**IOC (Inversion of Control) Container in Java**

**Inversion of Control (IoC)** is a design principle used in Java to achieve **loose coupling** between components. Instead of creating dependencies manually (new keyword), we delegate the control of object creation and lifecycle to a **container**.

The **IoC Container** manages dependencies and injects them wherever required. The most commonly used IoC containers in Java are:

1. **Spring IoC Container** (ApplicationContext, BeanFactory)
2. **Google Guice**
3. **Dagger (for Android and Java applications)**
4. **Jakarta CDI (Contexts and Dependency Injection)**

**🔹 1. IoC Container in Spring Framework**

Spring provides two types of IoC containers:

* **BeanFactory** (Basic container with lazy initialization)
* **ApplicationContext** (Advanced container with event handling, eager initialization, etc.)

**✅ Example: IoC with Spring @Component and @Autowired**

java

CopyEdit

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.stereotype.Component;

import org.springframework.context.ApplicationContext;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

// Service Interface

interface MessageService {

void sendMessage(String message);

}

// Implementation of Service

@Component

class EmailService implements MessageService {

public void sendMessage(String message) {

System.out.println("Email sent: " + message);

}

}

// Client Component (Dependent Class)

@Component

class NotificationSender {

private final MessageService messageService;

@Autowired // Spring injects EmailService automatically

public NotificationSender(MessageService messageService) {

this.messageService = messageService;

}

public void send(String msg) {

messageService.sendMessage(msg);

}

}

// Configuration and Running IoC

@Configuration

@ComponentScan("com.example") // Scan components in the package

class AppConfig {}

public class IoCExample {

public static void main(String[] args) {

// Create Spring IoC Container

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

// Get Bean and Use Service

NotificationSender sender = context.getBean(NotificationSender.class);

sender.send("Hello, IoC!");

}

}

🔹 **Explanation:**

* @Component marks EmailService and NotificationSender as Spring-managed beans.
* @Autowired injects EmailService into NotificationSender automatically.
* ApplicationContext creates and manages the beans.

**🔹 2. Manual IoC without Spring (Factory Pattern)**

Before Spring, **Factory Pattern** was used for dependency injection.

java

CopyEdit

// Service Interface

interface MessageService {

void sendMessage(String message);

}

// Implementation

class SMSService implements MessageService {

public void sendMessage(String message) {

System.out.println("SMS sent: " + message);

}

}

// Factory Class (IoC Container)

class ServiceFactory {

public static MessageService getMessageService() {

return new SMSService(); // Loose coupling

}

}

// Client

public class ManualIoC {

public static void main(String[] args) {

MessageService service = ServiceFactory.getMessageService(); // IoC

service.sendMessage("Hello, Factory IoC!");

}

}

🔹 **Drawback:** Manual IoC requires explicit factory classes and does not handle lifecycle management.

**🔹 3. IoC with Google Guice**

Google Guice is another IoC container that uses @Inject.

java

CopyEdit

import com.google.inject.\*;

// Interface

interface Database {

void connect();

}

// Implementation

class MySQLDatabase implements Database {

public void connect() {

System.out.println("Connected to MySQL");

}

}

// Dependency Module

class DatabaseModule extends AbstractModule {

@Override

protected void configure() {

bind(Database.class).to(MySQLDatabase.class);

}

}

// Client

class App {

private final Database db;

@Inject

public App(Database db) {

this.db = db;

}

void start() {

db.connect();

}

}

// Main Method

public class GuiceIoC {

public static void main(String[] args) {

Injector injector = Guice.createInjector(new DatabaseModule());

App app = injector.getInstance(App.class);

app.start();

}

}

🔹 **Google Guice handles IoC efficiently without XML or Spring context.**

**🔹 Summary**

| **IoC Container** | **Features** |
| --- | --- |
| **Spring IoC** | Most popular, supports @Autowired, XML, and Java Config. Handles lifecycle and AOP. |
| **Google Guice** | Lightweight, annotation-based, no XML needed. |
| **Dagger** | Used for dependency injection in Android and Java. |
| **Jakarta CDI** | Java EE standard for dependency injection. |

**When to Use Spring?**

* If you need **complete application management** with transaction support, AOP, security, etc.

**When to Use Guice?**

* If you want **lightweight** dependency injection without additional frameworks.

**🚀 Conclusion:** IoC **removes dependency creation responsibility from objects** and shifts it to a container. Spring IoC is widely used in **enterprise Java applications**, while **Guice and Dagger** are used for lightweight applications.