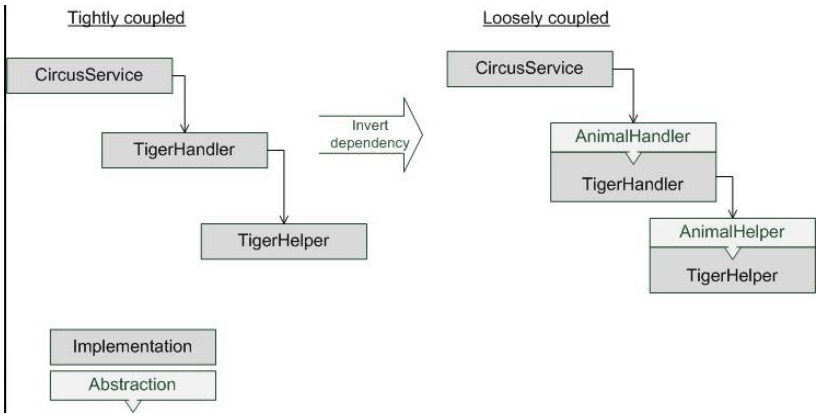
03: ♥♦ Spring DIP, DI & IoC in detail interview Q&As

Posted on August 25, 2014 by Arulkumaran Kumaraswamipillai

Video: Spring DIP, DI, and IoC

Q1. What do you understand by the terms Dependency Inversion Principle (**DIP**), Dependency Injection (**DI**) and Inversion of Control (**IoC**) container?

A1. **Dependency Inversion Principle (DIP)** is one of the 6 OO design principles abbreviated as “SOLID”, and is in some ways related to the Dependency Injection (DI) pattern. The idea of DIP is that higher layers of your application should not directly depend on lower layers. Dependency Inversion Principle does not imply Dependency Injection. This principle doesn’t say anything about how higher layers know what lower layer to use. This could be done as shown below by coding to interface using a factory pattern or through Dependency Injection by using an IoC container like Spring framework, Pico container, Guice, Apache HiveMind, and JEE 6.



DIP (Dependency Inversion Principle)

The **Dependency Inversion Principle (DIP)** states that

– High level modules should not depend upon low level modules. Both should depend upon abstractions.
– Abstractions should not depend upon details. Details should depend upon abstractions.

When this principle is applied, the higher level classes will not be working directly with the lower level classes, but with an abstract layer (i.e. an abstract class or an interface). This gives us the flexibility at the cost of increased effort. The "CircusService" dependes on the interface "AnimalHandler" and not on the concrete implementation "Tigerhandler". The implementation can be easily swapped to the"LionHandler" as long as it implements the interface "AnimalHandler".

#1. Tightly coupled "CircusService"

Helper implementation

```
1 public class TigerHelper {
2
3   public void help() {
4     System.out.println("TigerHelper in action")
5   }
6 }
7
```

Handler implementation

16 Technical Key Areas

80+ step by step Java Tutorials

```
1  public  class  TigerHandler  {
2
3    private  TigerHelper  helper;
4
5    public  void  handle()  {
6      System.out.println("TigerHandler  in  action
7      helper.help();
8    }
9
10   public  void  setHelper(TigerHelper  helper)
11     this.helper  =  helper;
12   }
13 }
14
```

Service implementation

```
1  public  class  CircusService  {
2
3    private  TigerHandler  handler;
4
5    public  void  setHandler(TigerHandler  handler
6      this.handler  =  handler;
7    }
8
9    public  void  process()  {
10     handler.handle();
11   }
12
13   public  static  void  main(String[]  args)  {
14     CircusService  service  =  new  CircusServic
15
16     //wire  up  dependencies
17     TigerHelper  helper  =  new  TigerHelper();
18     TigerHandler  handler  =  new  TigerHandler(
19
20     handler.setHelper(helper);
21     service.setHandler(handler);
22
23     //start
24     service.process();
25   }
26 }
```

If you want CircusService to work with a TigerHandler and a LionHelper (instead of TigerHelper), you need to modify the “TigerHandler” class as it depends directly on the “TigerHelper” implementation. You need to change in “TigerHandler”

1) FROM: private TigerHelper helper; **TO:** private LionHelper helper;
2) FROM: public void setHelper(TigerHelper helper) {…} **TO:** public void setHelper(LionHelper helper) {…} ;

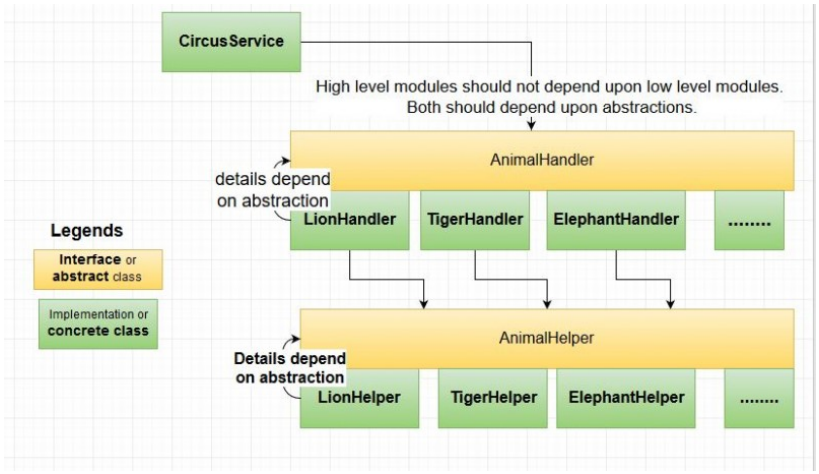
Output:

TigerHandler in action
TigerHelper in action

As TigerHandler and TigerHelper are tightly coupled. If you apply the DIP (i.e. coding to interface), you don't have to make any changes to "TigerHandler" to use "LionHelper" with it as shown below.

#2. Loosely coupled "CircusService"

by "**coding to interface**" as shown below.



DIP – Dependency Inversion Principle

Define interfaces

```
1 public  interface  AnimalHelper  {
2         abstract  void  help();
3 }
```

```
1 public  interface  AnimalHandler  {
2         abstract  void  handle();
3 }
4
```

Helper implementation coded to interface

```
1 public class TigerHelper implements AnimalHel
2
3   @Override
4   public void help() {
5     System.out.println("TigerHelper in action")
6   }
7 }
```

```
1 public class LionHelper implements AnimalHelp
2
3   @Override
4   public void help() {
5     System.out.println("LionHelper in action");
6   }
7 }
```

Handler implementation coded to interface

```
1  public class TigerHandler implements AnimalH
2
3    private AnimalHelper helper;
4
5    @Override
6    public void handle() {
7      System.out.println("TigerHandler in action
8      helper.help();
9    }
10
11   public void setHelper(AnimalHelper helper)
12     this.helper = helper;
13   }
14 }
15
```

```
1  public class LionHandler implements AnimalHa
2
3    private AnimalHelper helper;
4
5    @Override
6    public void handle() {
7      System.out.println("LionHandler in action"
8      helper.help();
9    }
10
11   public void setHelper(AnimalHelper helper)
12     this.helper = helper;
13   }
14 }
```

Service implementation that can swap helpers without having to modify the handler

```
1  public class CircusService {
2
3    private AnimalHandler handler;
```

```
4
5    public void setHandler(AnimalHandler handle
6      this.handler = handler;
7    }
8
9    public void process() {
10     handler.handle();
11   }
12 }
```

Now the newly created Client class that work with any combination of AnimalHelper and AnimalHandler without needing to modify the actual implementations.

```
1  public class Client {
2
3    public static void main(String[] args) {
4      CircusService service = new CircusServic
5
6      // wire up dependencies for tiger
7      TigerHelper helper = new TigerHelper();
8      TigerHandler handler = new TigerHandler(
9
10     handler.setHelper(helper);
11     service.setHandler(handler);
12
13     // start
14     service.process();
15
16     // wire up dependencies for a TigerHand
17     LionHelper helper2 = new LionHelper();
18
19     handler.setHelper(helper2);
20     service.setHandler(handler);
21
22     // start
23     service.process();
24   }
25 }
```

Output:

TigerHandler in action
TigerHelper in action
TigerHandler in action
LionHelper in action

#3. Singleton factory classes to wire up dependencies

If you have multiple client or callee classes of AnimalHandler, if you change AnimalHandler and AnimalHelper implementations, then all the callee classes need to be

modified. This is tight coupling of Clients (or callees) with the **AnimalHandler** class. This can be improved with the help of "**singleton**" **factory** classes as shown below.

```
/**
 *
 *  Singleton  factory  class
 */
public  final  class  AnimalHelperFactory  {

  private  static  AnimalHelper  instance  =  nu

  private  AnimalHelperFactory(){};

  public  synchronized  AnimalHelper  getInstanc

    if(instance  ==  null)  {
      instance  =  new  LionHelper();  //  or  T
    }

    return  instance;
  }

}
```

```
/**
 *
 *  Singleton  factory  class
 */
public  final  class  AnimalHandlerFactory  {

  private   AnimalHandler  instance  =  null;

  private  AnimalHandlerFactory(){};

  public  static  synchronized  AnimalHandler  g

    if(instance  ==  null)  {
      instance  =  new  TigerHandler();  //  or
      ((TigerHandler)instance).setHelper(helper)
    }

    return  instance;
  }

}
```

Revised Client using the factory classes

```
public  class  Client  {

  public  static  void  main(String[]  args)  {
    CircusService  service  =  new  CircusServic

    //  wire  up  dependencies  for  tiger
    AnimalHelper  helper  =  AnimalHelperFactory
    AnimalHandler  handler  =  AnimalHandlerFact
```

```
 9
10      service.setHandler(handler);
11
12      //  start
13      service.process();
14    }
15 }
```

Now, if any dependencies change like TigerHandler depending on TigerHelper, etc, change it only in the factories, and not in all the callees. But, a typical commercial project will have 300+ artifacts, and you will need to create too many singleton factory classes. This is where the DI (Dependency Injection) becomes useful to wireup dependnecy via an IoC (Inversion of Control) frameworks like Spring DI, Guice, CDI, etc as explained below.

Dependency Injection (DI) is a pattern of injecting a class's dependencies into it at runtime. This is achieved by defining the dependencies as interfaces, and then injecting in a concrete class implementing that interface to the constructor. This allows you to swap in different implementations without having to modify the main class. The Dependency Injection pattern also promotes high cohesion by promoting the Single Responsibility Principle (**SRP**), since your dependencies are individual objects which perform discrete specialized tasks like data access (via DAOs) and business services (via Service and Delegate classes).

#4. Spring as the IoC container to wire up dependencies via DI

Spring Ioc inverts the "flow of control".

```
1 <bean id="helper" class="LionHelper"/>
2
3 <bean id="handler" class="TigerHandler">
4     <!-- setter injection of helper into handler
5     <property name="helper" ref="helper" />
6 </bean>
7
```

The helper is injected into handler via setter injection in Spring.

The **Inversion of Control Container (IoC)** is a container that supports Dependency Injection. In this you use a central container like Spring framework, Guice, or HiveMind, which defines what concrete classes should be used for what dependencies throughout your application. This brings in an added flexibility through looser coupling, and it makes it much easier to change what dependencies are used on the fly. The basic concept of the Inversion of Control pattern is that you do not create your objects but describe how they should be created. You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file or via annotations. A container (in the case of the Spring framework, the IOC container) is then responsible for hooking it all up. Applying IoC, objects are given their dependencies at creation time by some external entity that coordinates each object in the system. That is, dependencies are injected into objects. So, IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

The real power of DI and IoC is realized in its ability to replace the **compile time** binding of the relationships between classes with binding those relationships at **runtime**. For example, in Seam framework, you can have a real and mock implementation of an interface, and at runtime decide which one to use based on a property, presence of another file, or some precedence values. This is incredibly useful if you think you may need to modify the way your application behaves in different scenarios. Another real benefit of DI and IoC is that it makes your code easier to unit test. There are other benefits like promoting looser coupling without any proliferation of factory and singleton design patterns, follows a consistent approach for lesser experienced developers to follow, etc. These benefits can come in at the cost of the added complexity to your application and has to be carefully manged by using them only at the right places where the real benefits are realized, and not just using them because many others are using them.

Note: The CDI (Contexts and Dependency Injection) is standard on Dependency Injection. CDI is a part of the Java EE 6 stack, meaning an application running in a Java EE 6 compatible container can leverage CDI out-of-the-box. Weld is the reference implementation of CDI.

Q2. In your experience, why would you use Spring framework?
A2. Spring framework has been very popular and been filling the gap in the Java EE stack for a number of years now. Java EE has finally caught up with DI, AOP, batch jobs, Managed Beans, etc promoting POJO driven development with JEE 6 and 7, hence for new Java EE projects JEE 6 or 7 might be the way to go. If you are after enhancing the current projects developed in Spring, or you are in more favor of Spring then this framewok has been popular for a number of reasons compared to pre JEE 6.

1) A key design principle in Spring in general is the "**Open for extension, closed for modification**" principle.So, some of the methods in the core classes are marked "final".

2) Spring is an IoC container that supports both constructor injection (supplying arguments to constructors) and setter-based injection (calling setters on a call) to get the benefits of Dependency Injection (DI) like loose coupling, easier to test, etc.

3) It is a light-weight framework with POJOs.

4) It is modular with spring core, spring batch, spring mvc, spring orm, etc.

5) It is very matured and has been used in many large Java/JEE applications.

6) It supports AOP to implement cross cutting concerns.

7) It supports transaction management.

8) It promotes uniformity to handle exceptions (e.g. checked vs unchecked) and integration with JMS, JDBC, JNDI, remoting, etc.

9) Closes the JDBC/JMS connection resources automatically without having to you write verbose try/catch/finally blocks with logic to close the resources.

Q3. In your experience, what do you don't like about Spring? Are there any pitfalls?
A3.

1) Spring has become very huge and bulky. So, don't over do it by using all its features because of the hype that Spring is good. Look at what parts of Spring really provides some benefits for your project and use those parts. In most cases, it is much better to use proven frameworks like Spring than create your own equivalent solution from a maintenance and applying the best practices perspective.

2) Spring MVC is probably not the best Web framework. There are other alternatives like angular js, Struts 2, and Wicket. Having said this, Spring integrates well with the other Web frameworks like Struts, JSF, etc.

3) The XML files can get bloated. This can be minimized by carefully considering other options like annotations and having separate XML configuration files. **JavaConfig** based configuration is an alternative.

Top 20+ Spring Interview Questions & Answers:

6 FAQ Spring interview questions and answers | 17 Spring FAQ interview Questions & Answers

Top 30+ Hibernate Interview Questions & Answers:

30+ FAQ Hibernate interview questions & answers

Popular Member Posts

♦ 11 Spring boot interview questions & answers

904 views

♦ Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers

816 views

001A: ♦ 7+ Java integration styles & patterns interview questions & answers

427 views

18 Java scenarios based interview Questions and Answers

408 views

♦ 7 Java debugging interview questions & answers

322 views

01b: ♦ 13 Spring basics Q8 – Q13 interview questions & answers

311 views

01: ♦ 15 Ice breaker questions asked 90% of the time in Java job interviews with hints

303 views

♦ 10 ERD (Entity-Relationship Diagrams) Interview Questions and Answers

301 views

♦ Q24-Q36: Top 50+ Core on Java classes, interfaces and generics interview questions & answers

251 views

001B: ♦ Java architecture & design concepts interview questions & answers

209 views

Bio | Latest Posts

Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

About Arulkumaran Kumaraswamipillai

Mechanical Eng to freelance Java developer in 3 yrs. Contracting since 2003, and attended 150+ Java job interviews, and often got 4 - 7 job offers to choose from. It pays to prepare. So, published Java interview Q&A books via Amazon.com in 2005, and sold 35,000+ copies. Books are outdated and replaced with this subscription based site.**945+** paid members. join my LinkedIn Group. **Reviews**

‹ ♦ 18 Agile Development interview Q&A for Java developers

04 ♦ 17 Spring bean life cycles & scopes FAQ interview Questions & Answers ›

Posted in Spring

Tags: Free Content, Java/JEE FAQs, JEE FAQs, Popular frameworks

Non-Technical Know Hows

☀ 6 Aspects that can motivate you to fast-track your career & go places ☀ Are you reinventing yourself as a Java developer? ☀ 8 tips to safeguard your Java career against offshoring ☀ My top 5 career mistakes

Prepare to succeed

☀ Turn readers of your Java CV go from “Blah blah” to “Wow”? ☀ How to prepare for Java job interviews? ☀ 16 Technical Key Areas ☀ How to choose from multiple Java job offers?

Select Category

© Disclaimer

 ↑