

Java-Success.com

Industrial strength Java/JEE Career Companion for those who want to go places

[Home](#)
[Java FAQs](#)
[600+ Java Q&As](#)
[Career](#)
[Tutorials](#)
[Member](#)
[Why?](#)
[Can u Debug?](#)
[Java 8 ready?](#)
[Top X](#)
[Productivity Tools](#)
[Judging Experience?](#)

[Home](#) › [Tech Key Areas](#) › [13 Technical Key Areas Interview Q&A](#) › [Design Concepts](#) › [Top 6 tips to go about writing loosely coupled Java applications](#)

Top 6 tips to go about writing loosely coupled Java applications

Posted on [August 11, 2014](#) by [Arulkumaran Kumaraswamipillai](#) — [No Comments](#) ↓

Q1. What is tight coupling?

A1. If class *OrderServiceImpl* relies on parts of class *PaymentServiceImpl* that are not part of class *PaymentServiceImpl*'s interface *PaymentService*, then the *OrderServiceImpl* and *PaymentServiceImpl* are said to be tightly coupled.

[9 tips to earn more](#) | [What can u do to go places?](#) | **945+** members. [LinkedIn Group](#). [Reviews](#)

600+ Full Stack Java/JEE Interview Q&As ♥Free ♦FAQs

[open all](#) | [close all](#)

[Ice Breaker Interview](#)

[Core Java Interview C](#)

[Java Overview \(4\)](#)

[Data types \(6\)](#)

[constructors-methc](#)

[Reserved Key Wor](#)

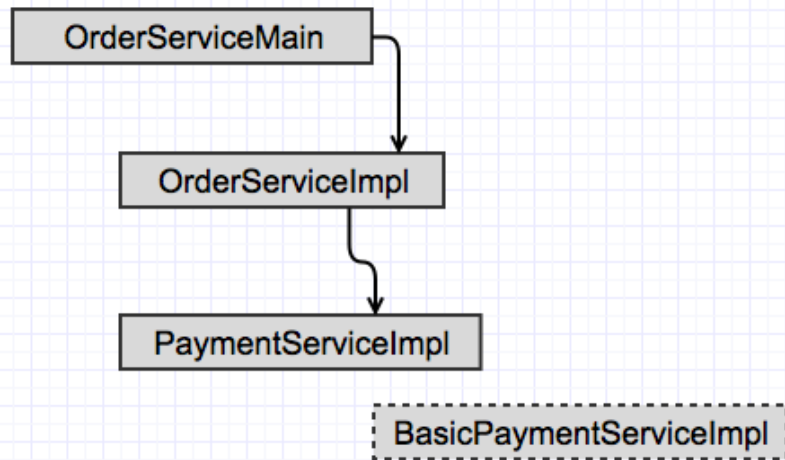
[Classes \(3\)](#)

[Objects \(8\)](#)

[OOP \(10\)](#)

[Design princip](#)

[30+ FAQ Java](#)



tightly coupled Java classes

In other words, *OrderServiceImpl* knows more than what it should about the way in which *PaymentServiceImpl* was implemented. If you want to change *PaymentServiceImpl* with a separate implementation *BasicPaymentServiceImpl*, then you need to modify the *OrderServiceImpl* class as well by changing *PaymentServiceImpl* to *BasicPaymentServiceImpl*.

Q2. What tips do you give to a junior developer about coupling or look for in reviewing others' code?

A2. Here are 6 tips to write loosely coupled Java applications.

Tip #1: Coding to interface will loosely couple classes.

Q3. What is loose coupling?

A3. If the only knowledge that class *OrderServiceImpl* has about class *PaymentServiceImpl*, is what class *PaymentServiceImpl* has exposed through its interface *PaymentService*, then class *OrderServiceImpl* and class *PaymentServiceImpl* are said to be loosely coupled. If you want to change *PaymentServiceImpl* with a separate implementations *BasicPaymentServiceImpl*, then you don't need to modify *OrderServiceImpl*. Change only *OrderServiceMain* from

- ◆ Why favor com
- 08: ◆ Write code
- Explain abstracti
- How to create a
- Top 5 OOPs tips
- Top 6 tips to go a
- Understanding C
- What are good r
- GC (2)
- Generics (5)
- FP (8)
- IO (7)
- Multithreading (12)
- Algorithms (5)
- Annotations (2)
- Collection and Data
- Differences Between
- Event Driven Progr
- Exceptions (2)
- Java 7 (2)
- Java 8 (24)
- JVM (6)
- Reactive Programn
- Swing & AWT (2)
- JEE Interview Q&A (3
- Pressed for time? Jav
- SQL, XML, UML, JSC
- Hadoop & BigData Int
- Java Architecture Inte
- Scala Interview Q&As
- Spring, Hibernate, & I
- Testing & Profiling/Sa
- Other Interview Q&A 1
- Free Java Interview

As a Java Architect

[Java architecture & design concepts](#)

```

1
2 PaymentService payService = new PaymentServiceIm
3

```

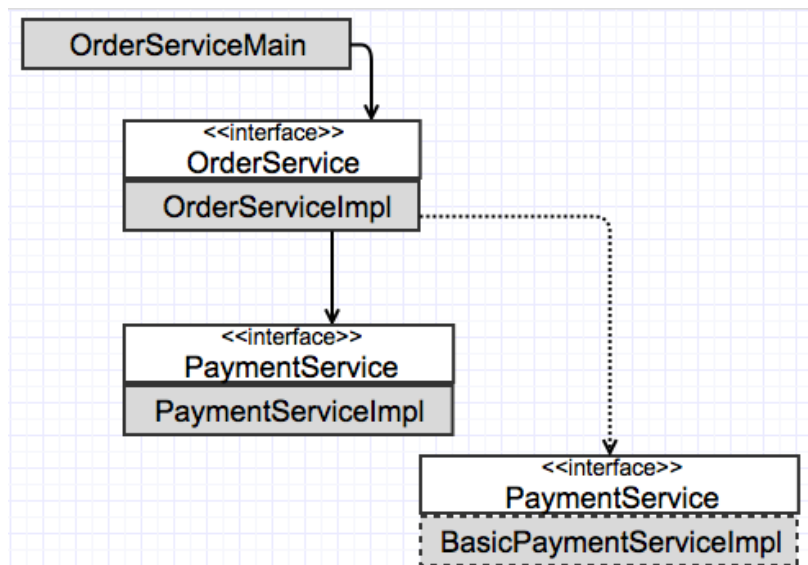
to

```

1
2 PaymentService payService = new BasicPaymentServ
3

```

This is what the **Dependency Inversion Principle (DIP)** states.



Loosely coupled by coding to interface

[interview Q&As with diagrams](#) | [What should be a typical Java EE architecture?](#)

Senior Java developers must have a good handle on

[open all](#) | [close all](#)

- [Best Practice \(6\)](#)
- [Coding \(26\)](#)
- [Concurrency \(6\)](#)
- [Design Concepts \(7\)](#)
- [Design Patterns \(11\)](#)
- [Exception Handling \(3\)](#)
- [Java Debugging \(21\)](#)
- [Judging Experience \(1\)](#)
- [Low Latency \(7\)](#)
- [Memory Management \(1\)](#)
- [Performance \(13\)](#)
- [QoS \(8\)](#)
- [Scalability \(4\)](#)
- [SDLC \(6\)](#)
- [Security \(13\)](#)
- [Transaction Management \(1\)](#)

Q4. Can the above classes be further improved in terms of coupling?

A4. Yes. Change the PaymentService method signature from

```

1
2 public abstract void handlePay(int accountNumber,
3

```

to

80+ step by step Java Tutorials

[open all](#) | [close all](#)

- [Setting up Tutorial \(6\)](#)
- [Tutorial - Diagnosis \(2\)](#)
- [Akka Tutorial \(9\)](#)
- [Core Java Tutorials \(2\)](#)
- [Hadoop & Spark Tuto](#)

```

1
2 public abstract void handlePay(PaymentDetail paym
3

```






Tip #2: Design method signatures carefully by avoiding long parameter lists.

As a rule, three parameters should be viewed as a practical maximum, and fewer is better (as recommended by Mr. Joshua Bloch.). This is not only from coupling perspective, but also in terms of readability and maintainability of your code.

It is likely that the `PaymentService` may need more parameters than account number and amount to process the payment. Every time you need to add a new parameter, your `PaymentService` interface method signature will change, and all other classes like `OrderService` that depends on `PaymentService` has to change as well to change its arguments to passed. But, if you create a value object like `PaymentDetail`, the method signature does not have to change. You add the new field to the `PaymentDetail` class.











Tip #3: Design patterns promote looser coupling

The *PaymentService* will not only be used by the *OrdersServiceMain*, but can be used by other classes like *RequestServiceMain*, *CancelServiceMain*, etc. So, if you want to change the actual implementation of *PaymentService* between *PaymentServiceImpl* and *BasicPaymentServiceImpl* without having to change *OrdersServiceMain*, *RequestServiceMain*, and *CancelServiceMain*, you can use the factory design pattern as shown by the *PaymentFactory* class. You only have to make a change to the *PaymentFactory* class to return the right *PaymentService* implementation.

-  [JEE Tutorials \(19\)](#)
-  [Scala Tutorials \(1\)](#)
-  [Spring & Hibernate T](#)
-  [Tools Tutorials \(19\)](#)
-  [Other Tutorials \(45\)](#)



Preparing for Java written & coding tests

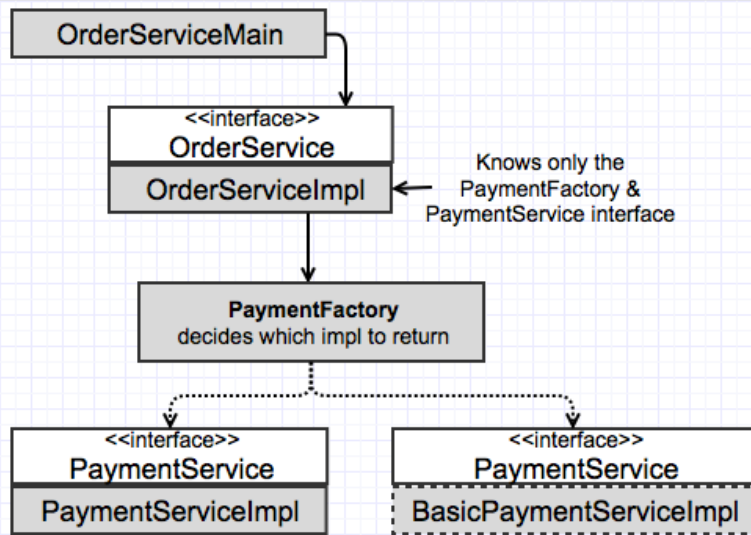
[open all](#) | [close all](#)

-  [♦ Complete the given](#)
-  [Can you write code? I](#)
-  [Converting from A to I](#)
-  [Designing your classe](#)
-  [Java Data Structures](#)
-  [Passing the unit tests](#)
-  [What is wrong with th](#)
-  [Writing Code Home A](#)
-  [Written Test Core Jav](#)
-  [Written Test JEE \(1\)](#)

How good are your...to go places?

[open all](#) | [close all](#)

-  [Career Making Know-](#)
-  [Job Hunting & Resum](#)



Design patterns promote loose coupling

```

1
2 package com.coupling;
3
4 import java.math.BigDecimal;
5
6 public class OrderServiceMain {
7
8     public static void main(String[] args) {
9         //loosely coupled as it knows only about the f
10        PaymentService payService = PaymentFactory.ge
11        OrderService orderService = new OrderServiceIm
12        orderService.process(12345, BigDecimal.valueOf
13    }
14 }
15

```

```

1
2 package com.coupling;
3
4 public final class PaymentFactory {
5
6     private static PaymentService instance = null;
7
8     private PaymentFactory(){}
9
10    public static PaymentService getPaymentService(
11        if(instance == null){
12            instance = new PaymentServiceImpl();
13        }
14
15    return instance;
16 }
17
18 }
19

```

Tip #4: Using Inversion of Control (IoC) Containers like Spring, Guice, etc.

Dependency Injection (DI) is a pattern of injecting a class's dependencies into it at run time. 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).

The Inversion of Control (IoC) container 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. 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.

For example, in Spring you will wire up the dependencies via an XML file:

```
1 <bean id="orderService" class="com.coupling.Ord
2 <constructor-arg ref="paymentService"/>
3 </bean>
4
5 <bean id="paymentService" class="com.coupling.Pay
```

You can also use annotations to inject dependencies. The `@Resource` annotation injects `PaymentService`.

```
1
2 package com.coupling;
3
4 import java.math.BigDecimal;
5 import javax.annotation.Resource;
6
7 public class OrderServiceImpl implements OrderSe
8
9 @Resource
10 PaymentService payService;
11
12 public OrderServiceImpl(PaymentService payServi
13     this.payService = payService;
14 }
15
16 public void process(int accountNumber, BigDecim
17     // some logic
18     payService.handlePay(new PaymentDetail(account
19     // some logic
20 }
21 }
22
23
```

Tip #5: High cohesion often correlates with loose coupling, and vice versa.

What is cohesion? Cohesion is the extent to which two or more parts of a system are related and how they work together to create something more valuable than the individual parts. You don't want a single class to perform all the functions (or concerns) like being a domain object, data access object, validator, and a service class with business logic. To create a more cohesive system from the higher and lower level perspectives, you need to break out the various needs into separate classes like `PaymentDetail`, `PaymentService`, `PaymentDao`, `PaymentValidator`, etc. Each class concentrates on one thing.

Coupling happens in between classes or modules, whereas cohesion happens within a class. So, think, tight encapsulation, loose (low) coupling, and high cohesion.

Tip #6: Favor composition over inheritance for code reuse

You will get a better abstraction with looser coupling with composition as composition is dynamic and takes place at run time compared to implementation inheritance, which is static, and happens at compile-time. The guide is that inheritance should be only used when subclass 'is a' super class. Don't use inheritance just to get code reuse. If there is no 'is a' relationship then use composition for code reuse. Overuse of implementation inheritance (uses the "extends" key word) can break all the subclasses, if the super class is modified. Do not use inheritance just to get polymorphism. If there is no 'is a' relationship and all you want is polymorphism then use interface inheritance with composition, which gives you code reuse. More elaborate explanation on this — Why favor composition over inheritance in Java OOP?

Popular Posts

♦ [11 Spring boot interview questions & answers](#)

861 views

♦ [Q11-Q23: Top 50+ Core on Java OOP Interview Questions & Answers](#)

829 views

[18 Java scenarios based interview Questions and Answers](#)

448 views

001A: ♦ [7+ Java integration styles & patterns interview questions & answers](#)

407 views

♦ [7 Java debugging interview questions & answers](#)

311 views

♦ [10 ERD \(Entity-Relationship Diagrams\) Interview Questions and Answers](#)

303 views

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

294 views

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

288 views

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

263 views

8 Git Source control system interview questions & answers

215 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.



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.

◀ How to create a well designed Java application?

◆ Java class loading interview Q&A to ascertain your depth of Java

knowledge ▶

Posted in Design Concepts, Judging Experience Interview Q&A, member-paid, OOP

Tags: Architect FAQs, TopX

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

The contents in this Java-Success are copy righted. The author has the right to correct or enhance the current content without any prior notice.

These are general advice only, and one needs to take his/her own circumstances into consideration. The author will not be held liable for any damages caused or alleged to be caused either directly or indirectly by these materials and resources. Any trademarked names or labels used in this blog remain the property of their respective trademark owners. No guarantees are made regarding the accuracy or usefulness of content, though I do make an effort to be accurate. Links to external sites do not imply endorsement of the linked-to sites.