Advanced Java Tutorial

Published On: July 5, 2024

Advanced Java Tutorial

Beyond the core concepts of Core Java, Advanced Java concepts cover a wide range of technologies, libraries, and frameworks. This advanced Java tutorial will help you learn the advanced concepts needed to create complicated Java applications effortlessly.

Advanced Java tutorial PDF

Introduction to Advanced Java

Advanced Java is a collection of technologies and tools that allow developers to create dynamic and secure applications.

The key components of advanced Java that ensure the application's authentication and authorization are as follows:

- JPA (Java Persistence API), which is used to manage relational databases,
- Spring Framework, which includes modules for dependency injection.
- Spring MVC Framework
- AOP (Aspect-Oriented Programming)
- Spring Security.

Features like JDBC (Java Database Connectivity), Servlets, and JSP (JavaServer Page) are used for creating interactive, dynamic content.

Benefits of Advanced Java

Some of the benefits of advanced Java are listed below:

- Advanced Java is used to develop dynamic web applications using **JSP (JavaServer Page) and Servlets**.
- **Enterprise JavaBeans (EJB)** provides scalability for big Java projects that meet heavy demands without interruption.
- Utilizing **Java Database Connectivity (JDBC)** for Efficient Data Management.

- It provides code reuse, and when combined with **advanced frameworks like Spring**, it encourages modular architecture and organizes your codebase.
- Advanced Java offers **robust security and a quick development process** to facilitate the integration of web services.

Advanced Java Syllabus PDF

JPA (Java Persistence API)

A framework known as Java Persistence API (JPA) offers several guidelines and interfaces for handling relational data in Java programs.

It is a component of the Java EE (Enterprise Edition) platform and enables programmers to use object-relational mapping, or ORM, to map Java objects to relational database tables.

Writing low-level SQL code is not necessary when interacting with databases because of JPA.

ORM: By enabling software engineers to interact with databases using notions from object-oriented programming, like classes, objects, and methods, instead of needing to write low-level SQL code, object-oriented query language (ORM) aims to facilitate software engineers' work with databases.

Implementation of JPA in Java

> You must first incorporate the JPA library into your project to use JPA. To achieve this, include the ensuing dependency in your project:
>
> *<dependency>*
>
> *<groupId>javax.persistence</groupId>*
>
> *<artifactId>persistence-api</artifactId>*
>
> *<version>2.2</version>*
>
> *</dependency>*

Advanced Java Interview Questions

Java Servlets

The Java programs that operate on a web server or application server with Java support are called Java Servlets.

They are employed to receive requests from the web server, manage them, process them, generate responses, and then reply to the web server with those responses.

ASP DOTNET Tutorial
Published On: July 31, 2024

ASP DOTNET Tutorial Microsoft created the web framework known as ASP.NET. It is employed in...

Artificial Intelligence Tutorial
Published On: July 30, 2024

Artificial Intelligence Tutorial Artificial intelligence (AI) is significant since it enhances many facets of society...

Appium Testing Tutorial
Published On: July 30, 2024

Appium Testing Tutorial Designed to make the UI automation of many app platforms easier, Appium...

The following are Servlets' features:

- On the server side, servlets function.
- Servlets can handle complex requests that are sent to the web server.

Architecture of Java Servlets

A Java application module called Servlet runs on the server side and manages client requests while implementing the Servlet interface.

As you can see in this image, a client submits a request to the server, which then produces, evaluates, and replies to the client.

Servlet Lifecycle

The servlet container utilizes ***javax.servlet.Servlet*** interface to comprehend and control the Servlet object oversees the full life cycle of a servlet.

The Java Servlet has four stages, as follows:

- Loading a Servlet
- Initializing the Servlet
- Request Handling
- Destroying the Servlet

Loading a Servlet

When a server boots up, the servlet container loads all servlets and deploys them.

Initializing the Servlet

Next, the init() method is called to initialize a servlet.

The Servlet container notifies that this Servlet instance has been successfully instantiated

Now, it is ready to be put into service by calling the Servlet.init() method.

Request Handling

Next, to handle a client's request and to notify the Servlet about the client's requests, the Servlet invokes the service() method.

Destroying the Servlet

Calling destroy() is the last step in terminating a servlet. The destroy() method marks the end of a Servlet instance and is only executed once in its lifetime.

The methods destroy() and init() are only invoked once. Ultimately,

a servlet is trash that the JVM's garbage collector gathers. Thus, the servlet life cycle comes to an end here.

Example: Java Servlet

Let's now take an example, in which we will create a basic login servlet and show the results in the browser.

> **Index.html**
>
> *<!DOCTYPE html>*
>
> *<html>*
>
> *<body>*
>
> *<form action="Login" method="post">*
>
> *<table>*
>
> *<tr>*
>
> *<td>Name:</td>*
>
> *<td><input type="text" name="userName"></td>*
>
> *</tr>*
>
> *<tr>*
>
> *<td>Password:</td>*
>
> *<td><input type="password" name="userPassword"></td>*
>
> *</tr>*
>
> *</table>*
>
> *<input type="submit" value="Login">*
>
> *</form>*
>
> *</body>*
>
> *</html>*

Java Class File

> *package SLA;*
>
> *import java.io.IOException;*
>
> *import java.io.PrintWriter;*
>
> *import javax.servlet.ServletException;*
>
> *import javax.servlet.http.HttpServlet;*
>
> *import javax.servlet.http.HttpServletRequest*
>
> *import javax.servlet.http.HttpServletResponse;*
>
> *public class Login extends HttpServlet*
>
> *{*
>
> *protected void doPost(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException*

```
PrintWriter pw=res.getWriter();

res.setContentType(“text/html”);

String user=req.getParameter(“userName”);

String pass=req.getParameter(“userPassword”);

pw.println(“Login Success...!”)

if(user.equals(“SLA”) && pass.equals(“SLA”))

pw.println(“Login Success...!”);

else

pw.println(“Login Failed...!”);

pw.close();

}

}
```

Adding mappings to the web.xml file is the final step after writing the Java class file.

Web.xml

```
<?xml version=”1.0”encoding=”UTF-8”?>

<web-app xmlns:xsi=”<a href=”<a href=”http://www.w3.org/2001/XMLSchema-
instance”>http://www.w3.org/2001/XMLSchema-instance</a>”><a
href=”http://www.w3.org/2001/XMLSchema-
instance”>http://www.w3.org/2001/XMLSchema-instance</a></a>” xmlns=”<a
href=”<a
href=”http://xmlns.jcp.org/xml/ns/javaee”>http://xmlns.jcp.org/xml/ns/javaee</a>”>
<a
href=”http://xmlns.jcp.org/xml/ns/javaee”>http://xmlns.jcp.org/xml/ns/javaee</a>
</a>”xsi:schemaLocation=”<a href=”<a
href=”http://xmlns.jcp.org/xml/ns/javaee”>http://xmlns.jcp.org/xml/ns/javaee</a>”>
<a
href=”http://xmlns.jcp.org/xml/ns/javaee”>http://xmlns.jcp.org/xml/ns/javaee</a>
</a> <a href=”<a href=”http://xmlns.jcp.org/xml/ns/javaee/web-
app_3_1.xsd”>http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd</a>”><a
href=”http://xmlns.jcp.org/xml/ns/javaee/web-
app_3_1.xsd”>http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd</a>
</a>”version=”3.1”>

<display-name>LoginServlet</display-name>

<servlet>

<servlet-name>Login</servlet-name>

<servlet-class>Edureka.Login</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>Login</servlet-name>
```

```
<url-pattern>/Login</url-pattern>

</servlet-mapping>

<welcome-file-list>

<welcome-file>index.html</welcome-file>

</welcome-file-list>

</web-app>
```

Session Tracking: The term “session” refers only to a specific time. Also referred to as session management in servlets, session tracking is a method for preserving user state (data). As a result, the server treats each request from a user as if it were their first.

Cookies: A cookie is a little data file that is kept in between requests from clients. A cookie consists of a name, a single value, and optional properties like a version number, a maximum age, path and domain qualifiers, and a comment.

JSP: Java Server Pages

Similar to Servlet technology, Java Server Pages is a technology used to construct web applications. It is an addition to Servlet since it offers features like JSTL, expression language, and other features that Servlets does not.

HTML tags and JSP tags forms a JSP page. Because we can keep designing and development separate, JSP pages are simpler to manage than Servlet pages.

Differences Between JSP and Servlets

- JSP is an extension to Servlet, but Servlet is not an extension to JSP.
- JSP is easy to maintain, but Servlet is quite complex.
- JSP doesn’t need to be recompiled or redeployed, while Servlet code needs to be recompiled.
- JSP contains less code than Servlet.

JSP Scripting Elements

Java code can be inserted inside JSPs due to the scripting components.

scriptlet tag – A scriptlet tag is used to execute Java source code in JSP.

Syntax : <% java source code %>

We have developed two files, index.html and welcome.jsp, for this example. The user’s username is obtained via the index.html page and the welcome.jsp file prints the username along with a

welcome message. Let's decipher the code now.

Index.html

```
<html>
<body>
<form action="welcome.jsp">
<input type="text" name="uname">
<input type="submit" value="go">
</form>
</body>
</html>
```

Welcome.jsp

```
<html>
<body>
<% String name=request.getParameter("uname"); print("welcome "+name); %>
</form>
</body>
</html>
```

expression tag: The response's output stream receives the code that is put inside the JSP expression tag. Thus, writing is not necessary.

To write data, use print(). Its primary function is to print method or variable values.

Syntax : <%= statement %>

We have utilized the Calendar class's getTime() method to show the current time.

Since the *getTime()* method returns an instance of the Calendar class, we have called it after using the *getInstance()* method to obtain an instance of the Calendar class.

Index.jsp

```
<html>
<body>
Current Time: <%= java.util.Calendar.getInstance().getTime() %>
</body>
</html>
```

declaration tag: Fields and methods must be declared using the

JSP declaration tag. An auto-generated servlet's service() function is beyond the scope of the code written inside the JSP declaration tag. Thus, memory is not obtained with every request.

Syntax: <%! field or method declaration %>

The method that returns the cube of a given integer is defined in the JSP declaration tag example below, and it is called from the JSP expression tag.

However, we can alternatively call the specified method using the JSP scriptlet tag.

> **Index.jsp**
>
> *<html>*
>
> *<body>*
>
> *<%! int cube(int n){ return n*n*n*; } %>*
>
> *<%= "Cube of 3 is:"+cube(3) %>*
>
> *</body>*
>
> *</html>*

That concludes our discussion of JSP scripting elements.

JSP Request and Response Objects

Every time a JSP request is made, the web container creates an implicit object of type HttpServletRequest.

It can be used to obtain request parameters, header data, remote address, server name, server port, content type, character encoding, and other information.

Additionally, it may be used to add, remove, and set characteristics within the scope of the JSP request.

> **index.html**
>
> *<form action="welcome.jsp">*
>
> *<input type="text" name="uname">*
>
> *<input type="submit" value="go">*
>
> *</form>*

welcome.jsp

<% String name=request.getParameter("uname"); print("welcome "+name); %>

JSP response implicit object: An implicit object of type HttpServletResponse is the response in JSP. The web container creates an instance of HttpServletResponse for every JSP request.

It can be utilized to add or modify responses, send errors, redirect responses to different resources, and so forth.

Example: index.html

> *<form action="welcome.jsp">*
>
> *<input type="text" name="uname">*
>
> *<input type="submit" value="go">*
>
> *</form>*

welcome.jsp

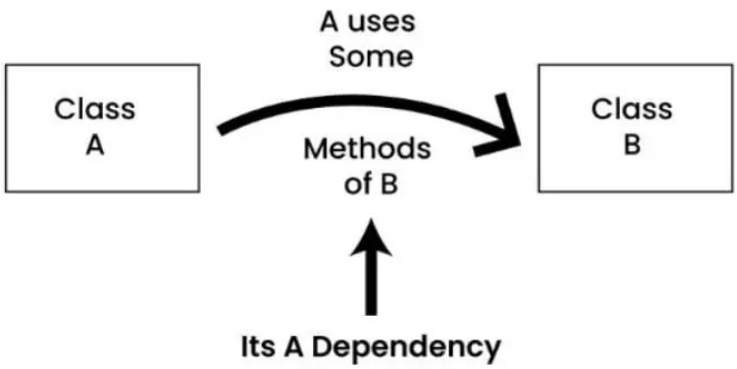
<% sendRedirect("http://www.google.com"); %>

Advanced Java Training

Design Pattern: Dependency Injection (DI)

Dependency injection is a highly prevalent design pattern. It's categorized as software design.

It is a program design technique that tries to separate objects from their dependencies. Objects obtain dependencies from outside sources rather than generating them internally.



Advanced java tutorial

Roles of Dependency Injection

When a class uses dependency injection, its dependencies are not created or managed internally; instead, they are injected from the outside. Four primary roles play this pattern:

- Client
- Server
- Injector
- Interface

Client: The class or component that is dependent on another class or module's services is known as the client. It depends on outside sources to supply the dependencies; it does not generate or manage the dependencies itself.

Service: A server is a class or component that offers a specific function or service that the customer needs. It is focused on delivering particular functionality and is made to be independent of the clients.

Injector: Injecting instances of services into the client is the injector's responsibility. It offers the required services during runtime and is aware of the client's requirements.

Interface: An interface is a contract or collection of procedures that a service needs to follow. As clients rely on these interfaces instead of particular implementations, flexibility and the capacity to switch implementations are encouraged.

Use Cases of Dependency Injection

Due to the ease of injecting or replacing dependencies, it is possible to add new features or modify current ones without requiring considerable code rework. The following are the main situations in which dependency injection is a useful strategy:

- For loose coupling and recyclability.
- Add test duplicates or mocks for dependencies.
- To maintain and set up frameworks.
- Organize intricate dependency structures and facilitate simpler expansion and scaling.
- integrating related issues to prevent code duplication and encourage a unified strategy.

Example

Code without Dependency Injection

```
public class NotificationService {

  private EmailProvider emailProvider = new EmailProvider(); // Tightly coupled to email

 public void sendNotification(String message, String recipient) {

    emailProvider.sendEmail(message, recipient);

 } }
```

Complexities in the above code are as follows:

Tight Coupling: Changing providers without modifying the code is challenging because of the NotificationService's close coupling to the EmailProvider.

Testability: Because NotificationService uses EmailProvider directly, it is difficult to test it in isolation.

Code with Dependency Injection

```
public interface NotificationProvider {
    void sendNotification(String message, String recipient);
}
public class EmailProvider implements NotificationProvider {
    @Override
    public void sendNotification(String message, String recipient) {
        // Send email logic
    }
}
public class SMSProvider implements NotificationProvider {
    @Override
    public void sendNotification(String message, String recipient) {
        // Send SMS logic
    }
}
public class NotificationService {
    private NotificationProvider notificationProvider;
    public NotificationService(NotificationProvider notificationProvider) { // Inject dependency
        this.notificationProvider = notificationProvider;
    }
    public void sendNotification(String message, String recipient) {
        notificationProvider.sendNotification(message, recipient);
    }
}
```

Benefits of Dependency Injection

There are numerous advantages that dependency injection can provide for your program's development. Here are a few main benefits:

- Increased modularity and maintainability.

- Improved testability.
- Reduced coupling and improved loose coupling.
- Easy Collaboration and Reusability.
- Encourages loose coupling and promotes dependency management.

Limitations of Dependency Injection

The following are some major drawbacks of dependency injection:

- Enhanced Intricacy
- Runtime Mistakes
- Performance and Overhead
- Evaluate Dependency Injection on Its Own
- Debugging Obstacles
- Decreased Transparency and Enhanced Abstraction

Microservices

A software application can be developed using the microservices architectural style as a group of discrete, autonomous services that interact with one another over a network.

Microservices divide an application into more manageable, loosely connected services as opposed to creating a monolithic application with every feature firmly integrated into a single codebase.

How do microservices work?

Microservices operate similarly to building a city out of modular, interconnected components, disassembling a large, complex application into smaller, autonomous parts that communicate and cooperate, offering flexibility, scalability, and simpler maintenance.

Java AWT

An API for creating Java window-based applications, or GUIs (Graphic User Interfaces), is called Java AWT, or Abstract Window Toolkit.

Platform-independent graphical application development is made possible via Java AWT, which is a component of the Java Foundation Classes (JFC).

For the following reasons, AWT remains platform-independent even after its components become platform-dependent:

Java Virtual Machine, or JVM: Because Java Virtual Machine depends on platforms

Abstract APIs: AWD gives the GUI an abstract layer. Java apps use

the platform-neutral Abstract API to communicate with AWT. Java's abstract API makes it possible to separate platform-specific information, enabling cross-system code portability.

Platform-Independent Libraries: The AWT libraries are completely platform-independent and are developed in Java. It guarantees that AWT functionality is consistent across many settings as a result.

Example

```
import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
// Driver Class
public class AWT_Example {
// main function
   public static void main(String[] args)
   {
      // Declaring a Frame and Label
      Frame frame = new Frame("Basic Program");
      Label label = new Label("Hello World!");
// Aligning the label to CENTER
      label.setAlignment(Label.CENTER);
// Adding Label and Setting
// the Size of the Frame
      frame.add(label);
      frame.setSize(300, 300);
// Making the Frame visible
      frame.setVisible(true);
// Using WindowListener for closing the window
      frame.addWindowListener(new WindowAdapter() {
         @Override
         public void windowClosing(WindowEvent e)
         {
            System.exit(0);
         }
      });
   }
```

```
}
```

Conclusion

Enjoy creating complex applications with our advanced Java tutorial. Learn comprehensively with hands-on exposure to real-time projects through our **Advanced Java Training in Chennai.**

Share on your Social Media

Softlogic Academy

Softlogic Systems

KK Nagar [Corporate Office]

No.10, PT Rajan Salai, K.K. Nagar, Chennai – 600 078.
Landmark: Karnataka Bank Building
Phone: +91 86818 84318
Email: enquiry@softlogicsys.in
Map: Google Maps Link

OMR

No. E1-A10, RTS Food Street
92, Rajiv Gandhi Salai (OMR),
Navalur, Chennai - 600 130.
Landmark: Adj. to AGS Cinemas
Phone: +91 89256 88858
Email: info@softlogicsys.in
Map: Google Maps Link

- Data Science
- Embedded
- Cloud Computing
- Hardware and Networking
- VBA Macros
- Mobile App Development
- DevOps

- Glassdoor
- Mouthshut
- Sulekha
- Justdial
- Ambitionbox
- Indeed
- Software Suggest
- Sitejabber