the Request

- If the JSP is invoked from an HTTP request with parameters (e.g. an HTML form), you can use a shorthand notation in the JSP
- This only works if the parameter names are exactly the same as the Bean's property names

```
1    <form action="http://localhost/SetStudent.jsp">
2      Student name:
3         <input type="text" name="name"><p>
4      Serial number:
5         <input type="text" name="serialNumber"><p>
6      GPA:
7        <input type="text" name="gpa"><p>
8      <input type="submit" value="Submit">
9    </form>
```

This page leads into the next page, which shows a special syntax for the setProperty action. Here we show a normal HTML form that references a JSP. In lines 3, 5 and 7, we define input widgets -- note that the widget names exactly match Bean property names.

Setting Properties from the Request, cont'd

- When you use the "*" syntax, the container iterates throughout the parameter names (in the **request** implicit variable) and matches them to the Bean's property names

```
<jsp:useBean id="theStudent"
             class="student.StudentBean"/>

<jsp:setProperty name="theStudent"
             property="*" />
```

This trick can reduce the number of lines in the JSP, but it requires communication between the JSP author and the HTML author (if they are not the same person). Note that parameter names are case-sensitive!

If there is no parameter name that matches a property name, the container does not change the property.

Retrieving Properties

- The **jsp:getProperty** action element retrieves the specified property (calls the "getter" method) and converts it to a String

```
1    <jsp:useBean id="theStudent"
2                 class="student.StudentBean" />
3
4    The student's gpa:
5    <jsp:getProperty name="theStudent"
6                     property="gpa" />
```

Lines 4 to 6 show retrieving a student Bean's "gpa" property and sending it to the output HTML along with a hard-code label.

In an MVC-architected web application, retrieving properties for display is one of the main jobs of JSPs, so JSP authors should become quite familiar with the getProperty syntax.

Property Type Conversion

- In a JSP, all attributes are quoted strings, but in the Bean, properties can be any Java type
- The container converts to/from String when the JSP accesses the property

Property Type	Conversion API (from String)
boolean	Boolean.valueOf (String)
byte	Byte.valueOf (String)
char	Character.valueOf (String)
double	Double.valueOf (String)
int	Integer.valueOf (String)
float	Float.valueOf (String)
long	Long.valueOf (String)

This table shows the standard Java methods that the container calls to convert the value specified in a setProperty action to a Bean's property type. Please refer to the API documentation for these calls for details on how the conversion take place (for example, the allowable formats for floating-point values). All of these classes are in the java.lang package.

Note that the opposite conversion takes place when the container fetches a property from a Bean -- the container must convert the property to a String. See the "toString" method in each of the classes (e.g. Double) for more information.

Calling Methods

- To call a method in a JavaBean, simply issue a standard Java method call from within a scriptlet
- Note that you could also use this technique to access properties, since properties are implemented as simple methods (not necessarily a good idea)

```
1    <jsp:useBean id="theStudent"
2                    class="student.StudentBean" />
3    <%
4        boolean b = theStudent.isHonorRoll();
5    %>
6
7    Honor roll status: <%= b %>
```

Here we show retrieving a bean reference in lines 1 and 2 and then writing a scriplet that calls a Bean method. Note that scriptlets use the variable name as defined by the "id" attribute in the useBean element.

In a well designed web application, you should avoid calling methods too much from JSPs, since it requires the JSP author to know a bit about Java syntax, types and so forth. Perhaps a better way to execute behaviors is to use custom action elements, which we will cover later in the course.

Chapter Summary

In this chapter, you learned:

- The basics of JavaBeans
- How to retrieve a Bean reference
- How to access a Bean's properties and methods