UNIT-V
JAVA SERVER PAGES AND BEANS

The Problem with the Servlets:

In one and only one class, the servlet alone has to do various tasks such as,

- Acceptance of request
- Processing of request
- Handling of business logic
- Generation of response.

Hence there are some problems that are associated with the servlets-

1. For developing a servlet based application, knowledge of Java as well as HTML code is necessary.
2. While developing any web based application, if a look and feel of the web based application needs to be changed then the entire source code to be changed and recompiled.
3. There are some **web page development tools** available using which the developer can develop the web based applications. But the servlets do not support such tools. Even if such tools are used for a servlet, we need to change the embedded HTML code manually, which is a time consuming, error prone and complicated process.

What is Java Server Pages?

The problems that are associated with servlets are due to one and only one reason and that is servlet has to handle all the tasks of request processing. Java Server Pages (JSP) is one technology that came up to overcome these problems. JSP is a technology in which request processing, business logic and presentations are separated out. JSP is a technology for developing web pages that support dynamic content which helps developers insert java code in HTML pages by making use of special JSP tags, most of which start with <% and end with %>. A Java Server Pages component is a type of Java servlet that is designed to fulfill the role of a user interface for a Java web application. Web developers write JSPs as text files that combine HTML or XHTML code, XML elements, and embedded JSP actions and commands. Using JSP, you can collect input from users through web page forms, present records from a database or another source, and create web pages dynamically. JSP tags can be used for a variety of purposes, such as retrieving information from a database or registering user preferences, accessing JavaBeans components, passing control between pages and sharing information between requests, pages etc.

Advantages of JSP:

Following is the list of other advantages of using JSP over other technologies:

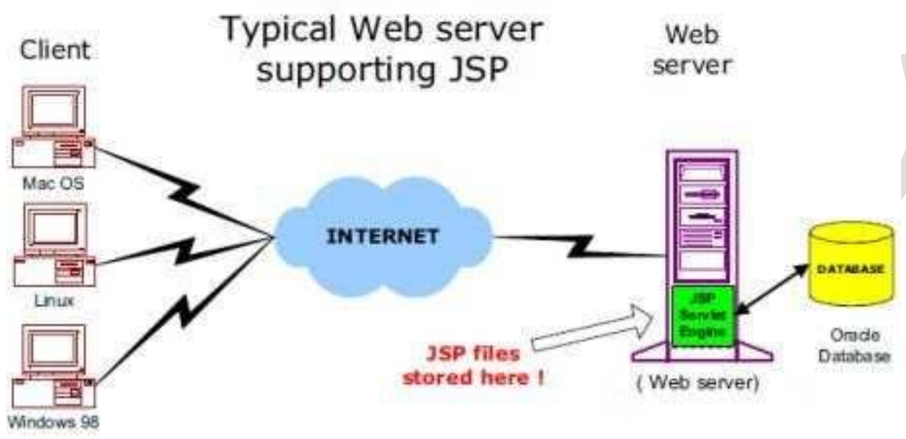
1. **Active Server Pages (ASP):** The advantages of JSP are twofold. First, the dynamic part is written in Java, not Visual Basic or other MS specific language, so it is more powerful and easier to use. Second, it is portable to other operating systems and non-Microsoft Web servers.
2. **Pure Servlets:** It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.
3. **Server-Side Includes (SSI):** SSI is really only intended for simple inclusions, not for

"real" programs that use form data, make database connections, and the like.

4. **JavaScript:** JavaScript can generate HTML dynamically on the client but can hardly interact with the web server to perform complex tasks like database access and image processing etc.
5. **Static HTML:** Regular HTML, of course, cannot contain dynamic information.

JSP Architecture:

The web server needs a JSP engine i.e., container to process JSP pages. The JSP container is responsible for intercepting requests for JSP pages. A JSP container works with the Web server to provide the runtime environment and other services a JSP needs. It knows how to understand the special elements that are part of JSPs. Following diagram shows the position of JSP container and JSP files in a Web Application.



The Anatomy of a JSP Page:

The JSP page is a simple web page which contains the **JSP elements** and **template text**. The template text can be any scripting code such as HTML, WML, XML, or a simple plain text. Various JSP elements can be action tags, custom tags, JSTL, and library elements. These JSP elements are responsible for generating dynamic contents.

JSP Code:

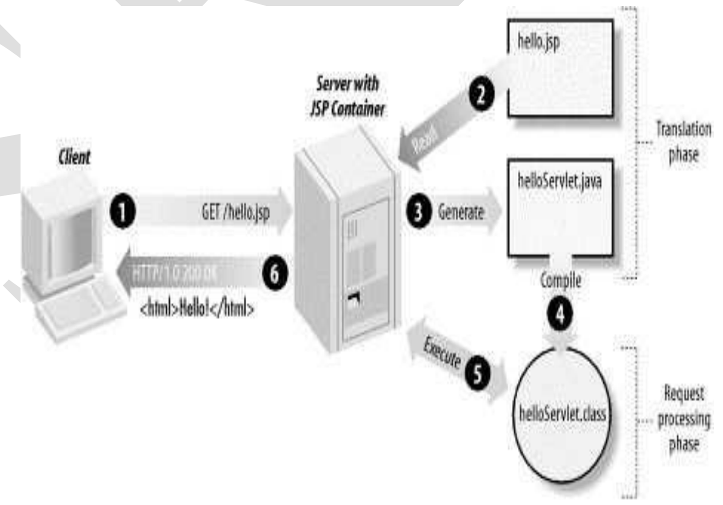
```
<% @page language = “java” contentType= “text/html”%>      ←—— JSP element
<html>
    <head>                                                  }
       <title> Simple JSP Program</title>                   }
     </head>                                                } ←—— Template Text
     <body bgcolor = “red”>                                 }
       <h1> Welcome to the JSP World </h1>                  }
       <p>From III Year CSE Students<br />
             <% out.println(“JSP is equal to HTML and JAVA”); %>   ←—— JSP Element
       </p>
       <h2>Today’s Date is: <% =new Date().toString() %></h2>
    </body>
</html>
```

JSP Processing:

The following steps explain how the web server creates the web page using JSP:

1. As with a normal page, your browser sends an HTTP request to the web server. The web server recognizes that the HTTP request is for a JSP page and forwards it to a JSP engine. This is done by using the URL or JSP page which ends with **.jsp** instead of .html.
2. The JSP engine loads the JSP page from disk and converts it into servlet content. This conversion is very simple in which all template text is converted to println( ) statements and all JSP elements are converted to Java code that implements the corresponding dynamic behavior of the page.
3. The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
4. A part of the web server called the servlet engine loads the Servlet class and executes it. During execution, the servlet produces an output in HTML format, which the servlet engine passes to the web server inside an HTTP response.
5. The web server forwards the HTTP response to your browser in terms of static HTML content.
6. Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

Typically, the JSP engine checks whether a servlet for a JSP file already exists and whether the modification date on the JSP is older than the servlet. If the JSP is older than its generated servlet, the JSP container assumes that the JSP hasn't changed and that the generated servlet still matches the JSP's contents. This makes the process more efficient than with other scripting languages (such as PHP) and therefore faster. So in a way, a JSP page is really just another way to write a servlet without having to be a Java programming wiz. Except for the translation phase, a JSP page is handled exactly like a regular servlet. All the above mentioned steps can be shown below in the following diagram:



JSP Life Cycle:

The key to understanding the low-level functionality of JSP is to understand the simple life cycle they follow. A JSP life cycle can be defined as the entire process from its creation till the destruction which is similar to a servlet life cycle with an additional step which is required to compile a JSP into servlet.

The following are the paths followed by a JSP

1. Compilation
2. Initialization
3. Execution
4. Destroying

The three major phases of JSP life cycle are very similar to Servlet Life Cycle and they are as follows:

1. JSP Compilation:

When a browser asks for a JSP, the JSP engine first checks to see whether it needs to compile the page. If the page has never been compiled, or if the JSP has been modified since it was last compiled, the JSP engine compiles the page. The compilation process involves three steps:

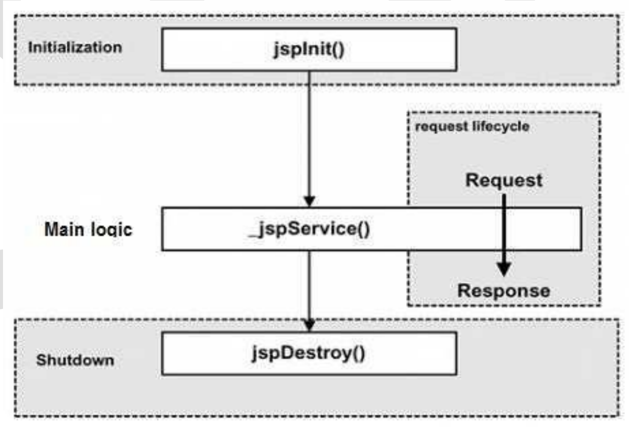
1. Parsing the JSP.
2. Turning the JSP into a servlet.
3. Compiling the servlet.

2. JSP Initialization:

When a container loads a JSP it invokes the jspInit() method before servicing any requests. If you need to perform JSP-specific initialization, override the jspInit() method:

```
public void jspInit() {
// Initialization code...
}
```

Typically initialization is performed only once and as with the servlet init method, you generally initialize database connections, open files, and create lookup tables in the jspInit method.



3. JSP Execution:

This phase of the JSP life cycle represents all interactions with requests until the JSP is destroyed. Whenever a browser requests a JSP and the page has been loaded and initialized, the JSP engine invokes the **_jspService()** method in the JSP. The _jspService() method takes an **HttpServletRequest** and an **HttpServletResponse** as its parameters as follows:

```
void _jspService(HttpServletRequest request, HttpServletResponse response)
{
// Service handling code...
}
```

The _jspService() method of a JSP is invoked once per a request and is responsible for generating the response for that request and this method is also responsible for generating responses to all seven of the HTTP methods ie. GET, POST, DELETE etc.

4. JSP Cleanup:
The destruction phase of the JSP life cycle represents when a JSP is being removed from use by a container. The **jspDestroy()** method is the JSP equivalent of the destroy method for servlets. Override jspDestroy when you need to perform any cleanup, such as releasing database connections or closing open files. The jspDestroy() method has the following form:

```
public void jspDestroy()
{
// Your cleanup code goes here.
}
```

JSP Application Design with MVC:

The design model of JSP application is called MVC model. The **MVC** stands for **M**odel-**V**iew-**C**ontroller. The basic idea in MVC design model is to separate out design logic into three parts – modeling, viewing, and controlling. Any server application is classified into three parts such as business logic, presentation, and request processing.

The **business logic** means the coding logic applied for manipulation of application data. The **presentation** refers to the code written for look and feel of the web page. For example: background color, font style, font size, placing of controls such as text boxes, command buttons and so on. The request processing is nothing but a combination of business logic and presentation. The request processing is always done in order to generate the response. According to the MVC design model, the Model corresponds to business logic, View corresponds to presentation and Controller corresponds to request processing.

Advantages of using MVC design Model:

The use of MVC architecture allows the developer to keep the separation between business logic, presentation and request processing. Due to this separation it becomes easy to make **changes** in presentation without disturbing the business logic. The changes in presentation are often required for accommodating the new presentation interfaces.

JSP Elements:

Java Server Page is facilitated with three types of elements. They are as follows:

i) JSP Directives
ii) JSP Actions
iii) JSP Scripting Elements

1) JSP Directives:

JSP directives provide directions and instructions to the container, telling it how to handle certain aspects of JSP processing. It usually has the following form:

<%@ directive attribute="value" %>

There are three types of directive tag are available in JSP:

Directive	Description
<%@ page ... %>	Defines page-dependent attributes, such as scripting language, error page, and buffering requirements.
<%@ include ... %>	Includes a file during the translation phase.
<%@ taglib ... %>	Declares a tag library, containing custom actions, used in the page.

i) The page Directive:

The **page** directive is used to provide instructions to the container that pertain to the current JSP page. You may code page directives anywhere in your JSP page. By convention, page directives are coded at the top of the JSP page.

Following is the basic syntax of page directive:

```
<%@ page attribute="value" %>
```

You can write XML equivalent of the above syntax as follows:

```
<jsp:page attribute="value" />
```

Attributes:
Following is the list of attributes associated with page directive:

Attribute	Purpose
buffer	Specifies a buffering model for the output stream.
autoFlush	Controls the behavior of the servlet output buffer.
contentType	Defines the character encoding scheme.
errorPage	Defines the URL of another JSP that reports on Java unchecked runtime exceptions.
isErrorPage	Indicates if this JSP page is a URL specified by another JSP page's errorPage attribute.
extends	Specifies a superclass that the generated servlet must extend
import	Specifies a list of packages or classes for use in the JSP as the Java import statement does for Java classes.
info	Defines a string that can be accessed with the servlet's getServletInfo() method.
language	Defines the programming language used in the JSP page.
session	Specifies whether or not the JSP page participates in HTTP sessions
isScriptingEnabled	Determines if scripting elements are allowed for use.

Example:

```
<%@ page import="java.io.*"   %>
```

ii) The include Directive:

The **include** directive is used to includes a file during the translation phase. This directive tells the container to merge the content of other external files with the current JSP during the translation phase. You may code *include* directives anywhere in your JSP page. The general usage form of this directive is as follows:

<%@ include file="relative-url" >

The filename in the include directive is actually a relative URL. If you just specify a filename with no associated path, the JSP compiler assumes that the file is in the same directory as your JSP. You can write XML equivalent of the above syntax as follows:

<jsp:include file="relative-url" />

iii) The taglib Directive:

The Java Server Pages API allows you to define custom JSP tags that look like HTML or XML tags and a tag library is a set of user-defined tags that implement custom behavior. The **taglib** directive declares that your JSP page uses a set of custom tags, identifies the location of the library, and provides a means for identifying the custom tags in your JSP page. The taglib directive follows the following syntax:

<%@taglib uri="uri" prefix="prefixOfTag">

2) JSP Actions:

JSP actions use constructs in XML syntax to control the behavior of the servlet engine. You can dynamically insert a file, reuse JavaBeans components, forward the user to another page, or generate HTML for the Java plugin. There is only one syntax for the Action element, as it conforms to the XML standard:

<jsp:action_name attribute="value" />

Action elements are basically predefined functions and there are following JSP actions available:

Syntax	Purpose
jsp:include	Includes a file at the time the page is requested
jsp:useBean	Finds or instantiates a JavaBean
jsp:setProperty	Sets the property of a JavaBean
jsp:getProperty	Inserts the property of a JavaBean into the output
jsp:forward	Forwards the requester to a new page
jsp:plugin	Generates browser-specific code that makes an OBJECT or EMBED tag for the Java plugin
jsp:element	Defines XML elements dynamically.
jsp:attribute	Defines dynamically defined XML element's attribute.
jsp:body	Defines dynamically defined XML element's body.
jsp:text	Use to write template text in JSP pages and documents.

Common Attributes:

There are two attributes that are common to all Action elements: the **id** attribute and the **scope** attribute.

- **Id attribute:** The id attribute uniquely identifies the Action element, and allows the action to be referenced inside the JSP page. If the Action creates an instance of an object the id value can be used to reference it through the implicit object PageContext
- **Scope attribute:** This attribute identifies the lifecycle of the Action element. The id attribute and the scope attribute are directly related, as the scope attribute determines the lifespan of the object associated with the id. The scope attribute has four possible values:
    a) page
    b) request
    c) session, and
    d) application

i) The <jsp:include> Action

This action lets you insert files into the page being generated. The syntax looks like this:

```
<jsp:include page="relative URL" flush="true" />
```

Unlike the **include** directive, which inserts the file at the time the JSP page is translated into a servlet, this action inserts the file at the time the page is requested. Following is the list of attributes associated with include action:

Attribute	Description
Page	The relative URL of the page to be included.
Flush	The boolean attribute determines whether the included resource has its buffer flushed before it is included.

Example:

Let us define following two files: date.jsp and main.jsp as follows:

date.jsp:

```
<p>
   Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

main.jsp:

```
<html>
        <head>
                <title>The include Action Example</title>
        </head>
        <body>
                <center>
                        <h2>The include action Example</h2>
                        <jsp:include page="date.jsp" flush="true" />
                </center>
        </body>
</html>
```

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like this:

```
The include action Example
Today's date: 12-Sep-2010 14:54:22
```

ii) The <jsp:useBean>Action:

The **useBean** action is quite versatile. It first searches for an existing object utilizing the id and scope variables. If an object is not found, it then tries to create the specified object.
The simplest way to load a bean is as follows:

```
<jsp:useBean id="name" class="package.class" />
```

Once a bean class is loaded, you can use **jsp:setProperty** and **jsp:getProperty** actions to modify and retrieve bean properties. Following is the list of attributes associated with useBean action:

Attribute	Description
Class	Designates the full package name of the bean.
Type	Specifies the type of the variable that will refer to the object.
beanName	Gives the name of the bean as specified by the instantiate () method of the java.beans.Beans class.

Let us discuss about **jsp:setProperty** and **jsp:getProperty** actions before giving a valid example related to these actions.

iii) The <jsp:setProperty> Action:

The **setProperty** action sets the properties of a Bean. The Bean must have been previously defined before this action. There are two basic ways to use the setProperty action:
You can use jsp:setProperty after, but outside of, a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... />
...
<jsp:setProperty name="myName" property="someProperty" .../>
```

In this case, the jsp:setProperty is executed regardless of whether a new bean was instantiated or an existing bean was found.

A second context in which jsp:setProperty can appear is inside the body of a jsp:useBean element, as below:

```
<jsp:useBean id="myName" ... >
...
<jsp:setProperty name="myName" property="someProperty" .../>
</jsp:useBean>
```

Here, the jsp:setProperty is executed only if a new object was instantiated, not if an existing one was found. Following is the list of attributes associated with setProperty action:
Following is the list of required attributes associated with setProperty action:

Attribute	Description
Name	The name of the Bean that has a property to be retrieved. The Bean must have been previously defined.
property	The property attribute is the name of the Bean property to be retrieved.

iv) The <jsp:getProperty> Action:

The **getProperty** action is used to retrieve the value of a given property and converts it to a string, and finally inserts it into the output. The getProperty action has only two attributes, both of which are required ans simple syntax is as follows:

Attribute	Description
Name	Designates the bean whose property will be set. The Bean must have been previously defined.
Property	Indicates the property you want to set. A value of "*" means that all request parameters whose names match bean property names will be passed to the appropriate setter methods.
Value	The value that is to be assigned to the given property. The the parameter's value is null, or the parameter does not exist, the setProperty action in ignored.
Param	The param attribute is the name of the request parameter whose value the property is to receive. You can’t use both value and param, but it is permissible to use neither.

```
<jsp:useBean id="myName" ... />
...
<jsp:getProperty name="myName" property="someProperty" .../>
```

Example:
Let us define a test bean which we will use in our example:

TestBean.java:
```
package action;
public class TestBean
{
    private String message = "No message specified";
    public String getMessage()
    {
        return(message);
    }
    public void setMessage(String message)
    {
        this.message = message;
    }
}
```

Compile above code to generated TestBean.class file and make sure that you copied TestBean.class in C:\apache-tomcat-7.0.2\webapps\WEB-INF\classes\action folder and CLASSPATH variable should also be set to this folder:

main.jsp:
```
<html>
        <head>
                <title>Using JavaBeans in JSP</title>
        </head>
        <body>
                <center>
                        <h2>Using JavaBeans in JSP</h2>
                        <jsp:useBean id="test" class="action.TestBean" />
                        <jsp:setProperty name="test" property="message" value="Hello JSP..." />
                        <p>Got message....</p>
                        <jsp:getProperty name="test" property="message" />
                </center>
        </body>
</html>
```

Now try to access main.jsp, it would display following result:

Using JavaBeans in JSP
Got message....
Hello JSP...

v) The <jsp:forward> Action:

The **forward** action terminates the action of the current page and forwards the request to another resource such as a static page, another JSP page, or a Java Servlet. The simple syntax of this action is as follows:

```
<jsp:forward page="Relative URL" />
```

Following is the list of required attributes associated with forward action:

Attribute	Description
Page	Should consist of a relative URL of another resource such as a static page, another JSP page, or a Java Servlet.

Example:

Let us reuse following two files (a) date.jps and (b) main.jsp as follows: Following is the content of date.jsp file:

```
<p>
        Today's date: <%= (new java.util.Date()).toLocaleString()%>
</p>
```

Here is the content of main.jsp file:

```
<html>
        <head>
                <title>The include Action Example</title>
        </head>
        <body>
                <center>
                        <h2>The include action Example</h2>
                        <jsp:forward page="date.jsp" />
                </center>
        </body>
</html>
```

Now let us keep all these files in root directory and try to access main.jsp. This would display result something like as below. Here it discarded content from main page and displayed content from forwarded page only.

Today's date: 12-Sep-2010 14:54:22

vi) The <jsp:plugin> Action:

The **plugin** action is used to insert Java components into a JSP page. It determines the type of browser and inserts the <object> or <embed> tags as needed. If the needed plugin is not present, it downloads the plugin and then executes the Java component. The Java component can be either an Applet or a JavaBean. The plugin action has several attributes that correspond to common HTML tags used to format Java components. The <param> element can also be used to send parameters to the Applet or Bean. Following is the typical syntax of using plugin action:

```
<jsp:plugin type="applet" codebase="dirname" code="MyApplet.class" width="60"
height="80">
<jsp:param name="fontcolor" value="red" />
<jsp:param name="background" value="black" />
<jsp:fallback>
        Unable to initialize Java Plugin
</jsp:fallback>
</jsp:plugin>
```

You can try this action using some applet if you are interested. A new element, the <fallback> element, can be used to specify an error string to be sent to the user in case the component fails.

3) JSP Scripting Elements:

JSP scripting elements can be used to embed the Java code into the JSP file, declare variables and also attach the results to the HTML content. There are four types of scripting elements are available. They are:

i) Scriptlets
ii) Declarations
iii) Expressions
iv) Comments

i) The Scriptlet:

A scriptlet can contain any number of JAVA language statements, variable or method declarations, or expressions that are valid in the java server page scripting language. Following is the syntax of Scriptlet:

<% code fragment %>

Any text, HTML tags, or JSP elements you write must be outside the scriptlet. Following is the simple example for JSP:

```
<html>
        <body>
        <%
                out.println("Your IP address is " + request.getRemoteAddr());
        %>
        </body>
</html>
```

ii) Declarations:

A declaration declares one or more variables or methods that you can use in Java code later in the JSP file. You must declare the variable or method before you use it in the JSP file.
Following is the syntax of JSP Declarations:

```
<%! declaration; [ declaration; ] ... %>
```

Following is the simple example for JSP Comments:

```
<%! int i = 0; %>
<%! int a, b, c; %>
<%! Circle a = new Circle(2.0); %>
```

iii) Expressions:

A JSP expression element contains a scripting language expression that is evaluated, converted to a String, and inserted where the expression appears in the JSP file. Because the value of an expression is converted to a String, you can use an expression within a line of text, whether or not it is tagged with HTML, in a JSP file. The expression element can contain any expression that is valid according to the Java Language Specification but you cannot use a semicolon to end an expression. Following is the syntax of JSP Expression:

```
<%= expression %>
```

Following is the simple example for JSP Expression:

```
<html>
	<head>
		<title>JSP Expressions</title>
	</head>
	<body>
		<p>
			Today's date: <%= (new java.util.Date()).toLocaleString()%>
		</p>
	</body>
</html>
```

This would generate following result:

```
Today's date: 10-Jan-2018 21:24:25
```

iv) Comments:

JSP comment marks text or statements that the JSP container should ignore. A JSP comment is useful when you want to hide or "comment out" part of your JSP page.
Following is the syntax of JSP comments:

```
<%--This is JSP comment --%>
```

Following is the simple example for JSP Comments:

```
<html>
	<head>
		<title>A Comment Test</title>
	</head>
	<body>
		<h2>A Test of Comments</h2>
```

```
                    <%--This comment will not be visible in the page source --%>
          </body>
</html>
```

JSP Implicit Objects:

JSP Implicit Objects are the Java objects that the JSP Container makes available to developers in each page and developer can call them directly without being explicitly declared. JSP Implicit Objects are also called predefined variables. JSP supports nine Implicit Objects which are listed below:

Object	Description
request	This is the **HttpServletRequest** object associated with the request.
response	This is the **HttpServletResponse** object associated with the response to the client.
out	This is the **PrintWriter** object used to send output to the client.
session	This is the **HttpSession** object associated with the request.
application	This is the **ServletContext** object associated with application context.
config	This is the **ServletConfig** object associated with the page.
pageContext	This encapsulates use of server-specific features like higher performance **JspWriters**.
page	This is simply a synonym for **this**, and is used to call the methods defined by the translated servlet class.
Exception	The **Exception** object allows the exception data to be accessed by designated JSP.

i) The request Object:

The request object is an instance of a "**javax.servlet.http.HttpServletRequest"** object. Each time a client requests a page the JSP engine creates a new object to represent that request. The request object provides methods to get HTTP header information including form data, cookies, HTTP methods etc.

ii) The response Object:

The response object is an instance of a "**javax.servlet.http.HttpServletResponse"** object. Just as the server creates the request object, it also creates an object to represent the response to the client. The response object also defines the interfaces that deal with creating new HTTP headers. Through this object the JSP programmer can add new cookies or date stamps, HTTP status codes etc.

iii) The out Object:

The out object is an instance of a **"javax.servlet.jsp.JspWriter**" object and is used to send content in a response. The initial JspWriter object is instantiated differently depending on whether the page is buffered or not. Buffering can be easily turned off by using the buffered='false' attribute of the page directive. The JspWriter object contains most of the same methods as the java.io.PrintWriter class. However, JspWriter has some additional methods designed to deal with buffering. Unlike the PrintWriter object, JspWriter throws IOExceptions. Following are the important methods which we would use to write boolean char, int, double, object, String etc.

Method	Description
out.print(dataType dt)	Print a data type value
out.println(dataType dt)	Print a data type value then terminate the line with new line character.
out.flush()	Flush the stream.

iv) The session Object:

The session object is an instance of **"javax.servlet.http.HttpSession**" and behaves exactly the same way that session objects behave under Java Servlets. The session object is used to track client session between client requests.

v) The application Object:

The application object is direct wrapper around the ServletContext object for the generated Servlet and in reality an instance of a **"javax.servlet.ServletContext"** object. This object is a representation of the JSP page through its entire lifecycle. This object is created when the JSP page is initialized and will be removed when the JSP page is removed by the jspDestroy() method. By adding an attribute to application, you can ensure that all JSP files that make up your web application have access to it.

vi) The config Object:

The config object is an instantiation of **"javax.servlet.ServletConfig**" and is a direct wrapper around the ServletConfig object for the generated servlet. This object allows the JSP programmer access to the Servlet or JSP engine initialization parameters such as the paths or file locations etc. The following config method is the only one you might ever use, and its usage is trivial: **config.getServletName();**

This returns the servlet name, which is the string contained in the <servlet-name> element defined in the WEB-INF\web.xml file.

vii) The pageContext Object:

The pageContext object is an instance of a **"javax.servlet.jsp.PageContext**" object. The pageContext object is used to represent the entire JSP page.

```
pageContext.removeAttribute("attrName", PAGE_SCOPE);
```

viii) The page Object:

This object is an actual reference to the instance of the page. It can be thought of as an object that represents the entire JSP page. The page object is really a direct synonym for the "**this"** object.

ix) The exception Object:

The exception object is a wrapper containing the exception thrown from the previous page. It is typically used to generate an appropriate response to the error condition.

Using Beans in JSP:

Java beans are reusable Java components. We can use simple Java bean in the JSP. This can help us in keeping the business logic separate from presentation logic. Beans are used in the JSP pages as the instance of class. We must specify the scope of the bean in the JSP page. Here scope of the bean means the range time span of the bean for its existence in JSP. When bean is present in particular scope its **id** also available in that scope. There are various scopes using which the bean can be used in JSP page.

1. **Page scope:** The bean object gets disappeared as soon as the current page is discarded. The default scope for a bean in JSP page is **page** scope.
2. **Request scope:** The bean object remains in existence as long as the request object is present.
3. **Session scope:** A session can be defined as the specific period of a time the user spends browsing the site.
4. **Application scope:** During application scope the bean will get stored to ServletContext. Hence particular bean is available to all the servlets in the same web application.

Example:
CounterDemo.java:
```
public class CounterDemo
{
        public int cnt;
        public CounterDemo()
        {
                cnt=0;
        }
        public int getCnt( )
        {
                return cnt;
        }
        public void setCnt(int cnt)
        {
                this.cnt=cnt;
        }
}
```
beanJSP.jsp:
```
<html>
        <head>
                <title>Using beans in JSP</title>
        </head>
        <body>
                <jsp:useBean id="bean1" class="CounterDemo" scope="session" />
                <%
                        bean1.setCnt(bean1.getCnt( )+1);
                %>
                Your count is: <%=bean1.getCnt( )%>
        </body>
</html>
```

Using Cookies:

Cookies are text files stored on the client computer and they are kept for various information tracking purposes. JSP transparently supports HTTP cookies using underlying servlet technology.

There are three steps involved in identifying returning users:

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user or may be for some other purpose as well.

Servlet Cookies Methods:

Following is the list of useful methods associated with Cookie object which you can use while manipulating cookies in JSP:

S.N.	Method & Description
1	**public void setDomain(String pattern)** This method sets the domain to which cookie applies, for example tutorialspoint.com.
2	**public String getDomain()** This method gets the domain to which cookie applies, for example tutorialspoint.com.
3	**public void setMaxAge(int expiry)** This method sets how much time (in seconds) should elapse before the cookie expires. If you don't set this, the cookie will last only for the current session.
4	**public int getMaxAge()**This method returns the maximum age of the cookie, specified in seconds, By default, -1 indicating the cookie will persist until browser shutdown.
5	**public String getName()** This method returns the name of the cookie. The name cannot be changed after creation.
6	**public void setValue(String newValue)** This method sets the value associated with the cookie.
7	**public String getValue()** This method gets the value associated with the cookie.
8	**public void setPath(String uri)** This method sets the path to which this cookie applies. If you don't specify a path, the cookie is returned for all URLs in the same directory as the current page as well as all subdirectories.
9	**public String getPath()** This method gets the path to which this cookie applies.
10	**public void setSecure(boolean flag)** This method sets the boolean value indicating whether the cookie should only be sent over encrypted (i.e. SSL) connections.
11	**public void setComment(String purpose)** This method specifies a comment that describes a cookie's purpose. The comment is useful if the browser presents the cookie

	to the user.
12	**public String getComment()**This method returns the comment describing the purpose of this cookie, or null if the cookie has no comment.

Example:
The following example illustrates how to create cookies and how to use them for processing. This example contains three files:
CookieDemo.html – This file allow us to enter name and value for cookie we want to create.
CookieDemo.jsp – This file process the input from the CookieDemo.html file.
CookieDisp.jsp – This file displays the created cookies.

CookieDemo.html:
```
<html>
    <head>
        <title>Cookie Demo</title>
        </head>
        <body><br>
        <center>
            <h1>Cookie Demo</h1>
            <form action="CookieDemo.jsp" method="get">
                <table>
                    <tr>
                        <td>Enter Cookie Name:</td>
                        <td><input type="text" name="cname"></td>
                    </tr>
                    <tr>
                        <td>Enter Cookie Value:</td>
                        <td><input type="text" name="cvalue"></td>
                    </tr>
                    <tr>
                        <td align="center"><input type="submit" value="Submit"
name="submit"></td>

                    </tr>
                </table>
            </form>
        </center>
    </body>
</html>
```

CookieDemo.jsp:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title>Cookie Demo</title>
      </head>
      <body>
            <center>
                  <%
                        String cn=request.getParameter("cname");
                        String cv=request.getParameter("cvalue");
                        Cookie c=new Cookie (cn,cv);
                        response.addCookie(c);
                        c.setMaxAge(50*50);
                  %>
                   <h1>Cookie Added</h1>
                  <form method="get" action="CookieDisp.jsp">
                        <input type="submit" value="List Cookies">
                  </form>
            </center>
      </body>
</html>
```

CookieDisp.html:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
            <title>Cookie Display</title>
      </head>
      <body>
            <center>
                  <h1>Cookies List</h1>
                  <%
                        Cookie[] c=request.getCookies();
                  %>
                  <table border=1>
                        <tr>
```

```
                    <%
                    out.println("<td><b> Cookie Name </td> <td> <b>Cookie value</b>
</td>");
                    for(int i=0;i<c.length;i++)
                    {
                      out.println("<h2> <tr> <td>
"+c[i].getName()+"</td><td>"+c[i].getValue()+"</td></tr>");
                    }
                    %>
                </tr>
            </table>
        </center>
    </body>
</html>
```

Using Sessions:

JSP makes use of servlet provided HttpSession Interface which provides a way to identify a user across more than one page request or visit to a Web site and to store information about that user. By default, JSPs have session tracking enabled and a new HttpSession object is instantiated for each new client automatically. Disabling session tracking requires explicitly turning it off by setting the page directive session attribute to false as follows:

<%@ page session="false" %>

The JSP engine exposes the HttpSession object to the JSP author through the implicit **session** object. Since **session** object is already provided to the JSP programmer, the programmer can immediately begin storing and retrieving data from the object without any initialization or getSession(). Here is a summary of important methods available through session object:

S.N.	Method & Description
1	**public Object getAttribute(String name)** This method returns the object bound with the specified name in this session, or null if no object is bound under the name.
2	**public Enumeration getAttributeNames()** This method returns an Enumeration of String objects containing the names of all the objects bound to this session.
3	**public long getCreationTime()** This method returns the time when this session was created, measured in milliseconds since midnight January 1, 1970 GMT.
4	**public String getId()** This method returns a string containing the unique identifier assigned to this session.

5	**public long getLastAccessedTime()** This method returns the last time the client sent a request associated with this session, as the number of milliseconds since midnight January 1, 1970 GMT.
6	**public int getMaxInactiveInterval()** This method returns the maximum time interval, in seconds, that the servlet container will keep this session open between client accesses.
7	**public void invalidate()** This method invalidates this session and unbinds any objects bound to it.
8	**public boolean isNew(** This method returns true if the client does not yet know about the session or if the client chooses not to join the session.
9	**public void removeAttribute(String name)** This method removes the object bound with the specified name from this session.
10	**public void setAttribute(String name, Object value)** This method binds an object to this session, using the name specified.
11	**public void setMaxInactiveInterval(int interval)** This method specifies the time, in seconds, between client requests before the servlet container will invalidate this session.

Example:

The following example illustrates how to use session state. A new session is created if one does not already exist. The **getAttribute( )** method is called to obtain the object that is bound to the name **"cnt"**.

Sessions.html:

```
<html>
	<head>
		<title>Session Demo</title>
	</head>
	<body>
		<form name="form1" action="SessionDemo.jsp" >
			<b> Session Demo</b><br>
			<input type="submit" value="Go to My Session Demo">
		</form>
	</body>
</html>
```

SessionDemo.jsp:

```
<%@ page import = "java.util.*" %>
<%
	String heading;
	Integer cnt=(Integer)session.getAttribute("cnt");
	if(cnt==null)
	{
```

```
                cnt=new Integer(0);
                heading="Welcome for the first time";
        }
        else
        {
                heading="Welcome once again";
                cnt=new Integer(cnt.intValue()+1);
        }
        session.setAttribute("cnt",cnt);
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>"+heading);
        out.println("The number of previous accesses: "+cnt);
        out.println("</body>");
        out.println("</html>");
%>
```

Connecting to Database in JSP:

Example: The following example contains two files:

i) **Login.html:** It allows us to enter user name and password to authenticate the user by checking the values are available in Database or not.

ii) **Login.jsp:** It processes the entered username and password and displays the message accordingly.

Login.html:

```
<html>
    <head>
        <title>Login Page</title>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
    </head>
    <body>
    <center>
        <h1>Login Form</h1>
        <form action="Login.jsp" method="post">
            <table>
                <tr>
                    <td>User name:</td>
                    <td><input type="text" name="uname"/></td>
                </tr>
                <tr>
                    <td>Password:</td>
                    <td><input type="password" name="pwd"/></td>
                </tr>
                <tr>
                    <td><input type="submit" value="Login"></td>
                    <td><input type="reset" value="Clear"/></td>
                </tr>
            </table>
        </form>
    </center>
    </body>
</html>
```

Login.jsp:

```
<%@page language="java" import="java.sql.*" %>
<%
    String un=request.getParameter("uname");
    String pwd=request.getParameter("pwd");
    Class.forName("com.mysql.jdbc.Driver");
    Connection con=DriverManager.getConnection("jdbc:mysql://localhost:3306/scce", "root",
"");
    Statement stmt=con.createStatement();
    out.println("<head><title>Login Page</title></head>");
    ResultSet rs=stmt.executeQuery("select * from Login where uname='"+un+"' and
pwd='"+pwd+"'");
    if(rs.next())
        out.println("<h1>Welcome "+rs.getString(1)+", You are logged in successfully</h1>");
    else
        out.println("<h1>Login Failed</h1>");
%>
```

INTRODUCTION:

Software components are self-contained software units developed according to the motto "Developed them once, run and reused them everywhere". Or in other words, reusability is the main concern behind the component model. A software component is a reusable object that can be plugged into any target software application. You can develop software components using various programming languages, such as C, C++, Java, and Visual Basic.

- A "Bean" is a reusable software component model based on sun's java bean specification that can be manipulated visually in a builder tool.
- The term software component model describe how to create and use reusable software components to build an application
- Builder tool is nothing but an application development tool which lets you both to create new beans or use existing beans to create an application.
- To enrich the software systems by adopting component technology JAVA came up with the concept called Java Beans.
- Java provides the facility of creating some user defined components by means of Bean programming.
- We create simple components using java beans.
- We can directly embed these beans into the software.

ADVANTAGES OF JAVA BEANS:

- The java beans posses the property of "Write once and run anywhere".
- Beans can work in different local platforms.
- Beans have the capability of capturing the events sent by other objects and vice versa

enabling object communication.

- The properties, events and methods of the bean can be controlled by the application developer. (ex. Add new properties)
- Beans can be configured with the help of auxiliary software during design time.(no hassle at runtime)
- The configuration setting can be made persistent.(reused)
- Configuration setting of a bean can be saved in persistent storage and restored later.

WHAT CAN WE DO/CREATE BY USING JAVABEANS:

- There is no restriction on the capability of a Bean.
- It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio. A Bean may be visible to an end user. One example of this is a button on a graphical user interface.
- Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally.
- Bean that provides real-time price information from a stock or commodities exchange.

DEFINITION OF A BUILDER TOOL:

Builder tools allow a developer to work with JavaBeans in a convenient way. By examining a JavaBean by a process known as Introspection, a builder tool exposes the discovered features of the JavaBean for visual manipulation. A builder tool maintains a list of all JavaBeans available. It allows you to compose the Bean into applets, application, servlets and composite components (e.g. a JFrame), customize its behavior and appearance by modifying its properties and connect other components to the event of the Bean or vice versa.

JAVABEANS BASIC RULES:

A JavaBean should:

- be public
- implement the Serializable interface
- have a no-argument constructor
- be derived from javax.swing.JComponent or java.awt.Component if it is visual

The classes and interfaces defined in the JavaBeans package enable you to create JavaBeans. The Java Bean components can exist in one of the following three phases of development:

- Construction phase
- Build phase
- Execution phase

It supports the standard **component architecture** features of

- Properties
- Events
- Methods
- Persistence
- Introspection (Allows Automatic Analysis of a java beans)
- Customization (To make it easy to configure a java beans component)

ELEMENTS OF A JAVABEAN:

i) Properties

It is similar to instance variables. A bean *property* is a named attribute of a bean that can affect its behavior or appearance. Examples of bean properties include color, label, font, font size, and display size.

ii) Methods

- Same as normal Java methods.
- Every property should have accessor (get) and mutator (set) method.
- All Public methods can be identified by the introspection mechanism.
- There is no specific naming standard for these methods.

iii) Events:

It is similar to Swing/AWT event handling.

THE JAVABEAN COMPONENT SPECIFICATION:

i) Customization:

It is the ability of JavaBean to allow its properties to be changed in build and execution phase.

ii) Persistence:

It is the ability of JavaBean to save its state to disk or storage device and restore the saved state when the JavaBean is reloaded.

iii) Communication:

It is the ability of JavaBean to notify change in its properties to other JavaBeans or the container.

iv) Introspection:

It is the ability of a JavaBean to allow an external application to query the properties, methods, and events supported by it.

SERVICES OF JAVABEAN COMPONENTS:

i) Builder support:

It enables you to create and group multiple JavaBeans in an application.

ii) Layout:

It allows multiple JavaBeans to be arranged in a development environment.

iii) Interface publishing:

It enables multiple JavaBeans in an application to communicate with each other.

iv) Event handling:

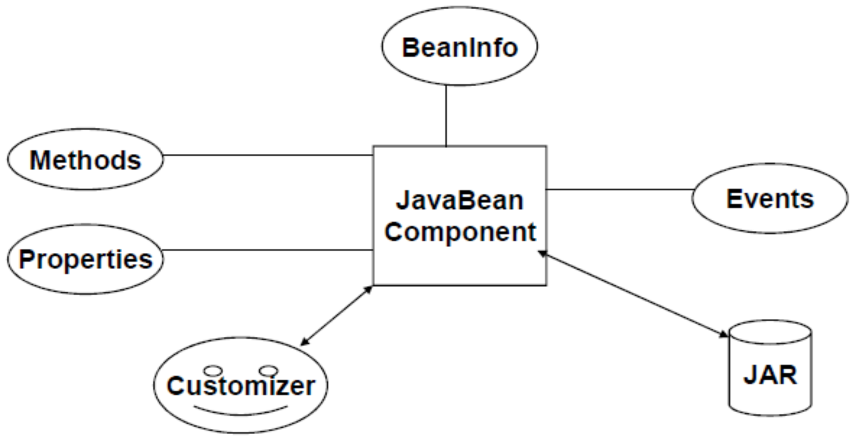
It refers to firing and handling of events associated with a JavaBean.

v) Persistence:

It enables you to save the last state of JavaBean.

FEATURES OF A JAVABEAN:

- Support for “introspection” so that a builder tool can analyze how a bean works.
- Support for “customization” to allow the customization of the appearance and behavior of a bean.
- Support for “events” as a simple communication metaphor than can be used to connect up beans.
- Support for “properties”, both for customization and for programmatic use.
- Support for “persistence”, so that a bean can save and restore its customized state.



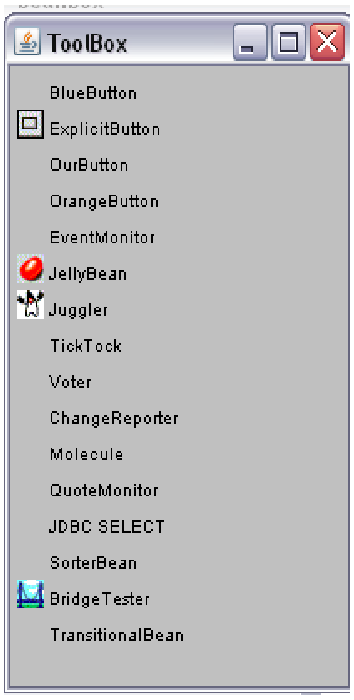
BEANS DEVELOPMENT KIT (BDK): It is a development environment to create, configure, and test JavaBeans.
The features of BDK environment are:

- Provides a GUI to create, configure, and test JavaBeans.
- Enables you to modify JavaBean properties and link multiple JavaBeans in an application using BDK.
- Provides a set of sample JavaBeans.
- Enables you to associate pre-defined events with sample JavaBeans.

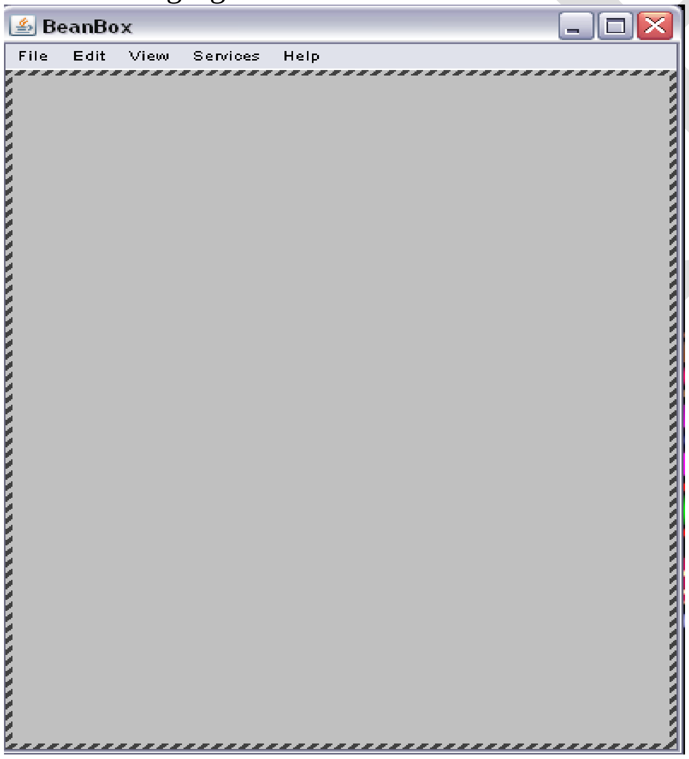
Identifying BDK Components

- Execute the run.bat file of BDK to start the BDK development environment.
- The components of BDK development environment are:
    - Tool Box
    - BeanBox
    - Properties-BeanBox
    - Method Tracer

i) **Tool Box Window**: Lists the sample JavaBeans of BDK.
The following figure shows the **Tool Box** window:

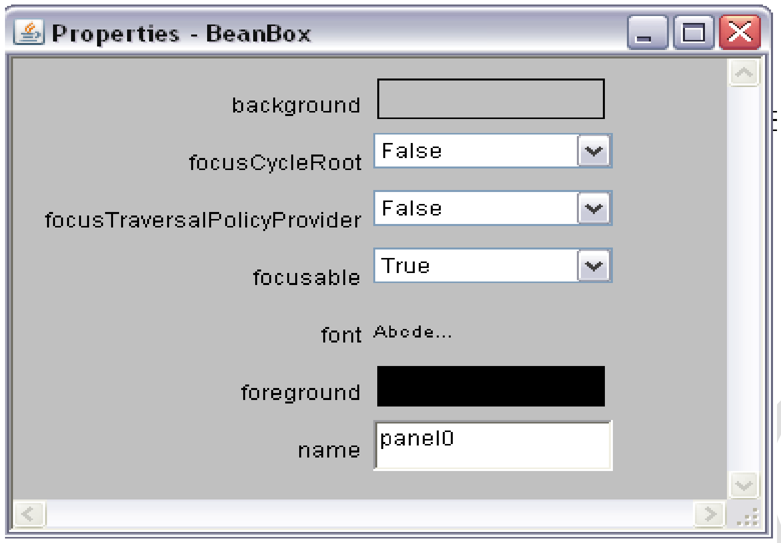


ii) **BeanBox Window**: It is a workspace for creating the layout of JavaBean application. The following figure shows the **BeanBox** window:



iii) **Properties Window**: It displays all the exposed properties of a JavaBean. You can modify JavaBean properties in the properties window.

The following figure shows the **Properties** window:



background

focusCycleRoot: False

focusTraversalPolicyProvider: False

focusable: True

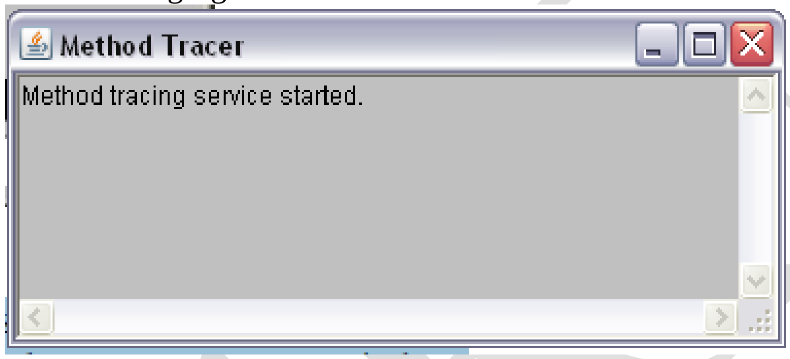
font: Abcde...

foreground

name: panel0

iv) **Method Tracer window**: It displays the debugging messages and method calls for a JavaBean application.

The following figure shows the **Method Tracer** window:



```
Method tracing service started.
```

STEPS TO DEVELOP A USER-DEFINED JAVABEAN:

1. Create a directory for the new bean
2. Create the java bean source file(s)
3. Compile the source file(s)
4. Create a manifest file
5. Generate a JAR file
6. Start BDK
7. Load Jar file
8. Test

1. Create a directory for the new bean

Create a directory/folder like C:\Beans

2. Create bean source file - MyBean.java

```
import java.awt.*;
public class MyBean extends Canvas
{
public MyBean()
{
setSize(70,50);
setBackground(Color.green);
}
}
```

3. Compile the source file(s)

C:\Beans >Javac MyBean.java

4. Create a manifest file

- The manifest file for a JavaBean application contains a list of all the class files that make up a JavaBean.
- The entry in the manifest file enables the target application to recognize the JavaBean classes for an application.

For example, the entry for the MyBean JavaBean in the manifest file is as shown:

```
Manifest-Version: 1.0
Name: MyBean.class
Java-Bean: True
```

Note: write that 2 as MyBean.mft

The rules to create a manifest file are:

- Press the Enter key after typing each line in the manifest file.
- Leave a space after the colon.
- Type a hyphen between Java and Bean.
- No blank line between the Name and the Java-Bean entry.

5. Generate a JAR file

- Syntax for creating jar file using manifest file

C:\Beans **>jar cfm MyBean.jar MyBean.mf MyBean.class**

JAR file:

- JAR file allows you to efficiently deploy a set of classes and their associated resources.
- JAR file makes it much easier to deliver, install, and download. It is compressed.

Java Archive File

- The files of a JavaBean application are compressed and grouped as JAR files to reduce the size and the download time of the files.
- The syntax to create a JAR file from the command prompt is:

jar <options> <file_names>

- The file_names is a list of files for a JavaBean application that are stored in the JAR file.

The various options that you can specify while creating a JAR file are:

- ❖ c: Indicates the new JAR file is created.
- ❖ f: Indicates that the first file in the file_names list is the name of the JAR file.
- ❖ m: Indicates that the second file in the file_names list is the name of the manifest file.
- ❖ t: Indicates that all the files and resources in the JAR file are to be displayed in a tabular format.
- ❖ v: Indicates that the JAR file should generate a verbose output.
- ❖ x: Indicates that the files and resources of a JAR file are to be extracted.
- ❖ o: Indicates that the JAR file should not be compressed.
- ❖ m: Indicates that the manifest file is not created.

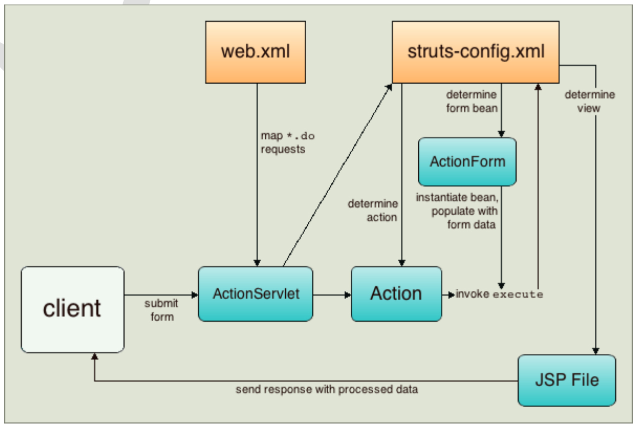
6. Start BDK

Go to->
C:\bdk1_1\beans\beanbox
Click on **run.bat** file. When we click on run.bat file the BDK software automatically started.
7. Load Jar file
Go to
Beanbox->File->Load jar. Here we have to select our created jar file when we click on ok, our bean (user-defined) MyBean appear in the ToolBox.
8. Test our created user defined bean
Select the MyBean from the ToolBox when we select that bean one + simple appear then drag that Bean in to the Beanbox. If you want to apply events for that bean, now we apply the events for that Bean.

INTRODUCTION TO STRUTS FRAMEWORK:
The **struts framework** is used to develop **MVC-based web application**. The struts framework was initially created by **Craig McClanahan** and donated to Apache Foundation in May, 2000 and Struts 1.0 was released in June 2001.

Struts Architecture and Flow:
When you use Struts, the framework provides you with a controller servlet, `ActionServlet`, which is defined in the Struts libraries that are included in the IDE, and which is automatically registered in the `web.xml` deployment descriptor as shown below. The controller servlet uses a `struts-config.xml` file to map incoming requests to Struts `Action` objects, and instantiate any `ActionForm` objects associated with the action to temporarily store form data. The `Action` object processes requests using its `execute` method, while making use of any data stored in the form bean. Once the `Action` object processes a request, it stores any new data (i.e., in the form bean, or in a separate result bean), and forwards the results to the appropriate view.

Developing a Struts application is similar to developing any other kind of web application in NetBeans IDE. However, you complement your web development toolkit by taking advantage of the Struts support provided by the IDE. For example, you use templates in the IDE to create Struts `Action` objects and `ActionForm` beans. Upon creation, the IDE automatically registers these classes in the `struts-config.xml` file and lets you extend this file very easily using menu items in the Source Editor's right-click menu. Because many web applications use JSP pages for the view, Struts also provides custom tag libraries which facilitate interaction with HTML forms. Within the IDE's Source Editor, you can invoke code completion and Javadoc support that helps you to work efficiently with these libraries.


web.xml
struts-config.xml
determine form bean
determine view
map *.do requests
ActionForm
determine action
instantiate bean, populate with form data
client
submit form
ActionServlet
Action
invoke execute
JSP File
send response with processed data