A beginners guide to Dependency Injection

Scope

This article presents a high level overview of Dependency Injection (DI). It aims to present the overall concept of Dependency Injection to the junior developer within the context of how it could be used in a variety of DI containers.

Dependency Injection (DI) or Inversion of Control (IOC) ?

A lot of current literature often refers to the same design pattern as either Dependency Injection or Inversion of Control. I prefer Dependency Injection from my belief that it is a more specific name. I can imagine many scenarios where one would find evidence of Inversion of Control, though the resultant pattern is unlikely to be focusing on resolving dependencies (e.g. moving from a console application to a Windows Event Loop) . However should you prefer IOC, you can pretty much read the remainder of the article by replacing DI with IOC. Within IOC there are various types (creatively called as type 1, type 2, type 3). The differences within these types are not particularly relevant to the scope of this article and thus not entered into.

Also note that I have used the term Service extensively through this article. This should not be confused with Service Oriented Architectures. It simply became a lot easier to clarify the roles by identifying “Clients” and “Services” as opposed to “Client Component” and “Server Component or Supplier Component”.

Simple Introduction to Dependency Injection

A simple introduction

Scenario 1

You work in an organization where you and your colleagues tend to travel a lot. Generally you travel by air and every time you need to catch a flight, you arrange for a pickup by a cab. You are aware of the airline agency who does the flight bookings, and the cab agency which arranges for the cab to drop you off at the airport. You know the phone numbers of the agencies, you are aware of the typical conversational activities to conduct the necessary bookings.

Thus your typical travel planning routine might look like the following :

Decide the destination, and desired arrival date and time

Call up the airline agency and convey the necessary information to obtain a flight booking.

Call up the cab agency, request for a cab to be able to catch a particular flight from say your residence (the cab agency in turn might need to communicate with the airline agency to obtain the flight departure schedule, the airport, compute the distance between your residence and the airport and compute the appropriate time at which to have the cab reach your residence)

Pickup the tickets, catch the cab and be on your way

Now if your company suddenly changed the preferred agencies and their contact mechanisms, you would be subject to the following relearning scenarios

The new agencies, and their new contact mechanisms (say the new agencies offer internet based services and the way to do the bookings is over the internet instead of over the phone)

The typical conversational sequence through which the necessary bookings get done (Data instead of voice).

Its not just you, but probably many of your colleagues would need to adjust themselves to the new scenario. This could lead to a substantial amount of time getting spent in the readjustment process.

Scenario 2

Now lets say the protocol is a little bit different. You have an administration department. Whenever you needed to travel an administration department interactive telephony system simply calls you up (which in turn is hooked up to the agencies). Over the phone you simply state the destination, desired arrival date and time by responding to a programmed set of questions. The flight reservations are made for you, the cab gets scheduled for the appropriate time, and the tickets get delivered to you.

Now if the preferred agencies were changed, the administration department would become aware of a change, would perhaps readjust its workflow to be able to communicate with the agencies. The interactive telephony system could be reprogrammed to communicate with the agencies over the internet. However you and your colleagues would have no relearning required. You still continue to follow exactly the same protocol as earlier (since the administration department did all the necessary adaptation in a manner that you do not need to do anything differently).

Dependency Injection ?

In both the scenarios, you are the client and you are dependent upon the services provided by the agencies. However Scenario 2 has a few differences.

You don't need to know the contact numbers / contact points of the agencies – the administration department calls you when necessary.

You don't need to know the exact conversational sequence by which they conduct their activities (Voice / Data etc.) (though you are aware of a particular standardized conversational sequence with the administration department)

The services you are dependent upon are provided to you in a manner that you do not need to readjust should the service providers change.

Thats dependency injection in “real life”. This may not seem like a lot since you imagine a cost to yourself as a single person – but if you imagine a large organization the savings are likely to be substantial.

Dependency Injection in a Software Context

Software components (Clients), are often a part of a set of collaborating components which depend upon other components (Services) to successfully complete their intended purpose. In many scenarios, they need to know “which” components to communicate with, “where” to locate them, and “how” to communicate with them. When the way such services can be accessed is changed, such changes can potentially require the source of lot of clients to be changed.

One way of structuring the code is to let the clients embed the logic of locating and/or instantiating the services as a part of their usual logic. Another way to structure the code is to have the clients declare their dependency on services, and have some "external" piece of code assume the responsibility of locating and/or instantiating the services and simply supplying the relevant service references to the clients when needed. In the latter method, client code typically is not required to be changed when the way to locate an external dependency changes. This type of implementation is considered to be an implementation of Dependency Injection and the "external" piece of code referred to earlier is likely to be either hand coded or implemented using one of a variety of DI frameworks.

Those attempting to map the above scenario to design idioms will notice that the analogy could be applied to three different design idioms – Dependency Injection, Factory and Program to an Interface and not an Implementation. As in this analogy these three idioms are often but not necessarily used together.

The conventional approach till a few years ago was to separate the interface from the implementation. The factory pattern even allowed for hiding the complexity of instantiation. However the mechanism to “locate” the services was often left to the clients. Moreover some parts of the software also needed to be aware of the dependencies between the various services themselves and thus implicitly had to work out the appropriate sequencing of such component initialization, and had to track and manage their life cycles.

As an example, while the J2EE specification uses JNDI as a mechanism to standardize the mechanism of locating object references, such implementations often require a lot of code change when say the clients need to work in much simpler environments where say JNDI is not available.

Usage of Dependency Injection requires that the software we write to declare the dependencies, and lets the framework or the container work out the complexities of service instantiation, initialization, sequencing and supplies the service references to the clients as required.

Some DI Frameworks

There are a number of frameworks available today to help the developer. The ones I have found useful are :

Spring Framework : A substantially large framework which offers a number of other capabilities apart from Dependency Injection.

PicoContainer : A fairly small tightly focused DI container framework.

HiveMind : Another DI container framework.

XWork : Primarily a command pattern framework which very effectively leverages Dependency Injection. While it is an independent framework in its own right, it is often used in conjunction with Webwork

In the following sections we shall take a scenario and implement the same first without using DI and then using the above frameworks respectively. This will allow us to get a flavor of how these frameworks can be leveraged. Note that I have tried to keep the code to the minimum with a strict focus on DI related functionalities.

Implemented Scenario

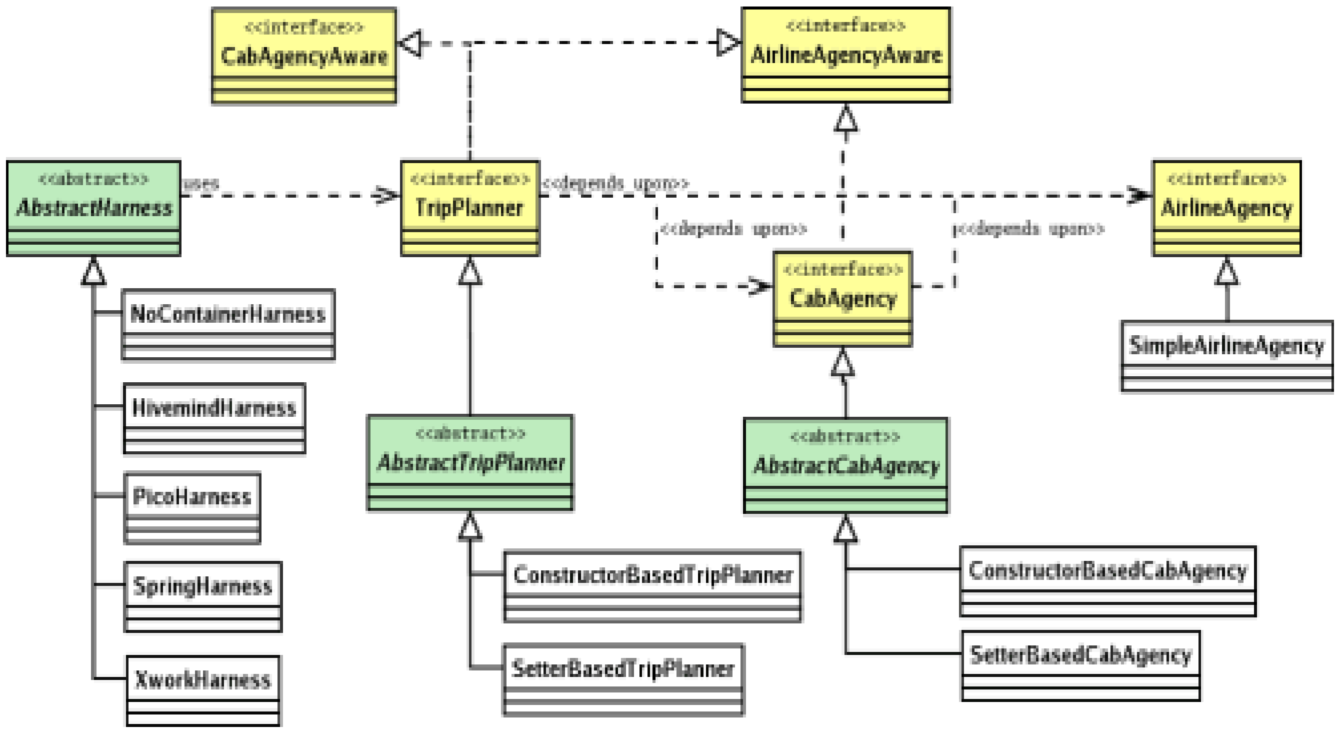
The attached source pretty much implements the scenario described earlier. The general flow (starting from the main method in each “*Harness” classes) is as follows :

- The Harness class is instantiated and the runHarness() method called.
- The runHarness() method (in AbstractHarness) first invokes the configure() method. The configure() method typically initializes the DI container and registers the implementation classes.
- The runHarness() method subsequently invokes the getPlanner() method. This typically invokes the lookup method on the underlying DI container. What is not immediately apparent, is that the lookup actually triggers the DI container to instantiate the implementation classes for AirlineAgency, CabAgency and TripPlanner interfaces, and resolve the dependencies between them.
- The runHarness() method subsequently invokes the planTrip() method which actually performs the ticket booking and cab pickup scheduling activities.

Some of the dependencies to be noted are :

- AirlineAgency and CabAgency are interfaces for the respective services. CabAgency requires a reference to the AirlineAgency to be able to fulfill its expectations successfully.
- TripPlanner is the interface for a service which can effectively plan out the trip in conjunction with the AirlineAgency and CabAgency. TripPlanner requires a reference to the AirlineAgency and CabAgency.

The UML diagram for the source is shown below :



View large image

The salient differences to be observed across different frameworks are:

- The configure() and getPlanner() method in each corresponding harness.
- How the interface / implementor relationships are described (sometimes in java and sometimes in xml files)
- How the dependencies between different services are declared (sometimes in xml, sometimes in the constructor and sometimes using a setter method)
- Any additional requirements expectations (e.g. enabler interfaces in case of xwork)

NoContainer implementation :

There is no "typical" no container implementation. Some implementations would do a lookup from a service (e.g. JNDI) or interface with a set of factories (which perhaps may be implemented as a singleton). In this case I have chosen to read the class names from a property file and in turn instantiate the objects for the services using reflection.

The property file is as follows (simply contains the class names)

```
airline-agency-class = com.dnene.ditutorial.common.impl.SimpleAirlineAgency
cab-agency-class = com.dnene.ditutorial.common.impl.ConstructorBasedCabAgency
```

The code to instantiate the agencies and trip planner is as follows :

```
AirlineAgency airlineAgency = null;
CabAgency cabAgency = null;
TripPlanner tripPlanner = null;

// Get class names from property file
InputStream propertyFile = getClass().getClassLoader().
 getResourceAsStream("nocontainer-agency.properties");
Properties properties = new Properties();
properties.load(propertyFile);

Class airlineAgencyClass = Class.forName
  (properties.getProperty("airline-agency-class"));
Class cabAgencyClass = Class.forName
  (properties.getProperty("cab-agency-class"));
Class tripPlannerClass = Class.forName
  (properties.getProperty("trip-planner-class"));
if (AirlineAgency.class.isAssignableFrom(airlineAgencyClass))
{
 // Instantiate airline agency
 airlineAgency = (AirlineAgency) airlineAgencyClass.newInstance();
}
```

```
if (CabAgency.class.isAssignableFrom(cabAgencyClass))
{
 // Instantiate CabAgency, and satisfy its dependency on an airlineagency.

 Constructor constructor = cabAgencyClass.getConstructor
  (new Class[]{AirlineAgency.class});
 cabAgency = (CabAgency) constructor.newInstance
  (new Object[]{airlineAgency});
}
if (TripPlanner.class.isAssignableFrom(tripPlannerClass))
{
 Constructor constructor = tripPlannerClass.getConstructor
  (new Class[]{AirlineAgency.class,CabAgency.class});
 tripPlanner = (TripPlanner) constructor.newInstance
  (new Object[]{airlineAgency,cabAgency});
}
return tripPlanner;
```

Note that the reference to the AirlineAgency is passed to the constructor of the CabAgency (to reflect the dependencies between the services).

The code to actually conduct the trip planning is in the AbstractHarness and AbstractTripPlanner and is reused across all harnesses.

AbstractHarness.java

```
public void runHarness() throws Exception
```

```
{
 this.configure();
 this.getPlanner().planTrip(
  "1, Acme Street",
  "11111",
  "22222",
  new Date(System.currentTimeMillis() + 7200000));
 }
```

AbstractTripPlanner.java

```
public void planTrip(String departingAddress, String departingZipCode,
  String arrivalZipCode, Date preferredArrivalTime)
{
 FlightSchedule schedule = airlineAgency.bookTicket
   (departingZipCode,arrivalZipCode,preferredArrivalTime);
 cabAgency.requestCab(departingAddress ,schedule.getFlightNumber());
}
```

PicoContainer Implementation

Container instantiation

This requires a simple instantiation of the DefaultPicoContainer class.

```
pico = new DefaultPicoContainer();
```

Service Registration

The implementation classes are registered with PicoContainer

```
pico.registerComponentImplementation(SimpleAirlineAgency.class);
pico.registerComponentImplementation(ConstructorBasedCabAgency.class);
pico.registerComponentImplementation(ConstructorBasedTripPlanner.class);
```

Dependency Declaration

Dependencies are declared by using a constructor containing the list of the (interface) classes the service is dependent upon. In the following case the ConstructorBasedCabAgency service is dependent upon the AirlineAgency interface.

```
public ConstructorBasedCabAgency(AirlineAgency airlineAgency)

{

 this.airlineAgency = airlineAgency;

}
```

Similarly the ConstructorBasedTripPlanner implicitly declares its dependencies on the AirlineAgency and the CabAgency through its constructor.

Dependency Resolution

Pico resolves dependencies by looking at the constructors. The service classes are registered with PicoContainer by registering the implementation class names. The client however requests for the services using the interface class names. PicoContainer is smart enough to traverse the interface implementation hierarchy and return the appropriate implementation as requested. (Note that the no container implementation discussed earlier also instantiated the implementation using the class names).

Since the TripPlanner is dependent upon AirlineAgency and CabAgency and the AirlineAgency and CabAgency is also dependent upon the AirlineAgency, PicoContainer will automatically instantiate the AirlineAgency first, pass its reference while constructing the CabAgency, and pass both the agency references while constructing the TripPlanner. This entire sequence gets executed upon the following request being made.

```
pico.getComponentInstanceOfType(TripPlanner.class);
```

HiveMind

XML Declarations

In this example, both the interface and implementation classes are declared in the xml file along. Note that the dependency between the services is not stated in the xml file, but follows later in the code. A “service-point” element in the xml file documents the data necessary for each service.

```
<?xml version="1.0"?>

<module id="com.dnene.ditutorial.hivemind" version="1.0.0">

 <service-point id="AirlineAgency" interface="com.dnene.ditutorial.common.AirlineAgency">

   <create-instance class="com.dnene.ditutorial.common.impl.SimpleAirlineAgency"/>

 </service-point>

 <service-point id="CabAgency" interface="com.dnene.ditutorial.common.CabAgency">

   <invoke-factory>

     <construct class="com.dnene.ditutorial.common.impl.SetterBasedCabAgency"/>

   </invoke-factory>

  </service-point>

  <service-point id="TripPlanner" interface="com.dnene.ditutorial.common.TripPlanner">

   <invoke-factory>

     <construct class="com.dnene.ditutorial.common.impl.SetterBasedTripPlanner"/>

   </invoke-factory>

  </service-point>

</module>
```

Container Initialization

This will automatically configure the registry based on the xml specified earlier.

```
Registry registry = RegistryBuilder.constructDefaultRegistry();
```

Dependency Declaration

The fact that the CabAgency requires the AirlineAgency is implicitly made clear to HiveMind by declaring the associated setter method.

```
public void setAirlineAgency(AirlineAgency airlineAgency)
{
 this.airlineAgency = airlineAgency;
}
```

Dependency Resolution

Again the dependency resolution is automatically performed when the reference to the trip planner is requested as shown in the following snippet. Hivemind is able to traverse the dependencies and instantiates AirlineAgency, CabAgency and TripPlanner in that order and invokes setAirlineAgency() on the cab agency, and setAirlineAgency() and setCabAgency() on the TripPlanner to resolve the dependencies across them.

```
registry.getService(TripPlanner.class);
```

Spring Framework

XML Declaration

The implementations that need to be instantiated are declared in the XML. Dependencies between services are also declared as “property” element. Spring framework will use this information to invoke the corresponding “setter” method.

```
<beans>
 <bean id="AirlineAgency" class="com.dnene.ditutorial.common.impl.SimpleAirlineAgency"
singleton="true"/>
```

```
 <bean id="CabAgency" class="com.dnene.ditutorial.common.impl.SetterBasedCabAgency"
singleton="true">

  <property name="airlineAgency">

   <ref bean="AirlineAgency"/>

  </property>

 </bean>

 <bean id="TripPlanner" class="com.dnene.ditutorial.common.impl.SetterBasedTripPlanner"
singleton="true">

  <property name="airlineAgency">

   <ref bean="AirlineAgency"/>

  </property>

  <property name="cabAgency">

   <ref bean="CabAgency"/>

  </property>

 </bean>

</beans>
```

Container Initialization

The container initialization typically requires a reference to the xml file and an instantiation of the BeanFactory.

```
ClassPathResource res = new ClassPathResource("spring-beans.xml");

BeanFactory factory = new XmlBeanFactory(res);
```

Dependency Resolution

References to the services are retrieved based on the 'id' specified in the xml (not the interface class). Again all the services are implicitly instantiated in the appropriate order and the setters called to resolve their dependencies.

```
 factory.getBean("TripPlanner");
```

XWork Implementation

XWork is actually a command pattern framework with some amount of DI functionality. It is in all likelihood the least sophisticated of all DI frameworks. However I have found it a useful framework especially in situations where one has already decided to use webwork, more specifically when injecting dependencies into the action classes.

XML Declaration

```
<components>
  <component>
    <scope>application</scope>
    <class>com.dnene.ditutorial.common.impl.SimpleAirlineAgency</class>
    <enabler>com.dnene.ditutorial.xwork.AirlineAgencyAware</enabler>
  </component>
  <component>
    <scope>application</scope>
    <class>com.dnene.ditutorial.common.impl.SetterBasedCabAgency</class>
    <enabler>com.dnene.ditutorial.xwork.CabAgencyAware</enabler>
  </component>
  <component>
    <scope>application</scope>
    <class>com.dnene.ditutorial.common.impl.SetterBasedTripPlanner</class>
    <enabler>com.dnene.ditutorial.xwork.TripPlannerAware</enabler>
    </component>

</components>
```

Note that XWork depends upon a notion called “enabler interface”. This is an interface which basically defines a setter method for the service. eg.

```
public interface CabAgencyAware

{

 public void setCabAgency(CabAgency cabAgency);

}
```

Unlike other containers, it requires us to define interfaces AirlineAgencyAware and CabAgencyAware and TripPlannerAware and implement them appropriately. To that extent this requires tends to be a bit more intrusive than other containers.

Container Initialization

This tends to be a little bit more complex

```
ComponentConfiguration config = new ComponentConfiguration();

InputStream in =

 XworkHarness.class.getClassLoader().getResourceAsStream("components.xml");

config.loadFromXml(in);

cm = new DefaultComponentManager();

config.configure(cm,"application");
```

Dependency Declaration

Classes declare their dependencies by implementing the necessary “enabler interfaces”. Thus each service declares its dependencies on other services by implementing the corresponding enabler interfaces. XWork uses the enabler interfaces to resolve the dependencies.

public class SetterBasedCabAgency extends AbstractCabAgency implements AirlineAgencyAware { ... }

Dependency Resolution

No surprises here. Usually a simple lookup from the component manager and all dependencies are automatically resolved.

cm.getComponent(TripPlannerAware.class);

Xwork and Webwork

XWork is often used in conjunction with Webwork. In such scenarios, all you need to do is define and implement the enabler interfaces and create the xml file and use the component interceptor. The component manager interactions are all implemented by Webwork so that no coding to the ComponentManager is required so long as the dependencies are being resolved for the Action classes only. In addition it is very easy to setup different component managers for different scopes (e.g. Application, Session and Request).

Conclusion

As you can see, each DI framework does implement its features a little bit differently. In all fairness they could be used a little bit differently as well (eg. PicoContainer can be used with Setter based Dependency Injection, whereas Spring Framework can be used with Constructor based Dependency Injection). I have just chosen to document one of the supported mechanisms.

Its a little hard to be able to clearly declare one better than the other in a comprehensive sense. I guess you need to make up your mind what suits you the best in your context.

What should've become fairly clear with the above scenarios is that using any DI container substantially reduces the complexity of instantiating various services and resolving various dependencies between them.

Additional Features :

Note that I have primarily focused on only two features viz.

Resolving Service dependencies for clients and

Resolving Service dependencies within services themselves.

DI Frameworks do offer more functionality than just that – primarily in the area of managing the configuration and lifecycle of the services. You should read up the associated documentation to find more about these features. Dependency Injection alone is likely to suffice for most applications of simple to moderate complexities. However you would perhaps want to exploit the other features as your application (and services) start getting more complex. Some of the resources listed at the end are likely to be good starting points for getting more information :

Some Contentious Topics

There are a few areas where there has been a fair degree of debate (often between the authors/supporters of the specific DI frameworks). Some of these are :

Constructor Based Dependencies OR Setter Based Dependencies

This refers to whether it is better to declare dependencies using constructors or by using setter methods. I am not sure if there is a clear case that can be made in favor of one or the other. I have however observed that setter based dependencies do seem simpler to manage as as the overall dependency structures start getting more complex.

XML OR Java

Some prefer the wiring up of the dependencies and their relationships to be done using Java code – many others prefer using XML. I think it is a simpler to work with Java code. However there is at least one scenario where XML is the only reasonable choice. This is when you ship binaries, but would like to allow for administrators / system integrators / end users to be able to modify / enhance the default functionality by plugging in different implementations for the services (deployment specific variations). Your end users are far more likely to be able to modify your XMLs (assuming you documented them well) rather than your source code (assuming you shipped it).

It would be important to point out that in the above scenarios, the debate is about the means not the end. Either way you implement things, you still are able to leverage Dependency Injection. Note that in most situations, you have both the choices (you just might need to delve into the API documentation a bit). If the framework documentation suggests XML – you are quite likely to be able to use Java for wiring up by just figuring out the API to do so. Also if the documentation seems to suggest that the framework you would like to use supports only constructor or setter based dependency injection, dig just a little deeper – there's a chance you might find support for the other scenario as well.

So what do I get using DI ?

An overview of DI as a design pattern and of the various frameworks might still leave you wondering – so what do I really get out of it ? Some of the benefits are :

If you look at your service dependencies – there are two very orthogonal sub-dependencies. One is the "domain" specific dependency on the API / contract of the service itself. The other is the dependency on the overall "technical" protocol of instantiation, configuration and lookups of the services. DI allows you to relocate your code handling the latter to the fringe boundaries of your code (in an XML file or in some method not far away from your "main" method perhaps). This allows the "technical" changes to be handled in a much easier fashion, while allowing the client developer to focus on what he / she knows best – the "domain" API of the service.

If you want to build pluggable infrastructures where your end-users / system integrators might want to extend/modify the service functionality and capablities – doing it the DI way really makes things a lot easier. DI makes it easier to build software that lives up to the Open-Closed principle. While in the examples I have talked about Spring as a DI container, Spring itself also substantially leverages DI. A variety of capabilities built into spring (e.g. AOP, Transactions etc.) are actually pluggable services which are plugged in using DI. The framework can also be extended by writing additional services and plugging them in again using the DI framework (e.g. springmodules : https://springmodules.dev.java.net)

If you are really building a large number of fine grained services with complex interdependencies in your application, there is a good chance you might need to rewire things more often than you might think. You'll be glad you used DI in such situations (if you did).

When you start enhancing your application where some of the original assumptions start changing. An example is that if you had services in a J2EE server, all locating each other through JNDI. Now if you want to move the code to a non JNDI environment – you'll be glad you used DI.

Ref: http://www.theserverside.com/news/1321158/A-beginners-guide-to-Dependency-Injection