Java Dependency Injection · Pattern Example Tutorial

APRIL 2, 2018 BY PANKAJ — 67 COMMENTS

Java Dependency Injection design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable. We can implement **dependency injection in java** to move the dependency resolution from compile-time to runtime.

Table of Contents [hide]

Java Dependency Injection

Java Dependency injection seems hard to grasp with theory, so I would take a simple example and then we will see how to use dependency injection pattern to achieve loose coupling and extendability in the application.

Let's say we have an application where we consume `EmailService` to send emails. Normally we would implement this like below.

```
package com.journaldev.java.legacy;

public class EmailService {

	public void sendEmail(String message, String receiver){
		//logic to send email
		System.out.println("Email sent to "+receiver+ " with
Message="+message);
	}
}
```

`EmailService` class holds the logic to send email message to the recipient email address. Our application code will be like below.

```
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = new EmailService();

	public void processMessages(String msg, String rec){
```

```
                //do some msg validation, manipulation logic etc
                this.email.sendEmail(msg, rec);
        }
}
```

Our client code that will use `MyApplication` class to send email messages will be like below.

```
package com.journaldev.java.legacy;

public class MyLegacyTest {

        public static void main(String[] args) {
                MyApplication app = new MyApplication();
                app.processMessages("Hi Pankaj", "pankaj@abc.com");
        }

}
```

At first look, there seems nothing wrong with above implementation. But above code logic has certain limitations.

- `MyApplication` class is responsible to initialize the email service and then use it. This leads to hard-coded dependency. If we want to switch to some other advanced email service in future, it will require code changes in MyApplication class. This makes our application hard to extend and if email service is used in multiple classes then that would be even more harder.
- If we want to extend our application to provide additional messaging feature, such as SMS or Facebook message then we would need to write another application for that. This will involve code changes in application classes and in client classes too.
- Testing the application will be very difficult since our application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

One can argue that we can remove the email service instance creation from `MyApplication` class by having a constructor that requires email service as argument.

```
package com.journaldev.java.legacy;

public class MyApplication {

	private EmailService email = null;

	public MyApplication(EmailService svc){
		this.email=svc;
	}

	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.email.sendEmail(msg, rec);
	}
}
```

But in this case, we are asking client applications or test classes to initializing the email service that is not a good design decision.

Now let's see how we can apply java dependency injection pattern to solve all the problems with above implementation. Dependency Injection in java requires at least following:

1. Service components should be designed with base class or interface. It's better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

Java Dependency Injection – Service Components

For our case, we can have `MessageService` that will declare the contract for service implementations.

```
package com.journaldev.java.dependencyinjection.service;

public interface MessageService {

	void sendMessage(String msg, String rec);
}
```

Now let's say we have Email and SMS services that implement above interfaces.

```
package com.journaldev.java.dependencyinjection.service;
```

```
public class EmailServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send email
		System.out.println("Email sent to "+rec+ " with Message="+msg);
	}

}
```

```
package com.journaldev.java.dependencyinjection.service;

public class SMSServiceImpl implements MessageService {

	@Override
	public void sendMessage(String msg, String rec) {
		//logic to send SMS
		System.out.println("SMS sent to "+rec+ " with Message="+msg);
	}

}
```

Our dependency injection java services are ready and now we can write our consumer class.

Java Dependency Injection – Service Consumer

We are not required to have base interfaces for consumer classes but I will have a `Consumer` interface declaring contract for consumer classes.

```
package com.journaldev.java.dependencyinjection.consumer;

public interface Consumer {

	void processMessages(String msg, String rec);
}
```

My consumer class implementation is like below.

```
package com.journaldev.java.dependencyinjection.consumer;
```

```
import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(MessageService svc){
		this.service=svc;
	}

	@Override
	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.service.sendMessage(msg, rec);
	}

}
```

Notice that our application class is just using the service. It does not initialize the service that leads to better "*separation of concerns*". Also use of service interface allows us to easily test the application by mocking the MessageService and bind the services at runtime rather than compile time.

Now we are ready to write **java dependency injector classes** that will initialize the service and also consumer classes.

Java Dependency Injection – Injectors Classes

Let's have an interface `MessageServiceInjector` with method declaration that returns the `Consumer` class.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;

public interface MessageServiceInjector {

	public Consumer getConsumer();
}
```

Now for every service, we will have to create injector classes like below.

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new EmailServiceImpl());
	}

}
```

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.SMSServiceImpl;

public class SMSServiceInjector implements MessageServiceInjector {

	@Override
	public Consumer getConsumer() {
		return new MyDIApplication(new SMSServiceImpl());
	}

}
```

Now let's see how our client applications will use the application with a simple program.

```
import com.journaldev.java.dependencyinjection.injector.SMSServiceInjector;

public class MyMessageDITest {

	public static void main(String[] args) {
		String msg = "Hi Pankaj";
		String email = "pankaj@abc.com";
		String phone = "4088888888";
		MessageServiceInjector injector = null;
		Consumer app = null;
```

```
                //Send email
                injector = new EmailServiceInjector();
                app = injector.getConsumer();
                app.processMessages(msg, email);

                //Send SMS
                injector = new SMSServiceInjector();
                app = injector.getConsumer();
                app.processMessages(msg, phone);
        }

}
```

As you can see that our application classes are responsible only for using the service. Service classes are created in injectors. Also if we have to further extend our application to allow facebook messaging, we will have to write Service classes and injector classes only.

So dependency injection implementation solved the problem with hard-coded dependency and helped us in making our application flexible and easy to extend. Now let's see how easily we can test our application class by mocking the injector and service classes.

Java Dependency Injection – JUnit Test Case with Mock Injector and Service

```
                                        public void sendMessage(String msg, String
rec) {
                                                System.out.println("Mock Message
Service implementation");

                                        }
                                });
                        }
                };
        }

        @Test
        public void test() {
                Consumer consumer = injector.getConsumer();
                consumer.processMessages("Hi Pankaj", "pankaj@abc.com");
        }

        @After
        public void tear(){
                injector = null;
        }
```

```
}
```

As you can see that I am using anonymous classes to *mock the injector and service classes* and I can easily test my application methods. I am using JUnit 4 for above test class, so make sure it's in your project build path if you are running above test class.

We have used constructors to inject the dependencies in the application classes, another way is to use setter method to **inject dependencies** in application classes. For setter method dependency injection, our application class will be implemented like below.

```
package com.journaldev.java.dependencyinjection.consumer;

import com.journaldev.java.dependencyinjection.service.MessageService;

public class MyDIApplication implements Consumer{

	private MessageService service;

	public MyDIApplication(){}

	//setter dependency injection
	public void setService(MessageService service) {
		this.service = service;
	}

	@Override
	public void processMessages(String msg, String rec){
		//do some msg validation, manipulation logic etc
		this.service.sendMessage(msg, rec);
	}

}
```

```
package com.journaldev.java.dependencyinjection.injector;

import com.journaldev.java.dependencyinjection.consumer.Consumer;
import com.journaldev.java.dependencyinjection.consumer.MyDIApplication;
import com.journaldev.java.dependencyinjection.service.EmailServiceImpl;

public class EmailServiceInjector implements MessageServiceInjector {

	@Override
```

```
        public Consumer getConsumer() {
                MyDIApplication app = new MyDIApplication();
                app.setService(new EmailServiceImpl());
                return app;
        }

}
```

One of the best example of setter dependency injection is Struts2 Servlet API Aware interfaces.

Whether to use Constructor based dependency injection or setter based is a design decision and depends on your requirements. For example, if my application can't work at all without the service class then I would prefer constructor based DI or else I would go for setter method based DI to use it only when it's really needed.

Dependency Injection in Java is a way to achieve **Inversion of control** (**IoC**) in our application by moving objects binding from compile time to runtime. We can achieve IoC through Factory Pattern, Template Method Design Pattern, Strategy Pattern and Service Locator pattern too.

Spring Dependency Injection, **Google Guice** and **Java EE CDI** frameworks facilitate the process of dependency injection through use of Java Reflection API and java annotations. All we need is to annotate the field, constructor or setter method and configure them in configuration xml files or classes.

Benefits of Java Dependency Injection

Some of the benefits of using Dependency Injection in Java are:

- Separation of Concerns
- Boilerplate Code reduction in application classes because all work to initialize dependencies is handled by the injector component
- Configurable components makes application easily extendable
- Unit testing is easy with mock objects

Disadvantages of Java Dependency Injection

Java Dependency injection has some disadvantages too:

- If overused, it can lead to maintenance issues because effect of changes are known at runtime.
- Dependency injection in java hides the service class dependencies that can lead to runtime errors that would have been caught at compile time.

Download Dependency Injection Project

That's all for **dependency injection pattern in java**. It's good to know and use it when we are in control of the services.

« PREVIOUS
Java Design Patterns – Example Tutorial

NEXT »
Factory Design Pattern In Scala

About Pankaj

If you have come this far, it means that you liked what you are reading. Why not reach little more and connect with me directly on **Google Plus**, **Facebook** or **Twitter**. I would love to hear your thoughts and opinions on my articles directly.

Recently I started creating video tutorials too, so do check out my videos on **Youtube**.

FILED UNDER: DESIGN PATTERNS

Comments

JohnsonMa says
JUNE 11, 2018 AT 8:38 PM
Really helpful,thank you!
Reply

Fagun Patel says
JUNE 10, 2018 AT 2:42 AM
Awesome…..Thanks a lot.
Reply

Hrishikesh Raskar says
JUNE 10, 2018 AT 12:53 AM
Thanks for the post. Hope you will continue such work.
Reply

Akshay shingan. says
MAY 19, 2018 AT 1:07 AM
well explained…thank you.
Reply

Gunjan says
APRIL 19, 2018 AT 11:51 PM
Thank you for the explanation
Reply

amir elhajjam says
APRIL 18, 2018 AT 7:32 AM
very helpful thank u
Reply

Mukund says
APRIL 2, 2018 AT 11:53 AM
very helpful and easy to understand for beginners
Reply

teja says
OCTOBER 13, 2017 AT 3:43 AM
Thank you
Reply

Karthik says
SEPTEMBER 6, 2017 AT 5:12 AM
confusing, lengthy and boring!
Reply

Priya says
AUGUST 23, 2017 AT 10:39 PM
Well Explained tutorial of Dependency Injection . Thanks for Sharing
Reply

saeed says
AUGUST 15, 2017 AT 2:16 AM
Thanks for your great article but your consumer class has 1 dependency(MessagingService interface). what if consumer class have multiple dependency or constructor params???
Reply

Lord Banks says
AUGUST 14, 2017 AT 1:33 PM
I think its rather confusing. I've seen more detailed but easier to follow examples. Good job still
Reply

john says
AUGUST 7, 2017 AT 3:39 AM
very bad explanation, very intuitive and confusing code, and website font is too large. please get better or refrain from writing “Tutorials” on the internet.
Reply

Gayn Dushantha says
MAY 31, 2017 AT 10:04 PM
Thank you!
Reply

Yaffa Harari says
MAY 1, 2017 AT 5:36 AM
so much clear and clean
thank you! □
Reply

Moustafa zein says
APRIL 14, 2017 AT 8:50 AM
Well done
Reply

Srini says
APRIL 17, 2017 AT 4:39 AM
Very helpful material
Reply

vinod says
APRIL 12, 2017 AT 6:52 PM
Excellent Job bro
Reply

pidgey says
APRIL 12, 2017 AT 1:10 PM
Nice tutorial. But I think your example is just fabric pattern.
Reply

Eduardo Ponzoni says
JULY 10, 2018 AT 2:35 PM
Absolutely agreed! For me this is nothing but a factory as there's no dependency injection as such anywhere.
Reply

Nix says
FEBRUARY 14, 2017 AT 11:00 PM
Nice article. Dependency Injection seems like Bridge Design Patter,
Reply

Sree says
NOVEMBER 30, 2016 AT 2:40 AM
Good Article Pankaj!!!.
Reply

Jagadeesh says
NOVEMBER 24, 2016 AT 3:58 AM
Simply Awesome…
Reply

sivateja says
NOVEMBER 10, 2016 AT 9:58 PM
The best tutorial for dependency injection , Very well explanation, I am very thankfull to you Pankaj sir.
Reply

paweu says
SEPTEMBER 4, 2016 AT 9:46 AM
Well done.
Reply

Sprung says
AUGUST 23, 2016 AT 2:26 PM
Way too long. DI via service should not take this long to explain.
Reply

progger says
SEPTEMBER 4, 2016 AT 8:41 AM
Actually I prefer this explanation to the others I found so far, because he actually goes to the effort of defining interfaces and changing the client to use those. I find other tutorials confusing in that respect, where, the clients continue to use the same concrete classes in the constructor or setter method, although the point of DI is to reduce those dependencies.
Reply

Jack says
NOVEMBER 2, 2016 AT 7:05 AM
Agreed!
Reply

HuangWei says
JULY 31, 2016 AT 3:58 AM
Wonderful, this post help me a lot to understand the DI technology which is the base of Spring FW.
Reply

Pankaj says
JULY 31, 2016 AT 7:01 AM
Thanks for liking it, appreciate your comment.
Reply

Vasily says
JULY 25, 2016 AT 7:30 AM
This is how to take a simple program and turn it into a swollen non-readable over engineered crap. Good job bro ☺
Reply

tomacco says
AUGUST 2, 2016 AT 1:31 AM
"simple program" is not that simple when you are constructing a whole system above it. This kind of decisions increase maintainability and extensibility, meaning a lot of money saved in the mid – long term. But of course, If you are building toys, of course you don't need to use any of these techniques.
Reply

Olu says
JANUARY 6, 2017 AT 1:29 PM
Well said. That's how you know the newbies □
Reply

sachindra pandey says
JULY 9, 2016 AT 3:08 PM
Good for Beginner and for Interview purpose .
Reply

Ashish says
JULY 5, 2016 AT 4:42 AM
very nice explanation ……..
Reply

Nikhil says
DECEMBER 30, 2015 AT 1:29 AM
Excellent Article for begineer

Reply

Virender says

NOVEMBER 3, 2015 AT 7:19 PM

How an Injector is different from a factory here?
An injector here is basically serving the factory pattern. I think there more to Injector patter then explained here. Do you have a follow up tutorial to explain more?
Thanks
Virender

Reply

EAT WORK says

OCTOBER 6, 2015 AT 8:22 AM

I have to say there is something that I don't understand.
You claim that one way (without DI) is to provide the service in the constructor, but you say that the clients then need to decide which service to provider and this is not a good practice.
But in the DI example, you do provide the service in the constructor, only that the client does not initiate the type of service

Reply

david won says

OCTOBER 30, 2015 AT 1:15 PM

I agree with all you guys that the DI example does nothing better than the non-DI example.
And if something does make difference in this example, that is all credited to "interface" mechanism in java language itself.

Reply

Kalinga says

SEPTEMBER 9, 2015 AT 6:28 AM

Please put the link for the source code zip in the beginning.
That will prevent someone making copy paste to create source code files and latter realize the entire effort has gone for toss as the source code is already shared as zip.
Thanks,
Kalinga

Reply

passage2 says
AUGUST 25, 2015 AT 3:34 PM
Hi Pankaj,
Quick question about your example. You are saying
…
//Send email
injector = new EmailServiceInjector();
app = injector.getConsumer();
app.processMessages(msg, email);
//Send SMS
injector = new SMSServiceInjector();
app = injector.getConsumer();
app.processMessages(msg, phone);
As you can see that our application classes are responsible only for using the service. Service classes are created in injectors. Also if we have to further extend our application to allow facebook messaging, we will have to write Service classes and injector classes only.
So dependency injection implementation solved the problem with hard-coded dependency and helped us in making our application flexible and easy to extend.
###
But this code still has a hard-coded injector that we need.
injector = new EmailServiceInjector();
My question: What is the difference between creating in my code an instance of Injector or an instance of Service? Both are hard-coded.
If I understand the DI concept correctly, to avoid hard-coded dependency, which Service to use should be resolved at run-time, no?
Could you please explain.
Thanks.
Reply

Ram Sharma says
APRIL 30, 2018 AT 4:45 AM
Yes, it is hardcoded, but at some point of time you have to specify which service that you want to create, whether an Email service or facebook service or Twitter service, and using DI we are not changing application class. Application class or the Consumer class only takes service as input and for each service there exist an Injector.
It is hard coded but in an intelligent way to separate the concerns.
Reply

Ragu says
MAY 30, 2018 AT 12:24 AM

@Pankaj: Thanks for the wonderful tutorial. It is really helpfull. I'm still facing few confusions. May be because I'm new to this technology :). Still Thanks a lot for your effort.
@Ram:
Hi,:)
You were right that at some point of time we have to specify which service that we want to use.
But, what I'm not understanding is
In Injector class,

```
public class EmailServiceInjector implements MessageServiceInjector {
@Override
public Consumer getConsumer() {
//Instead of below
return new MyDIApplication(new EmailServiceImpl());
//We can also use this (Also with method return type change)
return new EmailServiceImpl();
}
}
```

As mentioned in Tutorial, for new service class (like FB message service) only corresponding Injector need to be created.
Same is satisfied in the above approach.
So what is the point creating a consumer class which can hold all service type
and creating object of that service type and then passing required service type into it?
Could you please make me understand this?
Reply

Cris says
AUGUST 18, 2015 AT 8:36 AM
hey, good explanation about dependency injection, but the thing that I don't understand well is, why a injector? why not just create a specific xxxServiceImp? we can also can mock/fake that object in our test, instead create a messageInjector we can create directly a MessageService (still mock/fake) and in our app we can have also something like this:

```
Consumer app = null;
//Send email
app = new MessageServiceImp();
app.processMessages(msg, email);
//Send SMS
app = new SMSServiceImp();
app.processMessages(msg, phone);
```

which one is the real benefice to use that Injector clases?
Reply

Bret says
JUNE 16, 2017 AT 2:23 PM
Same question for me as above. What benefit do the injector classes give you vs. above code.
Reply

Ivan says
APRIL 18, 2018 AT 12:10 AM
The same question for me, I don't know why so many people saying this example is simple, not sure what's the purpose of Injector in this example?
Reply

Sushant says
AUGUST 18, 2015 AT 4:20 AM
Please add "Previous" and "Next" button at the end of every page.
Reply

Sriprem says
JULY 23, 2015 AT 5:37 AM
Have gone through so many samples but got an clear idea about DI through this article.Really helpful, good work dude.
Reply

M says
JUNE 23, 2015 AT 10:23 PM
A few points to re-consider, just for the sake-of-argument:
Quote:
<>
Comment:
I think, "Testing the application" will not be difficult. The correct statement would be "Testing different scenarios" E.G. testing different messaging scenario. Testing the application itself would still be easy (without DI).
Quote:
<>

Comment:
Here is the catch. Our application classes are not only responsible for using the service. They are actually responsible for instantiating the injector classes also. Here is an explanation:
MyMessageDITest -> new EmailServiceInjector() -> new EmailServiceImpl()
MyMessageDITest -> new SMSServiceInjector() -> new SMSServiceImpl()
Therefore, a question can be raised (asked mostly at the interview): "Why do we need a ServiceInjector object to instantiate MessageService object? After all, we are instantiating the ServiceInjector object anyway. For two different MessageService requirement, we are creating two instances of ServiceInjector [new EmailServiceInjector() and new SMSServiceInjector()]. If we are to create a new instance of ServiceInjector, can't we create a new instance of MessageService?"
Any thoughts? Please correct me if my understanding is wrong.
Reply

pankaj says
MAY 9, 2015 AT 10:50 AM
very nice explanation. I have a question. Why it is a bad design to create the service in client code and pass it to the constructor of the Application. Because we are ultimately creating injector related to that service in the client code, so how it is different?
Thank u
Reply

rizwan says
APRIL 9, 2015 AT 6:48 AM
Sir please create pdf book of your all spring tutorial liked design patterns book.
Reply

tester says
MARCH 10, 2015 AT 6:07 AM
I don't get it. It does not look simple at all.
Too many lines of code for this simple functionality
Reply

tomboy says
APRIL 3, 2015 AT 7:29 AM
Tester, try to write 50 different applications that all need to send email and do it without dependency injection.
Reply

George says
MAY 12, 2015 AT 6:44 PM
This is the most lucid explanation, that can actually be understood on the first read (I tried a few before settled on this one).
However, I think 'Tester' objected to complexity of implementation of dependency injection in strictly object-oriented language:
— 1 interface
— n classes declaring that each implements that interface
— n implementations of that interface in each class
— n injector classes
— 1 application class
— 1 consumer class
What is needed isn't more classes, but better language with functions as first-class objects. In such language you don't need dependency injection to be explained and remembered; it comes naturally.
To understand why pure object-oriented languages are a dead-end all one need is to read descriptions of dependency injection. Object-oriented is useful technique – it is not a successful methodology.
But I admit, talks about this or that pattern is excellent marketing tool for outsourcing companies.
Reply

Lanito says
JANUARY 10, 2015 AT 8:19 AM
That's fine, but this is not mocking it is faking.
Best Regards
Reply

Rafał says
JANUARY 9, 2015 AT 7:56 AM
Good job!
This video is also very helpful: https://www.youtube.com/watch?v=GB8k2-Egfv0
Reply

Ayaz says

JUNE 8, 2015 AT 7:19 AM

I would say the video tutorial is a perfect and easiest explaination. It is easiest because we use a framework which simplifies lots of stuff.

Reply

AJ says

DECEMBER 16, 2014 AT 7:16 PM

Job very well done. I am an experienced developer, but I was always confused about Dependency Injection pattern. Your example is simple and well-communicated, it now totally makes sense. Thanks. I have already subscribed to your mailing list and just downloaded your eBook – thanks for the hard work you put in.

Reply

theodore says

DECEMBER 11, 2014 AT 7:23 AM

Dude thanks for your time. Very nice tut and good explained! Greetings from Greece!

Reply

Snehal Masne says

NOVEMBER 27, 2014 AT 3:29 AM

Excellent tutorial for DI with simple example, exactly what one expects at the beginging. Keep it up.

Reply

Calderas says

SEPTEMBER 4, 2014 AT 8:47 PM

Another Thanks from Mexico.

Reply

Shashi Kanth says

AUGUST 31, 2014 AT 3:49 PM

Thank you for your effort.

A very concise and neatly explained article indeed.

Reply

Vijay says
AUGUST 26, 2014 AT 7:27 AM
Nice Article !
Reply

Ahmed Kamal says
AUGUST 23, 2014 AT 10:58 AM
Good and clear article. Thanks Pankaj for your great efforts.
Reply

Mustapha Naciri says
AUGUST 22, 2014 AT 2:38 AM
Thank' you very much for teach us your big knowledge in java
Reply

Thanga says
JULY 29, 2014 AT 9:34 PM
Well explained.
Reply

Leave a Reply

Your email address will not be published. Required fields are marked *
Comment

Name *

Email *

Save my name, email, and website in this browser for the next time I comment.

POST COMMENT

Search for tutorials...

DOWNLOAD ANDROID APP

DESIGN PATTERNS TUTORIAL

Java Design Patterns

Creational Design Patterns

- Singleton
- Factory
- Abstract Factory
- Builder
- Prototype

Structural Design Patterns

- Adapter
- Composite
- Proxy
- Flyweight
- Facade
- Bridge
- Decorator

Behavioral Design Patterns

- Template Method
- Mediator
- Chain of Responsibility
- Observer
- Strategy
- Command
- State
- Visitor
- Interpreter
- Iterator
- Memento

Miscellaneous Design Patterns

- Dependency Injection
- Thread Safety in Java Singleton

RECOMMENDED TUTORIALS

Java Tutorials

- Java IO
- Java Regular Expressions
- Multithreading in Java
- Java Logging
- Java Annotations
- Java XML
- Collections in Java
- Java Generics
- Exception Handling in Java
- Java Reflection
- Java Design Patterns
- JDBC Tutorial

Java EE Tutorials

- Servlet JSP Tutorial
- Struts2 Tutorial
- Spring Tutorial
- Hibernate Tutorial
- Primefaces Tutorial
- Apache Axis 2
- JAX-RS
- Memcached Tutorial