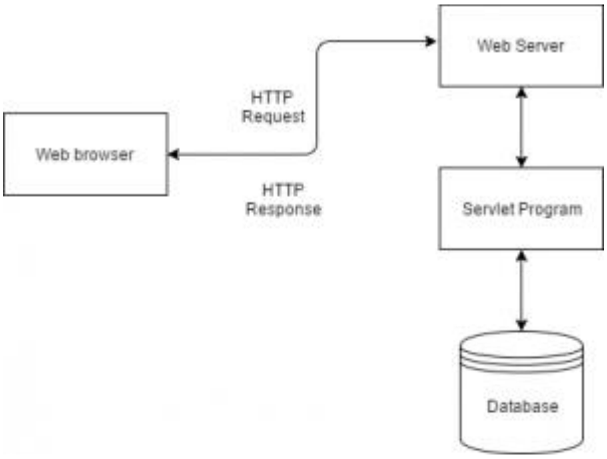
# Introduction to Java Servlets

Servlets are the Java programs that run on the Java-enabled web server or application server. They are used to handle the request obtained from the webserver, process the request, produce the response, then send a response back to the webserver.

Properties of Servlets are as follows:

- Servlets work on the server-side.
- Servlets are capable of handling complex requests obtained from the webserver.

Servlet Architecture is can be depicted from the image itself as provided below as follows:



Execution of Servlets basically involves six basic steps:

1. The clients send the request to the webserver.
2. The web server receives the request.
3. The web server passes the request to the corresponding servlet.
4. The servlet processes the request and generates the response in the form of output.
5. The servlet sends the response back to the webserver.
6. The web server sends the response back to the client and the client browser displays it on the screen.

Now let us do discuss eccentric point that why do we need For Server-Side extensions?

The **server-side extensions** are nothing but the technologies that are used to create dynamic Web pages. Actually, to provide the facility of dynamic Web pages, Web pages need a container or Web server. To meet this requirement, independent Web server providers offer some proprietary solutions in the form of **APIs**(Application Programming Interface).

These **APIs** allow us to build programs that can run with a Web server. In this case, **Java Servlet** is also one of the component APIs of **Java Platform Enterprise Edition** which sets standards for creating dynamic Web applications in Java.

The Servlet technology is similar to other Web server extensions such as **Common Gateway Interface**(CGI) scripts and **Hypertext Preprocessor** (PHP). However, Java Servlets are more acceptable since they solve the limitations of **CGI** such as low performance and low degree scalability.

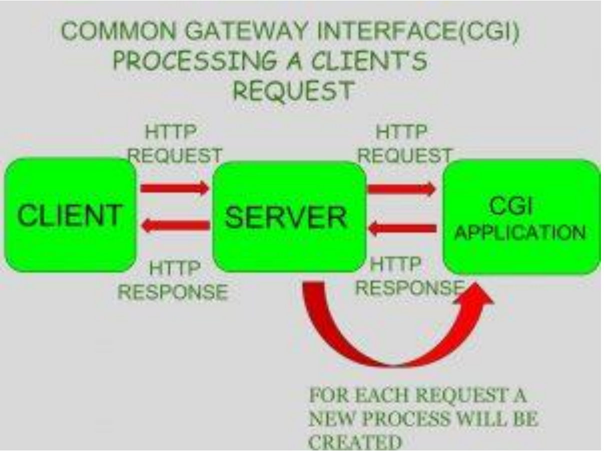
**What is CGI?**

**CGI** is actually an external application that is written by using any of the programming languages like **C** or **C++** and this is responsible for processing client requests and generating dynamic content.

In CGI application, when a client makes a request to access dynamic Web pages, the Web server performs the following operations :

- It first locates the requested web page *i.e* the required CGI application using URL.
- It then creates a new process to service the client's request.
- Invokes the CGI application within the process and passes the request information to the application.
- Collects the response from the CGI application.

- Destroys the process, prepares the HTTP response, and sends it to the client.


COMMON GATEWAY INTERFACE(CGI) PROCESSING A CLIENT'S REQUEST
HTTP REQUEST
HTTP REQUEST
CLIENT
SERVER
CGI APPLICATION
HTTP RESPONSE
HTTP RESPONSE
FOR EACH REQUEST A NEW PROCESS WILL BE CREATED

So, in **CGI** server has to create and destroy the process for every request. It's easy to understand that this approach is applicable for handling few clients but as the number of clients increases, the workload on the server increases and so the time is taken to process requests increases.

**Difference between Servlet and CGI**

| **Servlet** | **CGI(Common Gateway Interface)** |
|---|---|
| Servlets are portable and efficient. | CGI is not portable |
| In Servlets, sharing data is possible. | In CGI, sharing data is not possible. |
| Servlets can directly communicate with the webserver. | CGI cannot directly communicate with the webserver. |
| Servlets are less expensive than CGI. | CGI is more expensive than Servlets. |
| Servlets can handle the cookies. | CGI cannot handle the cookies. |

**Servlets API's:**

Servlets are build from two packages:

- javax.servlet(Basic)
- javax.servlet.http(Advance)

Various classes and interfaces present in these packages are:

| **Component** | **Type** | **Package** |
|---|---|---|
| Servlet | Interface | javax.servlet.* |

| Component | Type | Package |
|---|---|---|
| ServletRequest | Interface | javax.servlet.* |
| ServletResponse | Interface | javax.servlet.* |
| GenericServlet | Class | javax.servlet.* |
| HttpServlet | Class | javax.servlet.http.* |
| HttpServletRequest | Interface | javax.servlet.http.* |
| HttpServletResponse | Interface | javax.servlet.http.* |
| Filter | Interface | javax.servlet.* |
| ServletConfig | Interface | javax.servlet.* |

## Advantages of a Java Servlet

- Servlet is **faster** than CGI as it doesn't involve the creation of a new process for every new request received.
- Servlets, as written in Java, are platform-independent.
- Removes the overhead of creating a **new process** for each request as Servlet doesn't run in a separate process. There is only a single instance that handles all requests concurrently. This also saves the memory and allows a Servlet to easily manage the client state.
- It is a server-side component, so Servlet inherits the **security** provided by the Web server.
- The **API** designed for Java Servlet automatically acquires the advantages of the Java platforms such as platform-independent and portability. In addition, it obviously can use the wide range of APIs created on Java platforms such as **JDBC** to access the database.
- Many Web servers that are suitable for personal use or low-traffic websites are offered for free or at extremely **cheap costs** eg. Java servlet. However, the majority of commercial-grade Web servers are rather expensive, with the notable exception of Apache, which is free.

## The Servlet Container

**Servlet container**, also known as **Servlet engine** is an integrated set of objects that provide a run time environment for Java Servlet components.
In simple words, it is a system that manages Java Servlet components on top of the Web server to handle the Web client requests.

**Services provided by the Servlet container :**

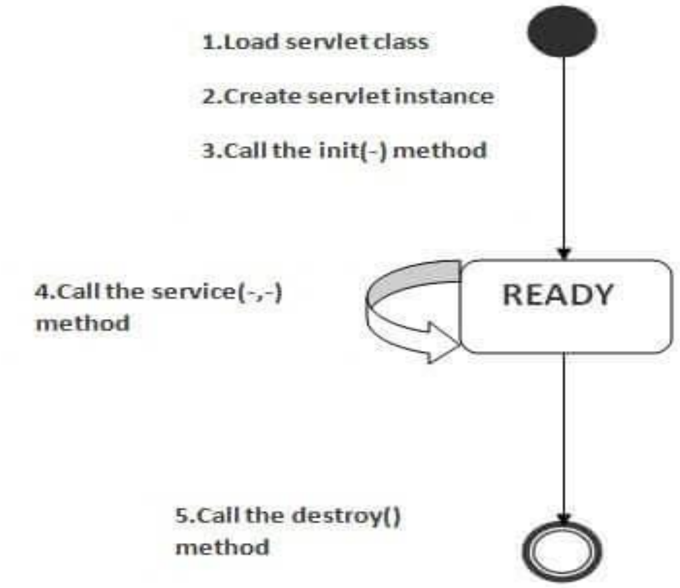
- **Network Services:** Loads a Servlet class. The loading may be from a local file system, a remote file system or other network services. The Servlet container provides the network services over which the request and response are sent.

- **Decode and Encode MIME-based messages:** Provides the service of decoding and encoding MIME-based messages.
- **Manage Servlet container:** Manages the lifecycle of a Servlet.
- **Resource management** Manages the static and dynamic resources, such as HTML files, Servlets, and JSP pages.
- **Security Service:** Handles authorization and authentication of resource access.
- **Session Management:** Maintains a session by appending a **session ID** to the URL path.

## Life Cycle of a Servlet (Servlet Life Cycle)

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

### 1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

### 2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

### 3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method given below:

```
public void init(ServletConfig config) throws ServletException
```

### 4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

```
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException
```

### 5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
public void destroy()
```

## Uses of Package

# javax.servlet

| Packages that use javax.servlet | |
|---|---|
| **javax.servlet** | The javax.servlet package contains a number of classes and interfaces that describe and define the contracts between a servlet class and the runtime environment provided for an instance of such a class by a conforming servlet container. |
| **javax.servlet.http** | The javax.servlet.http package contains a number of classes and interfaces that describe and define the contracts between a servlet class running under the HTTP protocol and the runtime environment provided for an instance of such a class by a conforming servlet container. |

| Classes in javax.servlet used by javax.servlet |
|---|
| **FilterChain**<br>A FilterChain is an object provided by the servlet container to the developer giving a view into the invocation chain of a filtered request for a resource. |
| **FilterConfig**<br>A filter configuration object used by a servlet container to pass information to a filter during initialization. |
| **RequestDispatcher**<br>Defines an object that receives requests from the client and sends them to any resource (such as a servlet, HTML file, or JSP file) on the server. |
| **Servlet**<br>Defines methods that all servlets must implement. |
| **ServletConfig**<br>A servlet configuration object used by a servlet container to pass information to a servlet during initialization. |
| **ServletContext**<br>Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file. |
| **ServletContextAttributeEvent**<br>This is the event class for notifications about changes to the attributes of the servlet context of a web application. |
| **ServletContextEvent**<br>This is the event class for notifications about changes to the servlet context of a web application. |
| **ServletException**<br>Defines a general exception a servlet can throw when it encounters difficulty. |
| **ServletInputStream**<br>Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time. |
| **ServletOutputStream**<br>Provides an output stream for sending binary data to the client. |
| **ServletRequest**<br>Defines an object to provide client request information to a servlet. |
| **ServletRequestAttributeEvent**<br>This is the event class for notifications of changes to the attributes of the servlet request in an application. |
| **ServletRequestEvent** |

| |
|---|
| Events of this kind indicate lifecycle events for a ServletRequest. |
| **ServletResponse**<br>Defines an object to assist a servlet in sending a response to the client. |

| Classes in javax.servlet used by javax.servlet.http |
|---|
| **GenericServlet**<br>Defines a generic, protocol-independent servlet. |
| **Servlet**<br>Defines methods that all servlets must implement. |
| **ServletConfig**<br>A servlet configuration object used by a servlet container to pass information to a servlet during initialization. |
| **ServletContext**<br>Defines a set of methods that a servlet uses to communicate with its servlet container, for example, to get the MIME type of a file, dispatch requests, or write to a log file. |
| **ServletException**<br>Defines a general exception a servlet can throw when it encounters difficulty. |
| **ServletInputStream**<br>Provides an input stream for reading binary data from a client request, including an efficient readLine method for reading data one line at a time. |
| **ServletRequest**<br>Defines an object to provide client request information to a servlet. |
| **ServletRequestWrapper**<br>Provides a convenient implementation of the ServletRequest interface that can be subclassed by developers wishing to adapt the request to a Servlet. |
| **ServletResponse**<br>Defines an object to assist a servlet in sending a response to the client. |
| **ServletResponseWrapper**<br>Provides a convenient implementation of the ServletResponse interface that can be subclassed by developers wishing to adapt the response from a Servlet. |

## Reading Servlet Parameters

The **ServletRequest** interface includes methods that allow you to read the names and values of parameters that are included in a client request. We will develop a servlet that illustrates their use. The example contains two files. A web page is defined in **PostParameters.html**, and a servlet is defined in **PostParametersServlet.java**.

The HTML source code for **PostParameters.html** is shown in the following listing. It defines a table that contains two labels and two text fields. One of the labels is Employee and the other is Phone. There is also a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies the servlet to process the HTTP POST request.

```
<html>

<body>

<center>

<form name="Form1" method="post"

action="http://localhost:8080/examples/servlets/

servlet/PostParametersServlet">

<table>

<tr>

<td><B>Employee</td>

<td><input type=textbox name="e" size="25" value=""></td> </tr>

<tr>

<td><B>Phone</td>

<td><input type=textbox name="p" size="25" value=""></td> </tr>

</table>

<input type=submit value="Submit"> </body>

</html>
```

The source code for **PostParametersServlet.java** is shown in the following listing. The **service( )** method is overridden to process client requests. The **getParameterNames( )** method returns an enumeration of the parameter names. These are processed in a loop. You can see that the parameter name and value are output to the client. The parameter value is obtained via the **getParameter( )** method.

```
import java.io.*; import java.util.*; import javax.servlet.*;

public class PostParametersServlet extends GenericServlet {
```

```
public void service(ServletRequest request, ServletResponse response)

throws ServletException, IOException {

// Get print writer.

PrintWriter pw = response.getWriter();

    Get enumeration of parameter names. Enumeration e = request.getParameterNames();

    Display parameter names and values. while(e.hasMoreElements()) {

String pname = (String)e.nextElement(); pw.print(pname + " = ");

String pvalue = request.getParameter(pname); pw.println(pvalue);
}

pw.close();

}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

Start Tomcat (if it is not already running).

Display the web page in a browser.

Enter an employee name and phone number in the text fields.

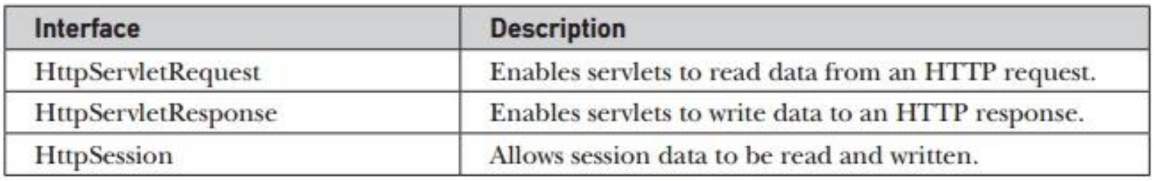
Submit the web page.

After following these steps, the browser will display a response that is dynamically generated by the servlet.

**The javax.servlet.http Package**

The preceding examples have used the classes and interfaces defined in **javax.servlet**, such as **ServletRequest**, **ServletResponse**, and **GenericServlet**, to illustrate the basic functionality of servlets. However, when working with HTTP, you will normally use the interfaces and classes in **javax.servlet.http**. As you will see, its functionality makes it easy to build servlets that work with HTTP requests and responses.

The following table summarizes the interfaces used in this chapter:



| Interface | Description |
|---|---|
| HttpServletRequest | Enables servlets to read data from an HTTP request. |
| HttpServletResponse | Enables servlets to write data to an HTTP response. |
| HttpSession | Allows session data to be read and written. |

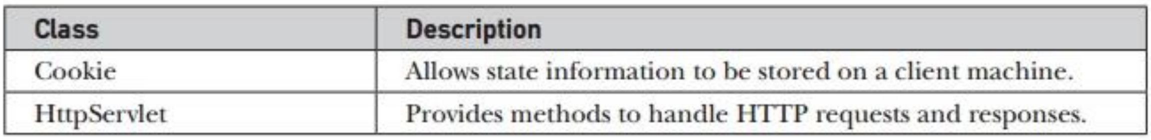
**Interface : Description**

HttpServletRequest : Enables servlets to read data from an HTTP request.

HttpServletResponse : Enables servlets to write data to an HTTP response.

HttpSession : Allows session data to be read and written.

The following table summarizes the classes used in this chapter. The most important of these is **HttpServlet**. Servlet developers typically extend this class in order to process HTTP requests.



| Class | Description |
|---|---|
| Cookie | Allows state information to be stored on a client machine. |
| HttpServlet | Provides methods to handle HTTP requests and responses. |

**Class : Description**

Cookie : Allows state information to be stored on a client machine.

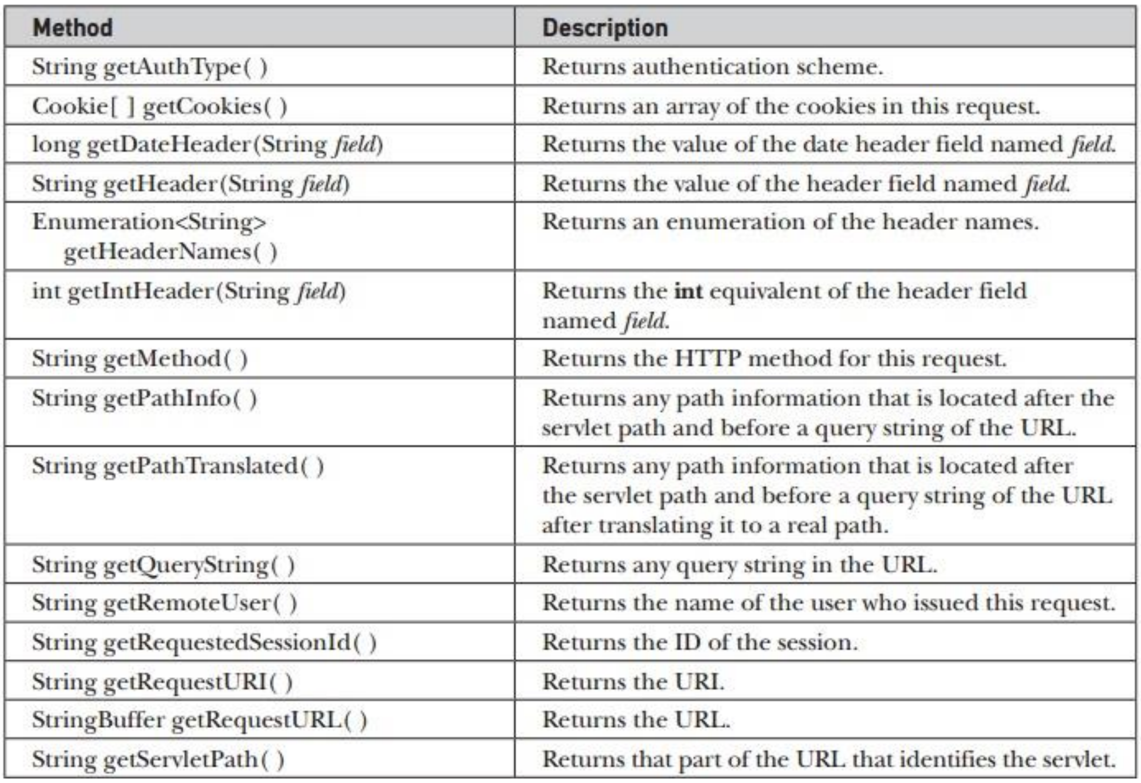
HttpServlet : Provides methods to handle HTTP requests and responses.

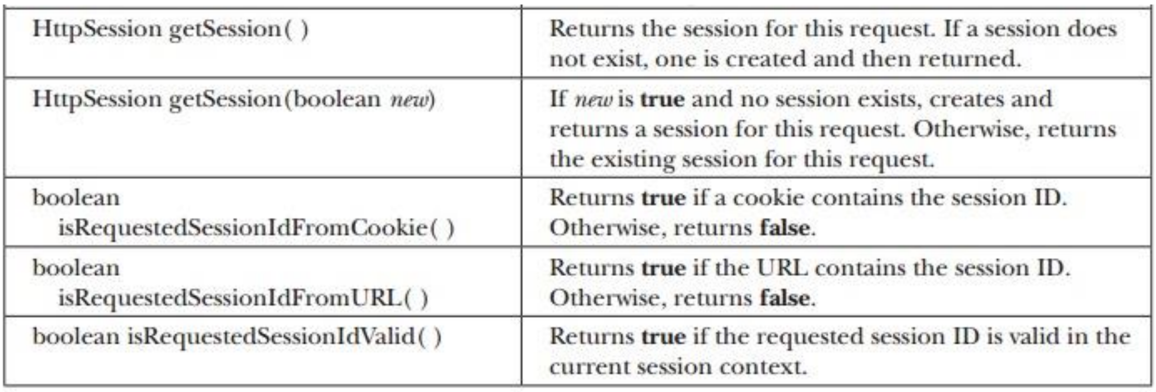
**The HttpServletRequest Interface**

The **HttpServletRequest** interface enables a servlet to obtain information about a client request. Several of its methods are shown in Table 38-5.

**The HttpServletResponse Interface**

The **HttpServletResponse** interface enables a servlet to formulate an HTTP response to a client. Several constants are defined. These correspond to the different status codes that can be assigned to an HTTP response. For example, **SC_OK** indicates that the HTTP





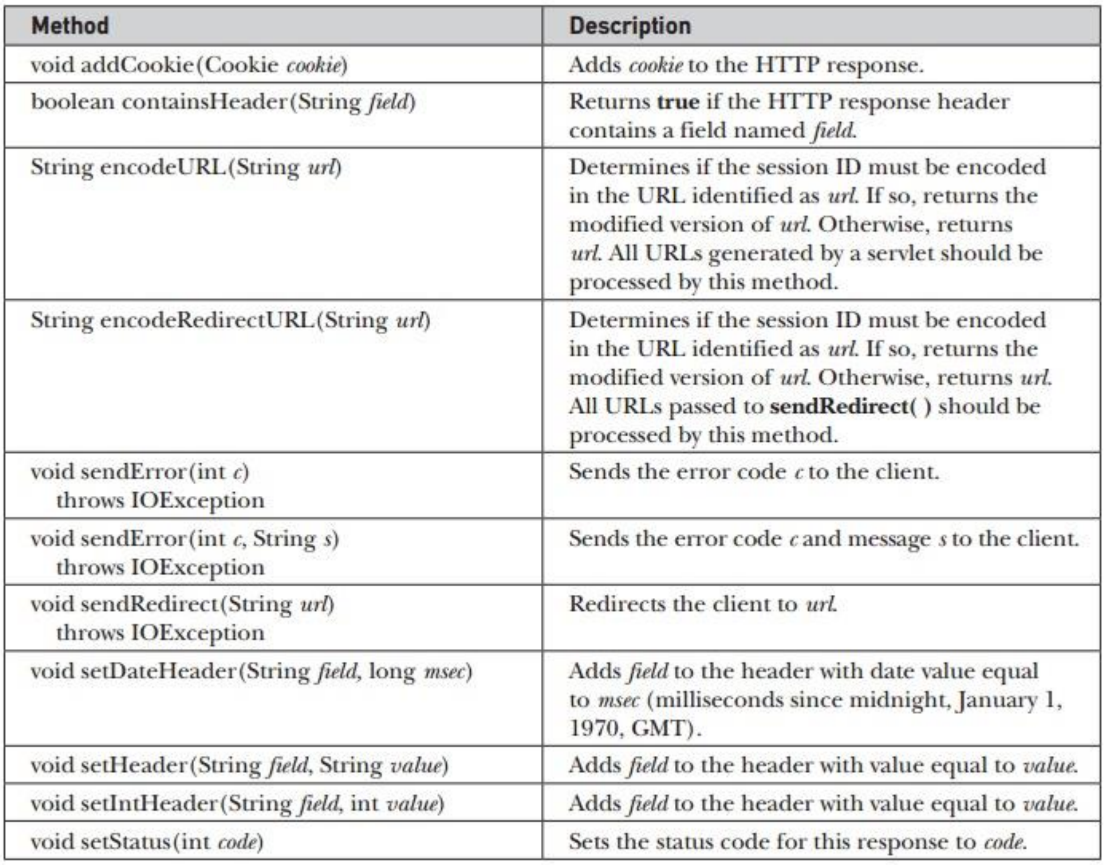
| Method | Description |
|---|---|
| String getAuthType( ) | Returns authentication scheme. |
| Cookie[ ] getCookies( ) | Returns an array of the cookies in this request. |
| long getDateHeader(String *field*) | Returns the value of the date header field named *field*. |
| String getHeader(String *field*) | Returns the value of the header field named *field*. |
| Enumeration<String> getHeaderNames( ) | Returns an enumeration of the header names. |
| int getIntHeader(String *field*) | Returns the **int** equivalent of the header field named *field*. |
| String getMethod( ) | Returns the HTTP method for this request. |
| String getPathInfo( ) | Returns any path information that is located after the servlet path and before a query string of the URL. |
| String getPathTranslated( ) | Returns any path information that is located after the servlet path and before a query string of the URL after translating it to a real path. |
| String getQueryString( ) | Returns any query string in the URL. |
| String getRemoteUser( ) | Returns the name of the user who issued this request. |
| String getRequestedSessionId( ) | Returns the ID of the session. |
| String getRequestURI( ) | Returns the URI. |
| StringBuffer getRequestURL( ) | Returns the URL. |
| String getServletPath( ) | Returns that part of the URL that identifies the servlet. |
| HttpSession getSession( ) | Returns the session for this request. If a session does not exist, one is created and then returned. |
| HttpSession getSession(boolean *new*) | If *new* is **true** and no session exists, creates and returns a session for this request. Otherwise, returns the existing session for this request. |
| boolean isRequestedSessionIdFromCookie( ) | Returns **true** if a cookie contains the session ID. Otherwise, returns **false**. |
| boolean isRequestedSessionIdFromURL( ) | Returns **true** if the URL contains the session ID. Otherwise, returns **false**. |
| boolean isRequestedSessionIdValid( ) | Returns **true** if the requested session ID is valid in the current session context. |

**Table 38-5** Various Methods Defined by **HttpServletRequest**

request succeeded, and **SC_NOT_FOUND** indicates that the requested resource is not available. Several methods of this interface are summarized in Table 38-6.

### The HttpSession Interface

The **HttpSession** interface enables a servlet to read and write the state information that is associated with an HTTP session. Several of its methods are summarized in Table 38-7. All of these methods throw an **IllegalStateException** if the session has already been invalidated.



| Method | Description |
|---|---|
| void addCookie(Cookie *cookie*) | Adds *cookie* to the HTTP response. |
| boolean containsHeader(String *field*) | Returns **true** if the HTTP response header contains a field named *field*. |
| String encodeURL(String *url*) | Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs generated by a servlet should be processed by this method. |
| String encodeRedirectURL(String *url*) | Determines if the session ID must be encoded in the URL identified as *url*. If so, returns the modified version of *url*. Otherwise, returns *url*. All URLs passed to **sendRedirect( )** should be processed by this method. |
| void sendError(int *c*)<br>throws IOException | Sends the error code *c* to the client. |
| void sendError(int *c*, String *s*)<br>throws IOException | Sends the error code *c* and message *s* to the client. |
| void sendRedirect(String *url*)<br>throws IOException | Redirects the client to *url*. |
| void setDateHeader(String *field*, long *msec*) | Adds *field* to the header with date value equal to *msec* (milliseconds since midnight, January 1, 1970, GMT). |
| void setHeader(String *field*, String *value*) | Adds *field* to the header with value equal to *value*. |
| void setIntHeader(String *field*, int *value*) | Adds *field* to the header with value equal to *value*. |
| void setStatus(int *code*) | Sets the status code for this response to *code*. |

**Table 38-6** Various Methods Defined by **HttpServletResponse**

## The Cookie Class

The **Cookie** class encapsulates a cookie. A *cookie* is stored on a client and contains state information. Cookies are valuable for tracking user activities. For example, assume that a user visits an online store. A cookie can save the user’s name, address, and other information. The user does not need to enter this data each time he or she visits the store.

A servlet can write a cookie to a user’s machine via the **addCookie( )** method of the **HttpServletResponse** interface. The data for that cookie is then included in the header of the HTTP response that is sent to the browser.

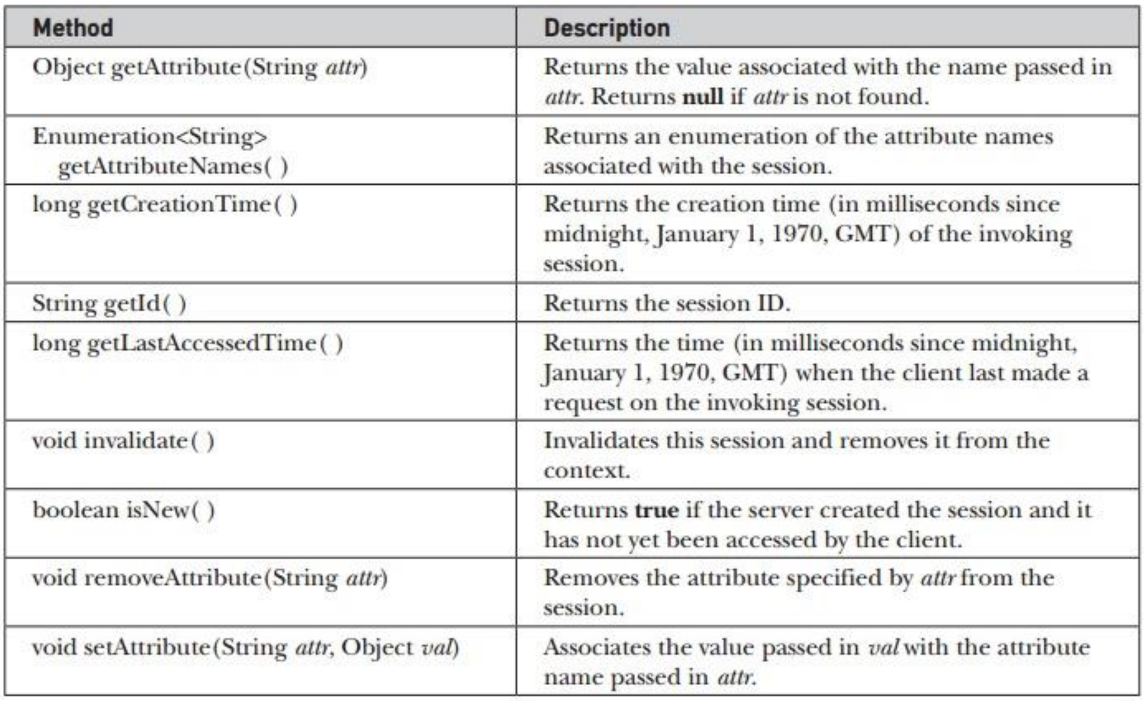
The names and values of cookies are stored on the user’s machine. Some of the information that can be saved for each cookie includes the following:

The name of the cookie

The value of the cookie

The expiration date of the cookie

The domain and path of the cookie



| Method | Description |
|---|---|
| Object getAttribute(String *attr*) | Returns the value associated with the name passed in *attr*. Returns **null** if *attr* is not found. |
| Enumeration<String> getAttributeNames( ) | Returns an enumeration of the attribute names associated with the session. |
| long getCreationTime( ) | Returns the creation time (in milliseconds since midnight, January 1, 1970, GMT) of the invoking session. |
| String getId( ) | Returns the session ID. |
| long getLastAccessedTime( ) | Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the client last made a request on the invoking session. |
| void invalidate( ) | Invalidates this session and removes it from the context. |
| boolean isNew( ) | Returns **true** if the server created the session and it has not yet been accessed by the client. |
| void removeAttribute(String *attr*) | Removes the attribute specified by *attr* from the session. |
| void setAttribute(String *attr*, Object *val*) | Associates the value passed in *val* with the attribute name passed in *attr*. |

**Table 38-7** Various Methods Defined by **HttpSession**

The expiration date determines when this cookie is deleted from the user's machine. If an expiration date is not explicitly assigned to a cookie, it is deleted when the current browser session ends.

The domain and path of the cookie determine when it is included in the header of an HTTP request. If the user enters a URL whose domain and path match these values, the cookie is then supplied to the web server. Otherwise, it is not.

There is one constructor for **Cookie**. It has the signature shown here: Cookie(String *name*, String *value*)

Here, the name and value of the cookie are supplied as arguments to the constructor. The methods of the **Cookie** class are summarized in Table 38-8.

**The HttpServlet Class**

The **HttpServlet** class extends **GenericServlet**. It is commonly used when developing servlets that receive and process HTTP requests. The methods defined by the **HttpServlet** class are summarized in Table 38-9.

**Method : Description**

Object clone( ) : Returns a copy of this object.

String getComment( ) : Returns the comment.

String getDomain( ) : Returns the domain.

int getMaxAge( ) : Returns the maximum age (in seconds).

String getName( ) : Returns the name.

String getPath( ) : Returns the path.

boolean getSecure( ) : Returns true if the cookie is secure. Otherwise, returns false.

String getValue( ) : Returns the value.

int getVersion( ) : Returns the version.

boolean isHttpOnly(?) : Returns true if the cookie has the HttpOnly attribute.

void setComment(String c) : Sets the comment to c.

void setDomain(String d) : Sets the domain to d.
void setHttpOnly(boolean httpOnly) : If httpOnly is true, then the HttpOnly attribute is added to the cookie. If httpOnly is false, the HttpOnly attribute is removed.

void setMaxAge(int secs) : Sets the maximum age of the cookie to secs. This is the number of seconds after which the cookie is deleted.

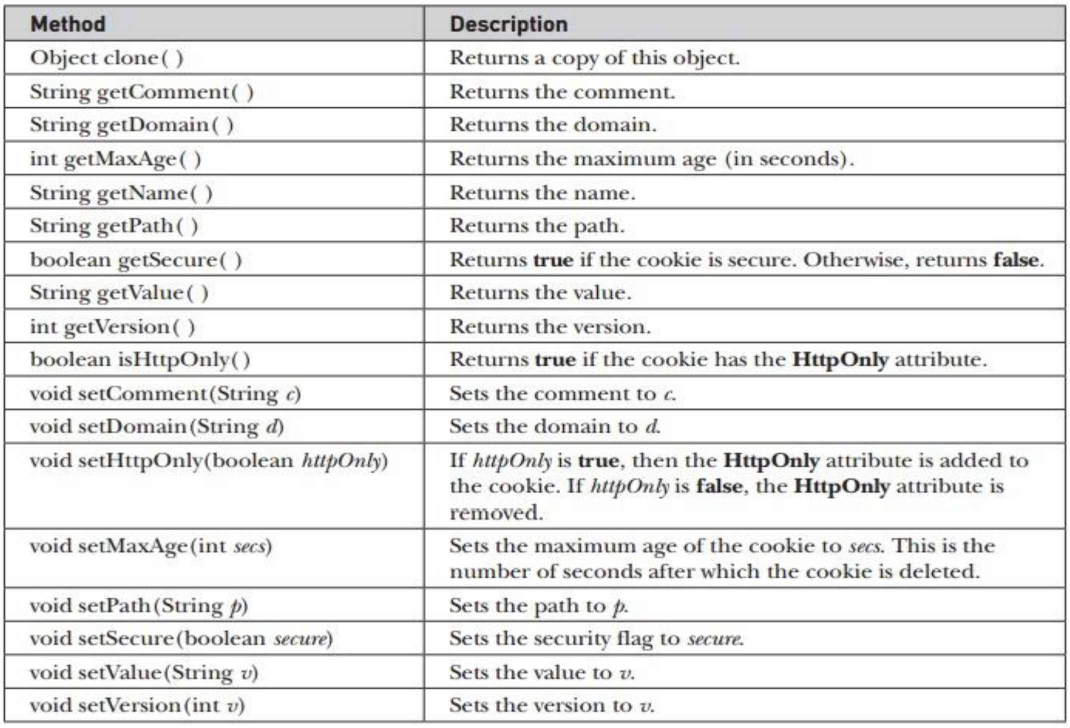
void setPath(String p) : Sets the path to p.

void setSecure(boolean secure) : Sets the security flag to secure.

void setValue(String v) : Sets the value to v.

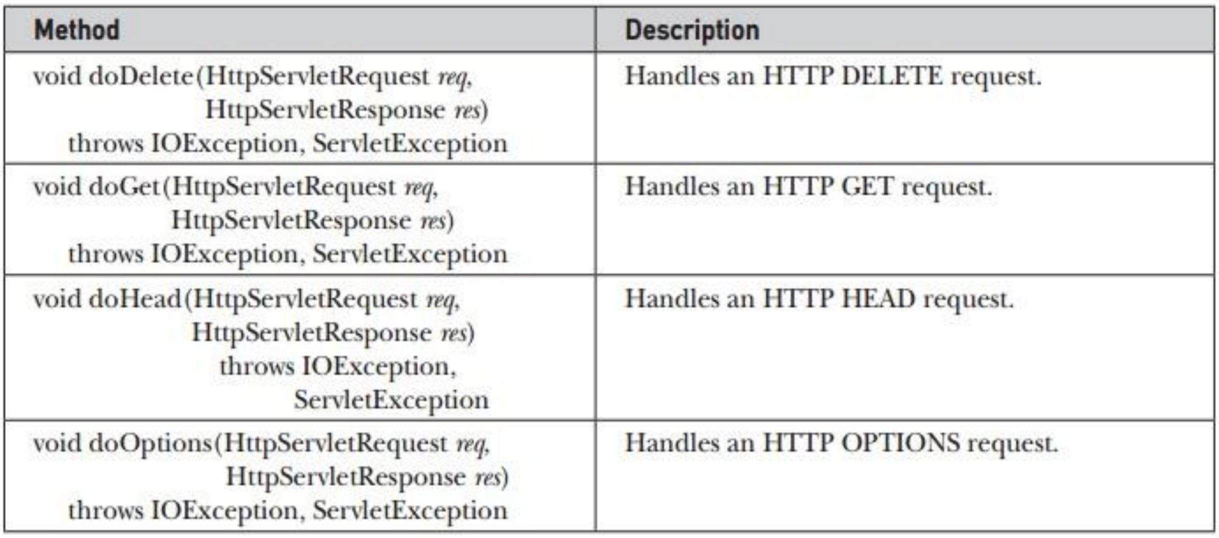
void setVersion(int v) : Sets the version to v.

**Table 38-8 The Methods Defined by Cookie**



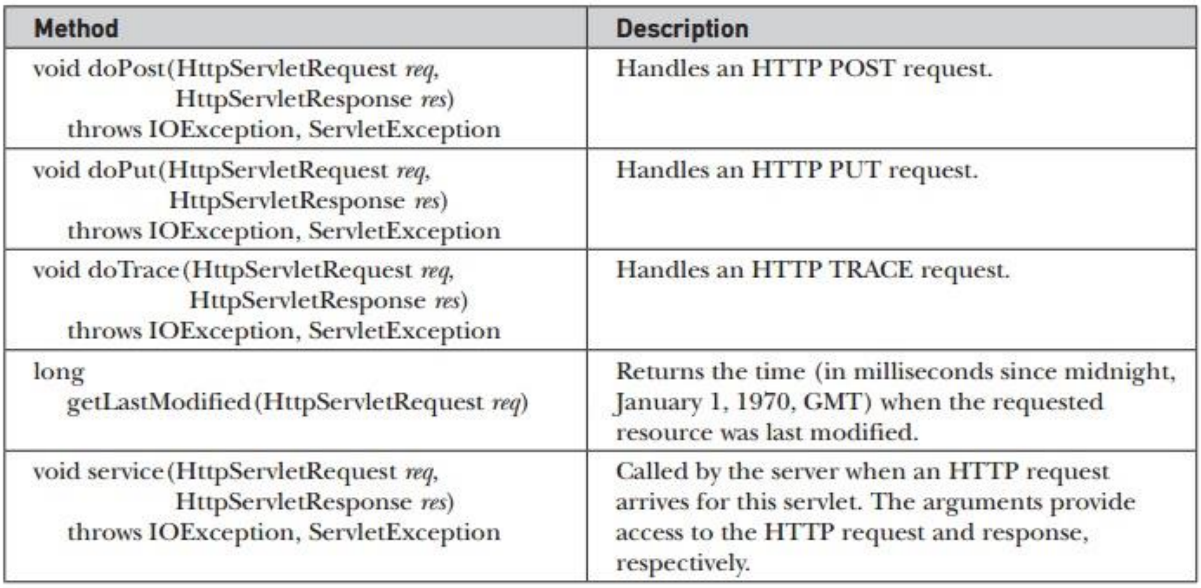
| Method | Description |
|---|---|
| Object clone( ) | Returns a copy of this object. |
| String getComment( ) | Returns the comment. |
| String getDomain( ) | Returns the domain. |
| int getMaxAge( ) | Returns the maximum age (in seconds). |
| String getName( ) | Returns the name. |
| String getPath( ) | Returns the path. |
| boolean getSecure( ) | Returns **true** if the cookie is secure. Otherwise, returns **false**. |
| String getValue( ) | Returns the value. |
| int getVersion( ) | Returns the version. |
| boolean isHttpOnly( ) | Returns **true** if the cookie has the **HttpOnly** attribute. |
| void setComment(String *c*) | Sets the comment to *c*. |
| void setDomain(String *d*) | Sets the domain to *d*. |
| void setHttpOnly(boolean *httpOnly*) | If *httpOnly* is **true**, then the **HttpOnly** attribute is added to the cookie. If *httpOnly* is **false**, the **HttpOnly** attribute is removed. |
| void setMaxAge(int *secs*) | Sets the maximum age of the cookie to *secs*. This is the number of seconds after which the cookie is deleted. |
| void setPath(String *p*) | Sets the path to *p*. |
| void setSecure(boolean *secure*) | Sets the security flag to *secure*. |
| void setValue(String *v*) | Sets the value to *v*. |
| void setVersion(int *v*) | Sets the version to *v*. |

**Table 38-8** The Methods Defined by **Cookie**



| Method | Description |
|---|---|
| void doDelete(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP DELETE request. |
| void doGet(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP GET request. |
| void doHead(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP HEAD request. |
| void doOptions(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP OPTIONS request. |

**Table 38-9** The Methods Defined by **HttpServlet**



| Method | Description |
|---|---|
| void doPost(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP POST request. |
| void doPut(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP PUT request. |
| void doTrace(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Handles an HTTP TRACE request. |
| long getLastModified(HttpServletRequest *req*) | Returns the time (in milliseconds since midnight, January 1, 1970, GMT) when the requested resource was last modified. |
| void service(HttpServletRequest *req*, HttpServletResponse *res*) throws IOException, ServletException | Called by the server when an HTTP request arrives for this servlet. The arguments provide access to the HTTP request and response, respectively. |

**Table 38-9** The Methods Defined by **HttpServlet** *(continued)*

## Handling HTTP Requests and Responses

The **HttpServlet** class provides specialized methods that handle the various types of HTTP requests. A servlet developer typically overrides one of these methods. These methods are **doDelete( )**, **doGet( )**, **doHead( )**, **doOptions( )**, **doPost( )**, **doPut( )**, and **doTrace( )**. A complete description of the different types of HTTP requests is beyond the scope of this book. However, the GET and POST requests are commonly used when handling form input. Therefore, this section presents examples of these cases.

## Handling HTTP GET Requests

Here we will develop a servlet that handles an HTTP GET request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorGet.html**, and a servlet is defined in **ColorGetServlet.java**. The HTML source code for **ColorGet.html** is shown in the following listing. It defines a form that contains a select element and a submit button. Notice that the action parameter of the form tag specifies a URL. The URL identifies a servlet to process the HTTP GET request.

```
<html>
<body>
<center>
<form
name="Form1" action="http://localhost:8080/examples/servlets/servlet/ColorGetServlet">
<B>Color:</B>
<select name="color" size="1"> <option value="Red">Red</option> <option value="Green">Green</option>
<option value="Blue">Blue</option> </select>
<br><br>
<input type=submit value="Submit"> </form>
</body>
</html>
```

The source code for **ColorGetServlet.java** is shown in the following listing. The **doGet( )** method is overridden to process any HTTP GET requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;
public class ColorGetServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
String color = request.getParameter("color"); response.setContentType("text/html"); PrintWriter pw = response.getWriter(); pw.println("<B>The selected color is: "); pw.println(color);
pw.close();
}
}
```

Compile the servlet. Next, copy it to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

Start Tomcat, if it is not already running.

Display the web page in a browser.

Select a color.

Submit the web page.

After completing these steps, the browser will display the response that is dynamically generated by the servlet.

One other point: Parameters for an HTTP GET request are included as part of the URL that is sent to the web server. Assume that the user selects the red option and submits the form. The URL sent from the browser to the server is

```
http://localhost:8080/examples/servlets/servlet/ColorGetServlet?color=Red
```

The characters to the right of the question mark are known as the *query string*.

## <u>Handling HTTP POST Requests</u>

Here we will develop a servlet that handles an HTTP POST request. The servlet is invoked when a form on a web page is submitted. The example contains two files. A web page is defined in **ColorPost.html**, and a servlet is defined in **ColorPostServlet.java**.

The HTML source code for **ColorPost.html** is shown in the following listing. It is identical to **ColorGet.html** except that the method parameter for the form tag explicitly specifies that the POST method should be used, and the action parameter for the form tag specifies a different servlet.

```
<html>

<body>

<center>

<form name="Form1" method="post"

action="http://localhost:8080/examples/servlets/servlet/ColorPostServlet">

<B>Color:</B>
```

```
<select name="color" size="1"> <option value="Red">Red</option> <option value="Green">Green</option> <option value="Blue">Blue</option> </select>
<br><br>

<input type=submit value="Submit"> </form>

</body>

</html>
```

The source code for **ColorPostServlet.java** is shown in the following listing. The **doPost( )** method is overridden to process any HTTP POST requests that are sent to this servlet. It uses the **getParameter( )** method of **HttpServletRequest** to obtain the selection that was made by the user. A response is then formulated.

```
import java.io.*; import javax.servlet.*;
import javax.servlet.http.*;

public class ColorPostServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String color = request.getParameter("color"); response.setContentType("text/html"); PrintWriter pw = response.getWriter(); pw.println("<B>The selected color is: "); pw.println(color);

pw.close();

}

}
```

Compile the servlet and perform the same steps as described in the previous section to test it.

**Using Cookies**

Now, let’s develop a servlet that illustrates how to use cookies. The servlet is invoked when a form on a web page is submitted. The example contains three files as summarized here:

**File : Description**

AddCookie.html : Allows a user to specify a value for the cookie named MyCookie.

AddCookieServlet.java : Processes the submission of AddCookie.html.

GetCookiesServlet.java : Displays cookie values.

The HTML source code for **AddCookie.html** is shown in the following listing. This page contains a text field in which a value can be entered. There is also a submit button on the page. When this button is pressed, the value in the text field is sent to **AddCookieServlet** via an HTTP POST request.

```
<html>

<body>

<center>

<form name="Form1" method="post"

action="http://localhost:8080/examples/servlets/servlet/AddCookieServlet"> <B>Enter a value for
MyCookie:</B>

<input type=textbox name="data" size=25 value=""> <input type=submit value="Submit">

</form>

</body>

</html>;
```

The source code for **AddCookieServlet.java** is shown in the following listing. It gets the value of the parameter named "data". It then creates a **Cookie** object that has the name "MyCookie" and contains the value of the "data" parameter. The cookie is then added to the header of the HTTP response via the **addCookie( )** method. A feedback message is then written to the browser.

```
import java.io.*; import javax.servlet.*;

import javax.servlet.http.*;

public class AddCookieServlet extends HttpServlet {

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

// Get parameter from HTTP request.

String data = request.getParameter("data");

// Create cookie.

Cookie cookie = new Cookie("MyCookie", data);

    Add cookie to HTTP response. response.addCookie(cookie);
    Write output to browser. response.setContentType("text/html"); PrintWriter pw =
response.getWriter(); pw.println("<B>MyCookie has been set to"); pw.println(data);
pw.close();

}

}
```

The source code for **GetCookiesServlet.java** is shown in the following listing. It invokes the **getCookies( )** method to read any cookies that are included in the HTTP GET request. The names and values of these cookies are then written to the HTTP response. Observe that the **getName( )** and **getValue( )** methods are called to obtain this information.

```
import java.io.*; import javax.servlet.*;

import javax.servlet.http.*;
```

```
public class GetCookiesServlet extends HttpServlet {
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    Get cookies from header of HTTP request. Cookie[] cookies = request.getCookies();
    Display these cookies. response.setContentType("text/html"); PrintWriter pw =
response.getWriter(); pw.println("<B>");
for(int i = 0; i < cookies.length; i++) { String name = cookies[i].getName(); String value =
cookies[i].getValue(); pw.println("name = " + name +
"; value = " + value);
}
pw.close();
 }
}
```

Compile the servlets. Next, copy them to the appropriate directory, and update the **web.xml** file, as previously described. Then, perform these steps to test this example:

Start Tomcat, if it is not already running.

Display **AddCookie.html** in a browser.

Enter a value for **MyCookie**.

Submit the web page.

After completing these steps, you will observe that a feedback message is displayed by the browser. Next, request the following URL via the browser:

http://localhost:8080/examples/servlets/servlet/GetCookiesServlet

Observe that the name and value of the cookie are displayed in the browser.

In this example, an expiration date is not explicitly assigned to the cookie via the **setMaxAge( )** method of **Cookie**. Therefore, the cookie expires when the browser session ends. You can experiment by using **setMaxAge( )** and observe that the cookie is then saved on the client machine.

## Session Tracking

HTTP is a stateless protocol. Each request is independent of the previous one. However, in some applications, it is necessary to save state information so that information can be collected from several interactions between a browser and a server. Sessions provide such a mechanism.

A session can be created via the **getSession( )** method of **HttpServletRequest**.
An **HttpSession** object is returned. This object can store a set of bindings that associate names with

objects. The **setAttribute( )**, **getAttributeNames( )**, and **removeAttribute( )** methods**)**, **getAttribute(**of **HttpSession** manage these bindings. Session state is shared by all servlets that are associated with a client.

The following servlet illustrates how to use session state. The **getSession()** method gets the current session. 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 "date". That object is a **Date** object that encapsulates the date and time when this page was last accessed. (Of course, there is no such binding when the page is first accessed.) A **Date** object encapsulating the current date and time is then created. The **setAttribute()** method is called to bind the name "date" to this object.

```
import java.io.*; import java.util.*; import javax.servlet.*;

import javax.servlet.http.*;

public class DateServlet extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

    Get the HttpSession object.
HttpSession hs = request.getSession(true);

    //Get writer.
    response.setContentType("text/html");
    PrintWriter pw = response.getWriter(); pw.print("<B>");


    //Display date/time of last access.


Date date = (Date)hs.getAttribute("date"); if(date != null) {

pw.print("Last access: " + date + "<br>");

}

// Display current date/time.

date = new Date(); hs.setAttribute("date", date);

pw.println("Current date: " + date);

}

}
```

When you first request this servlet, the browser displays one line with the current date and time information. On subsequent invocations, two lines are displayed. The first line shows the date and time when the servlet was last accessed. The second line shows the current date and time.

# JSP(JavaServer Pages)

JavaServer Pages (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.

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.

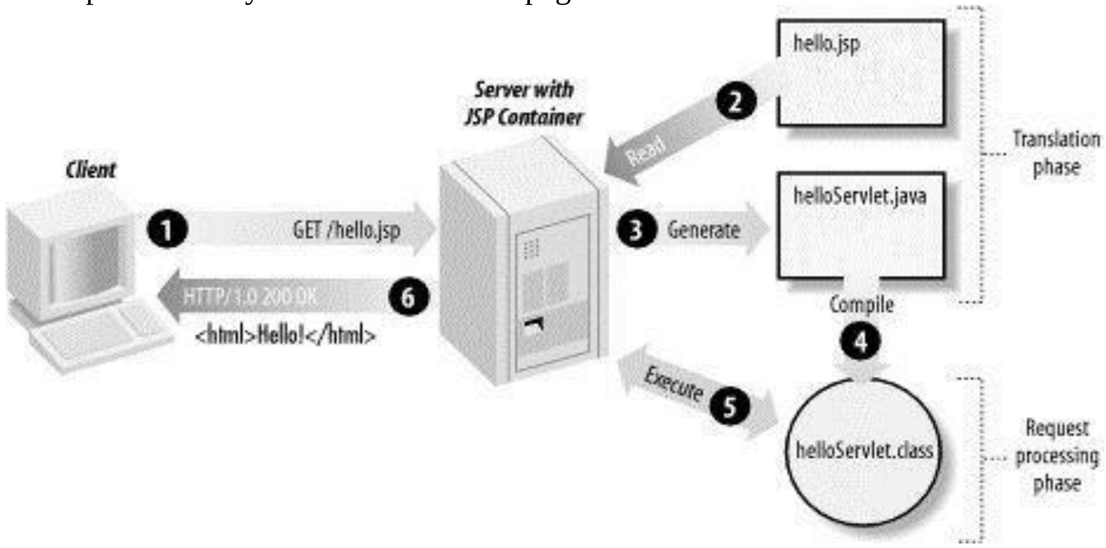
## Advantages of JSP:

**vs. Pure Servlets:** It is more convenient to write (and to modify!) regular HTML than to have plenty of println statements that generate the HTML.

## JSP Processing:

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

- 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.
- The JSP engine loads the JSP page from disk and converts it into a 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.
- The JSP engine compiles the servlet into an executable class and forwards the original request to a servlet engine.
- 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.
- The web server forwards the HTTP response to your browser in terms of static HTML content.
- Finally web browser handles the dynamically generated HTML page inside the HTTP response exactly as if it were a static page.

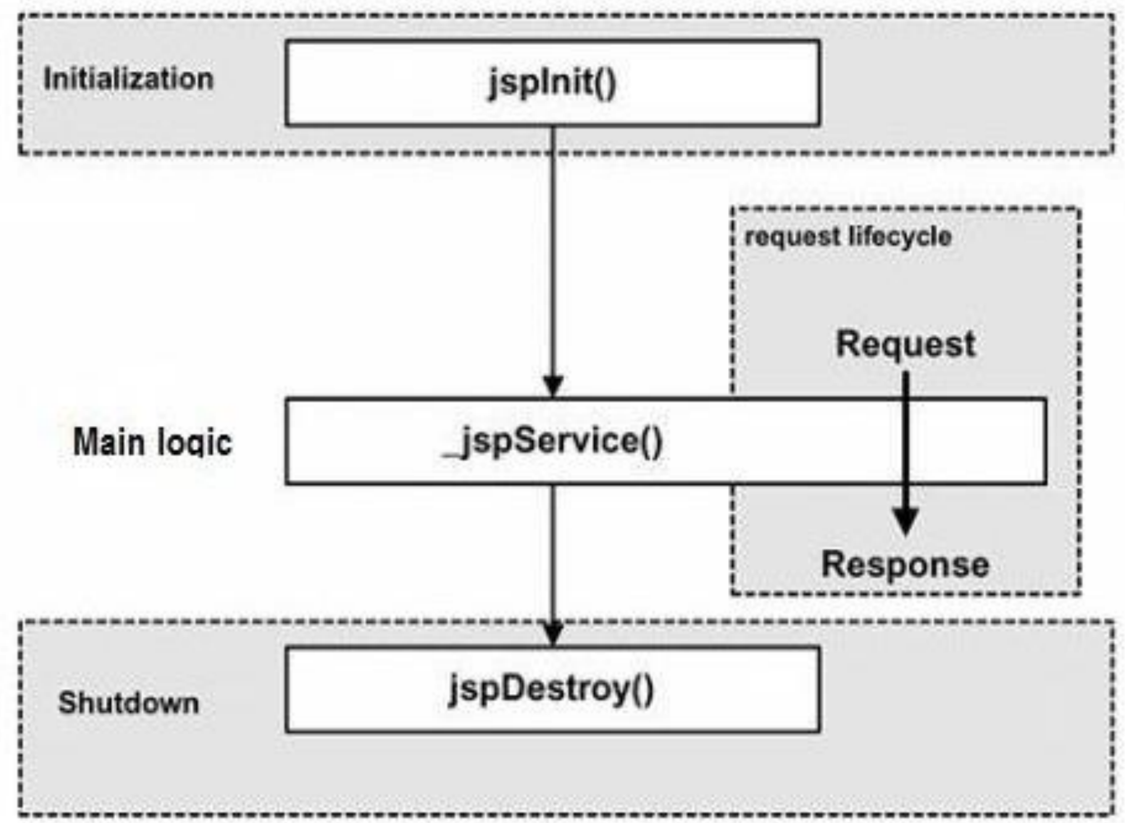


## LIFE CYCLE:

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

- Compilation
- Initialization
- Execution
- Cleanup



## 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:

- Parsing the JSP.
- Turning the JSP into a servlet.
- Compiling the servlet.

## 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.

## 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.

### 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 Elements :

1. **JSP Comments**
2. **JSP Scriptlets**
3. **JSP Expression**
4. **JSP Directives**
5. **JSP Declaration**

## 1. JSP Comments

Since JSP is built on top of HTML, we can write comments in JSP file like html comments as

```
<-- This is HTML Comment -->
```

These comments are sent to the client and we can look it with view source option of browsers.

We can put comments in JSP files as:

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

This comment is suitable for developers to provide code level comments because these are not sent in the client response.

## 2. JSP Scriptlets

Scriptlet tags are the easiest way to put java code in a JSP page. A scriptlet tag starts with `<%` and ends with `%>`.

Any code written inside the scriptlet tags go into the `_jspService()` method. For

example:

```
<%

Date d = new Date(); System.out.println("Current

Date="+d);

%>
```

## 3. JSP Expression

Since most of the times we print dynamic data in JSP page using *out.print()* method, there is a shortcut to do this through JSP Expressions. JSP Expression starts with `<%=` and ends with `%>`.

`<% out.print("Pankaj"); %>` can be written using JSP Expression as `<%= "Pankaj" %>`

Notice that anything between `<%= %>` is sent as parameter to `out.print()` method. Also notice that scriptlets can contain multiple java statements and always ends with semicolon (;) but expression doesn’t end with semicolon.

## 4. JSP Directives

JSP Directives are used to give special instructions to the container while JSP page is getting translated to servlet source code. JSP directives starts with `<%@` and ends with `%>`

For example, in above JSP Example, I am using *page* directive to to instruct container JSP translator to import the Date class.

## 5. JSP Declaration

JSP Declarations are used to declare member methods and variables of servlet class. JSP Declarations starts with `<%!` and ends with `%>`.

For example we can create an int variable in JSP at class level as `<%! public static int count=0; %>`

## JSP Implicit Objects:

JSP supports nine automatically defined variables, which are also called implicit objects. These variables are:

| Objects | 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. |

## JSP Directives:

A JSP directive affects the overall structure of the servlet class. It usually has the following form:

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

There are three types of directive tag:

| **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 |

# JSP Action Tags:

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. |

To give an example for a JSP code, first we are going to print the text "Hello Hiox". Try the the following syntax code.

***Example :***

```
<html>
<body>
<! -- This is the JSP file-->
<%
out.println ("Hello HIOX");
%>
</body>
</html>
```

**Result :**

```
Hello HIOX
```

## JSTL

# JSTL (JSP Standard Tag Library)

The JSP Standard Tag Library (JSTL) represents a set of tags to simplify the JSP development.

### Advantage of JSTL

1. **Fast Developement** JSTL provides many tags that simplifies the JSP.
2. **Code Reusability** We can use the JSTL tags in various pages.
3. **No need to use scriptlet tag** It avoids the use of scriptlet tag.

## JSTL Tags

There JSTL mainly provides 5 types of tags:

| Tag Name | Description |
|---|---|
| Core tags | The JSTL core tag provide variable support, URL management, flow control etc. The url for the core tag is **http://java.sun.com/jsp/jstl/core** . The prefix of core tag is **c**. |
| Function tags | The functions tags provide support for string manipulation and string length. The url for the functions tags is **http://java.sun.com/jsp/jstl/functions** and prefix is **fn**. |
| Formatting tags | The Formatting tags provide support for message formatting, number and date formatting etc. The url for the Formatting tags is **http://java.sun.com/jsp/jstl/fmt** and prefix is **fmt**. |
| XML tags | The xml sql tags provide flow control, transformation etc. The url for the xml tags is **http://java.sun.com/jsp/jstl/xml**and prefix is **x**. |
| SQL tags | The JSTL sql tags provide SQL support. The url for the sql tags is **http://java.sun.com/jsp/jstl/sql** and prefix is **sql**. |

## AJAX:

AJAX stands for **A**synchronous **Ja**vaScript and **X**ML. AJAX is a new technique for creating better, faster, and more interactive web applications with the help of XML, HTML, CSS, and Java Script.

- Ajax uses XHTML for content, CSS for presentation, along with Document Object Model and JavaScript for dynamic content display.

- Conventional web applications transmit information to and from the sever using synchronous requests. It means you fill out a form, hit submit, and get directed to a new page with new information from the server.

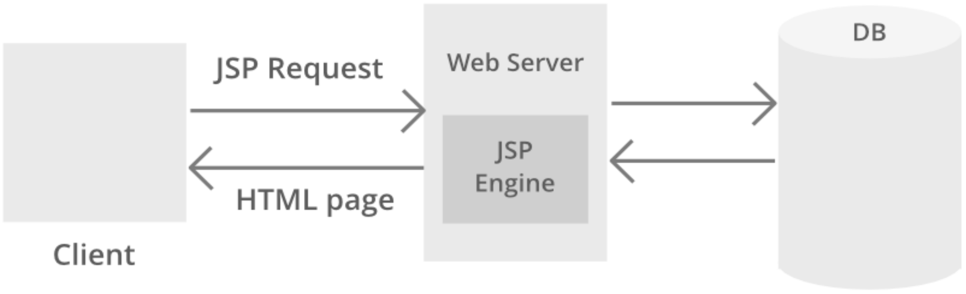
- With AJAX, when you hit submit, JavaScript will make a request to the server, interpret the results, and update the current screen. In the purest sense, the user would never know that anything was even transmitted to the server.

- XML is commonly used as the format for receiving server data, although any format, including plain text, can be used.

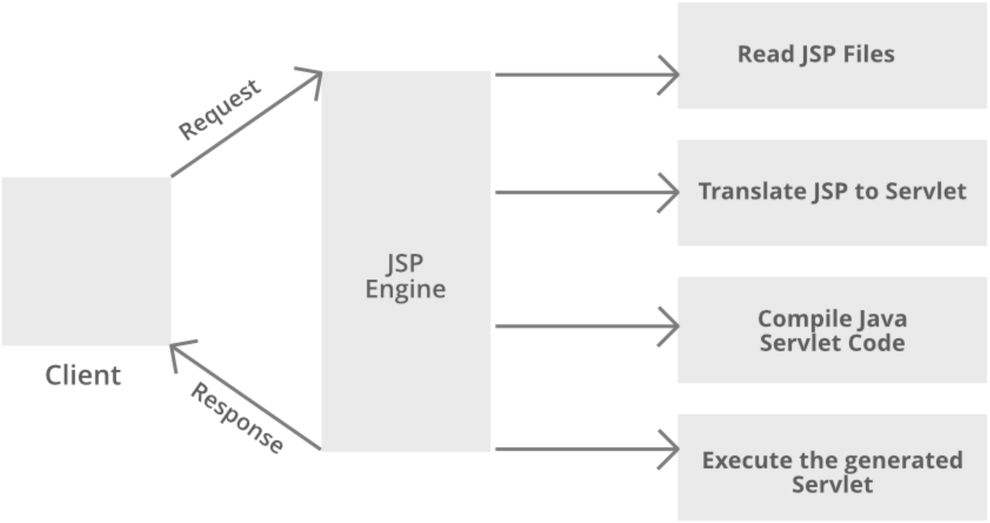
- AJAX is a web browser technology independent of web server software.
- A user can continue to use the application while the client program requests information from the server in the background.
- Intuitive and natural user interaction. Clicking is not required, mouse movement is a sufficient event trigger.
- Data-driven as opposed to page-driven.

## JSP Architecture

JSP architecture gives a high-level view of the working of JSP. JSP architecture is a 3 tier architecture. It has a Client, Web Server, and Database. The client is the web browser or application on the user side. Web Server uses a JSP Engine i.e; a container that processes JSP. For example, Apache Tomcat has a built-in JSP Engine. JSP Engine intercepts the request for JSP and provides the runtime environment for the understanding and processing of JSP files. It reads, parses, build Java Servlet, Compiles and Executes Java code, and returns the HTML page to the client. The webserver has access to the Database. The following diagram shows the architecture of JSP.



Now let us discuss JSP which stands for Java Server Pages. It is a server-side technology. It is used for creating web applications. It is used to create dynamic web content. In this JSP tags are used to insert JAVA code into HTML pages. It is an advanced version of Servlet Technology. It is a Web-based technology that helps us to create dynamic and platform-independent web pages. In this, Java code can be inserted in HTML/ XML pages or both. JSP is first converted into a servlet by JSP container before processing the client's request. **JSP Processing** is illustrated and discussed in sequential steps prior to which a pictorial media is provided as a handful pick to understand the JSP processing better which is as follows:



**Step 1:** The client navigates to a file ending with the ***.jsp extension*** and the browser initiates an HTTP request to the webserver. For example, the user enters the login details and submits the button. The browser requests a status.jsp page from the webserver.
**Step 2:** If the compiled version of JSP exists in the web server, it returns the file. Otherwise, the request is forwarded to the JSP Engine. This is done by recognizing the URL ending with **.jsp** extension.
**Step 3:** The JSP Engine loads the JSP file and translates the JSP to Servlet(Java code). This is done by converting all the template text into println() statements and JSP elements to Java code. This process is called **translation.**
**Step 4:** The JSP engine compiles the Servlet to an executable **.class** file. It is forwarded to the Servlet engine. This process is called **compilation** or **request processing phase.**
**Step 5:** The **.class** file is executed by the Servlet engine which is a part of the Web Server. The output is an HTML file. The Servlet engine passes the output as an HTTP response to the webserver.
**Step 6:** The web server forwards the HTML file to the client's browser.

## What is JSP?

In Java, **JSP** stands for **Java Server Pages**. It is a server-side technology which is used for creating web applications. It is used to create dynamic web content. JSP consists of both HTML tags and JSP tags. In this, JSP tags are used to insert JAVA code into HTML pages. It is an advanced version of **Servlet** Technology i.e. a web-based technology that helps us to create dynamic and platform-independent web pages. In this, Java code can be inserted in HTML/ XML pages or both. JSP is first converted into a servlet by the JSP container before processing the client's request. JSP has various features like JSP Expressions, JSP tags, JSP Expression Language, etc.

### How JSP more advantageous than Servlet?

- They are easy to maintain.
- No recompilation or redeployment is required.
- Less coding is required in JSP.
- JSP has access to the entire API of JAVA.
- JSP are extended version of Servlet.

### Features of JSP

- **Coding in JSP is easy** : As it is just adding JAVA code to HTML/XML.
- **Reduction in the length of Code** : In JSP we use action tags, custom tags etc.
- **Connection to Database is easier** : It is easier to connect website to database and allows to read or write data easily to the database.

- **Make Interactive websites** : In this we can create dynamic web pages which helps user to interact in real time environment.
- **Portable, Powerful, flexible and easy to maintain** : as these are browser and server independent.
- **No Redeployment and No Re-Compilation** : It is dynamic, secure and platform independent so no need to re-compilation.
- **Extension to Servlet** : as it has all features of servlets, implicit objects and custom tags
    1. **Declaration Tag** : It is used to declare variables.
    2. **Java Scriplets** : It allows us to add any number of JAVA code, variables and expressions.
    3. **JSP Expression** : It evaluates and convert the expression to a string.
    4. **JAVA Comments** : It contains the text that is added for information which has to be ignored.
        - Create html page from where request will be sent to server eg try.html.
        - To handle to request of user next is to create .jsp file Eg. new.jsp
        - Create project folder structure.
        - Create XML file eg my.xml.
        - Create WAR file.
        - Start Tomcat
        - Run Application
    5. It does not require advanced knowledge of JAVA
    6. It is capable of handling exceptions
    7. Easy to use and learn
    8. It contains tags which are easy to use and understand
    9. Implicit objects are there which reduces the length of code
    10. It is suitable for both JAVA and non JAVA programmer
    11. Difficult to debug for errors.
    12. First time access leads to wastage of time
    13. It’s output is HTML which lacks features.

## INTRODUCTION TO JAVA BEANS

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 JavaBean:**

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.

**Some Examples of Application Builder tools:**

| TOOL | VENDOR | DESCRIPTION |
|---|---|---|
| Java Workshop2.0 | Sun MicroSystems., Inc., | Complete IDE that support applet, application and bean development |
| Visual age for java | IBM | Bean Oriented visual development toolset. |
| Jbuilder | Borland Inc. | Suit of bean oriented java development tool |
| Beans Development Kit | SunMicroSystems., Inc., | Supports only Beans development |

## JavaBeans basic rules

A JavaBean should:

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

The classes and interfaces defined in the java.beans 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.

In addition Java Beans provides support for

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

## Elements of a JavaBean:

- **Properties**
  **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.**
- **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.
- **Events**
  **Similar to Swing/AWT event handling.**

## The JavaBean Component Specification:

Customization: Is the ability of JavaBean to allow its properties to be changed in build and execution phase.

**Persistence**:- Is the ability of JavaBean to save its state to disk or storage device and restore the saved state when the JavaBean is reloaded.

**Communication**:-Is the ability of JavaBean to notify change in its properties to other JavaBeans or the container.

**Introspection**:- 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

**Builder support**:- Enables you to create and group multiple JavaBeans in an application.

**Layout**:- Allows multiple JavaBeans to be arranged in a development environment.

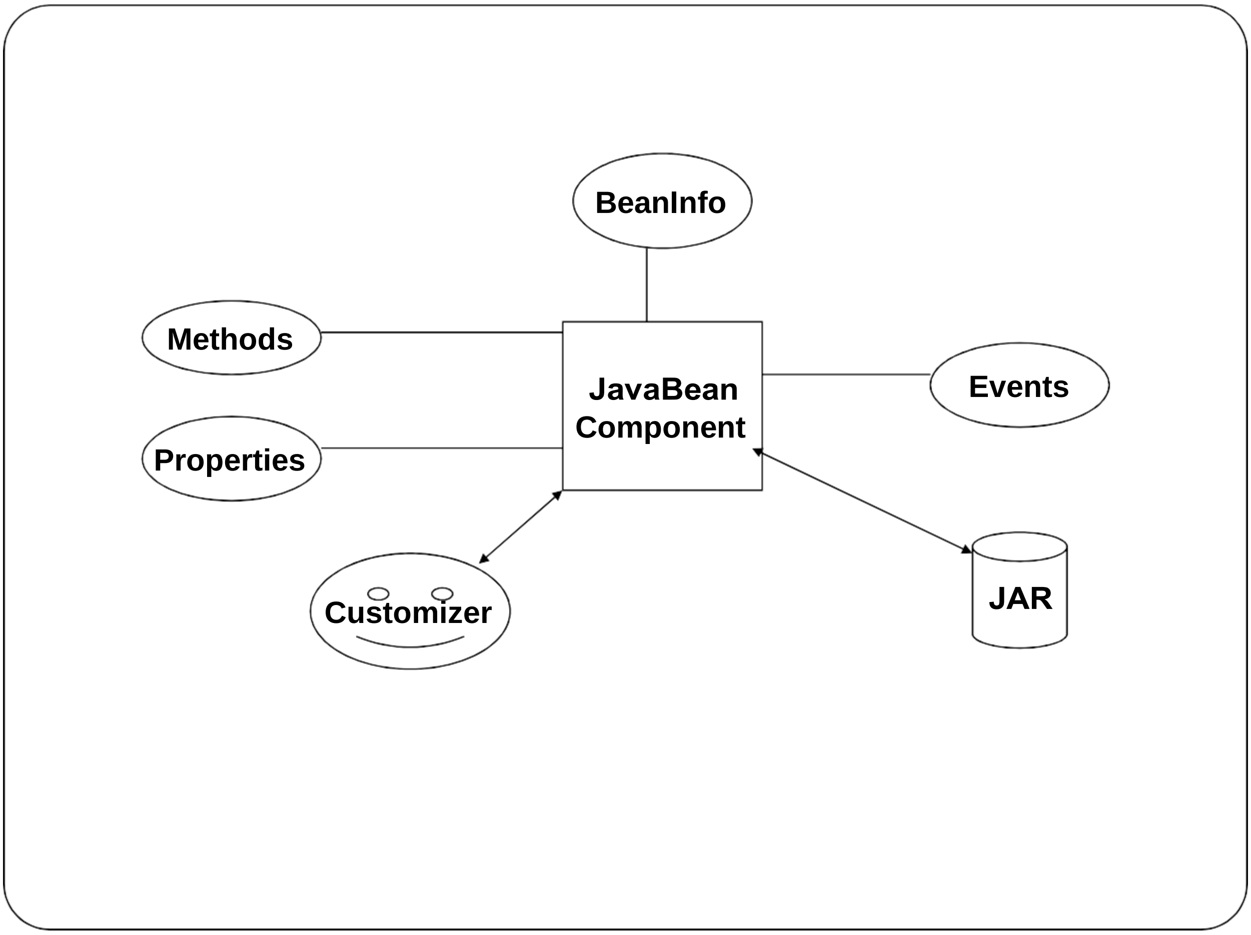
Interface publishing: Enables multiple JavaBeans in an application to communicate with each other.

**Event handling**:- Refers to firing and handling of events associated with a JavaBean.

**Persistence**:- 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 customisation of the appearance and behaviour 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.



**<u>Beans Development Kit</u>**

Is a development environment to create, configure, and test JavaBeans.

The features of BDK environment are:

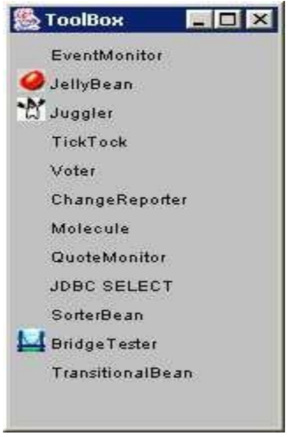
- Provides a GUI to create, configure, and test JavaBeans.
- Enables you to modify JavaBean properties and link multiple JavaBeans in an application using BDK.
- Provides a set of sample JavaBeans.
- Enables you to associate pre-defined events with sample JavaBeans.

Identifying BDK Components

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

**<u>ToolBox window</u>**: Lists the sample JavaBeans of BDK.

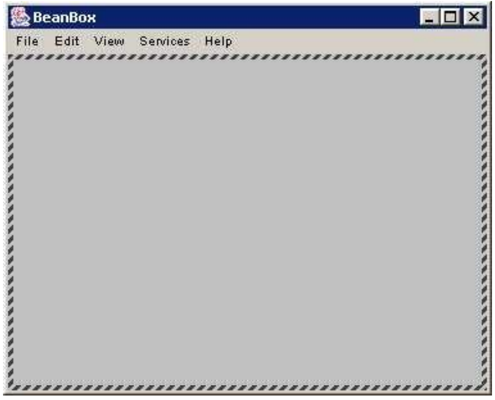
The following figure shows the **ToolBox** window:



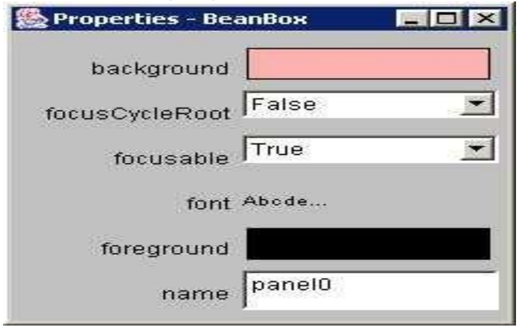
**<u>BeanBox window</u>**:

Is a workspace for creating the layout of JavaBean application.

- The following figure shows the **BeanBox** window:

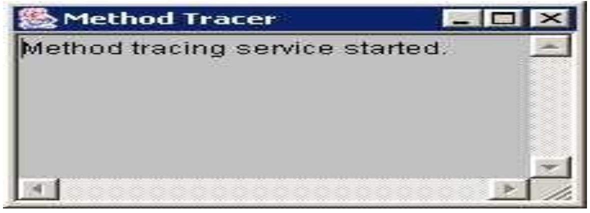


**Properties window**:
Displays all the exposed properties of a JavaBean. You can modify JavaBean properties in the properties window.
The following figure shows the **Properties** window:



background
focusCycleRoot False
focusable True
font Abcde...
foreground
name panel0

**Method Tracer window**:
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**

**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 lines code in the notepad and save that file as MyBean.mf**

**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(userdefined) 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.

**Introspection:**

- Introspection can be defined as the technique of obtaining information about bean properties, events and methods.
- Basically introspection means analysis of bean capabilities.
- Introspection is the automatic process by which a builder tool finds out which properties, methods, and events a bean supports.
- Introspection describes how methods, properties, and events are discovered in the beans that you write.
- This process controls the publishing and discovery of bean operations and properties
- Without introspection, the JavaBeans technology could not operate.

**BDK Introspection:**

- Allows automatic analysis of a java beans component
- Enables a builder tool to analyze how a bean works.

(Or)

- A mechanism that allows classes to publish the operations and properties they support and a mechanism to support the discovery of such mechanism.
- Introspection can be defined as the technique of obtaining information about bean properties, events and methods.
- Basically introspection means analysis of bean capabilities.

There are two ways in which the developer of a Bean can indicate which of its properties, events, and methods should be exposed by an builder tool. With the first method, simple naming conventions are used. These allow the introspection mechanisms to infer information about a Bean. In the second way, an additional class is provided that explicitly supplies this information.

**Design patterns for JavaBean Properties:-**

A property is a subset of a Bean’s state.

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.

Properties are the private data members of the JavaBean classes.

Properties are used to accept input from an end user in order to customize a JavaBean.

Properties can retrieve and specify the values of various attributes, which determine the behavior of a JavaBean.

Types of JavaBeans Properties

- Simple properties
- Boolean properties
- Indexed properties

**Simple Properties:**

Simple properties refer to the private variables of a JavaBean that can have only a single value.
Simple properties are retrieved and specified using the get and set methods respectively.

- A read/write property has both of these methods to access its values. The **get method** used to read the value of the property .The **set method** that sets the value of the property.

- The setXXX() and getXXX() methods are the heart of the java beans properties mechanism. This is also called getters and setters. These accessor methods are used to set the property .

The syntax of get method is:

**public return_type get<PropertyName>()**

**public T getN();**

**public void setN(T arg)**

- N is the name of the property and T is its type

**Ex:**

```
public double getDepth()
{
return depth;
}
```

Read only property has only a get method.
The syntax of set method is:

**public void set<PropertyName>(data_type value)**

**Ex:**

```
public void setDepth(double d)
{
Depth=d;
}
```

Write only property has only a set method.

## Boolean Properties:

- A Boolean property is a property which is used to represent the values True or False.

Have either of the two values, TRUE or FALSE.
It can identified by the following methods:
Syntax:

Let N be the name of the property and T be the type of the value then

```
public boolean isN();
public void setN(boolean parameter);
public Boolean getN();
```

- public boolean is<PropertyName>()
- public boolean get<PropertyName>()

First or second pattern can be used to retrieve the value of a Boolean.

- public void set<PropertyName>(boolean value)

For getting the values isN() and getN() methods are used and for setting the Boolean values setN() method is used.

**Ex**:

```
public boolean dotted=false;
public boolean isDotted()
{
return dotted;
}
public void setDotted(boolean dotted)
{
this.dotted=dotted;
}
```

## Indexed Properties:

Indexed Properties are consists of multiple values.
If a simple property can hold an array of value they are no longer called simple but instead indexed properties. The method’s signature has to be adapted accordingly. An indexed property may expose set/get methods to read/write one element in the array (so-called ’index getter/setter’) and/or so-called ’array getter/setter’ which read/write the entire array.

Indexed Properties enable you to set or retrieve the values from an array of property values.
Indexed Properties are retrieved using the following get methods:
**Syntax:**public int[] get<PropertyName>()
**Ex:**

```
private double data[];
 public double getData(int index)
   {
      return data[index];
      }
```

**Syntax:**public property_datatype get<PropertyName>(int index)
**Ex:**

```
public void setData(int index,double value)
{
Data[index]=value;
}
```

Indexed Properties are specified using the following set methods:
**Syntax**:

```
public void set<PropertyName>(int index, property_datatype value)
```

**EX**:

```
public double[] getData()
{
return data;
}
```

**Syntax** :

```
public void set<PropertyName>(property_datatype[] property_array)
```

**Ex**:

```
 public void setData(double[] values)
{
}
```

The properties window of BDK does not handle indexed properties. Hence the output can not be displayed here.

**Bound Properties**:

A bean that has a bound property generates an event when the property is changed.
Bound Properties are the properties of a JavaBean that inform its listeners about changes in its values.
Bound Properties are implemented using the **PropertyChangeSupport** class and its methods.
Bound Properties are always registered with an external event listener.
The event is of type ***PropertyChangeEvent*** and is sent to objects that previously egistered an interest in receiving such notifications
bean with bound property - Event source
Bean implementing listener -- event target

In order to provide this notification service a JavaBean needs to have the following two methods:

```
public void addPropertyChangeListener(PropertyChangeListener p) {
            changes.addPropertyChangeListener(p);
          }
          public void  removePropertyChangeListener(PropertyChangeListener p) {
            changes.removePropertyChangeListener(p);
          }
```

PropertyChangeListener is an interface declared in the java.beans package. Observers which want to be notified of property changes have to implement this interface, which consists of only one method:

```
public interface PropertyChangeListener extends EventListener {
public void propertyChange(PropertyChangeEvent e );
          }
```

## Constrained Properties:

It generates an event when an attempt is made to change it value
Constrained Properties are implemented using the ***PropertyChangeEvent*** class.
The event is sent to objects that previously registered an interest in receiving an such notification
Those other objects have the ability to veto the proposed change
This allows a bean to operate differently according to the runtime environment

A bean property for which a change to the property results in validation by another bean. The other bean may reject the change if it is not appropriate.
Constrained Properties are the properties that are protected from being changed by other JavaBeans.
Constrained Properties are registered with an external event listener that has the ability to either accept or reject the change in the value of a constrained property.
Constrained Properties can be retrieved using the get method. The prototype of the get method is:
**Syntax:**public string get<ConstrainedPropertyName>()
Can be specified using the set method. The prototype of the set method is:
**Syntax** :public string set<ConstrainedPropertyName>(String str)throws PropertyVetoException

# Design Patterns for Events:

**Handling Events in JavaBeans:**
Enables Beans to communicate and connect together.
Beans generate events and these events can be sent to other objects.
Event means any activity that interrupts the current ongoing activity.
**Example: mouse clicks, pressing key…**

User-defined JavaBeans interact with the help of user-defined events, which are also called custom events.
You can use the Java event delegation model to handle these custom events.
The components of the event delegation model are:

- **Event Source**: Generates the event and informs all the event listeners that are registered with it.
- **Event Listener**: Receives the notification, when an event source generates an event.
- **Event Object**: Represents the various types of events that can be generated by the event sources.

**Creating Custom Events:**

The classes and interfaces that you need to define to create the custom JavaBean events are:

- An event class to define a custom JavaBean event.
- An event listener interface for the custom JavaBean event.
- An event handler to process the custom JavaBean event.
- A target Java application that implements the custom event.

**Creating the Event Class:**

The event class that defines the custom event extends the EventObject class of the java.util package. For example,

```
        public class NumberEvent extends EventObject
        {
                public int number1,number2;
                public NumberEvent(Object o,int number1,int number2)
                {
                        super(o);
                        this.number1=number1;
                        this.number2=number2;
                }
        }
```

Beans can generate events and send them toother objects.

**Creating Event Listeners**

- When the event source triggers an event, it sends a notification to the event listener interface.
- The event listener interface implements the java.util.EventListener interface.
- Syntax:
- public void addTListener(TListener eventListener);
- public void addTListener(TListener eventListener)throws TooManyListeners;
- public void removeTListener(TListener eventListener);

- The target application that uses the custom event implements the custom listener. For example,

```
public interface NumberEnteredListener extends EventListener
{
public void arithmeticPerformed(NumberEvent mec);
}
```

**Creating Event Handler**

Custom event handlers should define the following methods:

- addXXListener(): Registers listeners of a JavaBean event.
- fireXX(): Notifies the listeners of the occurrence of a JavaBean event.
- removeXXListener(): Removes a listener from the list of registered listeners of a JavaBean.

**The code snippet to define an event handler for the custom event NumberEvent is:**

```
public class NumberBean extends JPanel implements ActionListener
{
        public NumberBean()
        {}
        NumberEnteredListener mel;
        public void addNumberListener(NumberEnteredListener mel)
        {
                this.mel = mel;
        }
}
```

## Persistence

Persistence means an ability to save properties and events of our beans to non-volatile storage and retrieve later.
It has the ability to save a bean to storage and retrieve it at a later time Configuration settings are saved It is implemented by Java serialization.
If a bean inherits directly or indirectly from Component class it is automatically Serializable.
Transient keyword can be used to designate data members of a Bean that should not be serialized.

- Enables developers to customize Beans in an application builder, and then retrieve those Beans, with customized features intact, for future use,perhaps in another environment.
- Java Beans supports two forms of persistence:
    - Automatic persistence
    - External persistence

**Automatic Persistence:**
Automatic persistence are java’s built-in serialization mechanism to save and restore the state of a bean.

**External Persistence:**
External persistence, on the other hand, gives you the option of supplying your own custom classes to control precisely how a bean state is stored and retrieved.

- Juggle Bean.
- Building an applet
- Your own bean

## Customizers:

The Properties window of the BDK allows a developer to modify the several properties of the Bean.
Property sheet may not be the best user interface for a complex component
It can provide step-by-step wizard guide to use component
It can provide a GUI frame with image which visually tells what is changed such as radio button, check box, ...
It can customize the appearance and behavior of the properties
Online documentation can also be provided. A Bean developer has great flexibility to develop a customizer that can differentiate his or her product in the marketplace.
To make it easy to configure a java beans component
Enables a developer to use an application builder tool to customize the appearance and behavior of a bean

# The Java Beans API:

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package. Table 25-2 lists the interfaces in **java.beans** and provides a brief description of their functionality. Table 25-3 lists the classes in **java.beans**.

Set of classes and interfaces in the java.beans package

### Package java.beans

Contains classes related to developing *beans* -- components based on the JavaBeans$^{TM}$ architecture

| Interface Summary | |
|---|---|
| AppletInitializer | This interface is designed to work in collusion with java.beans.Beans.instantiate. |
| BeanInfo | A bean implementor who wishes to provide explicit information about their bean may provide a BeanInfo class that implements this BeanInfo interface and provides explicit information about the methods, properties, events, etc, of their bean. |
| Customizer | A customizer class provides a complete custom GUI for customizing a target Java Bean. |
| DesignMode | This interface is intended to be implemented by, or delegated from, instances of java.beans.beancontext.BeanContext, in order to propagate to its nested hierarchy of java.beans.beancontext.BeanContextChild instances, the current "designTime" property. |
| ExceptionListener | An ExceptionListener is notified of internal exceptions. |
| PropertyChangeListener | A "PropertyChange" event gets fired whenever a bean changes a "bound" property. |
| PropertyEditor | A PropertyEditor class provides support for GUIs that want to allow users to edit a property value of a given type. |
| VetoableChangeListener | A VetoableChange event gets fired whenever a bean changes a "constrained" property. |
| Visibility | Under some circumstances a bean may be run on servers where a GUI is not available. |

| Class Summary | |
|---|---|
| BeanDescriptor | A BeanDescriptor provides global information about a "bean", including its Java class, its displayName, etc. |
| Beans | This class provides some general purpose beans control methods. |
| DefaultPersistenceDelegate | The `DefaultPersistenceDelegate` is a concrete implementation of the abstract `PersistenceDelegate` class and is the delegate used by default for classes about which no information is available. |
| Encoder | An `Encoder` is a class which can be used to create files or streams that encode the state of a collection of JavaBeans in terms of their public APIs. |
| EventHandler | The `EventHandler` class provides support for dynamically generating event listeners whose methods execute a simple statement involving an incoming event object and a target object. |
| EventSetDescriptor | An EventSetDescriptor describes a group of events that a given Java bean fires. |
| Expression | An `Expression` object represents a primitive expression in which a single method is applied to a target and a set of arguments to return a result - as in `"a.getFoo()"`. |

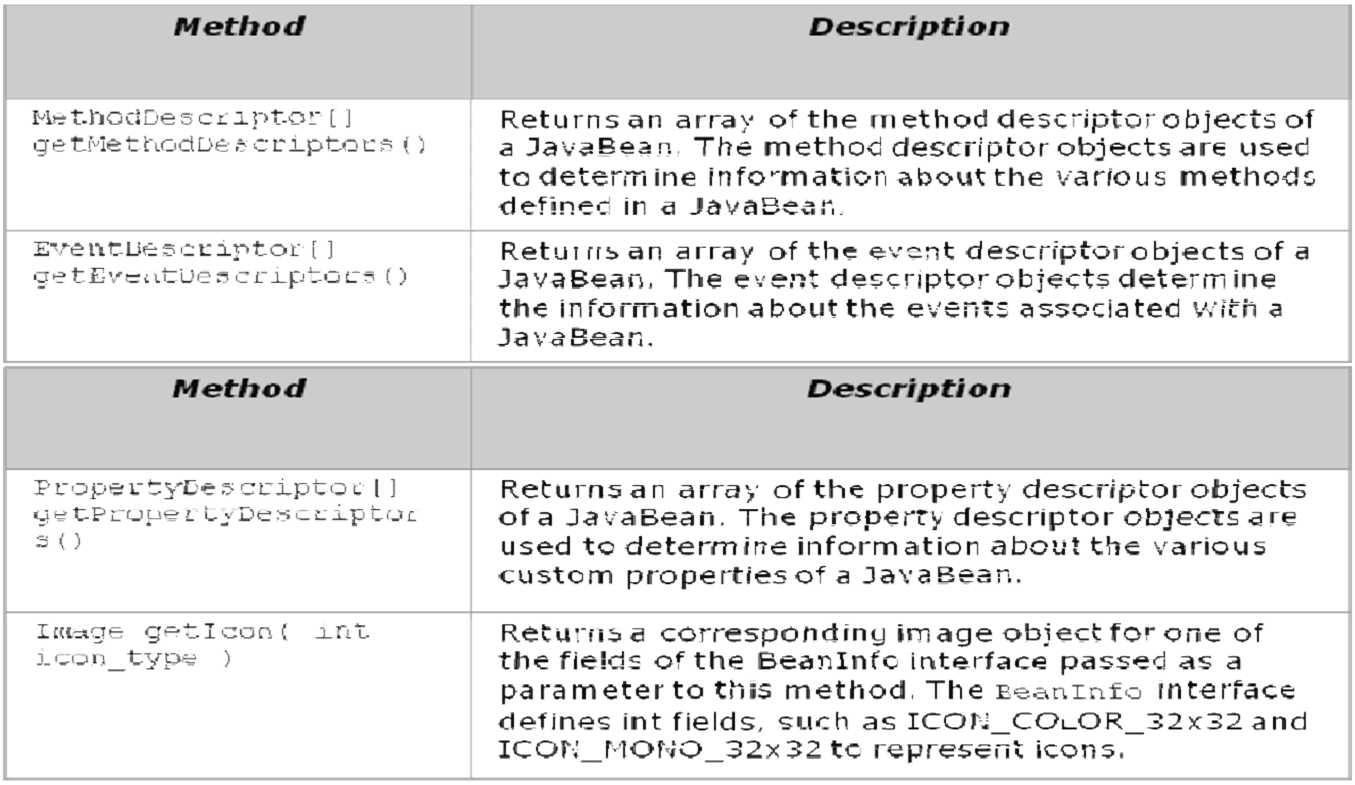
| | |
|---|---|
| FeatureDescriptor | The FeatureDescriptor class is the common baseclass for PropertyDescriptor, EventSetDescriptor, and MethodDescriptor, etc. |
| IndexedPropertyChangeEvent | An "IndexedPropertyChange" event gets delivered whenever a component that conforms to the JavaBeans specification (a "bean") changes a bound indexed property. |
| IndexedPropertyDescriptor | An IndexedPropertyDescriptor describes a property that acts like an array and has an indexed read and/or indexed write method to access specific elements of the array. |
| Introspector | The Introspector class provides a standard way for tools to learn about the properties, events, and methods supported by a target Java Bean. |
| MethodDescriptor | A MethodDescriptor describes a particular method that a Java Bean supports for external access from other components. |
| ParameterDescriptor | The ParameterDescriptor class allows bean implementors to provide additional information on each of their parameters, beyond the low level type information provided by the java.lang.reflect.Method class. |
| PersistenceDelegate | The PersistenceDelegate class takes the responsibility for expressing the state of an instance of a given class in terms of the methods in the class's public API. |
| PropertyChangeEvent | A "PropertyChange" event gets delivered whenever a bean changes a "bound" or "constrained" property. |
| PropertyChangeListenerProxy | A class which extends the `EventListenerProxy` specifically for adding a named `PropertyChangeListener`. |
| PropertyChangeSupport | This is a utility class that can be used by beans that support bound properties. |
| PropertyDescriptor | A PropertyDescriptor describes one property that a Java Bean exports via a pair of accessor methods. |
| PropertyEditorManager | The PropertyEditorManager can be used to locate a property editor for any given type name. |
| PropertyEditorSupport | This is a support class to help build property editors. |
| SimpleBeanInfo | This is a support class to make it easier for people to provide BeanInfo classes. |
| Statement | A `Statement` object represents a primitive statement in which a single method is applied to a target and a set of arguments - as in `"a.setFoo(b)"`. |
| VetoableChangeListenerProxy | A class which extends the `EventListenerProxy` specifically for associating a `VetoableChangeListener` with a "constrained" property. |
| VetoableChangeSupport | This is a utility class that can be used by beans that support constrained properties. |
| XMLDecoder | The `XMLDecoder` class is used to read XML documents created using the `XMLEncoder` and is used just like the `ObjectInputStream`. |
| XMLEncoder | The `XMLEncoder` class is a complementary alternative to the `ObjectOutputStream` and can used to generate a textual representation of a *JavaBean* in the same way that the `ObjectOutputStream` can be used to create binary representation of `Serializable` objects. |

## **The** BeanInfo **Interface :**

By default an Introspector uses the Reflection API to determine the features of a JavaBean.However, a JavaBean can provide its own BeanInfo which will be used instead by the Introspector to determine the discussed information. This allows a developer hiding specific properties, events and methods from a builder tool or from any other tool which uses the Introspector class. Moreover it allows supplying further details about events/properties/methods as you are in charge of creating the descriptor objects. Hence you can, for example, call the setShortDescription() method to set a descriptive description. A BeanInfo class has to be derived from the SimpleBeanInfo class and its name has to start with the name of the associated JavaBean. At this point it has to be underlined that the name of the BeanInfo class is the only relation between a JavaBean and its BeanInfo class.

The BeanInfo interface provides the methods that enable you to specify and retrieve the information about a JavaBean.

The following table lists some of the methods of BeanInfo interface:



| ***Method*** | ***Description*** |
|---|---|
| `MethodDescriptor[] getMethodDescriptors()` | Returns an array of the method descriptor objects of a JavaBean. The method descriptor objects are used to determine information about the various methods defined in a JavaBean. |
| `EventDescriptor[] getEventDescriptors()` | Returns an array of the event descriptor objects of a JavaBean. The event descriptor objects determine the information about the events associated with a JavaBean. |
| `PropertyDescriptor[] getPropertyDescriptors()` | Returns an array of the property descriptor objects of a JavaBean. The property descriptor objects are used to determine information about the various custom properties of a JavaBean. |
| `Image getIcon( int icon_type )` | Returns a corresponding image object for one of the fields of the BeanInfo interface passed as a parameter to this method. The `BeanInfo` interface defines int fields, such as ICON_COLOR_32x32 and ICON_MONO_32x32 to represent icons. |

# The BeanBox Menus

This section explains each item in the BeanBox File, Edit, and View menus.

## File Menu

### Save

Saves the Beans in the BeanBox, including each Bean’s size, position, and internal state. The saved file can be loaded via File|Load.

### SerializeComponent...

Saves the Beans in the BeanBox to a serialized (*.ser*) file. This file must be put in a `.jar` file to be useable.

### MakeApplet...

Generates an applet from the BeanBox contents.

### Load...

Loads Saved files into the BeanBox. Will not load `.ser` files.

### LoadJar...

Loads a Jar file’s contents into the ToolBox.

### Print

Prints the BeanBox contents.

### Clear

Removes the BeanBox contents.

### Exit

Quits the BeanBox *without offering to save*.

## Edit Menu

### Cut

Removes the Bean selected in the BeanBox. The cut Bean is serialized, and can then be pasted.

### Copy

Copies the Bean selected in the BeanBox. The copied Bean is serialized, and can then be pasted.

### Paste

Drops the last cut or copied Bean into the BeanBox.

### Report...

Generates an introspection report on the selected Bean.

### Events

Lists the event−firing methods, grouped by interface.

### Bind property...

Lists all the bound property methods of the selected Bean.

## View Menu

### Disable Design Mode

Removes the ToolBox and the Properties sheet from the screen. Eliminates all beanBox design and test behavior (selected Bean, etc.), and makes the BeanBox behave like an application.

### Hide Invisible Beans

Hides invisible Beans.