

**Java Dependency Injection** [design pattern](https://www.journaldev.com/1827/java-design-patterns-example-tutorial) 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.

**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.

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 an email message to the recipient email address. Our application code will be like below.

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.

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 the 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 the 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 harder.
* If we want to extend our application to provide an 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 an argument.

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 the above implementation. Dependency Injection in java requires at least the 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.

public interface MessageService {

void sendMessage(String msg, String rec);

}

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

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.

public interface Consumer {

void processMessages(String msg, String rec);

}

My consumer class implementation is like below.

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.

public interface MessageServiceInjector {

public Consumer getConsumer();

}

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

public class EmailServiceInjector implements MessageServiceInjector {

@Override

public Consumer getConsumer() {

return new MyDIApplication(new EmailServiceImpl());

}

}

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.

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 class MyDIApplicationJUnitTest {

private MessageServiceInjector injector;

@Before

public void setUp(){

//mock the injector with anonymous class

injector = new MessageServiceInjector() {

@Override

public Consumer getConsumer() {

//mock the message service

return new MyDIApplication(new MessageService() {

@Override

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](https://www.journaldev.com/996/java-inner-class) to *mock the injector and service classes* and I can easily test my application methods. I am using JUnit 4 for the 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 a setter method to **inject dependencies** in application classes. For setter method dependency injection, our application class will be implemented like below.

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);

}

}

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](https://www.journaldev.com/2203/get-servlet-session-request-response-context-attributes-struts-2-action).

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](https://www.journaldev.com/1392/factory-design-pattern-in-java), [Template Method Design Pattern](https://www.journaldev.com/1763/template-method-design-pattern-in-java), [Strategy Pattern](https://www.journaldev.com/1754/strategy-design-pattern-in-java-example-tutorial) and Service Locator pattern too.

[**Spring Dependency Injection**](https://www.journaldev.com/2410/spring-dependency-injection), [**Google Guice**](https://www.journaldev.com/2403/google-guice-dependency-injection-example-tutorial) and **Java EE CDI** frameworks facilitate the process of dependency injection through use of [Java Reflection API](https://www.journaldev.com/1789/java-reflection-example-tutorial) and [java annotations](https://www.journaldev.com/721/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 the 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.