

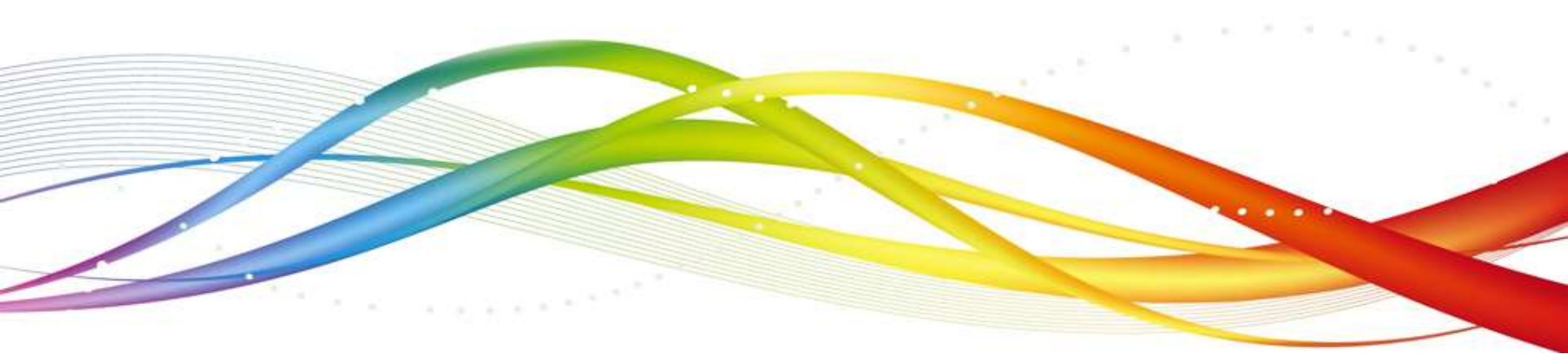


# Spring Basics and Inversion Of Control – II

## Dependency Injection

**Harish B Rao**

Talent Transformation | Wipro Technologies



# Inversion of Control / Dependency Injection

---

Java applications usually have objects that work together to present a working application for the end user.

When you develop a complex java application, the classes that are written as part of this application should be as independent as possible (from other java classes), so that you can easily reuse them. Creating such independent classes also facilitates easy unit testing.

Inversion of Control / Dependency injection helps in gluing these classes together while keeping them independent.

# Inversion of Control (IoC)

---

Spring Framework insists that Associations between Business objects should be externalized & the Client Applications should never be involved in doing these kinds of activities.

Instead of Clients having the control to establish relationship between Components, now the Framework carries this job

- which means that the Control is reversed from the Clients to the Framework
- that's why this principle is rightly termed as ***Inversion of Control***

# Inversion of Control (Contd.)

What is **I**nversion **o**f **C**ontrol?

Let us try to understand IoC, through the following simple Java code :

## Figure.java

```
interface Figure {  
    double area(int dim1, int dim2);  
}
```

## Rectangle.java

```
class Rectangle implements Figure {  
    public double area(int dim1, int dim2) {  
        double d = dim1*dim2;  
        return d;  
    }  
}
```

## Triangle.java

```
public class Triangle implements Figure {  
    public double area(int dim1, int dim2) {  
        return 0.5 * dim1 * dim2;  
    }  
}
```

We have an interface called Figure, which declares only one method area(int, int).

There are two classes, Rectangle and Triangle, which implement this interface.

Both these classes have the area method, which is the concrete implementation of the abstract method defined in the interface.

# Inversion of Control (Contd.)

## AreaClass.java

```
public class AreaClass { // line #1
    public static void main(String args[]){ // line #2
        Figure f = new Rectangle(); // line #3
        System.out.println("The area of figure is : "+f.area(9,5)); // line #4
        f = new Triangle(); // line #5
        System.out.println("The area of figure is : "+f.area(9,5)); // line #6
    }
}
```

When you execute the above code, it will print the following output on the console :

**The area of figure is : 45.0**

**The area of figure is : 22.5**

*Can you tell, instances of which classes are created in main method and which area methods are invoked in line #4 and line #6?*

*Yes, you can. It is very obvious that object of class Rectangle is created in line #3 and the code in line #4 invokes area method from Rectangle object. Similarly, when line #5 gets executed, it instantiates the class Triangle and the method area from Triangle is invoked thereafter.*

*The control here lies with the main method. The main method decides which object to instantiate and which method to invoke during runtime. In short, the main method of AreaClass is in control of the execution.*

# Inversion of Control (Contd.)

## Now Let us alter the code within AreaClass

```
class AreaClass {                                // line #1
    public static void main(String [] args) {    // line #2
        FindFigure x = new FindFigure();        // line #3
        Figure z = x.getFigure();              // line #4
        double area = z.area(9, 5);            // line #5
        System.out.println("Area of the figure is : "+area);
    }
}
```

*Now the question is, area method from which class is being invoked in line #5? Rectangle or Triangle? It may be either one of them.*

*Who decides, which area to invoke?*

*This depends on the class FindFigure. FindFigure has a method called getFigure, which will return the figure object and it is this class which instantiates the figure object and passes it onto the main method. So, the control now is with this FindFigure class rather than AreaClass. This is known as Inversion of Control, where the control of execution rests with some other entity and this entity is responsible for injecting the objects as and when required.*

*The code for FindFigure is given in the next page.*

# Inversion of Control (Contd.)

## FindFigure.java

```
class FindFigure {  
    Figure fig;  
    FindFigure() {  
        fig = new Triangle();  
    }  
    public Figure getFigure() {  
        return fig;  
    }  
}
```

*As you can see, the FindFigure class' constructor instantiates an object of class Triangle and when the getFigure method gets invoked, it passes the object of Triangle to the main method. So, the decision of which object to inject into main is made by the class FindFigure. Thus the control of the execution rests with FindFigure class and not with the client(main method in this case). This concept is known as **Inversion of Control**.*

# Inversion of Control (Contd.).

---

Spring is most closely identified with a flavor of Inversion of Control known as **Dependency Injection**

Dependency Injection is a form of IoC that removes explicit dependence on container APIs. (Finding it difficult to understand? Don't worry. Explanation available in the next slide)

The two major flavors of Dependency Injection are

**Setter Injection** (injection via JavaBean setters)

**Constructor Injection** (injection via constructor arguments).



# Inversion of Control (Contd.).

---

Dependency Injection is a form of IoC that removes explicit dependence on container APIs; Ordinary Java methods are used to *inject* dependencies such as collaborating objects or configuration values into application object instances.

Let us understand what this means. In traditional container architectures, such as EJB, a component might call the container to say "Where is object X, which I need to do my work".

With Dependency Injection, the container figures out that the component needs object X and provides it at runtime. The container does this figuring out based on method signatures(usually JavaBean properties or constructors) and configuration data such as XML.

---

*What is the advantage of such an architecture?*

*With traditional application development, we as developers need to write huge amount of code to create objects that are required during runtime. With Dependency injection, the container will figure out what objects are required and it instantiates and injects these objects automatically during runtime.*

# Use of Dependency Injection in Software Context

Software components (Clients), are often 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 like Spring.

# Dependency Injection

---

- Dependency Injection has several important benefits :
  - Because components don't need to look up collaborators at runtime, they're much simpler to write and maintain.
  - With a Dependency Injection approach, dependencies are explicit and evident in constructor or JavaBean properties
  - Easy to use objects either inside or outside the IoC container
  - Spring also provides unique support for instantiating objects from static factory methods or even methods on other objects managed by the IoC container
  - Your business objects can potentially be run in different Dependency Injection frameworks - or outside any framework - without code changes

# Dependency Injection (Contd.).

- There are two types of **Dependency Injection(DI)** techniques we can use:

- **Setter Injection**

- using setter methods in a bean class, the **Spring IOC** container will inject the dependencies

- **Constructor Injection**

- The constructor will take arguments based on number of dependencies required
  - You don't have option to reconfigure the dependencies at later point of time, since all the dependencies are resolved only at the time of invoking the constructor
  - Eg.:

**<constructor-arg index="0" type="java.lang.String" value="MyName"/>**

# Setter Injection

---

Setter based Dependency Injection is accomplished when the container calls setter methods on our beans after invoking a no argument constructor to instantiate the bean.

Example given in the next page illustrates how to accomplish setter injection in a Spring based application.

# Example on Setter Injection

Let us create a Java Project called SimpleProject2.

As in the previous examples, create a package named com.wipro and create three classes AreaClass, Rectangle and Triangle within this package. Also create an interface named Figure within the same package.

Create an xml file with any name(here this xml file is named as spring2.xml) under src folder.

*Please remember, for every Spring project that you create in Eclipse, you are supposed to include appropriate jar files in the build path.*

## **Figure.java**

```
package com.wipro;  
  
public interface Figure {  
    void area();  
}
```

# Example on Setter Injection (Contd.).

## Rectangle.java

```
package com.wipro;

public class Rectangle implements Figure {
    double d1, d2;
    public double getD1() {
        return d1;
    }
    public void setD1(double d1) {
        this.d1 = d1;
    }
    public double getD2() {
        return d2;
    }
    public void setD2(double d2) {
        this.d2 = d2;
    }
    public void area() {
        System.out.println("The area of Rectangle is " + (getD1()*getD2()));
    }
}
```

# Example on Setter Injection (Contd.).

## Triangle.java

```
package com.wipro;

public class Triangle implements Figure {
    double d1, d2;
    public double getD1() {
        return d1;
    }
    public void setD1(double d1) {
        this.d1 = d1;
    }
    public double getD2() {
        return d2;
    }
    public void setD2(double d2) {
        this.d2 = d2;
    }
    public void area() {
        System.out.println("The area of triangle is " + (0.5*getD1()*getD2()));
    }
}
```



# Example on Setter Injection (Contd.).

## AreaClass.java

```
package com.wipro;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class AreaClass {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring2.xml");
        Figure fig = (Figure)context.getBean("fig");
        fig.area();
    }
}
```

## spring2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="fig" class="com.wipro.Rectangle" >
        <property name="d1" value="50" />
        <property name="d2" value="20" />
    </bean>
</beans>
```

# Example on Setter Injection (Contd.).

---

As you can see, when we execute AreaClass as a java application, the container will inject the object of the class Rectangle by reading the configuration file, spring2.xml. When the Rectangle object is created, it will automatically invoke the setter methods of this class, setD1, SetD2 and passes the values defined in the xml file for the properties d1 and d2, to these setter methods. When the area method is invoked, it calculates the value of area by invoking the getter methods and prints the area of Rectangle as given below :

**The area of Rectangle is 1000.0**

# Constructor Injection

---

It is possible to inject the dependency by constructor.

The `<constructor-arg>` sub-element of `<bean>` is used for constructor injection.

Example given in the next page illustrates how to accomplish Constructor injection in a Spring based application.

# Example on Constructor Injection

Let us create a Java Project called SimpleProject3.

As in the previous examples, create a package named com.wipro and create two classes AreaClass and Triangle within this package. Also create an interface named Figure within the same package.

Create an xml file with any name(here this xml file is named as spring1.xml) under src folder.

*Please remember, for every Spring project that you create in Eclipse, you are supposed to include appropriate jar files in the build path.*

## **Figure.java**

```
package com.wipro;  
  
public interface Figure {  
    void draw();  
}
```

# Example on Constructor Injection (Contd.).

## Triangle.java

```
package com.wipro;  
  
public class Triangle implements Figure{  
    private String type;  
    private int height;  
  
    public Triangle() {  
    }  
  
    public Triangle(String type) {  
        this.type=type;  
    }  
  
    public Triangle(int height) {  
        this.height=height;  
    }  
}
```

*Continued...*

# Example on Constructor Injection (Contd.).

Triangle.java (continues from previous page)

```
public Triangle(String type, int height){
    this.type=type;
    this.height=height;
}

public String getType() {
    return type;
}

public String getHeight() {
    return height;
}

public void draw() {
    System.out.println(getType()+" Triangle is drawn of height "+getHeight());
}
}
```

# Example on Constructor Injection (Contd.).

## DrawFigure.java

```
package com.wipro;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DrawFigure {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring1.xml");
        Figure fig = (Figure)context.getBean("fig");
        fig.draw();
    }
}
```

## spring1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="fig" class="com.wipro.Triangle" >
        <constructor-arg index="1" value="80" />
        <constructor-arg index="0" value="Isoscles" />
    </bean>
</beans>
```

# Example on Constructor Injection (Contd.).

---

*When you execute the DrawFigure class, the container instantiates the figure(Triangle in this case) as per the bean configuration specified in the xml file. Appropriate constructor is invoked based on the constructor-arg specification in the xml file.*

When you execute the example given in the previous page, it prints

**Isoscles Triangle is drawn of height 80**



# The “ref” attribute

---

We have seen the use of setter injections and constructor injections. Till now, we have injected constructor arguments and properties, only with static values.

Values can also be injected by reference. i.e. One bean definition can be injected into another.

For injecting bean definitions into another, we use the constructor-arg or property's ref attribute instead of value attribute.

The ref attribute is used to refer to another bean definition's id.

# Example on “ref” attribute

Let us create a Java Project called SimpleProject4.

As in the previous examples, create a package named com.wipro and create two classes AreaClass ,Triangle and Point within this package. Also create an interface named Figure within the same package.

Create an xml file with any name(here this xml file is named as spring1.xml) under src folder.

*Please remember, for every Spring project that you create in Eclipse, you are supposed to include appropriate jar files in the build path.*

## **Figure.java**

```
package com.wipro;
```

```
public interface Figure {  
    void draw();  
}
```

# Example on “ref” attribute (Contd.).

## Triangle.java

```
package com.wipro;

public class Triangle implements Figure {
    private Point pointA;
    private Point pointB;
    private Point pointC;
    public Triangle() {

    }
    public Point getPointA() {
        return pointA;
    }
    public void setPointA(Point pointA) {
        this.pointA = pointA;
    }
}
```

*Continued..*

# Example on “ref” attribute (Contd.).

Triangle.java (continues from previous page)

```
public Point getPointB() {
    return pointB;
}
public void setPointB(Point pointB) {
    this.pointB = pointB;
}
public Point getPointC() {
    return pointC;
}
public void setPointC(Point pointC) {
    this.pointC = pointC;
}
public void draw() {
    System.out.println("Point A : "+getPointA().getX()+" "+getPointA().getY());
    System.out.println("Point B : "+getPointB().getX()+" "+getPointB().getY());
    System.out.println("Point C : "+getPointC().getX()+" "+getPointC().getY());
}
}
```

# Example on “ref” attribute (Contd.).

## Point.java

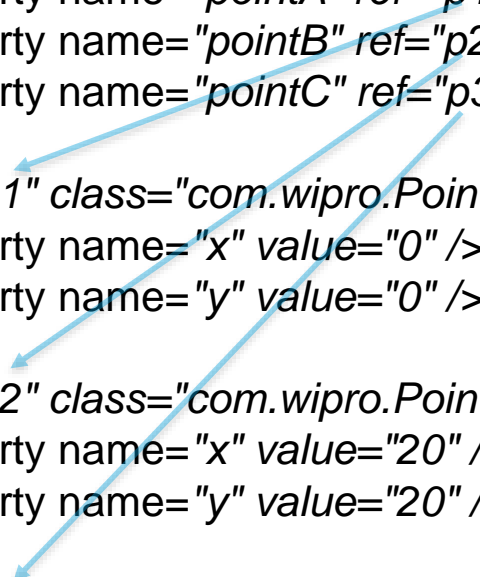
```
package com.wipro;

public class Point {
    private int x;
    private int y;
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
    }
    public int getY() {
        return y;
    }
    public void setY(int y) {
        this.y = y;
    }
}
```

# Example on “ref” attribute (Contd.).

## spring1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean id="fig" class="com.wipro.Triangle" >
        <property name="pointA" ref="p1" />
        <property name="pointB" ref="p2" />
        <property name="pointC" ref="p3" />
    </bean>
    <bean id="p1" class="com.wipro.Point" >
        <property name="x" value="0" />
        <property name="y" value="0" />
    </bean>
    <bean id="p2" class="com.wipro.Point" >
        <property name="x" value="20" />
        <property name="y" value="20" />
    </bean>
    <bean id="p3" class="com.wipro.Point" >
        <property name="x" value="40" />
        <property name="y" value="0" />
    </bean>
</beans>
```



The diagram illustrates the references in the XML. Three blue arrows originate from the 'ref' attributes in the 'fig' bean's property elements and point to the 'id' attributes of the 'p1', 'p2', and 'p3' beans respectively. The first arrow points from 'ref="p1"' to 'id="p1"', the second from 'ref="p2"' to 'id="p2"', and the third from 'ref="p3"' to 'id="p3"'. This visualizes how the 'fig' bean depends on the 'p1', 'p2', and 'p3' beans.

# Example on “ref” attribute (Contd.).

## DrawFigure.java

```
package com.wipro;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class DrawFigure {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("spring1.xml");
        Figure fig = (Figure)context.getBean("fig");
        fig.draw();
    }
}
```

*In the above example, the bean identified by fig has three properties, pointA, pointB and pointC, each one of this refers to an object of the type Point, referred by beans identified by p1, p2 and p3. Observe how ref attribute is used to refer to these three objects of Point.*

When you execute DrawFigure as a java application, it will display the following output on the console :

**Point A : 0 0**

**Point B : 20 20**

**Point C : 40 0**



**Thank You**

