What is java enterprise edition?

he **Java EE** stands for **Java Enterprise Edition**, which was earlier known as J2EE and is currently known as Jakarta EE. It is a set of specifications wrapping around Java SE (Standard Edition). The Java EE provides a platform for developers with enterprise features such as distributed computing and web services. Java EE applications are usually run on reference run times such as **micro servers** or **application servers**. Examples of some contexts where Java EE is used are e-commerce, accounting, banking information systems.

Java EE is a collection of specifications for developing and deploying enterprise applications.

In general, enterprise applications refer to software hosted on servers that provide the applications that support the enterprise.

The Java EE architecture provides services that simplify the most common challenges facing developers when building modern applications, in many cases through APIs, thus making it easier to use popular design patterns and industry-accepted best practices.

For example, one common challenge enterprise developers face is how to handle requests coming in from web-based clients. To simplify this challenge, Java EE provides the Servlet and JavaServer Pages (JSP) APIs, which provide methods for activities like finding out what a user typed into a text field in an online form or storing a cookie on a user's browser.

Another common task is how to store and retrieve information in a database. To address this goal, Java EE provides the Java Persistence API (JPA,) which makes it easy to map data used within a program to information stored in the tables and rows of a database. Also, creating web services or highly scalable logic components is simplified through the use of the Enterprise JavaBeans (EJB) specification. All of these APIs are well tested, relatively easy for Java developers to learn and can greatly simplify some of the hardest parts of enterprise development.

Java EE core technologies

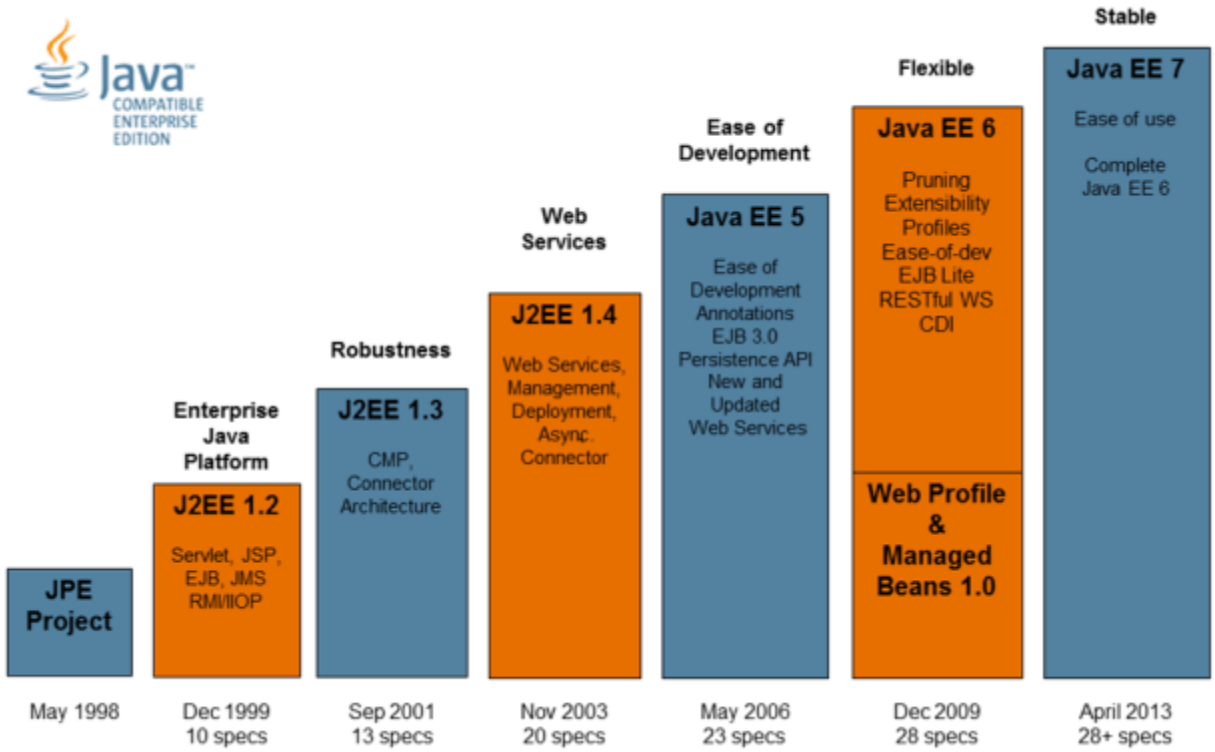
Along with the four aforementioned APIs, there are more than 30 Java APIs included as Java EE core technologies, with that number to approach 50 with the eventual release of Java EE 8. These Java EE core technologies broadly fall into the following file categories:

- **HTTP client technologies**. For dealing with HTTP-based clients, Java EE includes the Java API for WebSocket programming, an API for JSON Processing, the JSF and Servlet APIs and the JSP Standard Tag Library (JSTL).
- **Database and resource access technologies**. For interacting with external and back-end systems, Java EE includes JavaMail, a standard connector architecture, a Java Message Service (JMS) API and a Java Transaction API (JTA) for enforcing two-phase commits.
- **REST and web service technologies:** To help with the development and deployment of REST-, SOAP-, XML- and JSON-based web services, the Java APIs for RESTful Web Services (JAX-RS) and XML-based web services (JAX-WS) are included, along with APIs for XML messaging and XML registries (JAXR).
- **Java EE security and container management**: For implementing custom Java EE security and managing Java EE containers, software developers have access to the Java Authorization Contract for Containers and the Java Authentication Service Provider Interface for Containers.

JAVA EE Evolution

Java EE initially evolved as an enterprise application deployment platform that focused on robustness, Web services, and ease of deployment.

Continually shaped by feedback through the Java Community Process (JCP), Java EE represents a universal standard in enterprise IT, facilitating the development, deployment, and management of multi-tier, server-centric applications. Beginning with Java EE 5, focus shifted to increasing developer efficiency with the introduction of annotations, the Enterprise JavaBeans (EJB) 3.0 business component development model, new and updated Web services, and improvements to the persistence model.



Java EE 6 further streamlined the development process and increased the flexibility of the platform, thus enabling it to better address lightweight Web applications.

This is in part due to the introduction of the Web Profile as a subset of the Java EE specification targeted to Web applications. In addition, Java EE 6 embraced open source frameworks with hooks for more seamless integration, and began the process of pruning less relevant technologies.

Java EE 6 was particularly successful:

• As of May 2013, there have been over 50 million downloads of Java EE components, from Oracle and other industry vendors.

• It is the #1 choice for enterprise developers.

• It is the #1 application development platform.

• It has had the fastest adoption of any Java EE release with 18 compliant application server vendors.

Java EE 7 extends the benefits of Java EE 6 by leveraging the more transparent JCP process and community participation to deliver new and updated features, excelling in the areas expressed in Figure

• Java EE 7 enables developers to deliver HTML5 dynamic scalable applications. New to the platform, WebSockets reduce response time with low latency bi-directional data exchange while standard JSON support simplifies data parsing for portable applications. JAX-RS has been improved to deliver asynchronous, scalable, high performance RESTful Services. And much more.

• Java EE 7 increases developer productivity in multiple ways. It offers a simplified application architecture with a cohesive integrated platform; increased efficiency with reduced boiler-plate code and broader use of annotations; and enhanced application portability with standard RESTful Web service client support.

• Java EE 7 meets the most demanding enterprise requirements by breaking down batch jobs into manageable chunks for uninterrupted OLTP performance; easily defines multithreaded concurrent tasks for improved scalability; and delivers transactional applications with choice and flexibility.

GlassFish Server

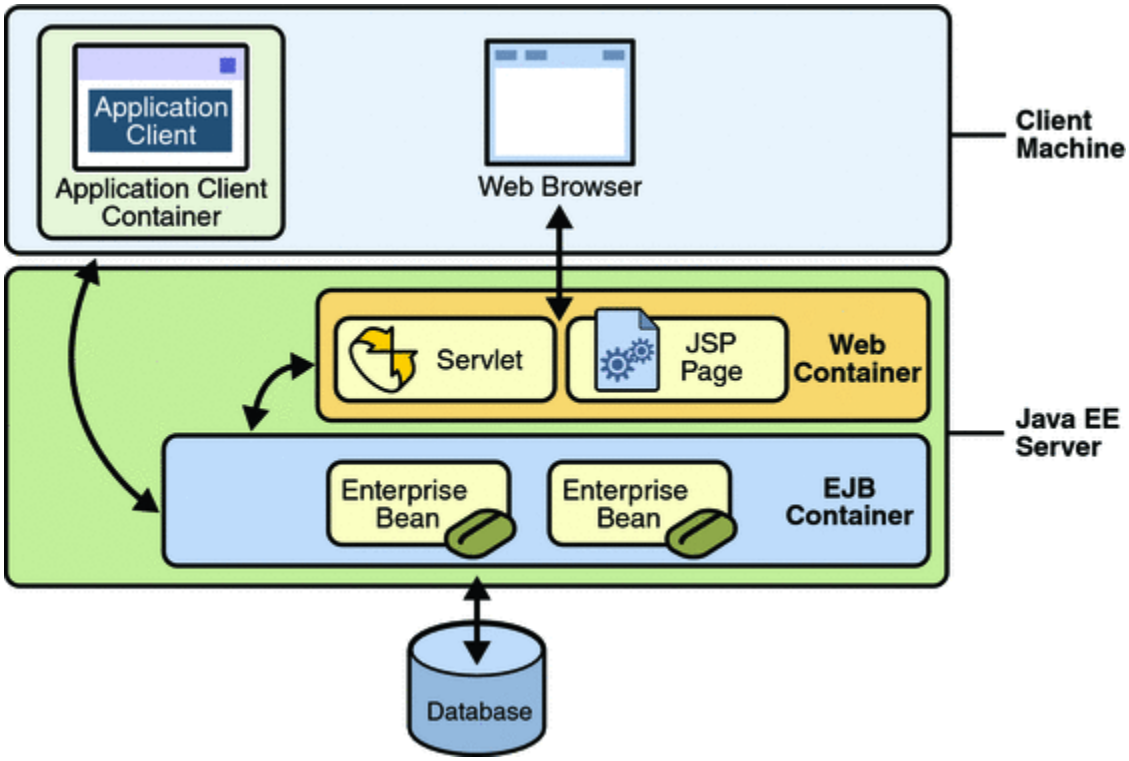
GlassFish is an open-source Jakarta EE platform application server project started by Sun Microsystems, then sponsored by Oracle Corporation, and now living at the Eclipse Foundation and supported by Payara, Oracle and Red Hat.[2] The supported version under Oracle was called Oracle GlassFish Server. GlassFish is free software and was initially dual-licensed under two free software licences: the Common Development and Distribution License (CDDL) and the GNU General Public License (GPL) with the Classpath exception. After having been transferred to Eclipse, GlassFish remained dual-licensed, but the CDDL license was replaced by the Eclipse Public License (EPL).

GlassFish is the reference implementation of Jakarta EE and as such supports EJB, JPA, JSF, JMS, RMI, JSP, servlets, etc. This allows developers to create enterprise applications that are portable and scalable, and that integrate with legacy technologies. Optional components can also be installed for additional services.

Built on a modular kernel powered by OSGi, GlassFish runs straight on top of the Apache Felix implementation. It also runs with Equinox OSGi or Knopflerfish OSGi runtimes. HK2 abstracts the OSGi module system to provide components, which can also be viewed as services. Such services can be discovered and injected at runtime.

GlassFish is based on source code released by Sun and Oracle Corporation's TopLink persistence system. It uses a derivative of Apache Tomcat as the servlet container for serving Web content, with an added component called Grizzly which uses Java New I/O (NIO) for scalability and speed.

Java EE Server and Containers



Java EE Servers

A Java EE server is a server application that the implements the Java EE platform APIs and provides the standard Java EE services. Java EE servers are sometimes called application servers, because they allow you to serve application data to clients, much like web servers serve web pages to web browsers.

Java EE servers host several application component types that correspond to the tiers in a multi-tiered application. The Java EE server provides services to these components in the form of a **container**.

Java EE Containers

Java EE containers are the interface between the component and the lower-level functionality provided by the platform to support that component. The functionality of the container is defined by the platform, and is different for each component type. Nonetheless, the server allows the different component types to work together to provide functionality in an enterprise application.

The Web Container

The web container is the interface between web components and the web server. A web component can be a servlet, a JavaServer Faces Facelets page, or a JSP page. The container manages the component's lifecycle, dispatches requests to application components, and provides interfaces to context data, such as information about the current request.

The Application Client Container

The application client container is the interface between Java EE application clients, which are special Java SE applications that use Java EE server components, and the Java EE server. The application client container runs on the client machine, and is the gateway between the client application and the Java EE server components that the client uses.

The EJB Container

The EJB container is the interface between enterprise beans, which provide the business logic in a Java EE application, and the Java EE server. The EJB container runs on the Java EE server and manages the execution of an application's enterprise beans.

Java Servlet Technology

Servlet technology is used to create a web application (resides at server side and generates a dynamic web page).

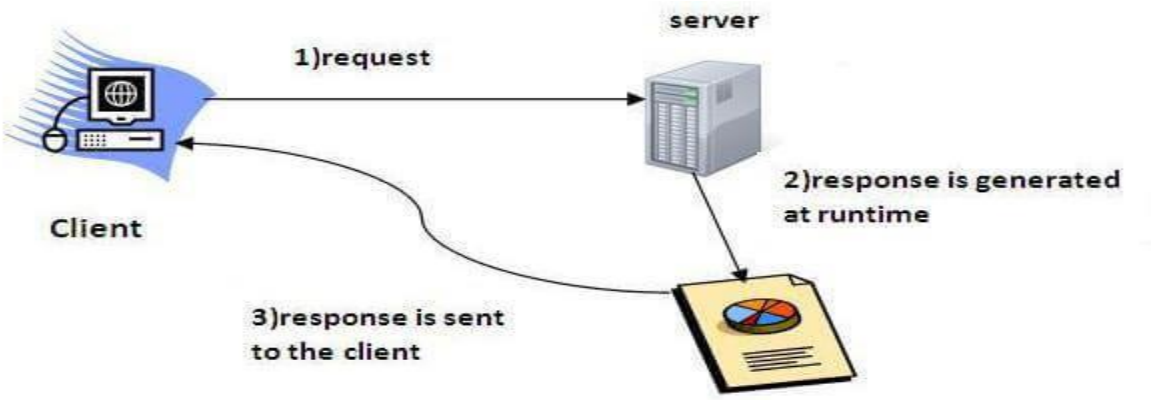
Servlet technology is robust and scalable because of java language. Before Servlet, CGI (Common Gateway Interface) scripting language was common as a server-side programming language. However, there were many disadvantages to this technology. We have discussed these disadvantages below.

There are many interfaces and classes in the Servlet API such as Servlet, GenericServlet, HttpServlet, ServletRequest, ServletResponse, etc.

What is a Servlet?

Servlet can be described in many ways, depending on the context.

- Servlet is a technology which is used to create a web application.
- Servlet is an API that provides many interfaces and classes including documentation.
- Servlet is an interface that must be implemented for creating any Servlet.
- Servlet is a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.
- Servlet is a web component that is deployed on the server to create a dynamic web page.



Servlet API

The javax.servlet and javax.servlet.http packages represent interfaces and classes for servlet api.

The **javax.servlet** package contains many interfaces and classes that are used by the servlet or web container. These are not specific to any protocol.

The **javax.servlet.http** package contains interfaces and classes that are responsible for http requests only.

Let's see what are the interfaces of javax.servlet package.

Interfaces in javax.servlet package

There are many interfaces in javax.servlet package. They are as follows:

1. Servlet
2. ServletRequest
3. ServletResponse
4. RequestDispatcher
5. ServletConfig
6. ServletContext
7. SingleThreadModel

8. Filter
9. FilterConfig
10. FilterChain
11. ServletRequestListener
12. ServletRequestAttributeListener
13. ServletContextListener
14. ServletContextAttributeListener

Classes in javax.servlet package

There are many classes in javax.servlet package. They are as follows:

1. GenericServlet
2. ServletInputStream
3. ServletOutputStream
4. ServletRequestWrapper
5. ServletResponseWrapper
6. ServletRequestEvent
7. ServletContextEvent
8. ServletRequestAttributeEvent
9. ServletContextAttributeEvent
10. ServletException
11. UnavailableException

Interfaces in javax.servlet.http package

There are many interfaces in javax.servlet.http package. They are as follows:

1. HttpServletRequest
2. HttpServletResponse
3. HttpSession
4. HttpSessionListener
5. HttpSessionAttributeListener
6. HttpSessionBindingListener
7. HttpSessionActivationListener
8. HttpSessionContext (deprecated now)

Classes in javax.servlet.http package

There are many classes in javax.servlet.http package. They are as follows:

1. HttpServlet
2. Cookie
3. HttpServletRequestWrapper
4. HttpServletResponseWrapper
5. HttpSessionEvent
6. HttpSessionBindingEvent
7. HttpUtils (deprecated now)

Servlets - Life Cycle

A servlet life cycle can be defined as the entire process from its creation till the destruction. The following are the paths followed by a servlet.

- The servlet is initialized by calling the **init()** method.
- The servlet calls **service()** method to process a client's request.
- The servlet is terminated by calling the **destroy()** method.
- Finally, servlet is garbage collected by the garbage collector of the JVM.

Now let us discuss the life cycle methods in detail.

The init() Method

The init method is called only once. It is called only when the servlet is created, and not called for any user requests afterwards. So, it is used for one-time initializations, just as with the init method of applets.

The servlet is normally created when a user first invokes a URL corresponding to the servlet, but you can also specify that the servlet be loaded when the server is first started.

When a user invokes a servlet, a single instance of each servlet gets created, with each user request resulting in a new thread that is handed off to doGet or doPost as appropriate. The init() method simply creates or loads some data that will be used throughout the life of the servlet.

The init method definition looks like this −

```
public void init() throws ServletException {
  // Initialization code...
}
```

The service() Method

The service() method is the main method to perform the actual task. The servlet container (i.e. web server) calls the service() method to handle requests coming from the client(browsers) and to write the formatted response back to the client.

Each time the server receives a request for a servlet, the server spawns a new thread and calls service. The service() method checks the HTTP request type (GET, POST, PUT, DELETE, etc.) and calls doGet, doPost, doPut, doDelete, etc. methods as appropriate.

Here is the signature of this method −

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

The service () method is called by the container and service method invokes doGet, doPost, doPut, doDelete, etc. methods as appropriate. So you have nothing to do with service() method but you override either doGet() or doPost() depending on what type of request you receive from the client.

The doGet() and doPost() are most frequently used methods with in each service request. Here is the signature of these two methods.

The doGet() Method

A GET request results from a normal request for a URL or from an HTML form that has no METHOD specified and it should be handled by doGet() method.

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  // Servlet code
}
```

The doPost() Method

A POST request results from an HTML form that specifically lists POST as the METHOD and it should be handled by doPost() method.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
  throws ServletException, IOException {
  // Servlet code
}
```

The destroy() Method

The destroy() method is called only once at the end of the life cycle of a servlet. This method gives your servlet a chance to close database connections, halt background threads, write cookie lists or hit counts to disk, and perform other such cleanup activities.

After the destroy() method is called, the servlet object is marked for garbage collection. The destroy method definition looks like this −

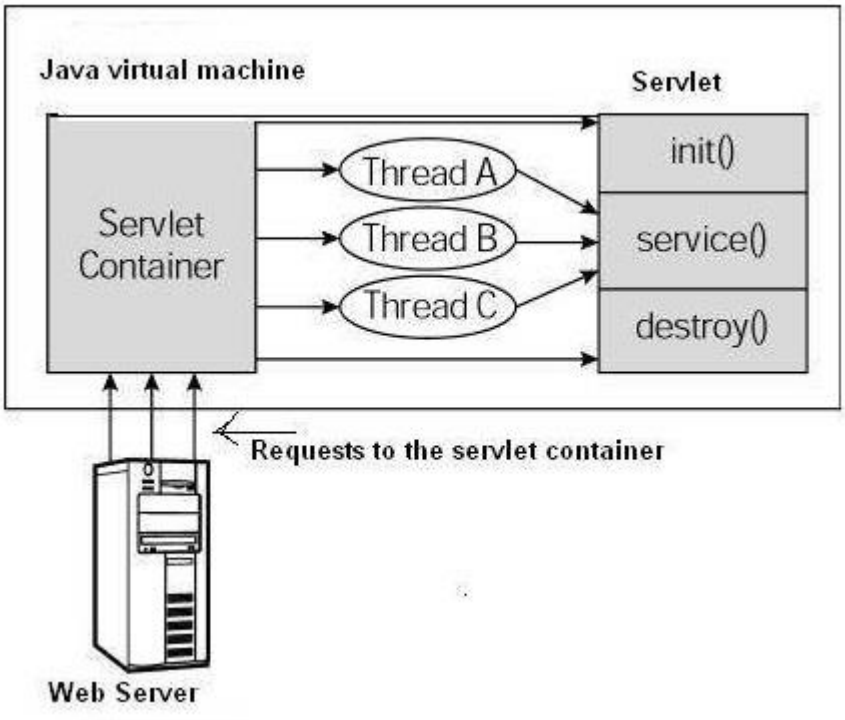
```
public void destroy() {
  // Finalization code...
}
```

Architecture Diagram

The following figure depicts a typical servlet life-cycle scenario.

- First the HTTP requests coming to the server are delegated to the servlet container.

- The servlet container loads the servlet before invoking the service() method.
- Then the servlet container handles multiple requests by spawning multiple threads, each thread executing the service() method of a single instance of the servlet.



Types of Servlet

Servlet Interface

Servlet interface provides commonbehaviorto all the servlets.Servlet interface defines methods that all servlets must implement.

Servlet interface needs to be implemented for creating any servlet (either directly or indirectly). It provides 3 life cycle methods that are used to initialize the servlet, to service the requests, and to destroy the servlet and 2 non-life cycle methods.

Methods of Servlet interface

There are 5 methods in Servlet interface. The init, service and destroy are the life cycle methods of servlet. These are invoked by the web container.

Method	Description
public void init(ServletConfig config)	initializes the servlet. It is the life cycle method of servlet and invoked by the web

	container only once.
public void service(ServletRequest request,ServletResponse response)	provides response for the incoming request. It is invoked at each request by the web container.
public void destroy()	is invoked only once and indicates that servlet is being destroyed.
public ServletConfig getServletConfig()	returns the object of ServletConfig.
public String getServletInfo()	returns information about servlet such as writer, copyright, version etc.

Example :- (with Annotation)

Input.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action=" servlet_an" method="get">
         Enter 1st No :<input type="text" name="t1"><br>
         Enter 2nd No :<input type="text" name="t2"><br>
         <input type="submit" name="add" value="+">
         <input type="submit" name="sub" value="-">
         <input type="submit" name="mult" value="*">
         <input type="submit" name="div" value="/">
      </form>
   </body>
</html>
```

servlet_an.java

```
package p3;

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

```
@WebServlet(name = "servlet_an", urlPatterns = {"/servlet_an"})

public class servlet_an implements Servlet
{
   @Override
   public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException
   {
      PrintWriter out=response.getWriter();
      out.println("Example of Servlet interface with annotation");
   }

   @Override
   public void init(ServletConfig config) throws ServletException
   {

   }

   @Override
   public ServletConfig getServletConfig()
   {
      return getServletConfig();
   }

   @Override
   public String getServletInfo()
   {
      return "";
   }

   @Override
   public void destroy()
   {

   }
}
```

Example :- (with Deployment Discriptor)

Input.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action="dd" method="get">
         Enter 1st No :<input type="text" name="t1"><br>
         Enter 2nd No :<input type="text" name="t2"><br>
         <input type="submit" name="add" value="+">
         <input type="submit" name="sub" value="-">
         <input type="submit" name="mult" value="*">
         <input type="submit" name="div" value="/">
      </form>
   </body>
</html>
```

servlet_dd.java

```
package p3;

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

public class servlet_dd implements Servlet
{
   @Override
   public void service(ServletRequest request, ServletResponse response) throws
ServletException, IOException
   {
      PrintWriter out=response.getWriter();
      out.println("Example of Servlet interface with web.xml(DD) file");
   }

   @Override
   public void init(ServletConfig config) throws ServletException
   {

   }

   @Override
```

```
    public ServletConfig getServletConfig()
    {
        return getServletConfig();
    }

    @Override
    public String getServletInfo()
    {
        return "";
    }

    @Override
    public void destroy()
    {

    }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app >

    <servlet>
        <servlet-name>servlet_dd</servlet-name>
        <servlet-class>p3.servlet_dd</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>servlet_dd</servlet-name>
        <url-pattern>/dd</url-pattern>
    </servlet-mapping>
</web-app>
```

GenericServlet class

GenericServlet class implements **Servlet**, **ServletConfig** and **Serializable** interfaces. It provides the implementation of all the methods of these interfaces except the service method.

GenericServlet class can handle any type of request so it is protocol-independent.

You may create a generic servlet by inheriting the GenericServlet class and providing the implementation of the service method.

Methods of GenericServlet class

There are many methods in GenericServlet class. They are as follows:

1. **public void init(ServletConfig config)** is used to initialize the servlet.
2. **public abstract void service(ServletRequest request, ServletResponse response)** provides service for the incoming request. It is invoked at each time when user requests for a servlet.
3. **public void destroy()** is invoked only once throughout the life cycle and indicates that servlet is being destroyed.
4. **public ServletConfig getServletConfig()** returns the object of ServletConfig.
5. **public String getServletInfo()** returns information about servlet such as writer, copyright, version etc.
6. **public void init()** it is a convenient method for the servlet programmers, now there is no need to call super.init(config)
7. **public ServletContext getServletContext()** returns the object of ServletContext.
8. **public String getInitParameter(String name)** returns the parameter value for the given parameter name.
9. **public Enumeration getInitParameterNames()** returns all the parameters defined in the web.xml file.
10. **public String getServletName()** returns the name of the servlet object.
11. **public void log(String msg)** writes the given message in the servlet log file.
12. **public void log(String msg,Throwable t)** writes the explanatory message in the servlet log file and a stack trace.

Example:- (using Annotation)

Input.html

```
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <form action="generic_an" method="get">
      Enter 1st No :<input type="text" name="t1"><br>
      Enter 2nd No :<input type="text" name="t2"><br>
      <input type="submit" name="add" value="+">
      <input type="submit" name="sub" value="-">
      <input type="submit" name="mult" value="*">
      <input type="submit" name="div" value="/">
    </form>
  </body>
```

```
</html>
```

generic_an.java

```
package p2;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet(name = "generic_an", urlPatterns = {"/generic_an"})

public class generic_an extends GenericServlet
{
   @Override
   public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException
   {
      PrintWriter out=response.getWriter();
      out.println("Example of GenericServlet class with annotation");
   }
}
```

Example:- (using Deployment Descritor)

Input.html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action="dd" method="get">
         Enter 1st No :<input type="text" name="t1"><br>
         Enter 2nd No :<input type="text" name="t2"><br>
         <input type="submit" name="add" value="+">
         <input type="submit" name="sub" value="-">
         <input type="submit" name="mult" value="*">
         <input type="submit" name="div" value="/">
```

```
    </form>
  </body>
</html>
package p2;

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

public class generic_dd extends GenericServlet
{
  @Override
  public void service(ServletRequest request, ServletResponse response) throws ServletException, IOException
  {
    PrintWriter out=response.getWriter();
    out.println("Example of GenericServlet class");
  }
}
web.xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  <servlet>
    <servlet-name>generic_dd</servlet-name>
    <servlet-class>p2.generic_dd</servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>generic_dd</servlet-name>
    <url-pattern>/dd</url-pattern>
  </servlet-mapping>

</web-app>
```

HttpServlet class

The HttpServlet class extends the GenericServlet class and implements Serializable interface. It provides http specific methods such as doGet, doPost, doHead, doTrace etc.

Methods of HttpServlet class

There are many methods in HttpServlet class. They are as follows:

1. **public void service(ServletRequest req,ServletResponse res)** dispatches the request to the protected service method by converting the request and response object into http type.
2. **protected void service(HttpServletRequest req, HttpServletResponse res)** receives the request from the service method, and dispatches the request to the doXXX() method depending on the incoming http request type.
3. **protected void doGet(HttpServletRequest req, HttpServletResponse res)** handles the GET request. It is invoked by the web container.
4. **protected void doPost(HttpServletRequest req, HttpServletResponse res)** handles the POST request. It is invoked by the web container.
5. **protected void doHead(HttpServletRequest req, HttpServletResponse res)** handles the HEAD request. It is invoked by the web container.
6. **protected void doOptions(HttpServletRequest req, HttpServletResponse res)** handles the OPTIONS request. It is invoked by the web container.
7. **protected void doPut(HttpServletRequest req, HttpServletResponse res)** handles the PUT request. It is invoked by the web container.
8. **protected void doTrace(HttpServletRequest req, HttpServletResponse res)** handles the TRACE request. It is invoked by the web container.
9. **protected void doDelete(HttpServletRequest req, HttpServletResponse res)** handles the DELETE request. It is invoked by the web container.
10. **protected long getLastModified(HttpServletRequest req)** returns the time when HttpServletRequest was last modified since midnight January 1, 1970 GMT.

Example:- (using Annotation)

Input.html

```
<html>
  <head>
    <title>TODO supply a title</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <form action="calculator" method="post">
      Enter 1st No :<input type="text" name="t1"><br>
      Enter 2nd No :<input type="text" name="t2"><br>
      <input type="submit" name="add" value="+">
      <input type="submit" name="sub" value="-">
      <input type="submit" name="mult" value="*">
      <input type="submit" name="div" value="/">
    </form>
```

```
   </body>
</html>
```

calculator.java

```
package p1;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet(name = "calculator", urlPatterns = {"/calculator"})
public class calculator extends HttpServlet
{

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
   {
      try
      {
         response.setContentType("text/html;charset=UTF-8");
         PrintWriter out=response.getWriter();
         String a=request.getParameter("t1");
         String b=request.getParameter("t2");
         int c=Integer.parseInt(a);
         int d=Integer.parseInt(b);

         if(request.getParameter("add")!=null)
         {
            int e=c+d;
            out.println("Addition="+e);
         }
         if(request.getParameter("sub")!=null)
         {
            int f=c-d;
            out.println("Subtraction="+f);
         }
         if(request.getParameter("mult")!=null)
         {
            int g=c*d;
            out.println("Multiplication="+g);
         }
         if(request.getParameter("div")!=null)
         {
            int h=c/d;
```

```
            out.println("Division="+h);
          }

       }catch(Exception e)
       { }
    }
}
```

Example:- (using Deployment Decriptor)

Input.Html

```
<html>
   <head>
      <title>TODO supply a title</title>
      <meta charset="UTF-8">
      <meta name="viewport" content="width=device-width, initial-scale=1.0">
   </head>
   <body>
      <form action="dd" method="post">
         Enter 1st No :<input type="text" name="t1"><br>
         Enter 2nd No :<input type="text" name="t2"><br>
         <input type="submit" name="add" value="+">
         <input type="submit" name="sub" value="-">
         <input type="submit" name="mult" value="*">
         <input type="submit" name="div" value="/">
      </form>
   </body>
</html>
```

calculator_dd.java

```
package p1;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class calculator_dd extends HttpServlet
{

   protected void doPost(HttpServletRequest request, HttpServletResponse response)
   {
      try
```

```
        {
            response.setContentType("text/html;charset=UTF-8");
            PrintWriter out=response.getWriter();
            String a=request.getParameter("t1");
            String b=request.getParameter("t2");
            int c=Integer.parseInt(a);
            int d=Integer.parseInt(b);

            if(request.getParameter("add")!=null)
            {
                int e=c+d;
                out.println("Addition="+e);
            }
            if(request.getParameter("sub")!=null)
            {
                int f=c-d;
                out.println("Subtraction="+f);
            }
            if(request.getParameter("mult")!=null)
            {
                int g=c*d;
                out.println("Multiplication="+g);
            }
            if(request.getParameter("div")!=null)
            {
                int h=c/d;
                out.println("Division="+h);
            }

        }catch(Exception e)
        { }
    }
}
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app >
    <servlet>
        <servlet-name>calculator_dd</servlet-name>
        <servlet-class>p1.calculator_dd</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>calculator_dd</servlet-name>
        <url-pattern>/dd</url-pattern>
    </servlet-mapping>
```

</web-app>