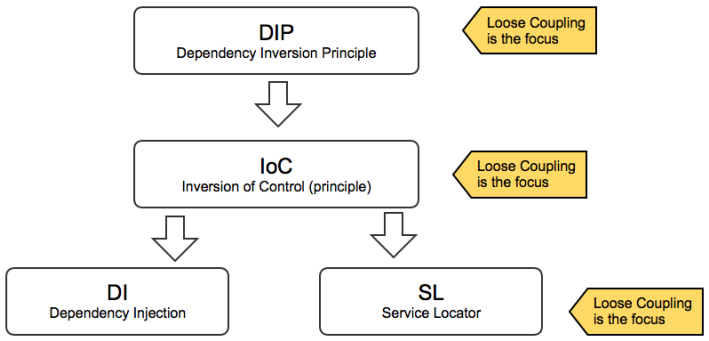
01: ♥♦ 13 Spring basics Q1 – Q7 interview questions & answers

Posted on December 2, 2014 by Arulkumaran Kumaraswamipillai — 3 Comments ↓

Spring interview questions are very popular with the job interviewers along with the Hibernate interview questions & answers.

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

A1. The differences are very subtle and can be hard to understand. Hence, explained via code samples.



DIP, DI & IoC

1) Dependency Inversion Principle (**DIP**): is one of the 6 OO design principles abbreviated as "SOLID", and "D" stands for DIP meaning that we should always only rely on interfaces and not on their implementations. The idea of DIP is that higher layers of your application should not directly depend on lower layers. DIP is the principle that guides us towards DI pattern. You will see in the example below that the higher layer module "MyServiceImpl" depends on the lower layer module interface "**Processor**" and NOT on the implementations "XmlProcessor" & "JsonProcessor". This is commented on the code shown below as "// code to interface" for the understanding.

2) Dependency Injection (**DI**): is a design pattern where instead of having your objects create a dependency or asking a factory object to make one for you, you pass the needed dependencies into the constructor or via setters from outside the class. This is achieved by defining the dependencies as interfaces, and then injecting in a concrete class implementing that interface via a constructor (i.e. constructor injection) or a setter method (i.e. setter injection). Dependency Injection is a design pattern that allows us to write loosely coupled code for better maintainability,

3) Inversion of Control (**IoC**): is a software design principle where the framework controls the program flow. Spring framework, Guice, etc are IoC containers that implement the IoC principle.

16 Technical Key Areas

80+ step by step Java Tutorials

Still not too clear?

Try Spring DIP, DI & IoC in detail with diagrams.

Q2. What are you “Inverting” in IoC?
A2. **Flow of control** is “inverted” by dependency injection because you are effectively delegating dependencies to some external system (e.g. IoC container or Service Locator) .

Q3. What are the different implementation patterns of IoC principle?
A3. The two **implementation patterns** of the IoC design principles are

1. Dependency Injection (**DI**) pattern: A class is given it’s dependencies from outside like **Spring IoC** or **JEE 7+** container. It neither knows, nor cares where the dependencies are coming from.
2. Service Locator (**SL**) pattern: A class is still responsible for creating its dependencies. It just uses the service locator to do it.

Here is a **DI** example with Spring **IoC** container…

Interface Processor

```
1 package com.mytutorial;
2
3 public interface Processor {
4     <T> T process();
5 }
```

Spring Configuration AppConfig

```
1  package com.mytutorial;
2
3  import org.springframework.context.annotation.Be
4  import org.springframework.context.annotation.Co
5  import org.springframework.context.annotation.Co
6
7  @Configuration
8  @ComponentScan("com.mytutorial")
9  public class AppConfig {
10
11     @Bean
```

open all | close all
Setting up Tutorial (6)
Tutorial - Diagnosis (2
Akka Tutorial (9)
Core Java Tutorials (2
Hadoop & Spark Tuto
JEE Tutorials (19)
Scala Tutorials (1)
Spring & HIbernate Tu
Tools Tutorials (19)
Other Tutorials (45)

```
12      MyServiceImpl myServiceImpl() {
13          return new MyServiceImpl();
14      }
15  }
16
```

Processor implementations “JsonProcessor” and “XmlProcessor”

```
1  package com.mytutorial;
2
3  import org.springframework.stereotype.Component;
4
5  @Component("JsonProcessor")
6  class JsonProcessor implements Processor {
7
8      public <T> T process() {
9          //...TODO:
10         System.out.println("jsonProcessor.....")
11         return null;
12     }
13 }
```

```
1  package com.mytutorial;
2
3  import org.springframework.stereotype.Component;
4
5  @Component("XmlProcessor")
6  class XmlProcessor implements Processor {
7
8      public <T> T process() {
9          //... TODO:
10         System.out.println("xmlProcessor.....");
11         return null;
12     }
13 }
```

Service class MyServiceImpl

```
1  package com.mytutorial;
2
3  import org.springframework.beans.factory.annotat
4  import org.springframework.beans.factory.annotat
5
6  class MyServiceImpl {
7
8      @Autowired
9      @Qualifier("XmlProcessor")
10     Processor xmlProcessor; // code to interface
11
12     @Autowired
13     @Qualifier("JsonProcessor")
14     Processor jsonProcessor;  //code to interfac
15
```

```
16      public void processXml() {
17          xmlProcessor.process();
18          // ....
19      }
20
21      public void processJson() {
22          jsonProcessor.process();
23          // ....
24      }
25  }
26
27
```

Standalone App to execute

```
1  package com.mytutorial;
2
3  import org.springframework.context.annotation.An
4
5  public class App
6  {
7      public static void main( String[] args )
8      {
9          AnnotationConfigApplicationContext ctx =
10         ctx.register(AppConfig.class);
11         ctx.refresh();
12         MyServiceImpl bean = ctx.getBean(MyServi
13         bean.processXml();
14         bean.processJson();
15         ctx.close();
16     }
17 }
18
```

Output:

```
1
2 xmlProcessor.....
3 jsonProcessor....
4
```

Service Locator (SL) type IoC example:

Not a common IoC pattern. Very rarely used. Same code as above can be modified to use a **Service Locator**.

Service Locator "ProcessorServiceLocatorFactory"

```
package com.mytutorial;

import org.springframework.stereotype.Service;

@Service
public class ProcessorServiceLocatorFactory {

    public Processor getProcessor(String process
        //lookup dynamically via JNDI or other M
        if("XmlProcessor".equalsIgnoreCase(proce
            return new XmlProcessor();
        } else {
            return new JsonProcessor();
        }
    }
}

```

Modified “MyServiceImpl” to use the Service Locator

```

package com.mytutorial;

import org.springframework.beans.factory.annotat

class MyServiceImpl {

    @Autowired
    ProcessorServiceLocatorFactory locatorServic

    public void processXml() {
        Processor processor = locatorService.get
        processor.process();
        // ....
    }

    public void processJson() {
        Processor processor = locatorService.get
        processor.process();
        // ....
    }
}

```

The core of the Spring Framework is its Inversion of Control (Ioc) container. The Spring IoC container manages Java objects from their instantiation to destruction via its BeanFactory. Java components that are instantiated by the IoC container are called beans, and the IoC container manages a bean’s scope (e.g. prototype vs singleton), lifecycle events (e.g. initialization, method callbacks & shutdown), and any AOP (Aspect Oriented Programming) features if configured.

The key focus of both types of IoC is to loosely couple dependencies among components like MyApp, MyService, and Procesor as per the above examples.

Q4. What are the different types of dependency injections?
A4. There are 4 types of dependency injection. Spring supports 3 types. 1, 2 & 4 shown below.

1) Constructor Injection (e.g. Spring): Dependencies are provided as **constructor parameters**.

```
1
2  package com.mytutorial;
3
4  import org.springframework.beans.factory.annotat
5  import org.springframework.beans.factory.annotat
6
7  class MyServiceImpl {
8
9      private final Processor xmlProcessor;
10     private final Processor jsonProcessor;
11
12     @Autowired
13     public MyServiceImpl(@Qualifier("XmlProcesso
14             @Qualifier("JsonProcessor") Processo
15         super();
16         this.xmlProcessor = xmlProcessor;
17         this.jsonProcessor = jsonProcessor;
18     }
19
20     public void processXml() {
21         xmlProcessor.process();
22         // ....
23     }
24
25     public void processJson() {
26         jsonProcessor.process();
27         // ....
28     }
29 }
30
```

2) Setter Injection (e.g. Spring): Dependencies are assigned through **setter methods**.

```
1
2  package com.mytutorial;
3
4  import org.springframework.beans.factory.annotat
5  import org.springframework.beans.factory.annotat
6
7  class MyServiceImpl {
8
```

```
 9      private Processor xmlProcessor;
10      private Processor jsonProcessor;
11
12      public void processXml() {
13          xmlProcessor.process();
14          // ....
15      }
16
17      public void processJson() {
18          jsonProcessor.process();
19          // ....
20      }
21
22      @Autowired
23      @Qualifier("XmlProcessor")
24      public void setXmlProcessor(Processor xmlPro
25          this.xmlProcessor = xmlProcessor;
26      }
27
28      @Autowired
29      @Qualifier("JsonProcessor")
30      public void setJsonProcessor(Processor jsonP
31          this.jsonProcessor = jsonProcessor;
32      }
33  }
34
```

3) Interface Injection (e.g. Avalon): Injection is done through an interface.

4) Field injection: Using annotations on fields and parameters.

```
1
2   class MyServiceImpl {
3
4       @Autowired
5       @Qualifier("XmlProcessor")
6       Processor xmlProcessor;
7
8
9       public void processXml() {
10              xmlProcessor.process();
11              // ....
12      }
13  }
14
15
```

Spring supports **1)** Constructor Injection, **2)** Setter Injection & **4)** Field injection with annotations.

Q5. Which ones are the most commonly used DIs?

A5. **1)** Constructor Injection, **2)** Setter Injection & **4)** Field injection with annotations.

Q6. When will you favor DI type “Constructor Injection” over “Setter Injection”?
A6. Using constructor injection allows you to hide immutable fields from users of your class. Immutable classes don’t declare setter methods. This also enforces that you have the valid objects at the construction time. It also prompts you to rethink about your design when you have too many constructor parameters.

Q7. When will you favor DI type “Setter Injection” over “Constructor Injection”?
A7. In some scenarios, the constructors may get a lot of parameters, which force you to create a lot of overloaded constructors for every way the object might be created. In these scenarios setter injection can be favored over constructor injection, but having too many constructor parameters may be an indication of a bad design.

Q8 to Q13: 13 Spring basics Q8 – Q13 interview questions & answers.

More FAQ Spring Interview Questions & Answers:

1. 9 Spring Bean Scopes Interview Q&As

2. 17 Spring FAQ interview Questions & Answers

3. 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

407 views

♦ 7 Java debugging interview questions & answers

322 views

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

310 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**

‹ ► Java Beginner Interview Questions and Answers on the Object class Video

Event Driven Programming in Java Example – Part 1 ›

Posted in Spring, Spring Job Interview Essentials

Tags: JEE FAQs

3 comments on "01: ♥♦ 13 Spring basics Q1 – Q7 interview questions & answers"

indrjeetkumar says:

July 8, 2016 at 7:48 pm

Very Good explanation of the concept and cover the topic in very easy way

Keep the good job and helping the people through Video.

Reply

Nitin says:

June 25, 2016 at 10:07 am

Excellent article.

Reply

Thaya says:

April 17, 2016 at 5:37 pm

Great, Concepts are clearly explained.

Reply

Leave a Reply

Logged in as geethika. Log out?

Comment

Post Comment

Empowers you to open more doors, and fast-track

Technical Know Hows

☀ Java generics in no time ☀ Top 6 tips to transforming your thinking from OOP to FP ☀ How does a HashMap internally work? What is a hashing function?
☀ 10+ Java String class interview Q&As ☀ Java auto un/boxing benefits & caveats ☀ Top 11 slacknesses that can come back and bite you as an experienced Java developer or architect

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

 ↑