Web Component Development Using Java

Session: 4

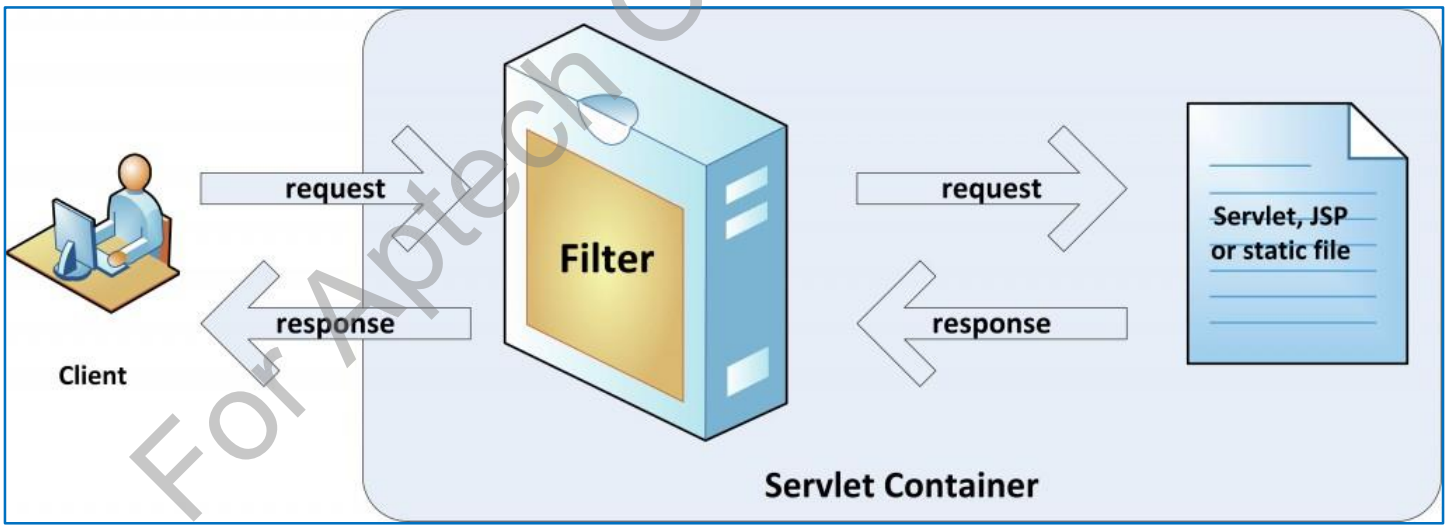
Filters and Annotations

Objectives

- Explain the concept and working of filters
- List the benefits of filters
- Explain the Filter API interfaces and there methods
- Explain the use of wrapper classes in managing Servlets
- Explain how to alter a request and response using filters
- Describe the basic concept of annotations
- Explain the different types of annotations supported in Servlet API
- Explain the steps to upload the file using HTML form elements
- Explain the mechanism to upload files on the server using Servlet

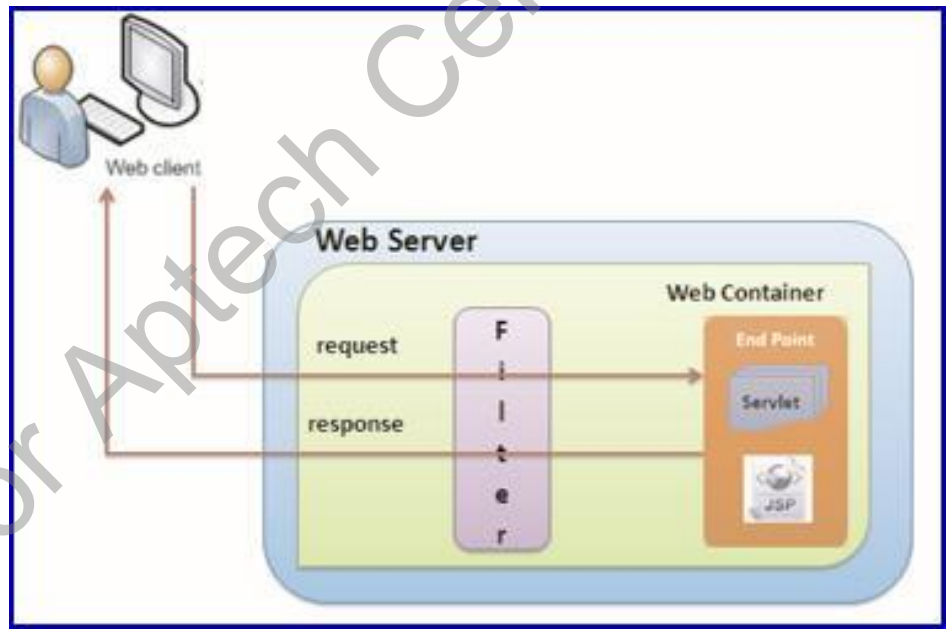
Introduction

- Java supports filter objects which performs processing of the request before it acquires the resource.
  - For example, data sent in the response from a Servlet must be encrypted to avoid any malfunction threat.
  - This needs an extra processing of encryption to be done on the received response from the Servlet.
- Before sending the response, filters perform manipulation of the responses sent to the client.
- Figure shows the processing of request and response done by filters.



Concept of Filter 1-2

- **Filters:**
  - Are Java classes.
  - Used to encapsulate the preprocessing and post processing functionalities common to a group of Servlets or JSP pages.
  - Were introduced as a Web component in Java Servlet specification.
  - Reside in the Web container along with the other Web components, such as Servlet or JSP pages.
- Figure depicts filters in a Web application.



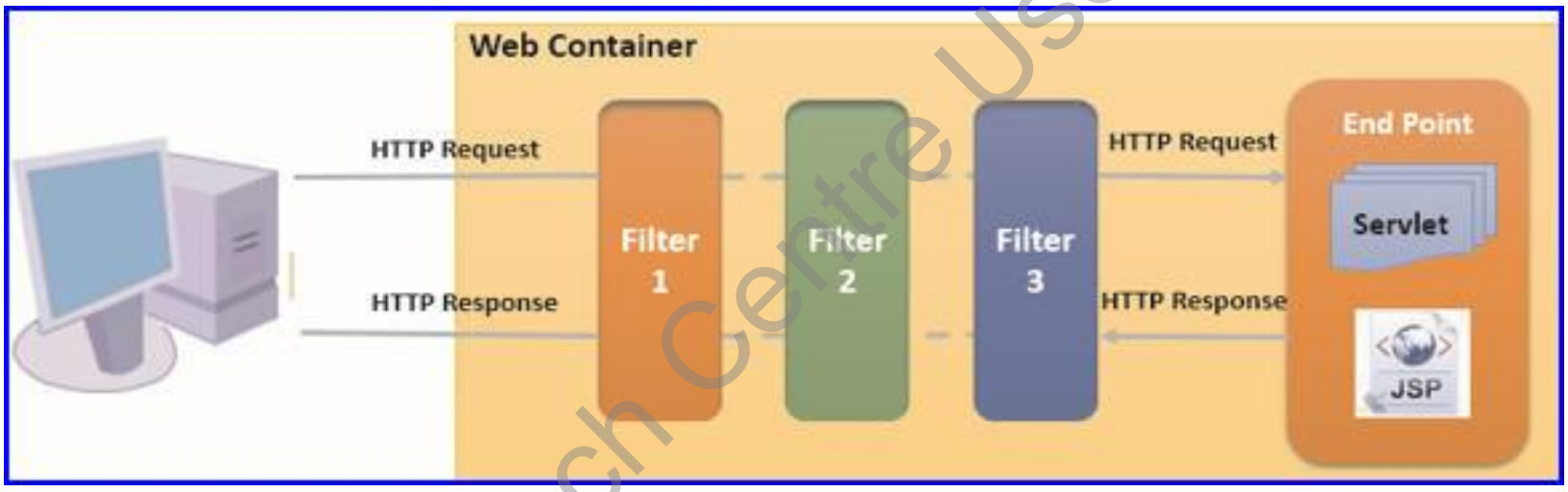
Concept of Filter 2-2

- Some of the important features provided by the filters to the Web applications are as follows:

  - Optimization of the time taken to send a response back to a client.
  - Compression of the content size sent from a Web server to a user over the Internet.
  - Conversion of images, texts, and videos to the required format.
  - Optimization of the available bandwidth on the network.
  - Authentication of the users in the secured Web site.
  - Encryption of the request and response header for addressing security concerns.

Filter Channing

- There can be more than one filter between the client and the Web resources, thus forming a filter chain.
- Figure shows channing of the filters.



- A request or a response is passed through one filter to the next in the filter chain.
- Each request and response is serviced by filters configured in the chain, before the Servlet is called by the Web container and after the Servlet responds.

Working of Filters 1-2

- A typical filter operates in the following way:

The Web container instantiates the filters and loads them for preprocessing.

The filter intercepts the request from the user to the servlet.

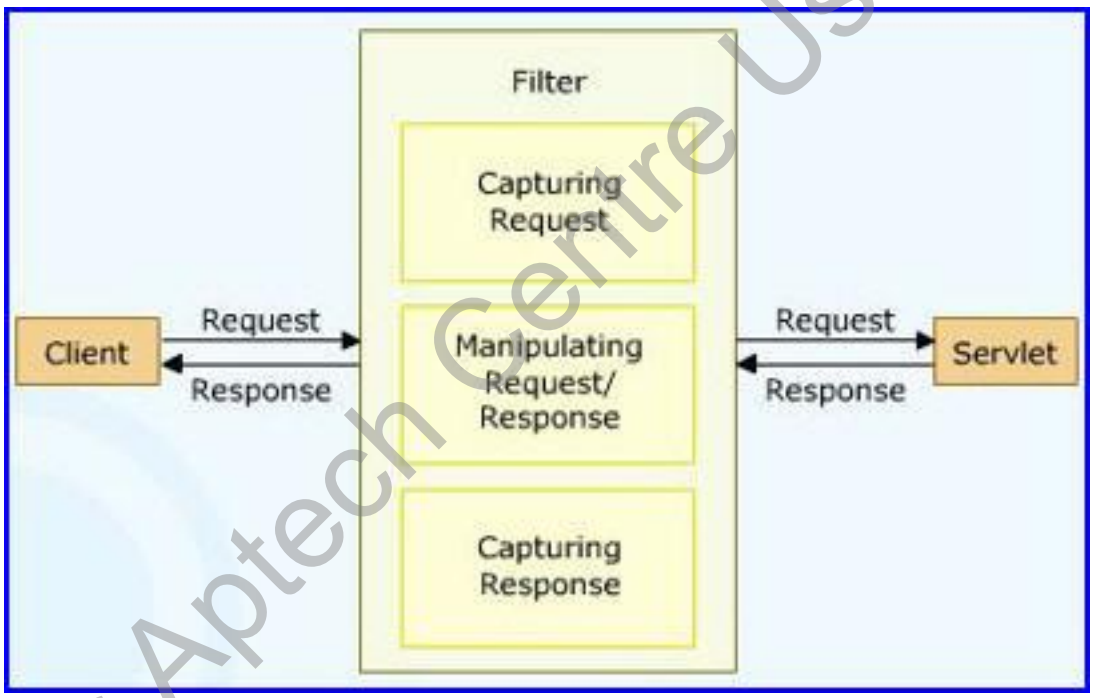
After the request is processed, the filter can do the following:

- Generate response and return to the client.
- Pass the modified or unmodified request to other filter in the chain.
- If the request is received by the last filter in the chain, it passes the request to the destination Servlet.
- It may route the request to a different resource rather than the associated Servlet.

Then, the filter sends the serviced request to the appropriate Servlet.

Working of Filters 2-2

- When the response is returned to the client, the response passes through same set of filters in reverse order.
- Figure depicts working of a typical filter.



Usage of Filters 1-2

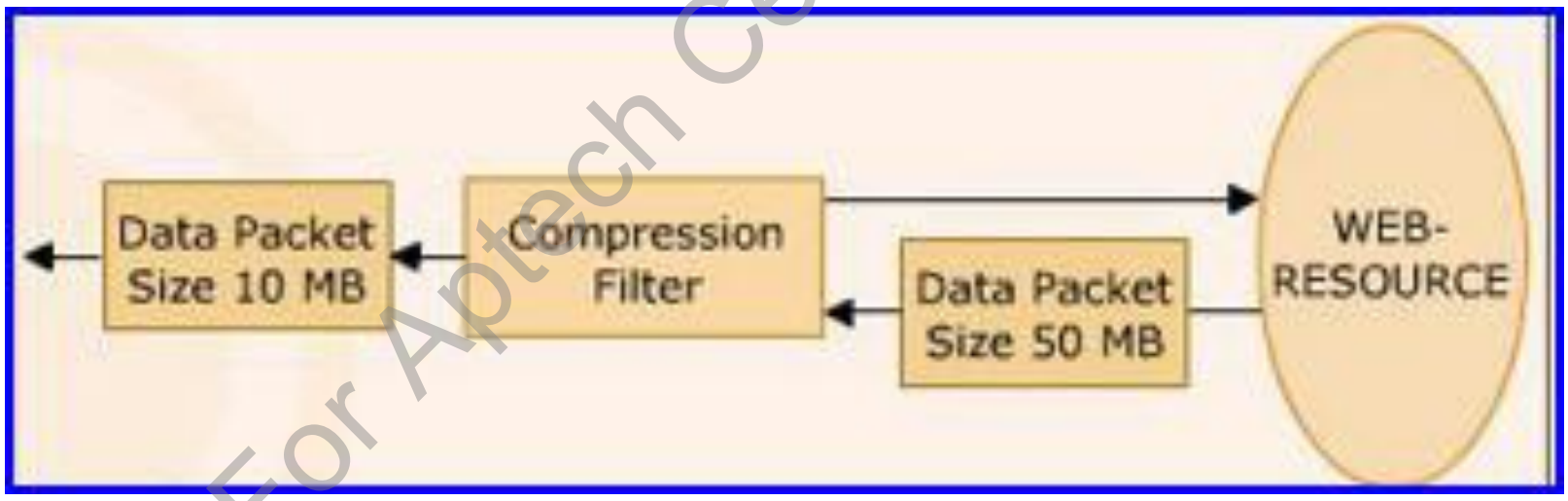
- Some of the filter categories are as follows:

Authentication filters	Allows the users to access any resource in secured Websites after providing proper username and password.
Logging and auditing filters	Tracks the activities of users on a Web application and log them.
Image conversion filters	Scales the image size or change the image type as per requirements.
Data compression filters	Helps in compressing uploaded or downloaded file size, thereby reducing the bandwidth requirement and time for downloading.
Encryption filters	Helps in encrypting the request and response header.

Usage of Filters 2-2

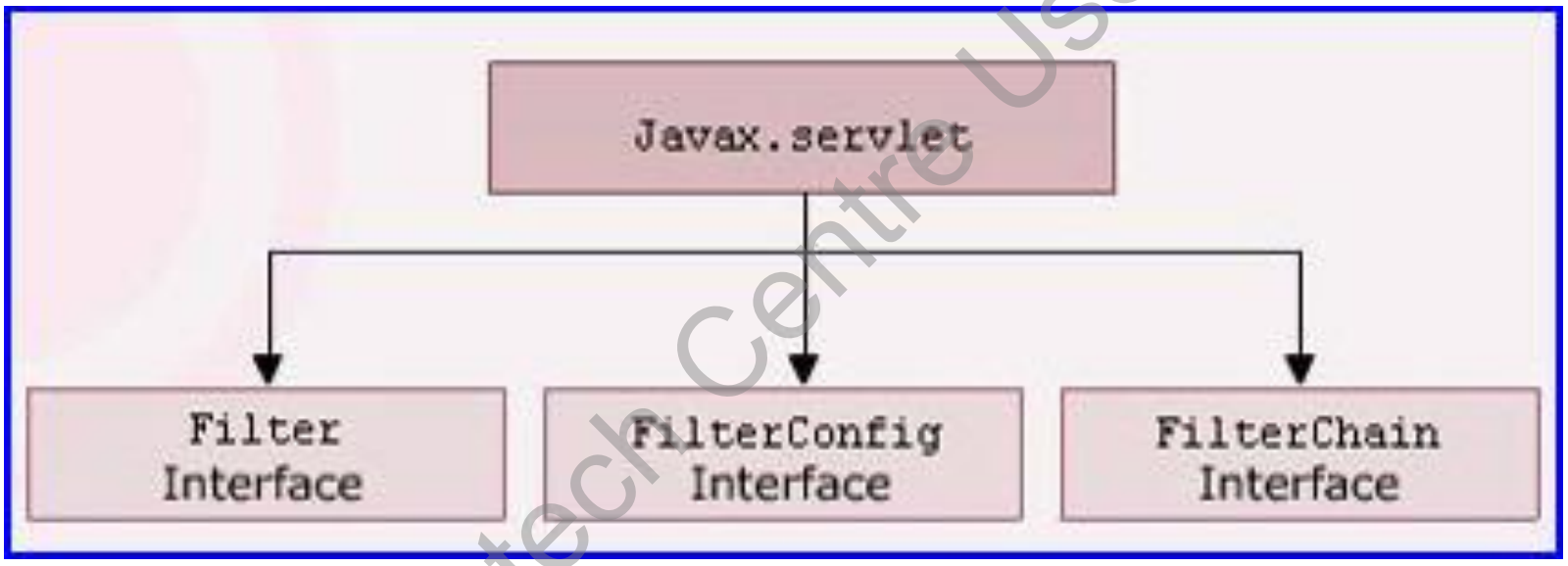
Filters that trigger resource access events	Helps in registering the particular database for a Web application before they retrieve or manipulate data according to the request made by the client.
XML transformation filters	Reads the XML data from the response and applies an XSLT transformation before sending it to the client browser.

- Figure depicts a compression filter.



Filter API

- It brings in all the required interfaces namely `Filter`, `FilterChain`, and `FilterConfig` for creating a filter class into the `javax.servlet` package.
- Figure depicts hierarchy structure of `Filter` API.



Filter Interface 1-4

- The `Filter` interface is part of the Servlet API.
- The `Filter` interface provides the life cycle methods of a filter. It must be implemented to create a filter class.
- The methods of the `Filter` interface are as follows:
  - **`init()`**
    - It is called by the servlet container to initialize the filter.
    - It is called only once.
    - The `init()` method must complete successfully before the filter is asked to do any filtering work.
    - **Syntax:**

```
public void init(FilterConfig filterConfig)

 throws ServletException
```

Filter Interface 2-4

- **`doFilter()`**
  - It is called by the container each time a request or response is processed.
  - It then examines the request or response headers and customizes them as per the requirements.
  - It also passes the request or response through the `FilterChain` object to the next entity in the chain.
  - **Syntax:**

```
public void doFilter(ServletRequest req,
ServletResponse res, FilterChain chain) throws
IOException, ServletException
```

- **`destroy()`**
  - It is called by the servlet container to inform the filter that its service is no more required.
  - It is called only once.
  - **Syntax:**

```
public void destroy()
```

Filter Interface 3-4

- Figure shows the template of a class implementing the `Filter` interface.

```
public class MyGenericFilter implements javax.servlet.Filter {

   public FilterConfig filterConfig;

  public void init(final FilterConfig filterConfig) {

            this. FilterConfig = filterConfig;
  }
  public void doFilter(final ServletRequest request,  final
     ServletResponse response, FilterChain chain) throws
     java.io.IOException, javax.servlet.ServletException {

            chain. DoFilter(request,response);
}

public void destroy() {  }

}
```

Filter Interface 4-4

- The code snippet demonstrates the implementation of filter to display a message before the invocation of the configured Servlet.

```
public class MessageFilter extends Filter {

private FilterConfig ;


public void doFilter(final ServletRequest request, final
ServletResponse response, FilterChain chain) throws
java.io.IOException, javax.servlet.ServletException {

    System.out.println("Invoking Filter");

    System.out.println("Preprocessing done by filter");


    chain.doFilter(request,response);

 }

}
```

Configuring Filter

- The configuration of a filter inside the `web.xml` file ensures that filter is the part of that application.
- In the deployment descriptor, filter is declared by the `<filter>` element.
- It defines a filter class and its initialization parameters.
- Following elements can be defined within a filter element:
  - `<display-name>`
  - `<description>`
  - `<filter-name>`
  - `<filter-class>`
  - `<init-param>`

Filter-mapping Element 1-2

- It specifies which filter to execute depending on a URL pattern or a Servlet name.
- This actually sets up a logical relation between the filter and the Web application for sequential control flow of request and response between them.
- Following elements are defined inside a `filter-mapping` element:

<filter-name>

- The element inside the deployment descriptor holds the name of the filter to which a URL pattern or a Servlet is mapped.
- It is a mandatory element for mapping a filter.

<url-pattern>

- The element describes a pattern used to resolve URLs.
- It is a mandatory element for mapping a filter.

<servlet-name>

- The element specifies the name of a servlet whose request and response will be modified by the filter.
- It is a mandatory element.

Filter-mapping Element 2-2

- The code snippet shows the mapping of `MessageFilter` with the `MyServlet` in the `web.xml` file.

```
<filter>
    <display-name>Filter</display-name>
    <description>This is my first filter
    </description>

        <filter-name>Message</filter-name>
    <filter-class>
                    servlet.filter.MessageFilter
  </filter-class>
</filter>

<filter-mapping>
    <filter-name>Message</filter-name>
    <servlet-name>MyServlet</servlet-name>
</filter-mapping>
. . .
```

Passing Initialization Parameters to Filter 1-3

- The `init()` method of a filter initializes the filter.
- It receives the object of `FilterConfig` object created by the Web container.
- Then, the initialization parameters are read in the `init()` using the methods of `FilterConfig`.
- Table lists some of the methods of `FilterConfig`.

Method	Description
`String getFilterName()`	Returns the name of the filter defined in the web.xml or deployment descriptor file in the Web application.
`String getInitParameter(String name)`	Returns the value of the named initialization parameter as a string. Returns null if the servlet has no attribute.
`Enumeration getInitParameterNames()`	Returns the names of the initialization parameter as an enumeration of String objects.
`ServletContext getServletContext()`	Returns the ServletContext object used by the caller to interact with its Servlet container and filter.

Passing Initialization Parameters to Filter 2-3

- The code snippet exhibits the use of the methods in the `FilterConfig` interface.

```
public class MessageFilter implements Filter {

private String message;
private FilterConfig config;

public void init(FilterConfig config) throws
ServletException
{
 this.config = config;
 message = config.getInitParameter("message");
}

public void doFilter(final ServletRequest request, final
ServletResponse response, FilterChain chain)
  throws
java.io.IOException,javax.servlet.ServletException {

 System.out.println("Entering MessageFilter");
 request.setAttribute("messageObj",message);
 chain.doFilter(request,response);
 System.out.println("Exiting MessageFilter");

 }
}
```

Passing Initialization Parameters to Filter 3-3

- Figure shows the configuration of filter with initialization parameters in the `web.xml` file.

```
<filter>
<filter-name>Message</filter-name>
 <filter-class>servlet.filter.MessageFilter</filter-class>

<!- Initialization parameters -->
  <init-param>
    <param-name>message</param-name>
    <param-value>Welcome to Filter</param-value>
  </init-param>

</filter>

<filter-mapping>
<filter-name>Message</filter-name>
<servlet-name>MyServlet</servlet-name>
 </filter-mapping>
```

Use of Wildcard in Mappings 1-2

- The `<url-element>` element in the deployment descriptor is used to specify URL pattern for the Web resources.
- It can either contain a specific URL or a even wild card character (*) can be used to match a set of URLs.
- All the URLs mapped with filters are applied before the Servlet are invoked.
- When a Web application is executed, it creates an instance of each filter that has been declared in the deployment descriptor.
- The sequence for execution of these filters is in the order, as they are declared in the deployment descriptor.

Use of Wildcard in Mappings 2-2

- The code snippet shows how to include multiple filters with a request to a mapped Servlet.

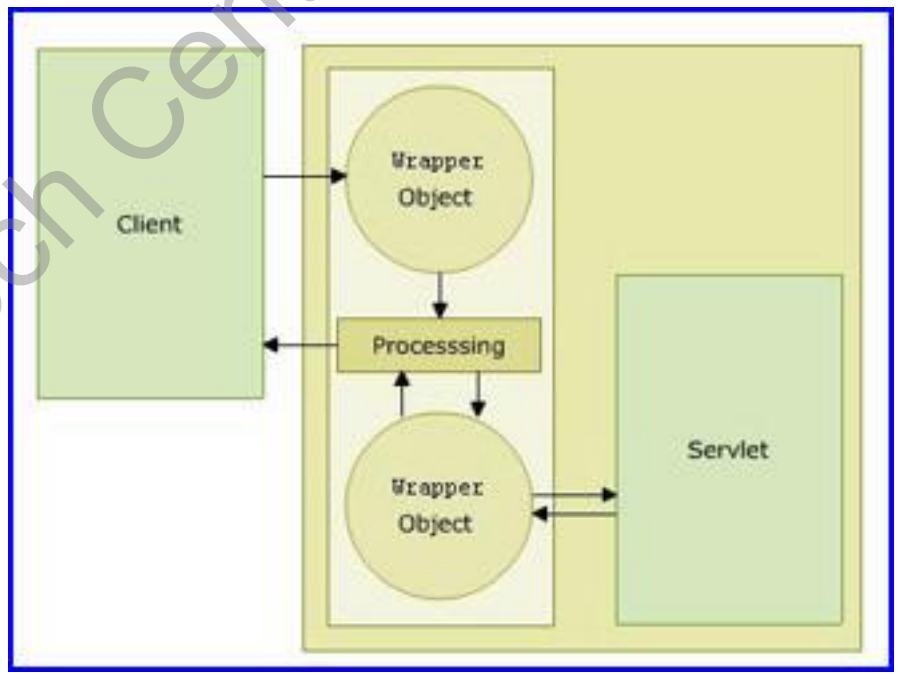
```
. . .
<filter-mapping>
    <filter-name>LogFilter</filter-name>
    <url-pattern>*.abc</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>AuditFilter</filter-name>
    <url-pattern>*.abc</url-pattern>
</filter-mapping>
. . .
```

- The code snippet shows the mapping for multiple patterns with the filter.

```
...
<filter-mapping>
    <filter-name>DispatchFilter</filter-name>
    <servlet-name>*</servlet-name>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
...
```

Manipulating Requests and Responses 1-4

- The filters can wrap the request or response objects in custom wrappers.
- The wrapper classes:
  - Helps to intercept the request or response, before they can reach their logical destination.
  - Create the object to capture the request and response, before they reach server and client respectively.
- Figure shows the working of wrapper objects used for processing of request and response in Servlets.



Manipulating Requests and Responses 2-4

- The classes defined by Servlet API for wrapping request and response are as follows:
  - **ServletRequestWrapper**
    - This class provides a convenient implementation of the `ServletRequest` interface.
  - **ServletResponseWrapper**
    - This class provides a convenient implementation of the `ServletResponse` interface.

Manipulating Requests and Responses 3-4

- The code snippet shows how to extend `HttpServletRequestWrapper` class.

```
class MyRequestWrapper extends HttpServletRequestWrapper
{
 ...
  public MyRequestWrapper(HttpServletRequest nested) {
  super(nested);
  ...
 }
 public void setParams(String key,String value) {
  ...
 }
 public String getParameter(String name) {
  String response;
  ...
  if (response==null){
   response=super.getParameter(name);
  }
  return response;
 }
}
```

Manipulating Requests and Responses 4-4

- The code snippet shows how to extend `HttpServletResponseWrapper` class.

```
public class MyResponseWrapper extends HttpServletResponseWrapper {


 public String toString() {
...
 }
 public MyResponseWrapper (HttpServletResponse response){
  super(response);
   ...
 }
    public PrintWriter getWriter(){
 ...
 }
}
```

Introduction to Annotations 1-2

- Annotations are one of the major advancement from Java EE 5.0 that are used to configure the server components.
- Using annotation in Java EE, makes the standard **`application.xml`** and **`web.xml`** deployment descriptors files optional.

Introduction to Annotations 2-2

- **Annotations**:
  - Can be defined as metadata information that can be attached to an element within the code to characterize it.
  - Simplifies the developer's work by reducing the amount of code to be written by moving the metadata information into the source code itself.
  - Can be added to program elements such as classes, methods, fields, parameters, local variables, and packages.
  - Never executed.
  - Is processed when the code containing it are compiled or interpreted by compilers, deployment tools, and so on.
  - Can result in the generation of code documents, code artifacts, and so on.

Advantages of Annotations

- Some of the advantages of using annotations are as follows:

Ease of Use

- Annotations are checked and compiled by the Java language compiler and are simple to use.

Portability

- Annotations are portable.

Type Checking

- Annotations are instances of annotation types and are compiled in their own class files.

Runtime Reflection

- Annotations are stored in the class files and accessed for runtime access.

Declaring Annotations 1-2

- The annotation type is used for defining an annotation and is used when custom annotation needs to be created.
- An annotation type takes an 'at `(@)`' sign, followed by the `interface` keyword and the annotation name.
- An annotation takes the form of an 'at' `(@)` sign, followed by the annotation type.
- The various standard annotations include `@Deprecated`, `@Override`, `@SuppressWarnings`, `@Documented`, and `@Retention`.

Declaring Annotations 2-2

- The rules and guidelines to be followed by a developer before using annotation are as follows:

A field or a method can be assigned any access qualifier.

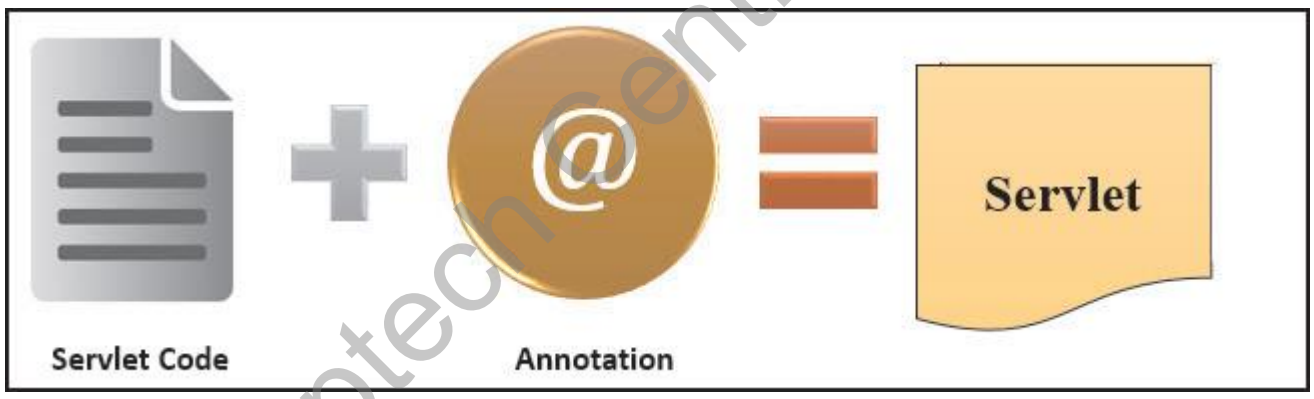
A field or method cannot be static.

The fields or methods of the main class that have been annotated for injection must be static.

Resource annotation can be specified in any of the classes or their superclasses.

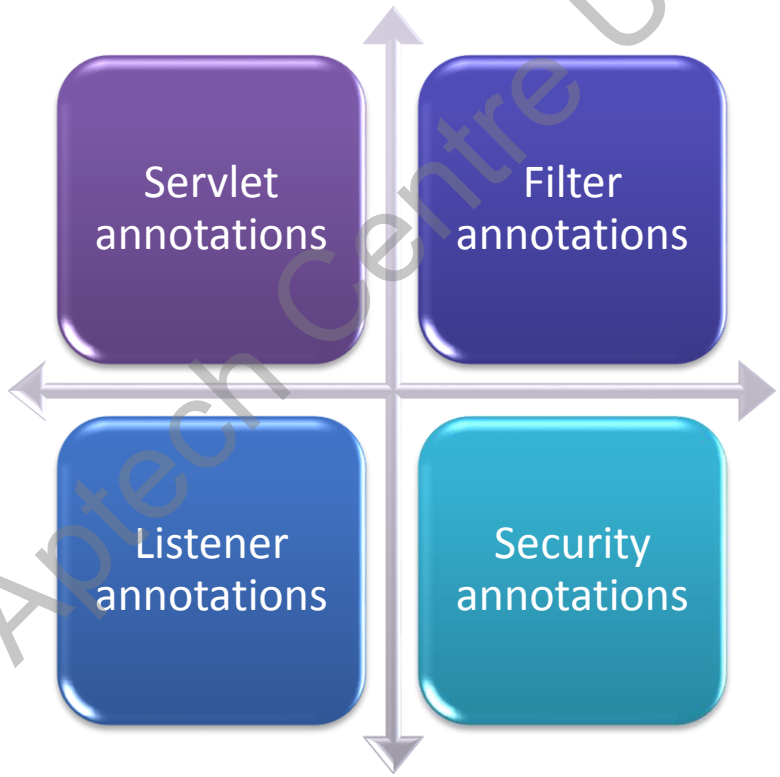
Support for Annotation

- Servlet API supports different types of annotations to provide information.
- The `javax.servlet.annotation` package provides annotations to declare Servlets by specifying metadata information in the Servlet class.
- The `javax.servlet.annotation` package also provides annotations to declare filters and listeners.
- Figure shows how to configure a Servlet by adding annotations to it.



Annotations API

- The `javax.servlet.annotation` package:
  - Contains a number of annotations Servlets, filters, and listeners.
  - Provides different types of annotations.
  - Figure shows the different types of annotations.



Servlet Annotations 1-5

- Some of the annotations used by the Servlet are as follows:
  - **WebServlet:**
    - This annotation is used to provide the mapping information of the Servlet.
    - Table shows the attributes of `@WebServlet` annotation.

Attribute Name	Description
`name`	Specifies the Servlet name. This attribute is optional.
`description`	Specifies the Servlet description and it is an optional attribute.
`displayName`	Specifies the Servlet display name, this attribute is optional.
`urlPatterns`	An array of url patterns use for accessing the Servlet, this attribute is required and should register one url pattern.
`asyncSupported`	Specifies whether the Servlet supports asynchronous processing or not, the value can be true or false.
`initParams`	An array of `@WebInitParam`, that can be used to pass servlet configuration parameters. This attribute is optional.
`loadOnStartup`	An integer value that indicates servlet initialization order, this attribute is optional.
`smallIcon`	A small icon image for the servlet, this attribute is optional.
`largeIcon`	A large icon image for the servlet, this attribute is optional.

Servlet Annotations 2-5

- The code snippet demonstrates how to assign `@WebServlet` annotation to the Servlet class.

```
. . .

@WebServlet(
    name = "webServletAnnotation",
    urlPatterns = {"/hello", "/helloWebApp"},
    asyncSupported = false
)

public class HelloAnnotationServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response)throws ServletException,
    IOException {

    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    out.write("<html><head><title>WebServlet Annotation</title></
head>");
    out.write("<body>");
    out.write("<h1>Servlet Hello Annotation</h1>");
    out.write("<hr/>");
    out.write("Welcome " +
    getServletConfig().getInitParameter("name"));
    out.write("</body></html>");
    out.close();

  }
}
```

Servlet Annotations 3-5

- To avoid this wastage of time, there is an option to disable annotation using the metadata-complete attribute in the `web.xml` file.
- The code snippet shows how to disable annotation in the Servlet files with version 2.5.

```
...
<web-app version="3.1"

 xmlns="http://xmlns.jcp.org/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
 http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
 metadata-complete="true">

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

Servlet Annotations 4-5

- **WebInitParam:**
  - This annotation is used to specify the initialization parameters.
  - This can be used with Servlets as well as filters.
  - Table shows the attributes of `@WebInitParam` annotation.

Attribute Name	Description
`name`	Specifies the names of the initialization parameters.
`value`	Specifies the value for the initialization parameters.

Servlet Annotations 5-5

- The code snippet shows how to apply the `@WebInitParam` annotation to the servlet.

```
@WebServlet(
    name = "webServletAnnotation", urlPatterns = {"/hello", "/
    helloWebApp"},
    asyncSupported = false,
    initParams = {
     @WebInitParam(name = "name", value = "admin"),
     @WebInitParam(name = "param1", value = "paramValue1"),
     @WebInitParam(name = "param2", value = " paramValue2")
    }
)
public class HelloAnnotationServlet extends HttpServlet {
 protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
. . .
 out.write("Welcome " + getServletConfig().
                        getInitParameter("name"));
. . .
}
```

Listener Annotations 1-4

- The Servlet API provides different types of listener interfaces to handle different events.
- To handle HttpSession life cycle events, the `HttpSessionListener` can be used.
- To declare a class as a listener class, the `@WebListener` annotation can be used.

Listener Annotations 2-4

- The `@WebListener` annotation is used to register the following types of listeners:

Context Listener	• `javax.servlet.ServletContextListener`
Context Attribute Listener	• `javax.servlet.ServletContextAttributeListener`
Servlet Request Listener	• `javax.servlet.ServletRequestListener`
Servlet Request Attribute Listener	• `javax.servlet.ServletRequestAttributeListener`
Http Session Listener	• `javax.servlet.http.HttpSessionListener`
Http Session Attribute Listener	• `javax.servlet.http.HttpSessionAttributeListener`

Listener Annotations 3-4

- The code snippet shows the implementation of the class that will log the session created or destroyed.

```
. . .
. . .
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class MySessionListener implements HttpSessionListener{

 @Override
 public void sessionCreated(HttpSessionEvent se) {
  System.out.println("Session Created:: ID="+ se.getSession().
getId());
 }

@Override
public void sessionDestroyed(HttpSessionEvent se) {
System.out.println("Session Destroyed:: ID="+ se.getSession().
getId());
    }

}
```

Listener Annotations 4-4

- The code snippet shows the Servlet class to create the `HttpSession` object.

```
@WebServlet("/MyServlet")

public class MyServlet extends HttpServlet {
 private static final long serialVersionUID = 1L;

 protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException
{
    HttpSession session = request.getSession();
    session.invalidate();
}

}
```

Filter Annotations 1-2

- In a Web application, the `@WebFilter` annotation is used to define a filter.
- It is specified on a class and contains metadata regarding the filter being declared.
- At least one URL pattern must be specified with the annotated filter.
- The `urlPatterns` or `value` attribute on the annotation accomplishes this.
- The other attributes are optional and have default settings.

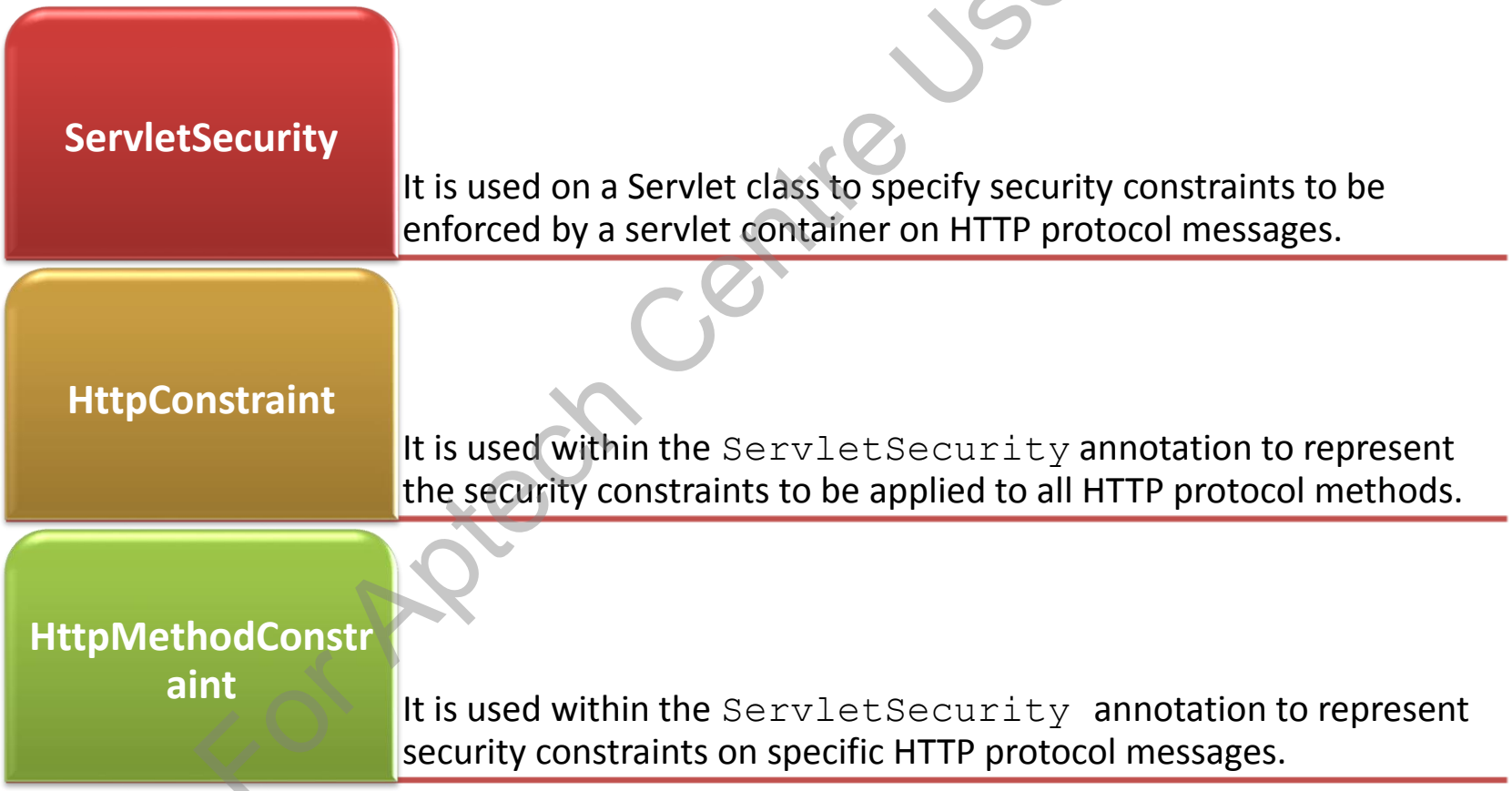
Filter Annotations 2-2

- The code snippet shows the creation of filter using `@WebFilter` annotation.

```
. . .
@WebFilter (value="/hello",
    initParams={@WebInitParam(name="message", value="Servlet
    says: ") }))

public MyFilter implements Filter {
 private FilterConfig _filterConfig;

 public void init(FilterConfig filterConfig)
  throws ServletException
 {
  _filterConfig = filterConfig;
 }

public void doFilter(ServletRequest req,
    ServletResponse res,
    FilterChain chain)
 throws ServletException, IOException
 {

PrintWriter out = res.getWriter();
    out.print(_filterConfig.getInitParameter("message"));
}

 public void destroy() {
 // destroy
 }
}
```

Security Annotations

- Security annotations can be used to specify permission on the methods of the Servlet class.
- Some of the security annotations are as follows:



Dependency Injection Annotations 1-2

- Dependency Injection (DI) mechanism allows the container to inject the dependent objects in the application components.
- The application components can be dependent on many resources to perform the necessary operations.
- The dependent objects are injected before the life cycle methods are invoked and before any reference is made to the dependent object.
- Servlet API supports various dependency injection annotations for different types of resources on which Servlet can be dependent. These include:
  - `@Resource`
  - `@RJB`
  - `@PersistenceUnit`
  - `@PersistenceContext`
  - `@WebServiceRef`

Dependency Injection Annotations 2-2

- The code snippet demonstrates how to obtain a reference for the data source in the Servlet class.

```
// Injecting the Datasource named EmployeeDB

 @Resource(name="jdbc/EmployeeDB")

 private javax.jdbc.DataSource myDB;

// Creating connection with the data source
 Connection con;

// Obtaining the connection
 con = myDb.getConnection();
```

Uploading Files with Servlet

- A very common requirement of a Web application is to upload the files on the server.
- File can be uploaded on the sever using the following two ways:

1. Client-side file upload

2. Server-side file upload

Client-side Uploading 1-3

- To upload the files on Web pages, there are some changes required in the HTML form.
- The requirements are as follows:
  - Create the input element with the attribute `type="file"`.
  - The form method supported for file upload will be `POST`, as `GET` method is not supported in file uploading.
  - Use the attribute named `enctype` with the `form` element.
- Table shows the values that can be assigned to the `enctype` attribute.

Value	Description
`application/x-www-form-urlencoded`	• This is the default value if not specified. • The value ensures that all characters are encoded before they are sent to the server.
`multipart/form-data`	• The value is provided when the form is using a file upload control. • It does not encode the characters before sending them in the request.
`text/plain`	• The value converts the spaces with the "+" symbols, however no other special characters are encoded.

Client-side Uploading 2-3

- The code snippet shows the `upload.jsp` page containing the HTML form element to upload the file on the server.

```
<html>
 <head>
  <title>File Uploading Form</title>
 </head>
 <body>
  <h3>File Upload:</h3>
  Select a file to upload: <br />

  <form action="UploadServlet" method="post"
     enctype="multipart/form-data">
     <input type="file" name="file" size="50"/>

     <br />
      <br />
    <input type="submit" value="Upload File" />
   </form>

 </body>
</html>
```

Client-side Uploading 3-3

- Figure shows the output of the JSP page.

Server-side File Upload 1-4

- The `HttpServletRequest` class includes two methods for handling the file upload. These are as follows:
  - **`Part getPart(String name)`**
    - This method extracts the individual parts of the file and returns each as a `Part` object.
    - The `Part` object belongs to the `Part` interface and provide methods that can be used to extract information from the returned `Part` object.
    - The information includes the name, the size, and the content-type of the file returned as a part.
    - It also provide methods for querying the header content submitted with the Http request object, writing the part to the external disk, and deleting the part.
  - **`Collection<Part> getParts()`**
    - This method returns a collection of `Part` objects which can be iterated to extract individual `Part` object from the collection.

Server-side File Upload 2-4

- The code snippet demonstrates the Servlet code that uploads the file on the server.

```
public class UploadServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
HttpServletResponse response) throws IOException {

    response.setContentType ("text/html");
    PrintWriter out = response.getWriter();

    boolean isMultipartContent = ServletFileUpload.
    isMultipartContent(request);

    if (!isMultipartContent) {
    out.println("No file uploaded<br/>");
      return;
  }

 out.println ("You are trying to upload:<br/>");

 FileItemFactory factory = new DiskFileItemFactory();

 ServletFileUpload upload = new ServletFileUpload(factory);

  try {
    List<FileItem> fields = upload.parseRequest(request);
  out.println("Number of fields: " + fields.size() + "<br/><br/>");
  Iterator<FileItem> it = fields.iterator();

  if (!it.hasNext()) {
   out.println("No fields found");
   return;
  }
```

Server-side File Upload 3-4

```
while (it.hasNext()) {

    FileItem = (FileItem)it.next();
    boolean isFormField = fileItem.isFormField();

         if (!isFormField) {
    String fileName = fileItem.getName();
    File = new File(fileName) ;
    fileItem.write(file) ;
    out.println("Uploaded Filename: " + fileName +
"<br>");

    }
   }
  }
  catch (Exception e) {
   e.printStackTrace();
  }
  }
}
```

Server-side File Upload 4-4

- The code snippet shows the corresponding `web.xml` file for configuring the Servlet.

```
 <servlet>
   <servlet-name>UploadServlet</servlet-name>
   <servlet-class>com.Upload.UploadServlet</servlet-
class>
 </servlet>

 <servlet-mapping>
   <servlet-name>UploadServlet</servlet-name>
   <url-pattern>/UploadServlet</url-pattern>
 </servlet-mapping>

 <welcome-file-list>
  <welcome-file>uploadfile.jsp</welcome-file>
 </welcome-file-list>
</web-app>
```

MultipartConfig Annotation 1-3

- The `javax.servlet.annotation.MultipartConfig` annotation is specified on a Servlet class.
- It provides file information to the server while uploading its content.
- The information provided by the `MultipartConfig` annotation is as follows:

The maximum size of the data that can be uploaded on the server

The default location where the file will be uploaded on the server

MultipartConfig Annotation 2-3

- Table lists the attributes of the `MultipartConfig` annotation.

Attribute	Description
`fileSizeThreshold`	• Specifies the size of the file in bytes.
`maxFileSize`	• Specifies the maximum file limit allowed to upload the file on the server.
`maxRequestSize`	• Specifies the maximum file size in bytes to be accepted for encrypted file data sent in the request.
`location`	• Specifies the default location for the file upload on the server.

MultipartConfig Annotation 3-3

- The code snippet shows the attributes of `@MultipartConfig` annotation that are applied to the Servlet.

```
@WebServlet(name="UploadServlet" ,
urlPatterns={"/Upload"})
@MultipartConfig(
      fileSizeThreshold=100,
      maxFileSize=400,
      maxRequestSize=50,
      location="C:/tmp")

public class UploadServlet extends HttpServlet{
protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws IOException {
  . . .
  . . .
}
```

Summary

- Filter acts as an interface between client and servlet inside the server. It intercepts the request or response to and from the server. More than one filter can also exist between a client and a servlet. This is called a filter chain.
- The Filter API is a part of the servlet package. It contain interfaces namely, Filter, FilterConfig, and FilterChain.
- All the information about the filter is described within the <filter> element inside the web.xml file. The filter also needs to be mapped to the servlet for which it will function. This is done by assigning URL name and servlet name inside the <filter-mapping> element.
- The request and response to and from the servlet is manipulated by the filter. The wrapper object of RequestWrapper or ResponseWrapper classes created by the filter intercepts the request or response and sends it to the filter.
- Annotation can be defined as metadata information that can be attached to an element within the code to characterize it. Annotation can be added to program elements such as classes, methods, fields, parameters, local variables, and packages.
- The javax.servlet.annotation package contains a number of annotations Servlets, filters and listeners. Apart from this, Servlet API also supports various dependency injection annotations.
- File can be uploaded on the sever using the following two ways namely, Client-side file upload and server-side file upload.
- The javax.servlet.annotation.MultipartConfig annotation is specified on a Servlet class to provides file information to the server while uploading its content.