DURGA SOFTWARE SOLUTIONS

Practical approach to Spring Framework

Spring Framework 4.x

Mr. Sriman

As part of this we cover Spring Core, AOP, Spring JDBC, Transactions, Spring MVC and ORM. This includes all the versions of spring starting with 2.0, 2.5, 3.0, 3.1.1 and latest 4.x (Spring Annotations as well).

Contents

Spring Framework

1 Spring Framework

1.1 Introduction to Spring Framework

Spring is a framework, but spring is not a framework similar to struts. Struts framework supports only developing Web Applications. Unlike Struts, spring supports various application development types.

In a Web application project we will not have only Web related aspects, apart from web we will have various other things like managing business logic and persisting data etc. So, in the above struts only will manages Web aspects and will not provide anything to manage business or persistency aspects of an application.

Spring not only manages Web, it can handle other aspects of the application as well, so it is usually termed as end to end application development framework. So using spring we can develop not only Web we can develop core java, JEE, remoting (rmi), distributed, persistency and various other application types.

How many application types spring is supporting. Spring supports all the application development types in par with JEE. If spring supports all the application types in par with JEE why should I go to spring, I can use JEE to get the benefits?

JEE is an API. API stands for Application Programming Interfaces; API contains interfaces and classes, using which we can build applications. Generally these API has more number of classes, and these classes are interlinked or inter-related with each other. For e.g. if I want to execute a sql query, I need statement Object in jdbc, but to create a Statement I need to know Connection and to create a Connection I should know DriverManager. In this way each class is inter-related with each other.

So, building an application with few features of JEE also it demands programs to know all the classes, unless he end up in learning all he will not be able to start developing the applications. So, API's are complex to understand and code.

As API's are huge (contains more number of classes), it takes more time in learning and learning curve would be high. Apart from this API's will not provide boiler plate code.

Boiler plate code means this is the piece of code that has to be repeated across various parts of the application with minimal or no changes. For e.g. I want to execute a sql query, to execute a sql query I need to write the following lines of code.

```
try{
  Class.forName("DriverClassName");
  Connection con = DriverManager.getConnection(url, un, pwd);
  Statement stmt = con.createStatement();
  ResultSet rs = stmt.executeQuery();
  If(rs.next()) {
    // grab the values
  }
}catch(SqlException sqe) {
   // print stack
}
```

Executing sql query is a common functionality that everyone has in their projects. To execute a sql query everyone has to write more or less the same piece of logic. When it is same, even then also I have to write the same duplicate logic all over so, it is called boiler plate code.

Instead if we go to spring framework to execute sql query, spring provides one simple class to do this, we never need to write the whole logic rather we should just call a method on the spring provided class. Internally spring classes will take care of doing the above and will provide output.

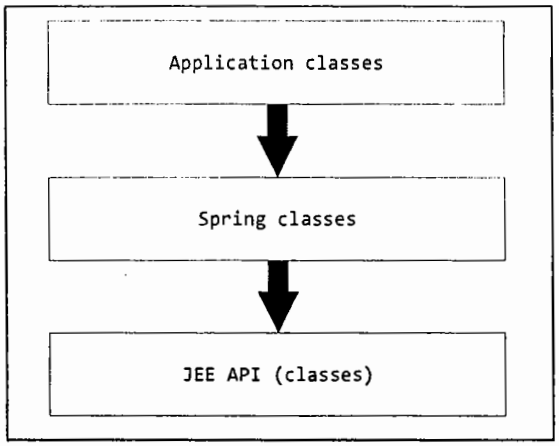
So, in API's boiler plate code has to be written by programmer, this leads to writing more and more number of lines of code. This has the following impacts

1. More amount of efforts are required as we need to write more code
2. Developers productivity will go down
3. As more lines of code chances of increasing the bugs will be also high
4. Maintenance cost involved in modifying the code also will be high

In spring as it provides boiler plate logic, we can avoid all the above said problems. That is the reason rather than working with JEE we are using spring.

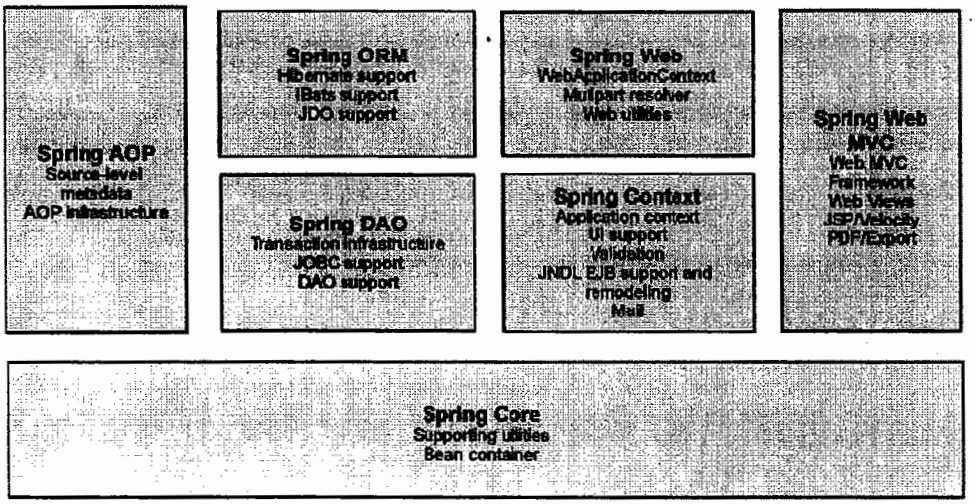
If spring supports all the development types in par with JEE, can I call spring as a replacement of JEE? Spring never replaces JEE rather spring complements JEE. Our application classes are going to talk to spring provided classes and spring classes internally talks to JEE provided classes. This means without JEE spring cannot work rather spring enriches and make JEE easier to use for the developer. This means spring is complementing JEE to use it easily.

Below diagram depicts the same.



If spring is providing various applications development types in par with JEE, is spring is light weight application development framework or heavy weight?

If we look at the spring architecture, it has been designed keeping in view of light weight. The basic module in the spring is core, it spans across the breath of the spring, it is the module on which all the rest of the modules are built-on. Apart from core there are AOP, Jdbc, Transactions and MVC. In this way spring has several modules and all are dependent and spring core module. Refer to the below diagram to understand the same.



Here if we observe carefully each and every module don't have intersection or cross lines with other, this means there is least or no dependency with other module. Let's say if we want to build persistence application, programmer has to learn/use only spring core and spring Dao (jdbc) to develop application, he don't need to use all the other modules to build the same. This shows spring is flexible enough in offering what you want rather than what is unnecessary. So it is called light weight application framework.

Following are the functionalities of each module.

1. Spring Core: - The core module is the heart of spring framework, it spans horizontally across all the other modules. Core module provides the essential functionality for spring framework. The primary component of core module is BeanFactory, an implementation of Factory pattern. BeanFactory allows you to separate your configuration information and dependency specification from your actual application code. Using spring core you can develop a core java application as well.
2. Spring Context: - The spring context module is a configuration file that allows you to provide the context information to the framework. The spring context may contain configuration about the pojo's, jndi, email, internationalization etc.
3. Spring AOP: - AOP stands for aspect oriented programming; it's a new programming technic that allows programmers to induce cross-cutting logic across several components of your application. Cross-cutting logic implies any logic or code that has to be applied across several components. For example transactions, logging, auditing and security are some of the examples of applications of AOP.
4. Spring DAO: - It is an abstraction of JDBC DAO. In an traditional JDBC application you have to write lot of boiler plate code like getting the connection, executing the statements, iterating over the result sets and managing the resources (e.g.. connection, statement, resultset etc.). Along with this while working with JDBC code you need to handle lot of annoying checked exceptions, spring JDBC avoids lot of boiler plate code and it can manages resources as well as has a meaningful exception hierarchy defined in the framework to handle several types of exceptions that JDBC code might throw.
5. Spring ORM: - spring framework plugins to several ORM frameworks to provide its Object Relation tool, the idea behind spring framework is it don't want to re-invent the wheel rather wants to make existing tools to be used easily in their applications. So, as part of this effort it has provided integrations to lot of ORM frameworks like Hibernate, iBatis, JPA etc.
6. Spring Web module:- The web context module is built on top application context module, providing the context for web-based applications. This module allows spring to integrate with various other web based frameworks like jakarta struts etc.
7. Spring Web MVC framework: - This module allows us to build Model-View-Controller architecture based applications which contains full features for building Web applications. It varies in many ways from normal web application frameworks like struts etc. The main advantage of going with spring web mvc module is view technology is abstracted from the client and you can have anything as a presentation tier.

Spring framework has been released in early 2004, at that time struts is the popular framework in the market and is being highly used. So, struts is peer competitor for spring, by overcoming lot of hurdles today spring turned to be

the top framework which replaced struts as well. What are the successful points that made spring framework to be top?

1) Spring is versatile application development framework. Spring can be integrated into existing projects which may be built-on any technologies to fill the gap in existing projects/frameworks. Spring is flexible enough to integrated into any existing technology/framework that is there in the market to solve the problems that are not addressed by those frameworks. This made spring to be consumed in bits and pieces and eventually turned to occupy the entire project on spring.
2) Spring is non-invasive application development framework. Spring will never force your project to implement/extend or use spring classes in your application code. Spring classes will be separate and our code is separate still we can leverage spring capabilities in our application. This makes our classes as pojo classes, which means even we remove spring framework dependencies still our classes can be compiled under the underlying Jdk and use the functionality of it.

As we understood spring core is the foundation module on which all the rest of the modules are built-on, our journey to spring has to start with spring core. Considering the importance of spring core module we have divided it into two parts I) Spring Core Basic and ii) Spring Core Advanced module.
With this separation spring core basic module acts as a foundation, which enables us to work on other modules (without spring core advanced). Advanced module covers the powerful features of spring core and is used rarely when the application demands.

2 Spring Core (Basic)

Spring core is the foundation module of the spring framework, as we know to work on any other module in spring we need to know spring core. Why spring core is so important, what exactly spring core offers to us. Let us try to understand.

In an application we will have multiple classes; all the classes will not play the same role. Few classes are called Java beans, few are called pojo and some other are called component or bean classes.

Java bean: If a class only contains attributes with accessor methods (setters & getters) those are called java beans. These classes generally will not contain any business logic rather those are used for holding some data in it.

Pojo: If a class can be executed with underlying Jdk, without any other external third-party libraries support then it is called Pojo.

Component/Bean: If a class has attributes with some member methods. Those member methods may use the attributes of that class and will fulfill some business functionality then it is called Component classes.

So a project cannot be built by just one. Say with Java beans we cannot complete the project rather we need business logic to perform something, so component classes will also exist in a project.

So, in a project do we have one component class or multiple component classes. By just having one we cannot complete rather we will have multiple component classes. Every component class in our project will talk to other or will not? Every component class cannot be isolated; it has to talk to some other component classes in the project to fulfill the functionality.

For e.g.

```
class A {
   public void m1() {
   }
}

class B {
   public int m2() {
   }
}
```

In the above code we have class A and class B. class A m1() method may have to talk to m2() method of the class B to fulfill business functionality, so these two classes are called dependent on each other as, unless B is there A cannot be used. So how to make the B available to A or how to manage the dependencies between A and B is what spring core is all about.

Now the question here is can I give any two classes and ask it to manage or spring cannot manage any arbitrary classes. Spring can manage any two arbitrary classes but in-order it to be effective spring recommends us to design our classes following some design guidelines, so that those components will be loosely-coupled. The spring can effective manage them.

Spring recommends us to design our classes following a design pattern called Strategy design pattern. Strategy design pattern is not belonging to spring, it is the pattern that comes from "Gang of four design patterns". If we design our classes following strategy design patterns the spring can manage them easily.

So, let's first examine the design principles and then we focus on Spring Core IOC.

2.1 Strategy pattern (Design principle spring recommends)

Strategy pattern lets you build software as loosely coupled collection of interchangeable parts, in contrast with tightly coupled system. This loosely coupling makes your software much more flexible, extensible, maintainable and reusable.

Strategy pattern recommends mainly three principles every application should follow to get the above benefits those are as follows

- Favor Composition over Inheritance
- Always design to Interfaces never code to Implementation
- Code should be open for extension and closed for modification

Let us examine all the three principles by taking an example

2.1.1 Favor Composition over Inheritance

Every class in order to perform a task has to talk with other class to get the things done. A class can use the functionality of other class in two ways.

1) Through Inheritance – Inheritance is the process of extending one class from another class to get its functionalities.
2) Composition – A class will hold reference of other classes as attributes inside it.

Using Inheritance you always express relation between two classes as IS-A relationship. Composition always expresses relation between two classes as HAS-A relationship.

When should I go for Inheritance and when should I go for composition?

The problem with most of the programmers is they will always choose the option of inheritance to use the functionality of other classes. But this may not be apt in all cases;

We have to go for inheritance only when all the behaviors (methods) of your base class can be commonly sharable across all the derived classes.

In case if my class wants to use only few behaviors of the other class, rather extending my class from the other I should go for composition.

If we look composition has more number of advantages when compared with inheritance as described below.

- Most of the use-cases are solvable through composition rather than inheritance. If we look no class wants to use all the services of other class, rather it wants to use only few method of other class. Just for the sake of using few methods of other class we should not go for inheritance rather than composition is recommended
- A class most of the time wants to use behavior of more than one classes, so to use the functionality of other class if we choose inheritance here we have to extend our class from more than one class. But most of the programming languages will not support extending a class from more than one class (multiple inheritance). Only alternate here is to use composition.
- When we go for inheritance our classes will become fragile. Fragile means those will become delicate or easily breakable. To understand this let's consider one example. I have a class A as shown below.

```
class A {
   int m1() {
     // some logic
     return 10;
   }
}
```

I have one more class B which extends from A and overriding method m1() as shown below.

```
class B extends A{
  @Override
   int m1() {
     // some logic
     int i = super.m1();
     // calculate
     return i;
   }
}
```

Now class A has been written by some other developer and class B has been written by other developer. After sometime author of the class A has felt the return type of the method needs to be changed and has modified from int to float as shown below.

```
class A {
   float m1() {
      // some logic
     return 10.24;
   }
}
```

Now author of the class A has changed the return type from int to float, immediately class B will not compile, why? When we are overriding the super class method and sub-class method signatures should be same. But in the super class for the method m1() return type is float and in sub-class the return type is int, so it is not an valid overriding.

Even it is not valid overloading, because parameters of my super class method m1() and parameters of my sub-class methods are same. It means we have two methods with same name in the class which is not allowed. So it results in compilation error.

Here the change in the super class will not just break the sub-class even the classes using the sub-class also will gets affected and all the components using your sub-class need to undergo changes. It results in high maintenance cost and chances of increasing the bugs will also be more.

If we go for composition in the above case which means class B is not extending from class A rather it is using the reference of A inside B to call its methods.

```
class B {
  private A a;

  int m1() {
    // some logic
    int i = a.m1();
    // calculate
    return i;
  }
}
```

Now if the return type of m1() in class A changed I just need to type cast the value in the B to int rather than modifying the return value as shown below.

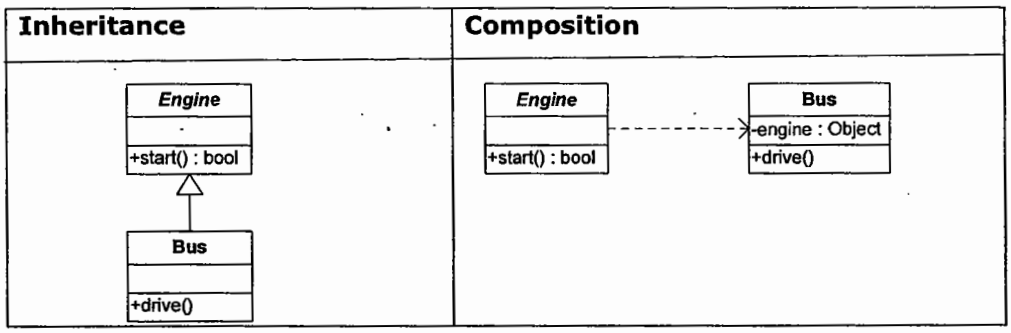
```
class B {
  private A a;

  int m1() {
    // some logic
    int i =(float) a.m1();
    // calculate
    return i;
  }
}
```

So, any changes to the class A can be handled in B and those changes will not affect the classes using B. Here B can suppress all those changes at its level, so that those do not propagate to other classes.

- Testability of the code will be easy when we go for composition instead of using inheritance.

For e.g. I have a class Engine and it has a method start(). I have another class Bus it has a method called drive().

If my Bus wants to use the Engine class I have two options one is inheritance and other one is composition. But inheritance makes my code difficult to test let's try how?



For example in the above case my Engine is not fully implemented, only the methods are declared, still the logic has to be written. But meanwhile the other developer as he knows the methods of the Engine he has written the logic for Bus.

Now to test class Bus I need the Engine, so to facilitate the testing I need to create dummy class for Engine, using which I can test my drive method of the Bus as shown below.

```
// dummy class or prototype class for the actual engine
class PrototypeEngine1 {
  public boolean start() {
    // do something.
    return true;
  }
}
```

Now I need to modify the Bus class to extend it from PrototypeEngine1 to test it as shown below.

```
class Bus extends PrototypeEngine1 {
  public void drive() {
    boolean isStarted = false;

    isStarted = super.start();
    if(isStarted == true) {
     // do something
    } else {
     // do something
    }
  }
}
```

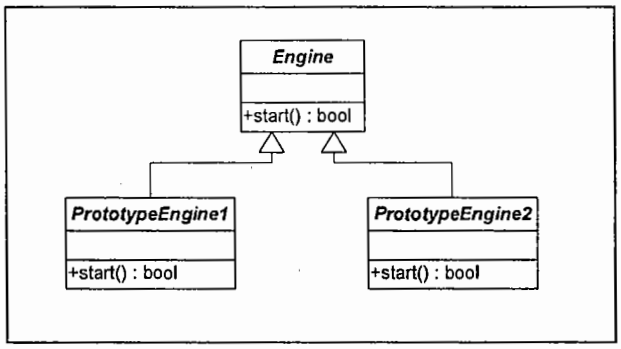
But just testing against one Dummy or Prototype class I cannot certify my Bus is working, rather I should be able to test it against lot of prototype engines in this way. So always I need to keep on changing the class from which I need to extend from.

```
class Bus extends
PrototypeEngine2 {
  public void drive() {
    boolean isStarted = false;

    isStarted = super.start();
    if(isStarted == true) {
     // do something
    } else {
     // do something
    }
  }
}
```

```
class Bus extends
PrototypeEngine3{
  public void drive() {
    boolean isStarted = false;

    isStarted = super.start();
    if(isStarted == true) {
     // do something
    } else {
     // do something
    }
  }
}
```

If you look as inheritance makes a class static reference to another class it makes harder to test my Bus class against different prototypes of Engine and demands code changes to test against different. Instead if I use composition it gives me lot of flexibility as shown below.

Create multiple prototype Engine classes extending from the abstract class Engine as shown below.



Now create Bus class which has reference of Engine inside it (composition).

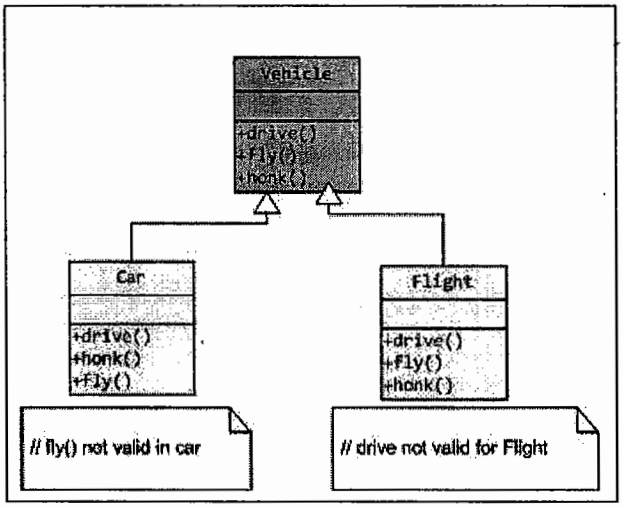
```
class Bus {
  private Engine engine;

  public Bus(Engine engine) {
    this.engine = engine;
  }

  public void drive() {
    boolean isStarted = false;

    isStarted = engine.start();
    if(isStarted == true) {
     // do something
    } else {
      // do something.
    }
  }
}
```

Now in my test class while testing the Bus pass different prototype Engines as input which makes your object to point to different Engines without modifying the code as shown below.

```
class TestBus {
  public void testDrive() {
     new Bus(new PrototypeEngine1()).drive();
     new Bus(new PrototypeEngine2()).drive();
  }
}
```

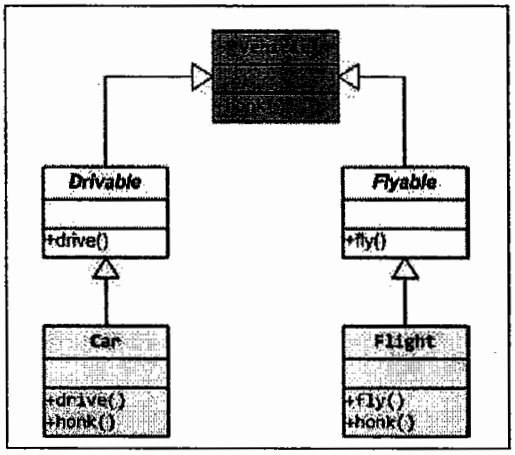
Now I can test Bus without modifying the code by making it point to several references of EnginePrototype which makes my code easily test.

So let's try to understand how to use inheritance by taking an example.



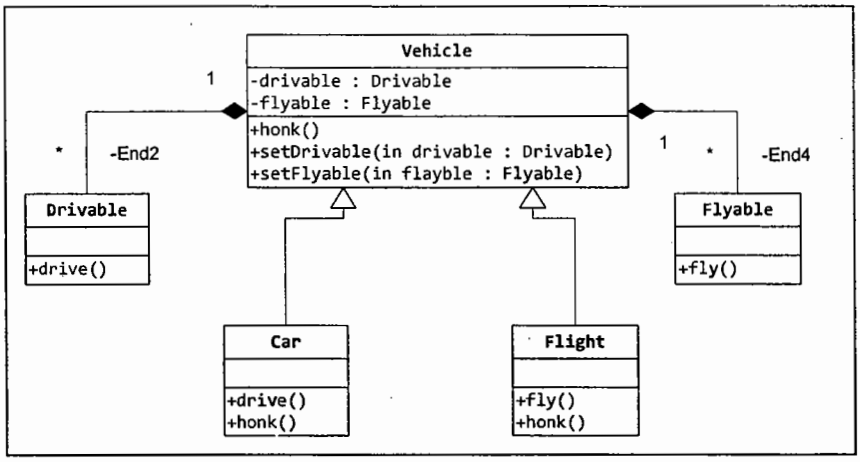
In the above diagram I have a vehicle which contain drive (), fly () and blow horn (honk ()). So I want to represent a Car and a Flight which is of Vehicle type. If I try to express this with Inheritance I will end up in implementing fly method in Car class as empty and drive method in flight class as empty (because the fly and drive are in-appropriate for Car and Flight respectively).

So to overcome this modifies your class structure into two hierarchies as Drivable and Flyable making Vehicle as sub-classes shown below.



But the problem with above approach is your design model is rigid (closed). Through the above design you are stating you can have a vehicle which can either fly or drive. But in future if I have a vehicle which can drive as well as fly then it is not possible to represent that kind of vehicle with the above design. This may leads to re-designing the entire application.

So, by the above example it is clear that we can't represent all the problems through inheritance. In order to solve this, Composition would be more apt than Inheritance. Below diagram shows the solution for the above described problem.



In the above diagram Vehicle is an abstract class. We created two more concrete classes Drivable and Flyable which contains logic for drive () and fly () respectively. As every vehicle may contain drive() or fly() or both, we declared both these classes as attributes inside Vehicle class.

Now class Car & Flight extends from vehicle. Let's say in the class Car we have the method drive() it will calls the drive() method on the super class attribute drivable as shown below.

```
class Car extends Vehicle {
   public Car() {
     setDrivable(new Drivable());
   }
   public void drive() {
     drivable.drive();
   }
}
```

```
class Flight extends Vehicle {
  public Flight() {
    setFlyable(new Flyable());
  }
  public void fly() {
    flyable.fly();
  }
}
```

In the same way Flight is another class extends from Vehicle and it only sets the flyable object reference and calls the fly() method to use the functionality. Tomorrow if we have a vehicle which can drive() and can fly() then in my class we need to set the references of both Drivable and Flyable and can use it.

Using composition now we have a flexibility of having both fly() and drive() as part of our sub-classes, this cannot be achieved by inheritance.

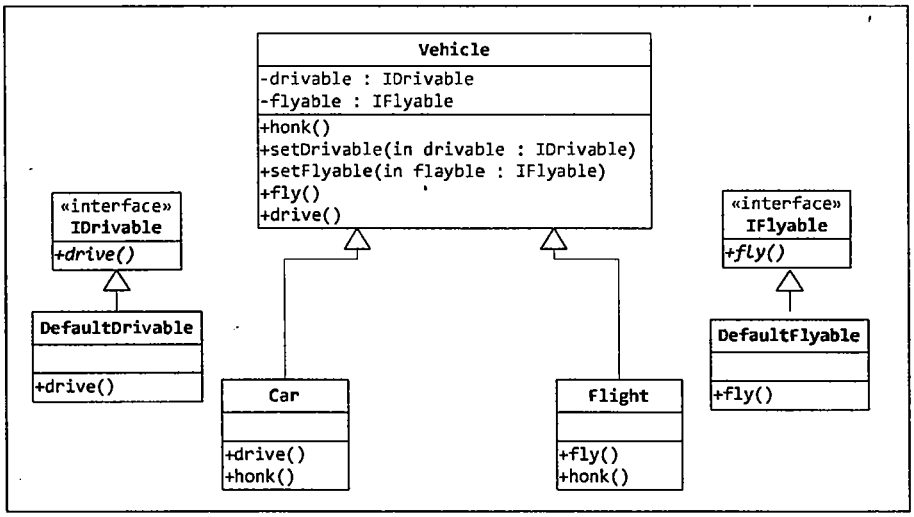
2.1.2 Always design to interfaces, never code to implementation

In our earlier discussion we understood most of the times we should prefer to use Composition rather than inheritance. Yes, but when we use composition, the coupling between the classes will be high.

For e.g When Vehicle is referring to Drivable or Flyable it means without the presence of Drivable and Flyable class Vehicle cannot be complete. Let's say we have a new car came which uses an Automatic Drive technic rather than normal one or a new Flight which needs to by fly with new technic. It cannot be changed as Vehicle holds the concrete references of Drivable and Flyable classes we cannot change it.

Now let's declare two interfaces IDrivable contains drive() and IFlyable contains fly() method. Declare these two interfaces as attributes in Vehicle class. Now create DefaultDrivable and DefaultFlyable as implementations of those interfaces.

As Vechicle holds the interface reference type we can set those to object of its implementation to those references. In future we need new drive() technic, we can create one more implementation of IDrivable interface and can plug-in. Here is the below diagram depicting the same.



2.1.1 Code should be open for extension and should be closed for modification

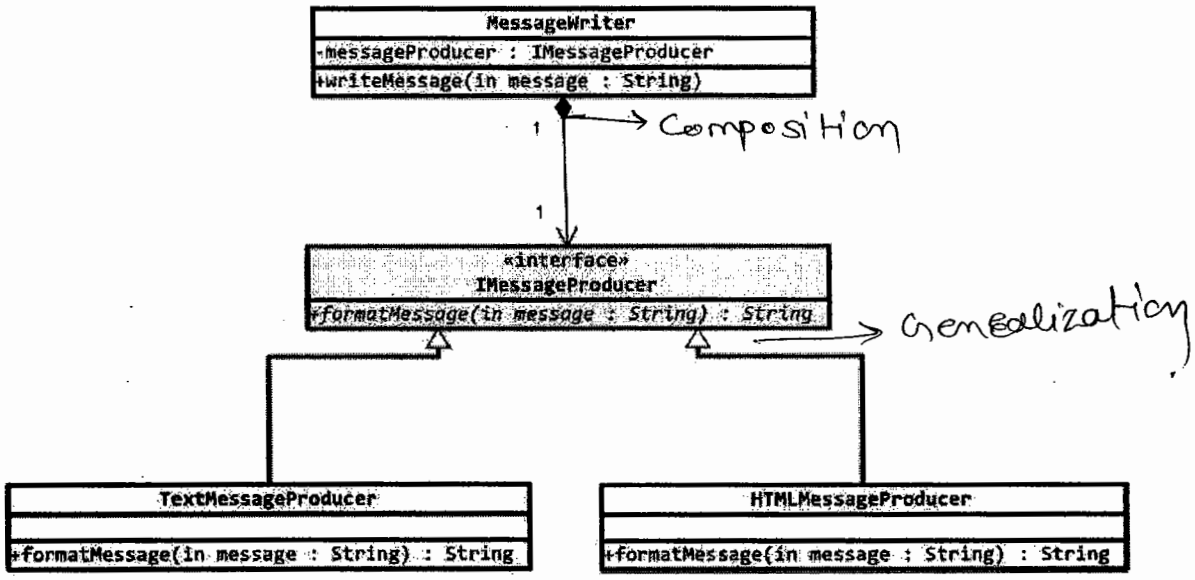
Let's say a new car came, and it uses a different drive technic, now we don't need to modify the DefaultDrivable class rather we can create one more implementation of the IDrivable interface and can set it as a reference to drivable reference. If we modify the existing code the chances of increasing the bugs would be more, rather than our design should be flexible enough to plug-in new implementations when required, our earlier design depicts the same.

2.2 Spring Inversion of Control (IOC)

Let's try to understand the IOC by taking one example, first let us design our classes by following strategy design pattern, and then we will identify the pit-falls in it and will understand how to simplify it using IOC

We have a MessageWriter class which will write the message to the console. The MessageWriter class will get the message from MessageProducer class. But we have multiple types of MessageProducers like TextMessageProducer, HTMLMessageProducer etc.

So the MessageWriter will talks to the message producers through an interface IMessageProducer, which defines a method formatMessage. This has been explained in the below diagram.



If you observe carefully the above design has followed all the design principles that spring recommends. Code fragment for the above classes are as shown below.

IMessageProducer.java

```
package com.ioc.beans;
public interface IMessageProducer {
        String formatMessage(String message);
}
```

TextMessageProducer.java

```
package com.ioc.beans;

public class TextMessageProducer
implements IMessageProducer {
    public String formatMessage(String
message) {
        return "Hello " + message + "!";
    }
}
```

HTMLMessageProducer.java

```
package com.ioc.beans;

public class HTMLMessageProducer
implements IMessageProducer {
   public String formatMessage(String
message) {
       return
"<HTML><HEAD></HEAD><BODY>" +
message + "</BODY></HTML>";
   }
}
```

MessageWriter.java

```
package com.ioc.beans;

public class MessageWriter {
  private IMessageProducer messageProducer;
  public void writeMessage(String message) {
     // instantiate messageProducer with concrete implementation class
     messageProducer = new HTMLMessageProducer();
     String formattedData = messageProducer.formatMessage(message);
     System.out.println(formattedData);
  }
}
```

MessageTest.java

```
package com.ioc.beans;

public class MessageTest {
   public static void main(String args[]) {
      MessageWriter writer = new MessageWriter();
      writer.writeMessage("Welcome to Spring");
   }
}
```

Even you followed the recommended design principles, it has two problems.

- In your MessageWriter class inside the writeMessage method you have hardcoded the concrete class name HTMLMessageProducer while instantiating messageProducer attribute. If you want to switch between HTMLMessageProducer to TextMessageProducer you need to modify the source code of the MessageWriter.

- MessageWriter to use the services of HTMLMessageProducer it is trying to create the object of it.If MessageWriter is trying to create the Object of HTMLMessageProducer or TextMessageProducer, it has to know the complete instantiation process of creating those. Some classes can be created out of calling new Operator, but some classes should be created using complex instantiation technic like sometimes let's say 'A' want 'B' it has to create 'B' but to create 'B' it is using 'C' so first create 'C' and then 'B' finally 'A' can use it.

This tells unless 'A' instantiates all the dependents that 'B' needs it cannot use, and the same piece of code has to be written by all the other classes which ever want to use 'B'. So, code is duplicated across the logic to create 'B'

To avoid the above problems instead of MessageWriter creating the Object of IMessageProducer implementation class, it should externalize this functionality to someone else, that's where factory classes come into picture.

In Jdbc Connection is an interface & Database vendors will have the implementation, if I want to switch between Oracle DB to MySql Database, I don't need to modify the code because my classes are talking to the interfaces, there are not talking to create implementations.

But in-order to create Connection I can't create the Connection as

```
Connection con = new Connection();
```

Because Connection is an interface, to instantiate it we need implementation of it. We don't know what is the implementation of Connection to instantiate? That's where DriverManager (factory) will find the implementation class and instantiate it.

So, here DriverManager acts as a factory for creating the object of Connection interface. Factories are the classes who will manufacture the objects of other classes. They hide the complexity in creating the objects of other classes.

Let's say I want a Car, but to drive a Car I don't need to know how to create a Car rather I just need to go to a Car factory, factory will create a car and gives to us.

Similarly if we want IMessageProducer Implementation, we can create MessageProducerFactory which will have a factory method, it creates the Object of one of the implementation of IMessageProducer and returns to use as shown below.

```
public class MessageProducerFactory {
   public static IMessageProducer createMessageProducer(String type) {
      if(type.equals("html")) {
        return new HTMLMessageProducer();
      }else if(type.equals("text")){
        return new TextMessageProducer();
      }
     return null;
   }
}
```

In the above class I have a factory method called createMessageProducer upon calling will create the one of the implementation of IMessageProducer and returns to us based on the 'type' we passed as input.

With the above we are able to avoid the two problems we discussed earlier, now we don't need to refer to the concreate class name of other class and we don't need to know the complex instantiation process of creating the object of other classes. The modified code for MessageWriter shown below.

Modified#1 - MessageWriter.java

```
package com.ioc.beans;

public class MessageWriter {
  private IMessageProducer messageProducer;
  public void writeMessage(String message) {
     messageProducer = MessageProducerFactory.createMessageProducer("html");
     String formattedData = messageProducer.formatMessage(message);
     System.out.println(formattedData);
  }
}
```

If we look at the above code, even we are able to avoid the concreate class name of other class still we are referring to the logical name of other class as 'html' or 'text' again there is a level of coupling.

By this we understood when we create the object of other class or when we try to get (pull) the object of other class through some factory, our classes will be tightly coupled to other classes. To avoid this, my class should not have the code for creating or pulling rather the dependent object need to be injected.

To inject we need to create a setter or constructor around the dependent attribute of our class. So, whoever wants to use MessageWriter will set the dependent and call the method?

Below is the modified version of MessageWriter for the same. Here test class is setting IMessageProducer implementation and calling the method writeMessage()

Modified#2 - MessageWriter.java

```
package com.ioc.beans;

public class MessageWriter {
  private IMessageProducer messageProducer;
  public void writeMessage(String message) {
     String formattedData = messageProducer.formatMessage(message);
     System.out.println(formattedData);
  }

  public void setMessageProducer(IMessageProducer messageProducer){
   this.messageProducer = messageProducer;
  }
}
```

Modified#1 – MessageTest.java

```
package com.ioc.beans;

public class MessageTest {
   public static void main(String args[]) {
      MessageWriter writer = new MessageWriter();
      IMessageProducer messageProducer = new HTMLMessageProducer();
      writer.setMessageProducer(messageProducer);
      writer.writeMessage("Welcome to Spring");
   }
}
```

If you see the MessageWriter class it is loosely coupled from HTMLMessageProducer and now it can talk to any IMessageProducer implementation classes. Second thing MessageWriter doesn't need to bother about how to instantiate IMessageProducer implementation class, because IMessageProducer implementation class will be injected by calling setter method on it.

Even though we made our business class loosely coupled from specific implementation, but still we hardcoded the HTMLMessageProducer in MessageTest class. If you want to use TextMessageProducer instead of HTMLMessageProducer your main method should be modified to instantiate TextMessageProducer class and should pass it to the MessageWriter via calling its setter.

So, at some place in your code you are hard coding the concrete class references, in order to avoid this you need to use the Spring IOC.

2.3 Types of IOC

IOC is a principle, IOC means collaborating objects and managing the lifecycle of it. There are two ways using which we can collaborate objects

IOC is of two types; again each type is classified into two more types as follows.

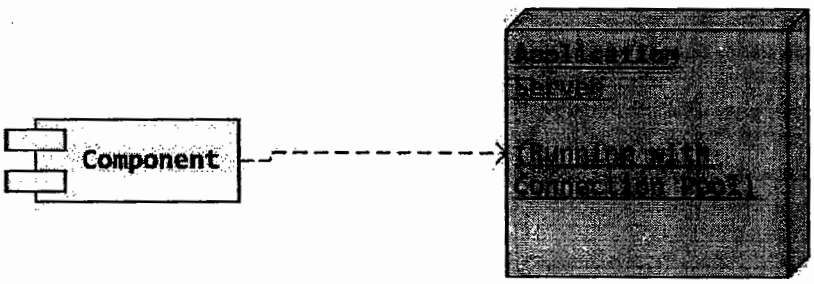
1) Dependency Lookup
   a. Dependency Pull
   b. Contextual Dependency lookup
2) Dependency Injection
   a. Setter Injection
   b. Constructor Injection

2.3.1 Dependency Lookup

Dependency lookup is a very old technic which is something exists already and most of the J2EE applications use. In this technic if a class needs another dependent object; it will write the code for getting the dependency. Again this has two variants as said above.

1) Dependency Pull

If we take a J2EE Web application as example, we retrieve and store data from database for displaying the web pages. So, to display data your code requires connection object as dependent, generally the connection object will be retrieved from a Connection Pool. Connection Pools are created and managed on an Application Server. Below figure shows the same.



Your component will look up for the connection from the pool by performing a JNDI lookup. Which means you are writing the code for getting the connection indirectly you are pulling the connection from the pool. So, it is called dependency pull.

Pseudo code

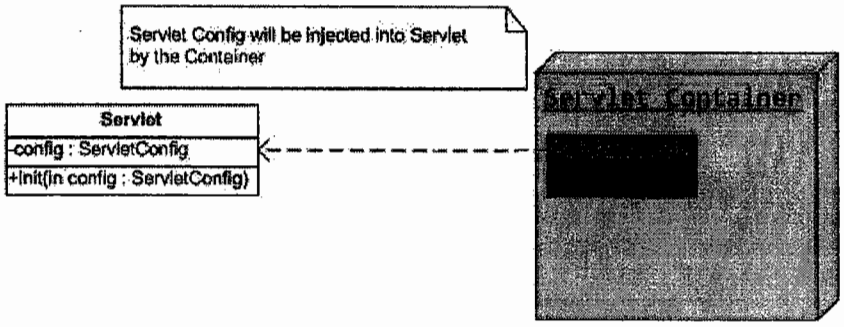
```
Context ctx = new InitialContext();
DataSource ds = ctx.lookup("jndi name of cp");
Connection con = ds.getConnection()
// pulling the connection
```


DURGA SOFTWARE SOLUTIONS, Plot No: 202, IInd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad-500038., Cell. 9246212143. Phone: 040-64512786

2) Contextual Dependency Lookup

In this technic your component and your environment/server will agree upon a contract/context, through which the dependent object will be injected into your code.

For e.g If you see a Servlet API, if your servlet has to access environment specific information (init params or to get context) it needs ServletConfig object. But the ServletContainer will create the ServletConfig object. So in order access ServletConfig object in your servlet, you servlet has to implement Servlet interface and override init(ServletConfig) as parameter. Refer the below figure.



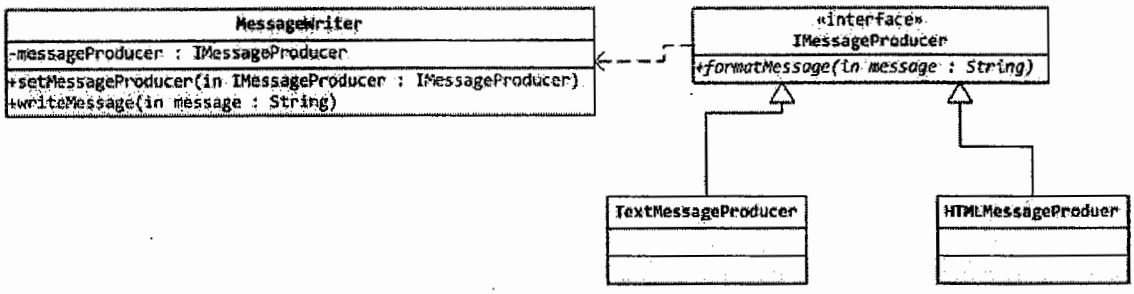
Then the container will pushes the config object by calling the init method of the servlet. Until your code implements from servlet interface, container will not pushes the config object, this indicates servlet interface acts as a contract between you and your servlet so, and it is called Context Dependency Lookup (CDL).

2.3.2 Dependency Injection

Even though spring supports the above mentioned two technics, the new way of acquiring the dependent objects is using setter injection or constructor injection. This is detailed as below.

1) Setter Injection

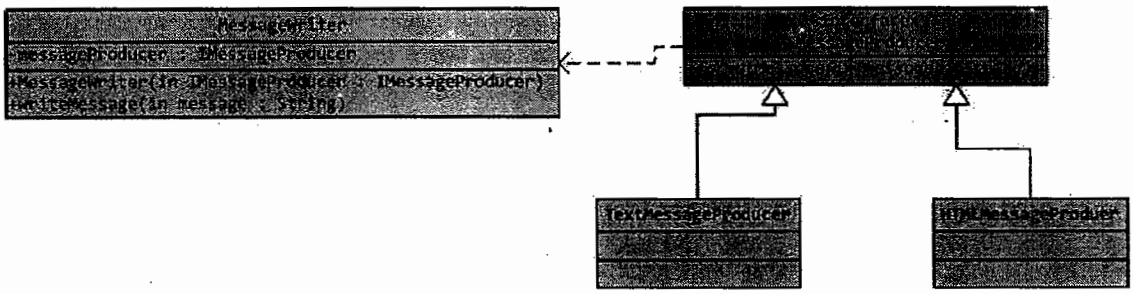
In setter injection if an object depends on another object then the dependent object will be injected into the target class through the setter method that has been exposed on the target classes as shown below.



In the above diagram MessageWriter is the target class onto which the dependent object IMessageProducer implementation will be injected by calling the exposed setter method.

2) Constructor Injection

In this technic instead of exposing a setter, your target class will expose a constructor, which will take the parameter of your dependent object. As your dependent object gets injected by calling the target class constructor, hence it is called constructor injection as shown below.



Let's modify the earlier example to inject the IMessageProducer into MessageWriter through setter injection and constructor injection.

If we want our class objects to be managed by spring then we should declare our classes as spring beans, any object that is managed by spring is called spring bean. Here the term manages refers to Object creation and Dependency injection (via setter, constructor etc...).

So in our example we want MessageWriter and IMessageProducer implementation classes to be created and injected by spring, so we need to declare them spring in a configuration file called "Spring Beans configuration".

"Spring Bean configuration" is an xml file in which we declare all the classes as beans, so that those will be managed by spring. We need to use <bean> tag to declare a class as spring bean. The below fragment shows how to create a class as spring bean.

Spring Beans Configuration (application-context.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="textMessageProducer" class="com.ioc.beans.TextMessageProducer" />
  <bean id="htmlMessageProducer" class="com.ioc.beans.HTMLMessageProducer" />
  <bean id="messageWriter" class="com.ioc.beans.MessageWriter"/>
</beans>
```

The org.springframework.beans and org.springframework.context packages are the basis for the Spring Framework's IOC Container. By just declaring our classes as beans in the spring bean configuration files automatically objects will not be created, rather we need to give this configuration as input to BeanFactory. The BeanFactory is an interface and the implementations of it know reading the configuration file and instantiating the beans.

We can declare the information about our classes in multiple ways for e.g. in properties file or xml or annotations. Based on the style of configuration we need to use one of the implementations of BeanFactory interface. Here we are using xml for configuring our class information, so we should use XMLBeanFactory implementation.

Creating IOC Container

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                          "com/injection/common/application-context.xml"));
```

Now to get the bean from the spring, we need to call the method factory.getBean("beanId"). Here beanId refers to the id with which we configured it in spring bean configuration.

Below is the code fragment explaining how to use the beans or objects created by spring.

```
package com.injection.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.ioc.beans.MessageWriter;

public class MessageWriterTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/ioc/common/application-context.xml"));
                MessageWriter messageWriter = factory.getBean("messageWriter",
                                MessageWriter.class);
                // now get message producer and set it to message writer
                IMessageProducer messageProducer =
factory.getBean("htmlMessageProducer", IMessageProducer.class);
                messageWriter.setMessageProducer(messageProducer);
                messageWriter.writeMessage("Welcome to Spring");
        }
}
```

In the above example we are not creating the beans rather spring itself has created the objects for our classes we are just getting those objects from Spring. After getting MessageWriter and HTMLMessageProducer, we are explicitly calling the setMessageProducer(..) method to inject messageProducer into MessageWriter class, because in our declaration (spring bean configuration file) we just asked spring to create the objects of our classes, we never asked him to manage the dependencies.

As both the objects created by spring, we got them and calling setter to inject one into another, so if I don't want to use HTMLMessageProducer instead I want to use TextMessageProducer, again I need to modify the code why? Because we are managing the depencdencies. So I should not create or should not pull rather pass the object of dependent HTMLMessageProducer to Target MessageWriter using spring.

Will spring does? Yes, as both the objects are created by spring, we can request spring to help us in calling a setter method or constructior other guy by passing HTMLMessageProducer or TextMessageProducer. How? Only way of talking to spring is through spring bean configuration file.

Instead of us injecting we want spring to create these objects and inject one into another, in order to do this we need to declare all the classes in configuration file including their dependency relations.

Spring Beans configuration File (application-context.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
       <bean id="textMessageProducer"
              class="com.ioc.beans.TextMessageProducer" />
       <bean id="htmlMessageProducer"
              class="com.ioc.beans.HTMLMessageProducer" />
       <bean id="messageWriter" class="com.ioc.beans.MessageWriter">
              <property name="messageProducer"
                                    ref="htmlMessageProducer"/>
       </bean>
</beans>
```

In order to inject htmlMessageProducer bean into messageWriter bean we need to declare the <property> tag. The name attribute of it refers to the target class attribute. Ref attribute refers to which bean has to be injected into name attribute (by calling its setter's).

<property> tag is used for performing setter injection and the dependent object will be injected into target class by callings the setter in target.

MessageTest.java

```
package com.injection.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.ioc.beans.MessageWriter;

public class MessageWriterTest {
       public static void main(String[] args) {
              BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                           "com/ioc/common/application-context.xml"));
              MessageWriter messageWriter = factory.getBean("messageWriter",
                           MessageWriter.class);
              messageWriter.writeMessage("Welcome to Spring");
       }
}
```

Now in the above code we are not calling the setter rather when we get the MessageWriter from spring it will be created by injecting the HTMLMessageProducer into it by spring. If I want to change from HTMLMessageProducer to TextMessageProducer, we can simply change the reference of it in spring bean

configuration file so, that I can switch between various dependencies. So my classes are completely loosely coupled.

Another way of managing the dependencies is using constructor, below example shows how to inject IMessageProducer into MessageWriter using constructor injection.

In order to perform constructor injection, your target class MessageWriter instead of exposing a setter, should expose a constructor which should take IMessageProducer as parameter.

Along with this need to modify the "Spring beans configuration" to instruct to inject by calling constructor using <constructor-arg> tag rather than setter injection.

Below code fragment shows how to implement the same.

Modified#2 - MessageWriter.java

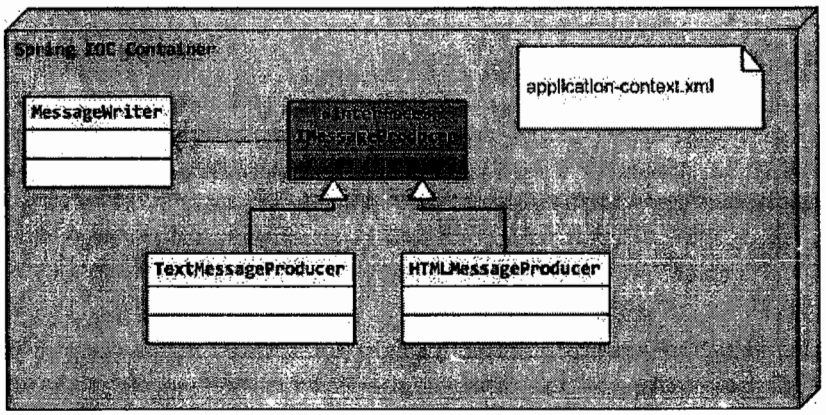
```
package com.ioc.beans;

public class MessageWriter {
  private IMessageProducer messageProducer;

  // constructor taking IMessageProducer parameter
  public MessageWriter(IMessageProducer messageProducer) {
     this.messageProducer = messageProducer;
  }

  public void writeMessage(String message) {
     String formattedData = messageProducer.formatMessage(message);
     System.out.println(formattedData);
  }
}
```

Spring Beans configuration File (application-context.xml)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
       <bean id="textMessageProducer"
             class="com.ioc.beans.TextMessageProducer" />
       <bean id="htmlMessageProducer"
             class="com.ioc.beans.HTMLMessageProducer" />
       <bean id="messageWriter" class="com.ioc.beans.MessageWriter">
             <constructor-arg ref="htmlMessageProducer"/>
       </bean>
</beans>
```

If you observe the above examples it's clearly evident that our application components are loosely coupled and instead of we creating the objects and managing the dependencies, spring is doing this for us. So, as we specify the things in configuration it is easy to maintain and modify without touching the source of our application.

Below diagram shows the IOC process through Setter or Constructor injection



- What will happen when we try to create the BeanFactory?

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/injection/common/application-context.xml"));
```

When we create a BeanFactory object, ClassPathResource will look for the application-context.xml under the class path of our project and loads and gives it as an input XmlBeanFactory. The XmlBeanFactory will take that Xml and performs well-formness and validity, once that Xml is well-formed and valid; it will place the Xml into the Jvm memory. This location where the spring bean objects and its metadata is places is called core container or ioc container. IOC Container is just a logical memory partition in which our spring beans will be maintained (it is similar to hashmap data structure).

The xml will be placed as in-memory metadata in the IOC container and gives the reference of BeanFactory object to us. When we call factory.getBean("messageWriter"), then the BeanFactory goes into its in-memory metadata of the IOC container search for the bean reference with that id. If found it will creates the object of the class by calling new operator and places the object in the IOC container with the given id and returns the reference of it.

So, all the beans we declared in spring bean configuration file are created and managed in the IOC Container in spring.

2.4 Constructor VS Setter Injection

As said above we have two types of Dependency Injection's, constructor injection and setter injection. Here are the points that make us understand when to use constructor inject and setter injection.

Constructor Injection	Setter Injection
• At the time of creating your target class object, the dependent objects are injected (can be accessed in the constructor of target class). • In case of constructor injection all the dependent objects are mandatory to be injected. If you don't provide any of the dependent objects through <contructor-arg> tag the core container will detects and throws BeanCreationException. • If classes have cyclic dependencies via constructor, these dependent beans cannot be configured through Constructor Injection.	• The dependent objects are not injected while creating the target classes object. Those will be injected after the target class has been instantiated, by calling the setter on the target object. • In case of setter injection your dependent objects are optional to be injected. Even you don't provide the <property> tag while declaring the bean; the container will creates the Bean and initializes all the properties to their default. • Cyclic dependencies are allowed in Setter Injection.

2.5 Resolving/Mapping Constructor Arguments

Even we pass the required amount of parameters in calling a constructor spring will not be able to determine the correct constructor for the arguments we passed and result in an error or in-correct mapping. This means spring will be confused sometimes in calling a constructor.

We can solve this confusion in three ways:

a) Using type attribute
b) Using index attribute
c) Using name attribute

Let's consider a case where spring cannot resolve the constructor arguments directly.

2.5.1 Using type attribute

Robot.java

```
package com.cc.beans;

public class Robot {
	private int id;
	private String name;
	private String type;

	public Robot(String name, String type) {
		this.type = type;
		this.name = name;
	}

	public Robot(int id, String type) {
		this.id = id;
		this.type = type;
	}

	@Override
	public String toString() {
		return "Robot [id=" + id + ", name=" + name + ", type=" + type + "]";
	}
```

}

In the above example the Robot class has two constructors one will take (String, String) and another takes (int, String) as parameter, if you configure this class as spring bean, you need to pass the argument one of these constructors. Let's say we are passing the values to the constructor as below

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="robot" class="com.cc.beans.Robot">
              <constructor-arg value="10"/>
              <constructor-arg value="Robot"/>
       </bean>
</beans>
```

Guess with the above bean definition which constructor will be called on our class? It will call (String, String) argument constructor, because by default any value that you configured in the spring bean configuration file will be treated as String, so even you pass "10" integer value spring will confuse and treats it as String rather than integer and invokes the two String argument constructor. To resolve this we need to use type attribute at the bean tag level as shown below.

Modified #1 - application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="robot" class="com.cc.beans.Robot">
              <constructor-arg value="10" type="int"/>
              <constructor-arg value="Robot"/>
       </bean>
</beans>
```

In the above configuration we are telling the value we are passing is of type "integer" and we are asking the spring to map it to an appropriate constructor as per it.

2.5.2 Using index attribute

Let us consider one more scenario where the same robot class contains one constructor as (int, String) as shown below.

Modified - #2 Robot.java

```
package com.cc.beans;

public class Robot {
	private int id;
	private String name;
	private String type;

	public Robot(int id, String name) {
		this.id = id;
		this.name = name;
	}

	@Override
	public String toString() {
		return "Robot [id=" + id + ", name=" + name + ", type=" + type + "]";
	}

}
```

If you try to configure this as spring bean, you need to configure the values in the same order of argument declaration. In case if the order mis-match then it will not be able to detect the relevant constructor. To resolve this you need to use index. Index lets you point the argument declaration to method parameters irrespective of the order in which those has been declared in configuration as shown below.

Modified - #2 application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="robot" class="com.cc.beans.Robot">
		<constructor-arg value="adf2" index="1"/>
		<constructor-arg value="10"/>
	</bean>
</beans>
```

When we use index, if there are n parameters for the constructor we need to specify the index value for n-1 parameters only.

2.5.1 Using name attribute

Instead of using index, the other way of mapping the parameters to the arguments of the constructor is using name attribute at the bean tag level, show as below.

```
package com.cc.beans;
import java.beans.ConstructorProperties;
public class Robot {
	private int id;
	private String name;
	private String type;
	@ConstructorProperties({ "id", "name" })
	public Robot(int id, String name) {
		this.id = id;
		this.name = name;
	}
	@Override
	public String toString() {
		return "Robot [id=" + id + ", name=" + name + ", type=" + type + "]";
	}
}
```

To use a name attribute at spring bean level to map the parameters, either we need to compile the code with –debug flag or we need to annotate our constructor with @ConstructorProperties as shown above.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="robot" class="com.cc.beans.Robot">
		<constructor-arg name="name" value="Andriod" />
		<constructor-arg name="id" value="10" />
	</bean>
</beans>
```

Run the above program you should see the output as expected.

2.6 Collection Injection

Spring not only supports inject primitives and objects to be injected as dependents into target classes, along with that spring even supports injected collections as dependent objects into the target class.

In spring you can inject four types of collections as dependent objects into your target classes. Those are List, Set, Map and Properties. Spring has provided convenient tags that allow you to create these objects in declarations and allow you to inject into your target classes.

1) Injecting List

Course.java

```
package com.cdi.beans;

import java.util.List;
import java.util.Properties;
import java.util.Set;

public class Course {
    private List<String> subjects;

    public void setSubjects(List<String> subjects) {
            this.subjects = subjects;
    }
    // toString();
}
```

In the above Course class we have List of subjects, while configuring the course class as a spring bean, we want to inject subjects list as well. In order to inject the list we need to use a <list> tag or <util:list> tag. Either using <list> or <util:list> has the same behavior, "util" namespace has been introduced from spring 2.0. The idea behind having the "util" namespace separately is to have namespace compartmentalization.
Below code snippet shows the configuration for injecting list.

application-context.xml

```
<bean id="bTechCS" class="com.cdi.beans.Course">
    <property name="subjects">
        <list value-type="java.lang.String">
            <value>C</value>
            <value>C++</value>
            <value>Java</value>
        </list>
    </property>
</bean>
```

2) Injecting Set

As you know the difference between list and set, list allows duplicates whereas set doesn't allow duplicates. You can inject set as dependent object using the tag <set>. Set has set of values, so the <set> tag contains <value> as child element, the same has been demonstrated in the below code.

Course.java

```
package com.cdi.beans;

import java.util.Set;

public class Course {
    private Set<String> faculties;

    public Course(Set<String> faculties) {
        this.faculties = faculties;
    }

    public void showFaculties() {
        System.out.println("Faculties :");
        for (String f : faculties) {
            System.out.println(f);
        }
    }
}
```

application-context.xml

```
<bean id="bTechCS" class="com.cdi.beans.Course">
    <constructor-arg>
        <set value-type="java.lang.String">
            <value>Mark</value>
            <value>John</value>
        </set>
    </constructor-arg>
</bean>
```

3) Injecting Map

Map is a collection which contains key and value pair. In case of Map the key can be any type and value can be any type. In order to create a map and inject into target class you need to use <map> tag. As map contains key and values the sub elements under it is <entry key=""><value></entry> tag. The below snippet shows the same.

University.java

```
package com.cdi.beans;

import java.util.Map;

public class University {
    private Map<String, Course> facultyCourseMap;

    public void setFacultyCourseMap(Map<String, Course> facultyCourseMap) {
        this.facultyCourseMap = facultyCourseMap;
    }

    public void showUniversityInfo() {
        System.out.println("University courses : ");
        for(String f : facultyCourseMap.keySet()) {
            System.out.println("********Course Info*****");
            Course c = facultyCourseMap.get(f);
            c.showSubjects();
            c.showFaculties();
            c.showFacultySubjects();
        }
    }
}
```

application-context.xml

```
<bean id="ou" class="com.cdi.beans.University">
    <property name="facultyCourseMap">
        <map key-type="java.lang.String" value-type="com.cdi.beans.Course">
            <entry key="mark">
                <ref bean="bTechCS"/>
            </entry>
            <entry key="john" value-ref="mca"/>
        </map>
    </property>
</bean>
```

4) Injecting Properties

Properties is also a Key and Value type collection, but the main difference between Map and Properties is Map can contain key and value as object type, but the Properties has key and value as string only.

In order to inject properties as dependent object, you need to use the tag <props>, this has sub elements <prop key="">value here</prop>.

Refer to the following example for the same.

Course.java

```
package com.cdi.beans;

import java.util.Properties;

public class Course {
    private Properties facultySubjects;

    public void setFacultySubjects(Properties facultySubjects) {
            this.facultySubjects = facultySubjects;
    }

    public void showFacultySubjects() {
            System.out.println("Faculty --> Subjects");
            for(Object o : facultySubjects.keySet()) {
                    System.out.print(o + " --> ");
                    System.out.println(facultySubjects.get(o));
            }
    }
}
```

application-context.xml

```
<bean id="mca" class="com.cdi.beans.Course">
    <property name="facultySubjects">
            <props>
                    <prop key="Mark">C</prop>
                    <prop key="John">S.E</prop>
            </props>
    </property>
</bean>
```

2.7 Bean Inheritance

Inheritance is the concept of reusing the existing functionality. In case of java you can inherit a class from an Interface or another class. When you inherit a class from another class, your child or derived class can use all of the functionalities of your base class.

A bean in spring is a class configured with some configuration around it using which spring creates the object for it. Bean inheritance is the concept of re-using the configuration property values of one bean inside another bean is called bean inheritance.

I have a class and I configured it as a bean with some values that has to be injected while creating. For the same class I want 10 beans of such type, so how many times I need to configure it as bean, with the same configuration changing the bean id I need to configure 10 beans.

This makes your configuration bulgier (more) and any change for one of the value of the property need to be changed across all the 10 beans, duplication of configuration results in more maintenance cost.

Is there any alternate to copy the values of one bean into another bean rather than re-declaring the entire configuration? That's where bean inheritance comes into picture.

For e.g. we own a car showroom and it has some cars, car is represented as an object of class Car. A showroom contains several cars to represent it we create more than one beans (objects) for each as shown below.

Car.java

```
package com.bi.beans;

public class Car {
	private int id;
	private String name;
	private String engineType;
	private String engineModel;
	private String classType;

	public void setId(int id) {
		this.id = id;
	}

	public void setName(String name) {
		this.name = name;
	}

	public void setEngineType(String engineType) {
		this.engineType = engineType;
	}

	public void setEngineModel(String engineModel) {
		this.engineModel = engineModel;
	}

	public void setClassType(String classType) {
		this.classType = classType;
	}
	// override to string
}
```

application-context.xml

```
<bean id="swift1" class="com.bi.beans.Car">
	<property name="id" value="1"/>
	<property name="name" value="Swift"/>
	<property name="engineType" value="Disel"/>
	<property name="engineModel" value="DDIS"/>
	<property name="classType" value="HatchBack"/>
</bean>
<bean id="swift2" class="com.bi.beans.Car">
	<property name="id" value="2"/>
	<property name="name" value="Swift"/>
	<property name="engineType" value="Disel"/>
	<property name="engineModel" value="DDIS"/>
	<property name="classType" value="HatchBack"/>
</bean>
```

In this way if we have 10 cars in the showroom we need to declare 10 beans in the spring bean configuration file, but if we closely observe all the beans contains the configuration values as same across. This results out in duplication of configuration. Rather than declaring the values again in all the beans here we can reuse the properties of swift1 in swift2 car.

Make the swift2 car inherit from swift1 car so that the values of the swift1 car will be copied into swift2 car and we can avoid declaring the properties again as shown below.

application-context.xml

```
<bean id="swift1" class="com.bi.beans.Car" parent="baseCar">
	<property name="id" value="1"/>
	<property name="name" value="Swift"/>
	<property name="engineType" value="Disel"/>
	<property name="engineModel" value="DDIS"/>
	<property name="classType" value="HatchBack"/>
</bean>
<bean id="swift2" class="com.bi.beans.Car" parent="swift1">
	<property name="id" value="2"/>
</bean>
```

Now the name, engineType, engineModel and classType property values will be copied from "swift1" car so we avoid duplication of configuration.

The bean that is being inherited is called parent bean, the bean that is getting inherited is called child bean of that parent. Always the parent bean property values will be copied to the child beans.

In the above example, the "swift2" car id will be different from "swift1" car id as every car has its own id, so "swift2" car has re-declared the value for the id property, it means the child bean property values will always overwrites the parent bean property values.

In the above example if we observe the "swift1" car is an active bean from which other cars are inheriting from. If any changes to "swift1" car will affect all the other cars as well. To avoid this we need to take one common base car which will contains common property values that can be used by all the other classes.

application-context.xml

```
<bean id="baseCar" class="com.bi.beans.Car">
        <property name="name" value="Swift"/>
        <property name="engineType" value="Disel"/>
        <property name="engineModel" value="DDIS"/>
        <property name="classType" value="HatchBack"/>
</bean>
<bean id="swift1" class="com.bi.beans.Car" parent="baseCar">
        <property name="id" value="1"/>
</bean>
<bean id="swift2" class="com.bi.beans.Car" parent="baseCar">
        <property name="id" value="2"/>
</bean>
```

Now in the above case as baseCar is a dummy bean that is declared to hold common values that can be reused, we can declare it as abstract bean so that spring will not create object for it rather it uses its configuration for inheritance.

```
<bean id="baseCar" class="com.bi.beans.Car" abstract="true">
        <property name="name" value="Swift"/>
        <property name="engineType" value="Disel"/>
        <property name="engineModel" value="DDIS"/>
        <property name="classType" value="HatchBack"/>
</bean>
```

Few points to remember:

- When we use bean inheritance, the parent bean and the child bean class types are not necessary to be same. All the properties of the parent bean must and should be present in child bean only
- When we inherit one bean from another bean, only the parent bean property values will be copied into child bean property values but the physical classes will never get extended.
- It is recommended to declare one of the beans as abstract bean which contains all the common values that should be inherited to the child. As it is an abstract bean we never modify any property unless it has to be affected to all.
- When we declare a bean as abstract, the class will not become abstract only the current bean definition will become abstract, which means spring will not create the object of that bean. Any call to factory.getBean("abstractbean") to an abstract bean will results out in an error.

```
package com.bi.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.bi.beans.Car;

public class BITest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/cc/common/application-context.xml"));
		Car car = factory.getBean("swift1", Car.class);
		System.out.println(car);
	}
}
```

In the above configuration we declared baseCar as abstract which means spring IOC container will not instantiate the object for the bean declaration. But it acts as a base bean from which its property values will be inherited to child beans.

Now when we run the above program, the swift1 car will inherit the values of name, engineType, engineModel, classType from baseCar.

2.8 Collection Merging

Let us try to understand the collection merging by taking one example, we have a class Course and it has list of subjects. And we create multiple beans of this class to represent different courses offered in a college, let's say btechCS1Yr1Sem and btechECE1Yr1Sem as shown below.

Course.java

```
package com.cm.beans;

import java.util.List;

public class Course {
	private List<String> subjects;

	public void setSubjects(List<String> subjects) {
		this.subjects = subjects;
	}

	// override toString() method
}
```

application-context.xml

```
<bean id="bTechCS1Yr1Sem" class="com.cm.beans.Course">
	<property name="subjects">
		<list>
			<value>c</value>
			<value>DMS</value>
		</list>
	</property>
</bean>
<bean id="bTechECE1Yr1Sem" class="com.cm.beans.Course">
	<property name="subjects">
		<list>
			<value>c</value>
			<value>DMS</value>
			<value>S.E</value>
		</list>
	</property>
</bean>
```

Now if you see the above configuration few subjects are common across different courses even then also we need to re-declare those. Instead of re-declaring them to reuse we can use bean inheritance. So, now inherit the bTechECE1Yr1Sem bean from bTechCS1Yr1Sem bean as shown below.

application-context.xml

```
<bean id="bTechCS1Yr1Sem" class="com.cm.beans.Course">
	<property name="subjects">
		<list>
			<value>c</value>
			<value>DMS</value>
		</list>
	</property>
</bean>
<bean id="bTechECE1Yr1Sem" class="com.cm.beans.Course"
parent="bTechCS1Yr1Sem">
	<property name="subjects">
		<list>
			<value>S.E</value>
		</list>
	</property>
</bean>
```

Can you guess how many subjects will be there in bTechECE1Yr1Sem bean, it will have only on that is 'S.E'! why? Always in case of inheritance the parent bean property value will be overwritten by child bean property values, so the 'C' & 'DMS' will be overwritten by 'S.E' value. But we want the bTechECE1Yr1Sem bean to have all the values including 'C' & 'DMS' is it possible? Yes for that you need to use collection merging.

If the parent bean property is a collection type, in the child bean if we have a similar property of the same collection type, we can merge the values of parent property collection with the child using collection merging as shown below.

```
<bean id="bTechECE1Yr1Sem" class="com.cm.beans.Course"
parent="bTechCS1Yr1Sem">
        <property name="subjects">
                <list merge="true">
                        <value>S.E</value>
                </list>
        </property>
</bean>
```

Now the bTechECE1Yr1Sem subjects will have all the three including the parent one as well as we said "merge=true".

Few points to remember about collection merging:

a) Collection merging always comes into picture in-case of bean inheritance only.
b) The parent property collection type and child property collection type we are merging should be of same type.
c) The parent property collection generic type and the child property collection generic type should always be same

Here is the code to test the same.

```
package com.cm.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.cm.beans.Course;

public class CMTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/cm/common/application-context.xml"));
                Course course = factory.getBean("bTech1Yr1Sem", Course.class);
                System.out.println(course);
        }
}
```

2.9 Inner Beans

An inner bean is the concept similar to Inner classes in Java. As how you can create a class inside another class, you can inject a bean into another bean by declaring it inline. Below snippet shows the same.

BiCycle.java

```
package com.ib.beans;

public class BiCycle {
	private Chain chain;

	public void setChain(Chain chain) {
		this.chain = chain;
	}
	// override toString()
}
```

Chain.java

```
package com.ib.beans;

public class Chain {
	private String type;

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}
	// override toString()
}
```

application-context.xml

```
<bean id="biCycle" class="com.ib.beans.BiCycle">
	<property name="chain">
		<bean class="com.ib.beans.Chain">
			<property name="type" value="t1"/>
		</bean>
	</property>
</bean>
```

```
package com.ib.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.ib.beans.BiCycle;

public class IBTest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/ib/common/application-context.xml"));
		BiCycle bc = factory.getBean("biCycle", BiCycle.class);
		System.out.println(bc);
	}
}
```

In the above case if you observe Chain cannot be used independently without a BiCycle, so why it should have an independent existence?

As no one is going to use Chain and as it is only being used by BiCycle, if we declare the Chain inside the BiCycle all the related configuration related to BiCycle will be there at one place and it would be easy to manage and understand the configuration at single shot.

2.10 Using IDRef

In some cases a bean wants to use the id of another bean, so how to inject "id" of a bean into another bean, here we need to use IDRef.

For example Car needs an Engine to run, to use Engine inside Car we can inject Engine into the Car, but I don't want to inject rather we want to pull, to pull we need the id of the Engine inside the car. So, we are injecting the "id" of the engine into the Car as shown below.

Car.java

```
package com.idref.beans;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.idref.beans.Engine;
public class Car {
	private String beanId;

	public void setBeanId(String beanId) {
		this.beanId = beanId;
	}

	public void run() {
		Engine engine = null;
		BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("com/idref/common/application-context.xml"));
		engine = factory.getBean(beanId, Engine.class);
		engine.start();
		System.out.println("Running....");
	}
}
```

Engine.java

```
package com.idref.beans;

public class Engine {
	public void start() {
		System.out.println("Started...");
	}
}
```

application-context.xml

```
<bean id="car" class="com.idref.beans.Car">
      <property name="beanId" value="engine"/>
</bean>

<bean id="engine" class="com.idref.beans.Engine" />
```

```
package com.idref.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.idref.beans.Car;

public class IDRefTest {
      public static void main(String[] args) {
            BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                  "com/idref/common/application-context.xml"));
            Car car= factory.getBean("car", Car.class);
            System.out.println(car);
      }
}
```

Will the above program work's? Yes but there is a problem with the above piece of code. The problem here is we are passing the bean id "engine" as value into the Car, If the bean id has been changed let's say from "engine" to "engine1" then it would be difficult to track and change the as the property is configured as value.

- Developer will not be able to understand upon looking into configuration whether to modify or not as it looks like a simple value.
- Spring will try to create the IOC Container even the configuration is in-consistent, which will results in an runtime exception while getting the Engine using factory.getBean

To avoid the above problems it is recommended to pass the id of another bean as value using idref tag. "ref" means referring an object, "idref" means ref to an id of another bean. Below example that shows the same.

application-context.xml

```
<bean id="car" class="com.idref.beans.Car">
      <property name="beanId">
         <idref bean="engine"/>
      </property>
</bean>

<bean id="engine" class="com.idref.beans.Engine" />
```

With the above spring while creating the Car bean in the IOC container, it checks is any bean with id as "engine" exists if found then only creates the car and injects the value "engine" into the Car. Otherwise will stop creating the object and throws an exception.

2.11 Bean Aliasing

In spring when you configure a class as Bean you will declare an id with which you want to retrieve it back from the container. Along with id you can attach multiple names to the beans, and these names acts as alias names with which you can look up the bean from the container.

Prior to spring 2.0 in order to declare multiple names you need to declare an "name" attribute at the bean tag level whose value contains bean names separated with ",".

Following code snippet shows the same.

```
package com.ba.beans;

public class Robot {
	private int id;
	private String name;

	// setters
	@Override
	public String toString() {
		return "Robot [id=" + id + ", name=" + name + "]";
	}
}
```

application-context.xml

```
<bean id="robot" name="agent, machine" class="com.ba.beans.Robot">
	<property name="id" value="10"/>
	<property name="name" value="Robot-1"/>
</bean>
```

You can retrieve the above bean with either robot or agent or machine names. You can even get all the names of the bean using factory.getAliases("onename").

In general bean aliasing is used for ease maintenance of the configuration.

In spring 2.0 a new tag has been introduced <alias> using which you can declare multiple names for the bean. The syntax is as follows

```
<bean id="robot" class="com.ba.beans.Robot"/>
<alias name="agent" alias="robot"/>
<alias name="machine" alias="robot"/>
```

```
package com.ba.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.ba.beans.Robot;

public class BATest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/ba/common/application-context.xml"));
                // this will work as agent is alias name of robot bean
                Robot robot= factory.getBean("agent", Robot.class);
                System.out.println(robot);

               String[] aliases = factory.getAliases("robot");
               for(String alias : aliases) {
                  System.out.println("alias : " + alias);
              }
        }
}
```

2.12 Null String

The concept of Null string is how to pass Null Value for a Bean property. Let's consider a case where a class has attribute as String or other Object. We are trying to inject the value of this attribute using Constructor Injection.

In case of constructor injection the dependent object is mandatory to be injected in configuration. In case if the dependent object is not available, you can pass null for the dependent object in the target class constructor as shown below.

Motor.java

```
package com.un.beans;

public class Motor {
        private String id;

        public Motor(String id) {
                this.id = id;
        }
        // toString()
}
```

application-context.xml

```
<bean id="m1" class="com.un.beans.Motor">
        <constructor-arg>
                <null/>
        </constructor-arg>
</bean>
```

If you see the declaration, in order to pass null as value for the dependent object you need to use the tag <null/>.

```
package com.un.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.un.beans.Motor;

public class UNTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/un/common/application-context.xml"));
                Motor motor= factory.getBean("motor", Motor.class);
                System.out.println(motor);
        }
}
```

With the above the Motor object will be created by initializing the String id to null.

2.13 Bean Scopes

In spring when you declare a class as a bean by default the bean will be created under singleton scope. Before understanding about scopes we need to understand what singleton class is and when to use it.

What is singleton, when to use.

When we create a class as singleton, it means we have only one instance of the class within the classloader.

We need to use singleton class in the below scenarios.

1) If a class has absolutely no state then declare those classes as singleton, as the class doesn't contains any attributes and it has only behavior's, calling the methods with object1 or object 2 doesn't makes any difference. So, instead of floating multiple objects in memory, we can have one object calling any methods of that class.

2) If a class has some state, but the state is read-only in nature then all the objects of that class sees the same state, so using the class behavior's with one object or n objects doesn't makes any difference, so we can use only one object of the class rather than multiple.

3) If a class has some state, but the state can be shared across multiple objects of the class. The state it contains is not only sharable but also it is very huge in nature, so instead of declaring multiple objects of that class we can allow to access the shared state through one object. But should allow the write/read access to the state via serialized order (synchronized fashion). For e.g.. all the cache classes has shared state which is shared across multiple objects of that class, but write/read operations to the cache data will be allowed in a synchronized manner.

In all the above scenarios we need to declare the class as Singleton. If a class is inverse of the above principles we should not declare the class as singleton rather should create multiple instances to access it.

In spring you can declare a bean with 5 different scopes as follows.

1) Singleton – by default every bean declared in the configuration file is defaulted to singleton (unless specified explicitly). This indicates when you try to refer the bean through injection or factory.getBean() the same bean instance will be returned from the core container.

2) Prototype – When we declare a bean scope as prototype this indicates every reference to the bean will return a unique instance of it.

3) Request – When we declare a bean scope as request, for every HTTPRequest a new bean instance will be injected

4) Session – For every new HttpSession, new bean instance will be injected.

5) Global Session – the globalsession scope has been removed from Spring 3.0. This is used in Spring MVC Portlet framework where if you want to inject a new bean for a Portal Session you need to use this scope.

By the above it is clear that you can use request and session in case of web applications. So we will postpone the discussion on these till Spring MVC.

Let's understand how to use singleton and prototype.

DateUtil.java

```
package com.bs.beans;

import java.text.SimpleDateFormat;
import java.util.Date;

public class DateUtil {
	public String formatDate(Date dt, String pattern) {
		String s = null;
		SimpleDateFormat sdf = new SimpleDateFormat(pattern);
		s = sdf.format(dt);
		return s;
	}
}
```

application-context.xml

```
<bean id="dateUtil" class="com.bs.beans.DateUtil" scope="prototype"/>
```

BeanScopeTest.java

```
package com.ba.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;
import com.bs.beans.DateUtil;

public class BSTest {
        public static void main(String[] args) {
BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/bs/common/application-context.xml"));
                DateUtil du1 = factory.getBean("dateUtil", DateUtil.class);
                DateUtil du2 = factory.getBean("dateUtil", DateUtil.class);
                System.out.println(du1 == du2);
        }
}
```

As in the configuration we declare the dateUtil bean scope as prototype the comparision between du1 == du2 will returns false. If we set the scope as singleton it will yields to true.

2.14 Bean Autowiring

In spring when you want to inject one bean into another bean, we need to declare the dependencies between the beans using <property> or <constructor-arg> tag in the configuration file. This indicates we need to specify the dependencies between the beans and spring will reads the declarations and performs injection.

But when it comes to autowiring, instead of we declaring the dependencies we will instruct spring to automatically detect the dependencies and perform injection between them.

So, in order to do this we need to enable autowiring on the target bean into which the dependent has to be injected. You can enable autowiring in 4 modes.

1) byname – If you enable autowiring by name, spring will finds the attribute name which has setter on the target bean, and finds the bean in configuration whose name is matching with the attribute name and performs the injection by calling the setter. Following code demonstrates the same.

Humpty.java

```
package com.ba.beans;

public class Humpty {
	private Dumpty dumpty;

	public void setDumpty(Dumpty dumpty) {
		System.out.println("Setter");
		this.dumpty = dumpty;
	}

	public void showHumpty() {
		System.out.println("I am working with Dumpty : " + dumpty.getName());
	}
}
```

Dumpty.java

```
package com.ba.beans;

public class Dumpty {
	private String name;

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

application-context.xml

```
<bean id="humpty" class="com.ba.beans.Humpty" autowire="byName"/>

<bean id="dumpty" class="com.ba.beans.Dumpty">
	<property name="name" value="Dumpty12"/>
</bean>
```

If you see the above configuration, on the humpty we enabled autowire byName. With this configuration, it will tries to find the Humpty beans attributes which has setters, with that attribute name "dumpty" it will tries to find a relevant bean with the same name "dumpty" as the bean is available it will injects the dumpty bean into Humty attribute.

2) byType – If we enable autowire byType, if will find the attributes type in the target class and tries to identify a bean from the configuration file of the same type and then injects into target class by calling setter on top of it. Below configuration demonstrates the same.

application-context.xml

```
<bean id="humpty" class="com.ba.beans.Humpty" autowire="constructor"/>

<bean id="dumpty12" class="com.ba.beans.Dumpty">
        <property name="name" value="Dumpty12"/>
</bean>
```

In the above case the bean attribute name is "dumpty", and in the configuration the bean name is "dumpty12" even though the names are not matching still the dumpty12 bean will be injected into humpty. Because the type of the attribute and the bean type is matching.

Note:- if multiple bean declarations of the same type is found it will throw an ambiguity error without instantiating the core container.

3) Constructor – If we enable autowire in constructor mode, now it will tries to find a bean whose class type is same as constructor parameter type, if a matching constructor is found it will passes the bean reference to its constructor and performs injection. This means it is similar to byType but instead of calling setter it will call constructor to perform injection.

Humpty.java

```
package com.ba.beans;

public class Humpty {
        private Dumpty dumpty;

        public Humpty() {
                super();
        }
        public Humpty(Dumpty dumpty) {
                System.out.println("Constructor");
                this.dumpty = dumpty;
        }

        public void showHumpty() {
                System.out.println("I am working with Dumpty : " + dumpty.getName());
        }
}
```

There is no change in configuration apart from declaring on the bean autowire="constructor"

4) Autodetect – this has been removed from spring 3.0 onwards as it is quite confusing. In this it will tries to perform injection by finding a relevant constructor by type if not found then it will finds the setter by type and performs injection.

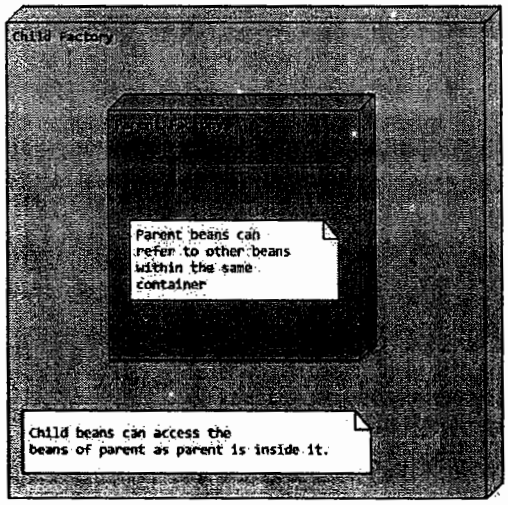
Drawback with autowiring – The problem with autowiring is we don't have control over which beans has to get injected into what, so it is least recommended to use autowiring for a large applications. For pilot projects where RAPID application development is needed we use autowiring.

2.15 Nested BeanFactories

If we have two bean factories in an application, we can nest one bean factory into another to allow the beans in one bean factory to refer to the beans of other factory. In this we declare one bean factory as parent bean factory and will declare the other as child.

This is similar to the concept of base class and derived classes. Derived class can access the attributes of base class, but base class cannot access the attributes of derived class.

In the same way child bean factory beans can refer to the parent bean factory beans. But parent bean factory beans cannot refer to child beans declared in child factory. Pictorial representation of it is shown below.



Below example shows how to use it.

EMICalculator.java

```
package com.nbf.beans;

public class EMICalculator {
	public float compute(long principal, float rateOfInterest, int years) {
		return 343.34f;
	}
}
```

CustomerLoanApprover.java

```
package com.nbf.beans;

public class CustomerLoanApprover {
	private EMICalculator emiCalculator;

	public void approve(double grossSalary, long principalAmount, int years) {
		float emi = emiCalculator.compute(principalAmount, 13.5f, years);
		System.out.println("Emi : " + emi);
		if (emi > 0) {
			System.out.println("Approved");
		} else {
			System.out.println("Rejected");
		}
	}

	public void setEmiCalculator(EMICalculator emiCalculator) {
		this.emiCalculator = emiCalculator;
	}

}
```

If the EMICalculator class has been declared in one configuration file and CustomerLoanApprover class has been declared in second configuration file. In order to inject EMICalculator into CustomerLoanApprover class we need to nest their factories as shown below.

loan-beans.xml

```
<bean id="emiCalculator" class="com.nbf.beans.EMICalculator"/>
```

customer-beans.xml

```
<bean id="customerLoanApprover" class="com.nbf.beans.CustomerLoanApprover">
	<property name="emiCalculator">
		<ref parent="emiCalculator"/>
	</property>
</bean>
```

In the above configuration in order for your bean declaration to refer to parent beans, it has to use the tag <ref parent = ""/>. Apart from parent attribute it has local which indicates refer to the local bean. Along with it we have bean attribute as well, which indicates look in local if not found the search in parent factory and perform injection.

NBFTest.java

```
public static void main(String[] args) {
        BeanFactory pf = new XmlBeanFactory(new ClassPathResource(
                        "com/nbf/common/loan-beans.xml"));
        BeanFactory cf = new XmlBeanFactory(new ClassPathResource(
                        "com/nbf/common/customer-beans.xml"), pf);

        CustomerLoanApprover cla = cf.getBean("customerLoanApprover",
                        CustomerLoanApprover.class);
        cla.approve(3423.3f, 3535, 324);
}
```

If you observe the above code while creating the "cf" factory we passed the reference of "pf" to create it. This indicates "cf" factory has been nested from "pf".

This completes the Spring Core basic concepts and enables us to proceed for advanced spring core concepts.

3 Spring Core (Advanced)

3.1 Using P & C – Namespace

If we want to perform setter injection on a spring bean we need to use <property> tag. Instead of writing length <property> tag declaration under the <bean> tag, we can replace with short form of representing the same with p-namespace.

In order to use the p-namespace, you first need to import the "http://www.springframework.org/schema/p" namespace in the spring bean configuration file. Once you have imported it, you have to write the attribute at the <bean> tag level to perform the injection as **p:propertyname="value"** or **p:propertyname-ref="refbean"**.

C-Namespace has been introduced in spring 3.1.1, in order to perform constructor injection we need to use <constructor-arg> tag. Instead of writing the length <constructor-arg> tag, we can replace it with c-namespace. The syntax for writing the C-Namespace is **c:argument="value"** or **c:-argument-ref="refbean"**

Course.java

```
package com.pnamespace.beans;

public class Course {
	private int id;
	private String name;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

}
```

Person.java

```
package com.pnamespace.beans;

public class Person {
	private Course course;

	public Person(Course course) {
		this.course = course;
	}

	public void whichCourse() {
		System.out.println("Course : " + course.getName());
	}
}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns:c="http://www.springframework.org/schema/c"
	xmlns:p="http://www.springframework.org/schema/p"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="course" class="com.pnamespace.beans.Course" p:id="34"
p:name="Java"/>
	<bean id="person" class="com.pnamespace.beans.Person" c:course-
ref="course"/>
</beans>
```

3.2 Dependency Check

In case of constructor injection, all the dependent objects are mandatory to be passed via <constructor-arg> tag while declaring the bean, but in case of setter injection the dependent objects are not mandatory to be injected. So, if want setter properties mandatory like similar to constructor properties then we need to use Dependency Check.

Dependency check has been removed from Spring 2.5. From spring 2.5 it has been replaced with @Required annotation, we will discuss about it in spring annotations support.

In order to perform the mandatory check on setter properties you need to enable dependency check. In order to enable dependency check you have to write the attribute **dependency-check="mode"**. Dependency check can be enabled in three modes.

1) Simple – when you turn the dependency check in simple mode, it will check all the primitive attributes of your bean (attributes contains setters) whether those has been configured with values in configuration file. If any primitive attribute is not configured with <property> or p-namespace in configuration automatically the core container will detects and throws error without creating the container.

2) Object- when you turn on the dependency check in object mode, it will check all the Object type attributes on your target class whether those has been configured the property injection in configuration, if not will throw exception as said above.

3) all – when you turn on the dependency check as all mode, it will check for both simple and objects types of your class attributes, if those values are not injected through configuration it will throw exception.

Refer the example for the same.

Engine.java

```
package com.dc.beans;

public class Engine {
	private int id;
	private String name;

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}
}
```

Motor.java

```
package com.dc.beans;

public class Motor {
	private Engine engine;

	public void setEngine(Engine engine) {
		this.engine = engine;
	}

	public void run() {
		System.out.println("Running with engine : " + engine.getName());
	}
}
```

application-context.xml

```
<bean id="engine" class="com.dc.beans.Engine" dependency-check="simple">
	<property name="id" value="24"/>
	<property name="name" value="100 cc"/>
</bean>

<bean id="motor" class="com.dc.beans.Motor" dependency-check="objects">
	<property name="engine" ref="engine"/>
</bean>
```

The main drawback with dependency check is you don't have control on which attribute should be made as mandatory and which is optional, either you can the dependency check at simple level or object or all.

3.3 Depends-On

If a class uses the functionality of other class inside it, then those classes are directly dependent on each other. But every class may not directly dependent on other; sometimes there may be indirect dependency. Let us consider a use case to understand.

We have a class LoanCalculator, it has a method calInst to it we pass principle, noofmonths and city name as input. When we call this method it should find the applicable rate of interest for the city we passed by fetching it from properties file.

```
package com.don.beans;
import java.util.Properties;

public class LoanCalculator {
	public double calInst(Long principle, int n, String city) throws Exception {
		float ri = 0.0f;
		double amt = 0.0f;
		Properties cityInsProps = null;

		// read the data from properties file
		// load into properties collection
		// now find the rate of interest for the given city

		amt = (principle * n * ri) / 100;
		return amt;
	}
}
```

The problem with above code is we will read the same data repeatedly for every call to the calIns method of the LoanCalculator class.

So instead we want to cache the data so that the LoanCalculator now can go and get the data from cache. So for this when we call calIns method we will check whether the data is available with cache or not. If it is not available we will read the city and rate of Interest values from properties file as key and values and place in properties collection.

Now we will store this properties collection into the Cache as shown below. As now the data is available in cache we can pull the data from Cache itself.

```
package com.don.beans;
public class LoanCalculator {
	public double calInst(Long principle, int n, String city) throws Exception {
		float ri = 0.0f;
		double amt = 0.0f;
		Cache cache = null;
		Properties cityInsProps = null;

		cache = Cache.getInstance();
		if (cache.containsKey("cityRI") == false) {
			// read the data from properties file
			// make it as properties collection
			// store in the cache with key as "cityRI"
		}
		cityInsProps = cache.get("cityRI");
		ri = Float.parseFloat(cityInsProps.getProperty(city));
		amt = (principle * n * ri) / 100;
		return amt;
	}
}
```

Is there any problem in the above piece of logic, yes the problem is here the city and rate of interest values may not only required for LoanCalculator there may be several other classes in the application want to read the data from Cache. So, we need to write the logic for checking the data is available in the cache if not load the data into cache and use it.

So, we end up in duplicating the same logic across various places in the application. Instead can we write the logic for populating the data into cache in cache constructor? Yes it seems a good idea because constructor of the cache will be called only once as it is singleton and we need to read the data only once, the better place to write the logic for populating is in constructor of the class as shown below.

```
package com.don.beans;

public class LoanCalculator {
	public double calInst(Long principle, int n, String city) throws Exception {
		float ri = 0.0f;
		double amt = 0.0f;
		Cache cache = null;
		Properties cityInsProps = null;
		cache = Cache.getInstance();
		cityInsProps = (Properties) cache.get("cityRI");
		if (cityInsProps == null) {
			throw new Exception("Internal error");
		}
		if (cityInsProps.containsKey(city) == false) {
			throw new Exception("City not valid");
		}
		ri = Float.parseFloat(cityInsProps.getProperty(city));
		amt = (principle * n * ri) / 100;
		return amt;
	}
}
```

```
package com.don.util;

public class Cache {
	private Map<String, Object> dataMap;

	public Cache() {
		dataMap = new HashMap<String, Object>();
		// write the logic for reading the data from properties file
		// put that into properties collection
		// and store it into dataMap
	}
	public void put(String key, Object val) {
		dataMap.put(key, val);
	}
	public Object get(String key) {
		return dataMap.get(key);
	}
	public boolean containsKey(String key) {
		return dataMap.containsKey(key);
	}
}
```

Is there any problem with the above code, yes? Here the cache is exposed to the details of the source from where the data is coming from. Due to which any changes to the underlying source again will affect the cache.

Instead we should never write the logic for populating the data into cache rather we should write that logic in separate class called CacheManager. CacheManager is the one responsible for reading the data from underlying source system and massaging the data and storing the data into Cache.

```
package com.don.util;

public class CacheManager {
	public CacheManager() {
		init();
	}
	public void init() {
		Cache cache = null;

		cache = Cache.getInstance();
		// read the data and populate into cache
	}
}
```

Again is there any problem with above code? Yes, as the data comes from several source systems again we will write multiple access related logic in one single class. This may become complicated and difficult to manage and modify.

Instead write the logic for reading the data from source system in Accessor class. Now CacheManager will talk to Accessors in retrieving the data and populating into cache.

Now if we observe LoanCalculator is dependent on Cache or CacheManager to read the data. LoanCalculator never talks to CacheManager rather it reads the data from Cache. But inorder to populate the data into Cache, the CacheManager should be created first than LoanCalculator.

So here the LoanCalculator and CacheManager are indirectly dependent on each other. Before the LoanCalculator gets created, always the CacheManager has to be created first. This is called creational dependencies these can be managed using depends on as shown below.

```
package com.don.util;

import java.util.Map;

import com.don.accessor.IAccessor;

public class CacheManager {
	private Cache cache;
	private Map<String, IAccessor> accessorMap;

	public CacheManager(Cache cache, Map<String, IAccessor> accessorMap) {
		this.cache = cache;
		this.accessorMap = accessorMap;
		init();
	}

	public void init() {
		Object data = null;
		IAccessor accessor = null;

		for (String dataKey : accessorMap.keySet()) {
			accessor = accessorMap.get(dataKey);
			data = accessor.getData();
			cache.put(dataKey, data);
		}
	}
}
```

```
package com.don.util;

import java.util.HashMap;
import java.util.Map;

public class Cache {
	private Map<String, Object> dataMap;

	public Cache() {
		dataMap = new HashMap<String, Object>();
	}

	public void put(String key, Object val) {
		dataMap.put(key, val);
	}

	public Object get(String key) {
		return dataMap.get(key);
	}

	public boolean containsKey(String key) {
		return dataMap.containsKey(key);
	}

}
```

```
package com.don.accessor;
public interface IAccessor {
	Object getData();
}
```

```
package com.don.accessor;
import java.util.Properties;
import java.util.ResourceBundle;
public class CityInterestAccessor implements IAccessor {
	@Override
	public Object getData() {
		ResourceBundle rb = null;
		Properties props = null;

		rb = ResourceBundle.getBundle("com/don/common/cityins");
		props = new Properties();
		for (String key : rb.keySet()) {
			props.put(key, rb.getString(key));
		}
		return props;
	}
}
```

```
package com.don.beans;

import java.util.Properties;
import com.don.util.Cache;
public class LoanCalculator {
	private Cache cache;
	public double calInst(Long principle, int n, String city) throws Exception {
		float ri = 0.0f;
		double amt = 0.0f;
		Properties cityInsProps = null;
		cityInsProps = (Properties) cache.get("cityRI");
		if (cityInsProps == null) {
			throw new Exception("Internal error");
		}
		if (cityInsProps.containsKey(city) == false) {
			throw new Exception("City not valid");
		}
		ri = Float.parseFloat(cityInsProps.getProperty(city));
		amt = (principle * n * ri) / 100;
		return amt;
	}
	public void setCache(Cache cache) {
		this.cache = cache;
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="loanCalculator" class="com.don.beans.LoanCalculator" depends-
on="cacheManager">
		<property name="cache" ref="cache" />
	</bean>
	<bean id="cache" class="com.don.util.Cache" />
	<bean id="cacheManager" class="com.don.util.CacheManager">
		<constructor-arg ref="cache" />
		<constructor-arg>
			<map key-type="java.lang.String" value-
type="com.don.accessor.IAccessor">
				<entry key="cityRI" value-ref="cityInsAccessor" />
			</map>
		</constructor-arg>
	</bean>
	<bean id="cityInsAccessor" class="com.don.accessor.CityInterestAccessor" />
</beans>
```

cityins.properties

```
hyd=12.12
chn=11.12
blr=10.23
```

```
package com.don.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.don.beans.LoanCalculator;

public class DONTest {
	public static void main(String[] args) throws Exception {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/don/common/application-context.xml"));
		LoanCalculator lc = factory.getBean("loanCalculator",
				LoanCalculator.class);
		double amt = lc.calInst(100L, 12, "hyd");
		System.out.println("Interest amount : " + amt);
	}
}
```

In the above code for the bean id loanCalculator we added one attribute depends-on pointing to the cacheManager. This means before creating the loanCalculator always Spring IOC container should create cacheManager first.

3.4 Bean Lifecycle

Bean lifecycle allows a way to provide spring beans to perform initialization or disposable operations.

If you consider a Servlet, J2EE containers will call the init and destroy methods after creating the servlet object and before removing it from the web container respectively. In the Servlet init method, developer has to write the initialization logic to initialize the Servlet and destroy method he needs to write the code for releasing the resources to facilitate the destruction process.

If you consider a POJO in Java, initialization logic has to be written in the constructor and resource release logic should be written in the finalize method.

So, for a spring bean also can we use Constructor and finalize as the lifecycle methods. The answer here is "No". When we are performing initialization always we don't want to initialize with default or hardcoded values sometimes we wanted to initialize the state of a bean with user supplied values. How to pass the values as input while creating the bean in spring. There are two ways one is constructor injection and other

is setter injection. If we recall the setter injection values are not accessible within the constructor of the class as those gets injected after the class has been created. This indicates we cannot initialize the state of the bean in constructor as all the values are not available.

We need a separate initialization hookup that will be called after all the values are injected so, that we can write the initialization logic. That's where BeanLifecycle comes into picture.

In order to do this spring provides three ways to work with Bean Lifecycle.

1) Declarative approach
2) Programmatic approach
3) Annotation based approach

3.4.1 Declarative approach

In the declarative approach you will declare the init and destroy methods of a bean in spring beans configuration file. In the bean class declare the methods whose signature must be public and should have return type as void and should not take any parameters, any method which follows this signature can be used as lifecycle methods.

After writing the methods in your bean, you need to declare those methods as lifecycle methods in spring configuration file, at the <bean> tag level, you need to declare two attributes **init-method = "methodname"** and **destroy-method = "destroymethod"**.

IOC container after creating the bean (this includes after all injections); it will automatically calls the init method to perform the initialization. But spring IOC container cannot automatically calls the destroy-method of the bean class, because in spring core application IOC container will not be able to judge when the bean is available for garbage collection or how many references of this bean is held by other beans.

In order to invoke the destroy-method on the bean class you need to explicitly call on the ConfigurableListableBeanFactory, destroySingleton or destroyScopedBean("beanscope") method, so that IOC container delegates the call to all the beans destroy-methods in that IOC container.

Refer to the below example on how to work with declarative approach.

Robot.java

```
package com.blc.beans;

public class Robot {
  private String name;
  private SensorDriver driver;

  public Robot(SensorDriver driver) {
        this.driver = driver;
  }

  public void setName(String name) {
        this.name = name;
  }

  public void startup() {
        System.out.println("Driver Type : " + driver.getType());
        System.out.println("Name : " + name);
  }

  public void release() {
        System.out.println("releasing resources....");
  }

}
```

SensorDriver.java

```
package com.blc.beans;

public class SensorDriver {
  private String type;

  public String getType() {
        return type;
  }

  public void setType(String type) {
        this.type = type;
  }

}
```

application-context.xml

```
<bean id="robot" class="com.blc.beans.Robot" init-method="startup" destroy-
method="release">
        <constructor-arg ref="driver"/>
        <property name="name" value="Robot 1"/>
</bean>

<bean id="driver" class="com.blc.beans.SensorDriver">
        <property name="type" value="IR"/>
</bean>
```

BLCTest.java

```
package com.blc.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.blc.beans.Robot;
import com.blc.beans.ShutdownHookThread;

public class BLCTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/blc/common/application-context.xml"));
                Robot robot = factory.getBean("robot", Robot.class);
                ((ConfigurableListableBeanFactory)factory).destroySingletons();
        }
}
```

3.4.2 Programmatic approach

The problem with declarative approach is developer has provide the init and destroy method declarations for each bean of that class in configuration, if he misses for atleast one bean, then the initialization or destruction will not happen. It would be tough to maintain the configuration, lets say if the init or destroy method name changed in class, all the references to those methods in the configuration has to be modified to match with class method names.

In order to avoid the problems with this spring has provided programmatic approach. In this the bean class has to implement from two interfaces InitializingBean interface to handle initialization process and DisposableBean interface to handle destruction process and should override afterPropertiesSet and destroy methods respectively.

Now the developer don't need to declare these methods as init-method or destroy-method rather the core container while creating these beans will detects

these beans as InitializingBean type or DisposableBean type and calls the afterPropertiesSet and destroy method automatically.

The Robot.java has been modified to depict the same.

Modified# 1 - Robot.java

```
package com.blc.beans;

import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Robot implements InitializingBean, DisposableBean {
  private String name;
  private SensorDriver driver;

  public Robot(SensorDriver driver) {
        this.driver = driver;
  }

  public void setName(String name) {
        this.name = name;
  }

  @Override
  public void destroy() throws Exception {
        System.out.println("From destroy()");
        System.out.println("destroying dependents....");
  }

  @Override
  public void afterPropertiesSet() throws Exception {
        System.out.println("From afterPropertiesSet()");
        System.out.println("Driver Type : " + driver.getType());
        System.out.println("Name : " + name);
  }
}
```

We will discuss about the annotation approach while we are dealing with annotations.

3.5 Aware Interfaces

Many times a bean has to refer to the environment in which it is living. If you recall the discussion of Contextual Dependency lookup, you will understand.

Servlet wants to access the environment in which it is running. To access the environment it needs ServletContext, but to get the ServletContext it needs ServletConfig.

A Servlet cannot create a ServletConfig or it cannot get it by looking up in a registry because it is not an exposed object, rather it is internal object of the ServletContainer. To get the internal objects of the container, we need to implement or extend from container specific interface or class resply and should override the method. In this case we need to implement Servlet interface or extend from HttpServlet class and should override the init() method. ServletContainer calls this method at the appropriate phase during the creation of Servlet object and passes the ServletConfig object as parameter.

So, if we want to get the container internal objects we need to implement the container provider interface and should override then the container is going to provide those objects.

In case of our spring bean also if it wants to access the environment in which it is living, we need to implements the Spring IOC container provided interface and should override a method, so that IOC container will automatically calls this method which creating the bean and injects the dependent object.

Let's consider one example to understand it.

I have a class "Car" it has a method run() and to "run" the car we need to call the start() method on the Engine. How Car will talk to the Engine, either we can use Composition or Inheritance. As it is recommended to use Composition we declare it as attribute. But we should not create the Object of Engine so, as we are using spring we can inject Engine into the Car.

But I don't want to create and don't want to inject, how can I use engine inside the car. I should pull the Engine.

To pull the Engine in the Car we need the reference of the BeanFactory. How to get the reference of the BeanFactory inside the bean. BeanFactory is not an external object rather bean factory is an implicit object of IOC container. To get the container internal objects we need to implement container provided interfaces and should override the method, which will be called by container.

So, here our Car should implements an interface provided by spring, BeanFactoryAware and should override the method, so IOC container will call that method and injects the BeanFactory as shown below.

```
package com.ai.beans;

public interface IEngine {
	void start();
}
```

```
package com.ai.beans;

public class EngineImpl implements IEngine {

	@Override
	public void start() {
		System.out.println("starting...");
	}

}
```

```
package com.ai.beans;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class Car implements BeanFactoryAware {
	private IEngine engine;
	private String beanId;
	private BeanFactory factory;


	public Car(String beanId) {
		System.out.println("Car()");
		this.beanId = beanId;
	}

	public void run() {
		engine = factory.getBean(beanId, IEngine.class);
		engine.start();
		System.out.println("running...");
	}

	@Override
	public void setBeanFactory(BeanFactory factory) throws BeansException {
		System.out.println("setBeanFactory()");
		this.factory = factory;
	}

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="car" class="com.ai.beans.Car">
		<constructor-arg>
			<idref bean="engine" />
		</constructor-arg>
	</bean>
	<bean id="engine" class="com.ai.beans.EngineImpl" />
</beans>
```

```
package com.ai.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.ai.beans.Car;

public class AITest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/ai/common/application-context.xml"));
		Car car = factory.getBean("car", Car.class);
		car.run();
	}
}
```

- When will the IOC container performs the Aware injection?
  - When we request for the bean from the IOC container, first IOC container identifies the bean definition and then starts instantiating. After performing a series of checks. Then while creating the object for the bean it performs constructor injection, after that setter injection and then detects is the class is implementing any Aware interfaces, If yes it calls the corresponding method and performs the injection.

3.6 Static Factory Method

When you configure a class as spring bean, IOC container will instantiate the bean by calling the new operator on it. But in java not all the classes can be instantiated out of new operator. For example Singleton classes cannot be instantiated using new operator, rather you need to call the static factory method that has been exposed by the class to create object of that class.

So, for the classes which cannot be created out of new operator, those which has to be created by calling the static method that has been exposed in the class itself, you need to use static factory injection technic.

In this, after configuring the class as a bean, at the <bean> tag level you need to declare an attribute factory-method = "factorymethod" which is responsible for creating the object of that class. For example an alarm class needs an Calendar as dependent object, but java.util.Calendar class object cannot instantiated using new operation we need to call the getInstance() static method that is exposed in that class to get object of that class. Following example depicts the same.

Alarm.java

```
package com.sf.beans;

import java.util.Calendar;

public class Alarm {
	private Calendar time;

	public void setTime(Calendar time) {
		this.time = time;
	}
	public void ring() {
		System.out.println("Rining at : " + time.getTime());
	}
}
```

application-context.xml

```
<bean id="cal" class="java.util.Calendar" factory-method="getInstance" />

<bean id="alarm" class="com.sf.beans.Alarm">
	<property name="time" ref="cal" />
</bean>
```

```
package com.sf.beans;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.ai.beans.Car;

public class AITest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/sf/common/application-context.xml"));
		Alarm alarm = factory.getBean("alarm", Alarm.class);
		alarm.ring();
	}
}
```

3.7 Instance Factory Method

As said above not all class objects cannot be created out of new operator, few can be constructed by calling factory method on the class, but few objects can be constructed by calling methods on other classes.

If you consider EJB's are any other distributed objects, those cannot be constructor out of new operator or a static method. In order to create an EJB object you need to look up the reference of home object in the JNDI and then you need to call the create() method on it to get the EJB Remote object. This indicates that creation of object involves some sequence of steps or operations to be performed, so these steps are coded in a separate class methods and generally those classes are called ServiceLocator's, these are the classes who knows how to instantiate the objects of other classes.

So, in order to create such type of objects you need to instantiate the ServiceLocator or helper class on which in-turn you need to call the factory method to create our class object. Instance Factory method injection allows you to instantiate classes object by calling method on other (ServiceLocator) class. For this you need to declare the (Service Locator) class on which you call the factory method as spring bean, along with this configure the (Target) bean which has to be created by the Service Locator and declare on the <bean> tag factory-bean is your service locator bean and factory-method is your method in service locator class which will creates your target bean, which is shown below.

GoogleMapRenderer.java

```
package com.inf.beans;

public class GoogleMapRenderer {
	private MapEngine mapEngine;

	public void render(String source, String destination) {
		String directions[] = mapEngine.getDirections(source, destination);
		System.out.println("Directions :");
		for (String d : directions) {
			System.out.println(d);
		}
	}

	public void setMapEngine(MapEngine mapEngine) {
		this.mapEngine = mapEngine;
	}
}
```

MapEngine.java

```
package com.inf.beans;

public interface MapEngine {
	String[] getDirections(String source, String destination);
}
```

IndiaMapEngine.java

```
package com.inf.beans;

public class IndiaMapEngine implements MapEngine {

	@Override
	public String[] getDirections(String source, String destination) {
		return new String[] { "a", "b", "c" };
	}
}
```

USMapEngine.java

```
package com.inf.beans;

public class USMapEngine implements MapEngine {

	@Override
	public String[] getDirections(String source, String destination) {
		return new String[] { "x", "y", "z" };
	}
}
```

MapEngineLocator.java

```
package com.inf.beans;

public class MapEngineLocator {

	public MapEngine getIndiaMapEngine() {
		return new IndiaMapEngine();
	}

	public MapEngine getUSMapEngine() {
		return new USMapEngine();
	}
}
```

application-context.xml

```
<bean id="mapEngineLocator" class="com.inf.beans.MapEngineLocator" />

<bean id="indiaMapEngine" factory-bean="mapEngineLocator"
	factory-method="getMapEngine" />
<bean id="usMapEngine" factory-bean="mapEngineLocator"
	factory-method="getMapEngine"/>

<bean id="googleMapRenderer" class="com.inf.beans.GoogleMapRenderer">
	<property name="mapEngine" ref="indiaMapEngine" />
</bean>
```

```
package com.inf.beans;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class AITest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/inf/common/application-context.xml"));
		GoogleMapRenderer gmr = factory.getBean("googleMapRenderer",
GoogleMapRenderer.class);
		gmr.render("ameerpet", "kukatpally");
	}
}
```

Problem with Static Factory method and Instance Factory method instantiation

The problem with static factory method and instance factory method injection is the factory method you configure should take zero arguments. In case if the factory method of your class takes parameters, it cannot be declared as factory-method in the configuration file.

The above statement indicates if your factory methods are not taking parameters means you cannot instantiate the object of your class with user supplied inputs, always the objects will be created with default values, it can be consider as a biggest limitation.

3.8 Factory Bean

Factory classes are meant for creating the Object of other classes, here the name refers "Factory Bean" itself says these are the classes meant for creating the Bean's.

If we cannot create a class using "new" operator, we will write the code for creating the object of such classes within factory beans, this class will makes those objects as beans in the IOC container.

In order to work with Factory Beans, you need to create a class it should implement from FactoryBean interface and need to override three methods getObject, getObjectType and isSingleton. In the getObject method you need to create the object of the class you want to make as bean and return the Object.

In the getObjectType you need to return the class type of the Object and isSingleton indicates the object you want to create through the Factory is singleton or prototype.

As per the above we need "to write the logic for creating the Calendar class object in getObject(), need to return the type Calendar.class in getObjectType() and if you want the Calendar to place in the IOC container as a singleton bean then you should return "true" in isSingleton method as shown below.

```
package com.fb.beans;
import java.util.Calendar;

public class Alarm {
	private Calendar time;
	public void sayTime() {
		System.out.println(time.getTime());
	}
	public void setTime(Calendar time) {
		this.time = time;
	}
}
```

```
package com.fb.beans;

import java.util.Calendar;
import org.springframework.beans.factory.FactoryBean;

public class CalendarFactoryBean implements FactoryBean<Calendar> {
	private int day;
	private int month;
	private int year;

	public CalendarFactoryBean(int day, int month, int year) {
		this.day = day;
		this.month = month;
		this.year = year;
	}

	@Override
	public Calendar getObject() throws Exception {
		Calendar calendar = null;
		System.out.println("getObject()");
		calendar = Calendar.getInstance();
		calendar.set(year, month, day);
		return calendar;
	}

	@Override
	public Class<?> getObjectType() {
		System.out.println("getObjectType()");
		return Calendar.class;
	}

	@Override
	public boolean isSingleton() {
		System.out.println("isSingleton()");
		return true;
	}
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="alarm" class="com.fb.beans.Alarm">
		<property name="time" ref="calendar" />
	</bean>
	<bean id="calendar" class="com.fb.beans.CalendarFactoryBean">
		<constructor-arg value="10"/>
		<constructor-arg value="1"/>
		<constructor-arg value="2014"/>
	</bean>
</beans>
```

```
package com.fb.test;

import java.util.Calendar;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

public class FBTest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/fb/common/application-context.xml"));
		Alarm alarm = factory.getBean("alarm", Alarm.class);
		alarm.sayTime();
	}
}
```

In the above configuration if you observe into the "time" property we are injection "calendar" reference. Here "time" is Calendar type attribute and "calendar" is an "CalendarFactoryBean" type object. Is it possible to injection "CalendarFactoryBean" object into "time" attribute?

Yes when IOC container instantiating the "calendar" bean it takes the class of it "CalendarFactoryBean" and starts instantiating while creating the object of CalendarFactoryBean it will notice this is implementing from FactoryBean interface, by which it will understand this class is meant for making other class objects as beans, so it understands that, programmer want me to create the object returned by the FactoryBean as bean. So, it calls the getObject() method on the CalendarFactoryBean class and calls the isSingleton method. Based on the isSingleton the object will be placed as bean in the IOC container with the name "calendar". Here bean calendar refers to "Calendar" class object that is created by CalendarFactoryBean, hence it should be able to perform injection.

3.9 Method Replacement

There are many cases in an application we want to replace the logic inside a method with new logic. To replace the logic we can use multiple ways

1) Comment the logic inside the existing method and write the new logic as part of it

2) Write one more class extending the existing class and override the method and write the new logic

If you observe in the above two mechanisms we need to modify the existing source of the application. This pulls lot of questions

1) What if the new logic we are implementing has not certainty (not guaranteed to work)?

2) Sometimes we want to implement the new logic in experimental way.

If the new logic we are writing has uncertainty or experimental after writing the logic by modifying the existing source, it needs to be tested and should be deployed. But if after moving into production if the logic is not working then to revert the logic back to original again it needs to modify as we modified it has to be tested and should be redeployed. This incurs the cost equal to writing a new logic, just to revert back to the earlier logic which is a severe loss to the client.

So, if we manage to replace the logic inside the application without modifying the existing code inside the application, then if new logic is failing reverting to the old will not incurs cost as we haven't touched the existing code.

This can be done through the Method replacement. For replacing a method on a bean the method should not be final and should not be static. In order to replace the method logic, you need to write new class which implement from MethodReplacer interface and should override reimplement method.

The reimplement method takes Object target, java.lang.Method method and Object[] arguments as parameters. These parameters are the values with which you called the original method.

We can replace the logic of a method with new, only when we have information about the original method and its inputs with which we called it. So if we look at the signature of the reimplement, it has the parameters as Object pointing to the actual object with which we called, Method points to the method we are replacing and Object[] args contains the inputs we passed to actual method.

Using the inputs now we can implement the logic to replace the actual as shown below.

```
package com.mr.beans;

public class PlanFinder {
	public String findPlans(int age, String zipCode, int coverageType,
			int networkType) {
		return "Jeevan Anand";
	}

	public String findPlans(int age, String zipCode) {
		return "Jeevan Abhaya";
	}
}
```

```
package com.mr.replacer;

import java.lang.reflect.Method;

import org.springframework.beans.factory.support.MethodReplacer;

public class FindPlansReplacer implements MethodReplacer {

	@Override
	public Object reimplement(Object target, Method method, Object[] args)
			throws Throwable {
		if (method.getName().equals("findPlans")) {
			System.out.println("age : " + args[0]);
			System.out.println("zipCode : " + args[1]);
			System.out.println("coverageType : " + args[2]);
			System.out.println("networkType : " + args[3]);

			// complex logic to replace actual method here
			return "Jeeval Saral";
		}
		return null;
	}

}
```

By writing the FindPlansReplacer class by implementing MethodReplacer the logic will not gets replaced rather we need to instruct the IOC container while creating the object of PlanFinder bean to replace the logic with reimplement as below.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="planFinder" class="com.mr.beans.PlanFinder">
		<replaced-method name="findPlans" replacer="findPlanReplacer"/>
	</bean>
	<bean id="findPlanReplacer" class="com.mr.replacer.FindPlansReplacer" />
</beans>
```

```
package com.mr.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.mr.beans.PlanFinder;

public class MRTest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/mr/common/application-context.xml"));
		PlanFinder pf = factory.getBean("planFinder", PlanFinder.class);
		String plan = pf.findPlans(10, "11111", 1, 0);
		System.out.println("Plan': " + plan);
	}
}
```

Spring will replaces the findPlan method logic with reimplement method of findPlanReplacer at runtime by generating a new class extending from PlanFinder class and override the method findPlan with the logic that is there in reimplement method.

In order to replace the logic and to generate new class spring uses CGLIB runtime byte code generation libraries.

We need to remember some best practices while working with MethodReplacement

- It is good to have if condition to check whether the method we want to replace is right one or not. This makes code more clear and avoids configuration issues
- For every method we want to replace we need to use a separate replacer class, we should not replace multiple methods with the same replacer even it is technically possible also

If our class contains overloaded methods then in the <replaced-method> tag when we specify the name as "findplans" it will not differentiate and replace the method rather it ignores replacing.
To avoid this we can pass an extra <arg> tag inside the <replaced-method> tag to tell the exact signature of the method want to replace as shown below.

In case of Overloaded methods we can use the below configuration.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

       <bean id="planFinder" class="com.mr.beans.PlanFinder">
              <replaced-method name="findPlans" replacer="findPlanReplacer">
                     <arg-type>int</arg-type>
                     <arg-type>java.lang.String</arg-type>
                     <arg-type>int</arg-type>
                     <arg-type>int</arg-type>
              </replaced-method>
       </bean>
       <bean id="findPlanReplacer" class="com.mr.replacer.FindPlansReplacer" />
</beans>
```

3.10 Lookup Method Injection

In spring when a non-singleton bean is getting injected into a singleton bean the result will be always singleton. For example if LoanApprover is a singleton class and LoanInfo is a non-singleton class. If we inject LoanInfo into LoanApprover class as shown below.

```
<bean id="loanApprover" class="com.lmi.beans.LoanApprover">
       <property name="loanInfo" ref="loanInfo"/>
</bean>
<bean id="loanInfo" class="com.lmi.beans.LoanInfo" scope="prototype"/>
```

As the loanApprover is a singleton class, only one object of LoanApprover will be created by core container and hence LoanInfo bean will be injected once via setter injection into LoanApprover class. This leads to LoanInfo as singleton.

Ideally speaking the above design is wrong, because if we want to declare a class as singleton, it should not contain any state. As the LoanApprover is using the LoanInfo, instead of injecting LoadInfo via setter or constructor injection, LoanApprover has to fetch the LoanInfo from the container and should use and dispose it.

The below table depicts the matrix of combinations of single-ton and non-singleton.

Target Bean	Dependent Bean	Result
Singleton	Singleton	✓
Singleton	Non-Singleton	x (Should not use setter or constructor, use lookup method injection)
Non-Singleton	Singleton	✓
Non-Singleton	Non-Singleton	✓

Let's try to understand this better by taking one more example. The Web container upon receiving a request from the client, it will try to process the request by creating a new Object of RequestHandler class and populates the request data to RequestHandler.

If you observer the above scenario, here the Web container is a singleton class and RequestHandler is a non-singleton class, for every incoming request to the container it has to create a new RequestHandler class object and populate the request info and then should handover the control to process the request. So, the Web container will fetch the RequestHandler (so that each lookup of request handler will returns new object) class object upon receiving the request and populates data to process.

The Web Container class can fetch the Object of RequestHandler by implement BeanFactoryAware and can call getBean method to get the RequestHandler object shown below.

```
package com.lmi.beans;

public class WebContainer implements BeanFactoryAware {
	private BeanFactory factory;

	public void process(String data) {
		RequestHandler rh = factory.getBean("requestHandler",
RequestHandler.class);
		rh.setData(data);
		rh.handle();
	}

	public void setBeanFactory(BeanFactory factory) {
		this.factory = factory;
	}
}
```

In the above code we have used factory.getBean("requestHandler"), this indicates that we have hardcoded the logical name of the bean in our code, so that our code is tightly coupled with that specific bean. Second issue with that code is we have implemented our code from spring specific interface which indicates we lose the benefit of non-invasive feature of spring.

To avoid the above problems we need to use Lookup Method Injection. In this your class doesn't need to implement from any spring specific class or interface. You don't need to code for getting the reference of dependent object. Instead you declare to spring asking it to write the code of getting the dependent object through configuration, so that your code is loosely coupled with a spring bean. This is shown in the below example.

WebContainer.java

```
package com.lmi.beans;

abstract public class WebContainer {
	public void process(String data) {
		RequestHandler rh = getRequestHandler();
		rh.setData(data);
		rh.handle();
	}
	abstract public RequestHandler getRequestHandler();
}
```

RequestHandler.java

```
package com.lmi.beans;

public class RequestHandler {
	private String data;

	public void setData(String data) {
		this.data = data;
	}

	public void handle() {
		System.out.println("Processing request with data : " + data);
	}
}
```

In the above code we declared a method getRequestHandler() as a abstract method which returns RequestHandler object. We don't know how to write the logic in this method to get the RequestHandler object, so we declared this as abstract asking spring to implement this method for us. We will tell spring to implement the getRequestHandler method in configuration as shown below.

application-context.xml

```
<bean id="requestHandler" class="com.lmi.beans.RequestHandler"
scope="prototype"/>
<bean id="webContainer" class="com.lmi.beans.WebContainer">
	<lookup-method name="getRequestHandler" bean="requestHandler"/>
</bean>
```

In the above declaration we declared lookup method tag in WebContainer bean stating spring to implement the method getRequestHandler to return the requestHandler bean.

So your code has to call the getRequestHandler method to get the object of it. Spring will generates a class at runtime by using CGLib libraries and overrides this method to return the RequestHandler object.

There are few things we need to remember as part of this

- We should not go for injection when our Target bean is singleton and our dependent beans is prototype
- Even a class is abstract also we can declare it as a bean in spring bean configuration file, if the abstract method is declared as lookup method.

3.11 Property Editors

In spring when you configure a class as spring bean, you can inject all the dependent attributes of the bean via setter or constructor injection. The attributes that you want to inject via spring injection could be of any type like int, float, long, any arbitrary object, String[], File etc.

Spring when you configure an attribute value using <property> tag, the value will be converted into class attribute type by using the property editor and then injects into the bean.

So there are lot of in-built property editors that are available in spring which converts the string values you configured in the configuration file to the target bean attribute types. For example when you configure a file path as string in <property> or <constructor-arg> tag it will be injected as a File object into the target class attribute using the file path. So these types of conversions will be done using PropertyEditor as shown below.

```
package com.pe.beans;

import java.io.File;
import java.util.Arrays;
import java.util.Date;

public class Payslip {
	private int empId;
	private String name;
	private Date paidDt;
	private File paySlipFile;
	private String[] verifiers;

	// setters and getters

	@Override
	public String toString() {
		return "Payslip [empId=" + empId + ", name=" + name + ", paidDt="
				+ paidDt + ", paySlipFile=" + paySlipFile + ", verifiers="
				+ Arrays.toString(verifiers) + "]";
	}

}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<bean id="payslip" class="com.pe.beans.Payslip">
		<property name="empId" value="10" />
		<property name="name" value="john" />
		<property name="paidDt" value="02/10/2014" />
		<property name="paySlipFile" value="C:\axis-1_4\README"/>
		<property name="verifiers" value="rama,laxmana"/>
	</bean>
</beans>
```

```
package com.pe.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.pe.beans.Payslip;

public class PETest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/pe/common/application-context.xml"));
		Payslip ps = factory.getBean("payslip", Payslip.class);
		System.out.println(ps);
	}
}
```

As we understood the purpose of PropertyEditor, let's try to understand how to create our own custom property editor by taking an example.

ComplexNumber.java

```
package com.pe.beans;
public class ComplexNumber {
	private int base;
	private int expo;
	public ComplexNumber(int base, int expo) {
		this.base = base;
		this.expo = expo;
	}
	public int getBase() {
		return base;
	}
	public void setBase(int base) {
		this.base = base;
	}
	public int getExpo() {
		return expo;
	}
	public void setExpo(int expo) {
		this.expo = expo;
	}
}
```

MathCalculator.java

```
package com.pe.beans;

public class MathCalculator {
	private ComplexNumber complexNumber;

	public void setComplexNumber(ComplexNumber complexNumber) {
		this.complexNumber = complexNumber;
	}

	public void calculate() {
		System.out.println("Calculating with complex number base : "
				+ complexNumber.getBase() + " expo : "
				+ complexNumber.getExpo());
	}
}
```

In order to inject ComplexNumber into MathCalculator, the configuration looks as follows.

application-context.xml

```
<bean id="complexNumber" class="com.pe.beans.ComplexNumber">
	<property name="base" value="24"/>
	<property name="expo" value="34"/>
</bean>
<bean id="mathCalculator" class="com.pe.beans.MathCalculator">
	<property name="complexNumber" ref="complexNumber"/>
</bean>
```

The above configuration works without any issue, but instead of configuring the complexNumber as a bean and injecting via property reference, we want to configure the complexNumber as a String literal value as 24, 34 where 24 is the base and 34 is the expo as shown below.

```
<bean id="mathCalculator" class="com.pe.beans.MathCalculator">
	<property name="complexNumber" value="24,34"/>
</bean>
```

In order to achieve the above, we need to write a custom property editor which will reads the string value "23, 34" in the configuration and coverts it to the target class attribute type, in this case ComplexNumber type and injects it.

So, to develop a custom property editor you need to write a class which extends from PropertyEditorSupport and should register this with the BeanFactory as shown below.

ComplexNumberEditor.java

```
package com.pe.beans;

import java.beans.PropertyEditorSupport;

public class ComplexNumberEditor extends PropertyEditorSupport {

	@Override
	public void setAsText(String value) throws IllegalArgumentException {
		int base = 0;
		int expo = 0;

		base = Integer.parseInt(value.substring(0, value.indexOf(",")));
		expo = Integer.parseInt(value.substring(value.indexOf(",") + 1,
				value.length()));
		ComplexNumber complexNumber = new ComplexNumber(base, expo);
		setValue(complexNumber);
	}
}
```

After developing a custom property editor class, you need to register this with the BeanFactory after creating the factory shown below.

PETest.java

```
package com.pe.test;

import org.springframework.beans.PropertyEditorRegistrar;
import org.springframework.beans.PropertyEditorRegistry;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.pe.beans.ComplexNumber;
import com.pe.beans.ComplexNumberEditor;
import com.pe.beans.MathCalculator;

public class PETest {
	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/pe/common/application-context.xml"));

		((ConfigurableListableBeanFactory) factory)
				.addPropertyEditorRegistrar(new
PropertyEditorRegistrar() {

					@Override
					public void registerCustomEditors(
							PropertyEditorRegistry registry) {

	registry.registerCustomEditor(ComplexNumber.class,
							new ComplexNumberEditor());
					}
				});
		MathCalculator mc = factory.getBean("mathCalculator",
MathCalculator.class);
		mc.calculate();
	}

}
```

So, after creating the Custom Property editor we need to register that with the PropertyEditorRegistry. PropertyEditorRegistry is an internal object in the IOC container so we wrote a class implementing PropertyEditorRegistrar and overridden the method. We registered the PropertyEditorRegistrar as a registrar to the container.
So, container will automatically class the registerCustomeEditors by passing the registry using which we can register our custom property editors.

3.12 Internationalization (i18n)

Internationalization is the process of designing the application to adapt and display the content specific to the locale from which the user is accessing from.

JEE has support for designing internationalized applications. It has provided a class "ResourceBundle", it takes the base bundle name and locale and retrieves the values from the locale specific property files.

Here is the sample snipped of code showing the same.

messages.properties

```
HOME_WELCOME_MSG=Welcome to my application
```

messages_cn_CH.properties

```
HOME_WELCOME_MSG=欢迎来到我的应用
```

We created two properties file with key as HOME_WELCOME_MSG one containing value as English and other one in Chinese language. Now based on the locale from which the user is accessing we will pull the value from appropriate locale specific bundle and display as part of application as shown below.

```
ResourceBundle rb = ResourceBundle.getBundle("messages", Locale.geDefault());
String message = rb.getMessage("HOME_WELCOME_MSG");
System.out.println(message);
```

There are lots of problems with JEE approach in working with Internationalization as described below.

- To display content in a Jsp page we should not hardcode rather should read the text from Message Bundles (properties) files and should display the content. This means we have to create the "ResourceBundle" object across the servlet's and should make it available to the Jsp by binding to request scope. But with this we end up in writing the code across all the classes of our application, instead we can write a filter which creates the "ResourceBundle" and make it available for all the requests. This indicates programmer has to decide the strategy of using the ResourceBundle object in the application.
- An application cannot be completed with one properties file, we may need to externalize the values across multiple properties files. If we have multiple properties file we need to create one ResourceBundle object per Properties file to read the values. This leads to referring multiple ResourceBundle objects across the application to retrieve values from different properties files.
- Sometimes in an application we want to read the value for a key in multiple Locale bundles. To read the same message in different languages, we need to create one ResourceBundle object per locale. In this case also we have to refer more ResourceBundle objects in application.

Spring has a better support for internationalization when compared with JEE internationalization. It has provided multiple classes like StaticMessageBundleSource, ResourceBundleMessageSource and ReloadableMessageSource. These take multiple properties files as inputs and will retrieve the messages from various properties.

As we are going to configure this as a bean, by default it is singleton, so for entire application we will have only one object of this. And these classes has a method getMessage and it takes the "key" and Locale as a parameter. This indicates the message can be read from different locales without creating multiple objects.

Now instead of directly reading the messages from these classes we can read the messages from ApplicationContext as well. This is one of the differences between ApplicationContext and BeanFactory. BeanFactory doesn't have internationalization support only ApplicationContext has.

Spring has provided methods to access the messages from the resource bundle in ApplicationContext. Now we can call convienent method getMessage() on application context in turn reads the messages using the help of the above mentioned classes. In the ApplicationContext, it declared an attribute messageSource of type ResourceBundleMessageSource as shown below.

So in order to load properties file you need to declare a bean whose type is ResourceBundleMessageSource, it will takes an attribute baseName, for which you need to pass the name of the properties file. This bean has to be named with messageSource so that when we call getMessage() method on ApplicationContext, it delegates the call to the bean whose name is "messageSource" to get the messages.

Note: - You have to place the properties file under classes' folder (place it in src, will automatically shipped as part of classes).

application-context.xml

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename" value="label"/>
</bean>
```

I18NTest.java

```
package com.i18n.test;
import java.util.Locale;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class I18NTest {
        public static void main(String[] args) {
                ApplicationContext context = new ClassPathXmlApplicationContext(
                                "com/i18n/common/application-context.xml");
                System.out.println("Message : "
                                + context.getMessage("empid.label", new Object[] {"is not",
"valid"}, Locale.getDefault()));
        }
}
```

3.13 Bean Post Processor

In spring, we use Bean Lifecycle to perform initialization on a bean. But using init-method, destroy-method or InitializingBean, DisposableBean we do initialization or destruction process for a specific bean which implement these.

If we have a common initialization process which has to be applied across all the beans in the configuration, Bean lifecycle cannot handle this. We need to use BeanPostProcessor.

BeanPostProcessor has two methods postProcessBeforeInitialization and postProcessAfterInitialization, these methods will be invoked for all the beans on the core container. postProcessBeforeInitialization method will be called after the core container has created the bean and before it performs injection. postProcessAfterInitialization method will be called after the core container has perfomed injections. For both of these methods the bean will be passed as parameter along with beanName on which the methods are fired.

In these methods the developer can write the Initialization logic to initialize the beans. So, you need to write a class which implements from BeanPostProcessor interface and override the methods and provide the logic for initialization. The same has been depicted in the below example.

EmployeeDelegate.java

```
package com.bpp.beans;

abstract public class EmployeeDelegate {
	private EmployeeVO employeeVO;
	private EmployeeDao employeeDao;

	public void insert() {
		employeeVO = lookupEmployeeVO();
		employeeVO.setEmpID("E32542");
		employeeVO.setName("John");
		employeeVO.setSalary(353.34f);
		employeeDao.insert(employeeVO);
	}

	public abstract EmployeeVO lookupEmployeeVO();

	public void setEmployeeDao(EmployeeDao employeeDao) {
		this.employeeDao = employeeDao;
	}
}
```

EmployeeDao.java

```
package com.bpp.beans;

public class EmployeeDao {
	public void insert(EmployeeVO employeeVO) {
		System.out.println("inserting employee : "
				+ employeeVO.getLastModifiedDate());
	}
}
```

EmployeeVO.java

```
package com.bpp.beans;

public class EmployeeVO extends BaseVO {
	private String empID;
	private String name;
	private float salary;

	// setters and getters
}
```

BaseVO.java

```
package com.bpp.beans;

import java.util.Date;

abstract public class BaseVO {
	private Date lastModifiedDate;

	public Date getLastModifiedDate() {
		return lastModifiedDate;
	}

	public void setLastModifiedDate(Date lastModifiedDate) {
		this.lastModifiedDate = lastModifiedDate;
	}

}
```

BaseVOBeanPostProcessor.java

```
package com.bpp.beans;

import java.util.Date;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class BaseVOPostProcessor implements BeanPostProcessor {

	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName)
			throws BeansException {
		if (bean instanceof BaseVO) {
			((BaseVO) bean).setLastModifiedDate(new Date());
		}
		return bean;
	}

	@Override
	public Object postProcessBeforeInitialization(Object arg0, String arg1)
			throws BeansException {
		return arg0;
	}

}
```

application-context.xml

```
<bean id="employeeVO" class="com.bpp.beans.EmployeeVO" scope="prototype"/>
<bean id="employeeDao" class="com.bpp.beans.EmployeeDao" />
<bean id="employeeDelegate" class="com.bpp.beans.EmployeeDelegate">
	<look-up-method name="lookupEmployeeVO" bean="employeeVO"/>
	<property name="employeeDao" ref="employeeDao" />
</bean>
<bean id="bpp" class="com.bpp.beans.BaseVOPostProcessor"/>
```

Once you create the BeanPostProcessor class, you need to register the postprocessor with the bean factory as shown below.

BPPTest.java

```
package com.bpp.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.bpp.beans.BaseVOPostProcessor;
import com.bpp.beans.EmployeeDelegate;

public class BPPTest {

	public static void main(String[] args) {
		BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
				"com/bpp/common/application-context.xml"));
		BaseVOPostProcessor bpp = factory.getBean("bpp",
				BaseVOPostProcessor.class);

		((ConfigurableListableBeanFactory)
factory).addBeanPostProcessor(bpp);
		EmployeeDelegate ed = factory.getBean("employeeDelegate",
EmployeeDelegate.class);
		ed.insert();
	}

}
```

So, in the above code after registering the bean postprocessor, when you try to fetch the EmployeeDelegate bean from the core container, while creating the delegate it will try to create EmployeeVO and EmployeeDao to inject into it.

So after the container creates these before performing injection it will call the postProcessBeforeInitialization and after performing the injection, it will call postProcessAfterInitialization.

3.14 Bean Factory Post Processor

If we want to perform some post processing on the BeanFactory after it has been created and before it instantiates the beans then we need to go for BeanFactoryPostProcessor.

If we want to change the configuration, we can always change it by modifying the physical configuration file. Instead if want to modify the configuration dynamically at runtime within the IOC container metadata then we can use BeanFactoryPostProcessor.

There are many benefits of using BeanFactoryPostProcessors. We can modify the configuration during the deployment time by using some build tools like ant. But if we want to modify the configuration again it demands the rebuild and redeployment. Instead if we go for BeanFactoryPostProcessor as the configuration is modified in the IOC container metada at runtime.

If we want to perform post initialization after creating the BeanFactory and before creating the beans, we need to use BeanFactoryPostProcessor. It's a one of the kind of extension hooks that spring has provided to the developer to perform customizations on the configuration that is loaded by the Factory.

Spring has provided built-in BeanFactoryPostProcessor's to perform post processing, based on the type of post processor you use, you will get the respective behavior in your application. One of it is PropertyPlaceHolderConfigurer.

PropertyPlaceHolderConfigurer is a post processor which will reads the messages from the properties file and replaces the ${} place holder's of a bean configuration after the BeanFactory has loaded it.

ConnectionManager.java

```
package com.bfpp.beans;

public class ConnectionManager {
        private String url;
        private String userName;
        private String password;

        // setters & getters
```

application-context.xml

```
<bean id="connectionManager" class="com.bfpp.beans.ConnectionManager">
        <property name="url" value="${db.url}"/>
        <property name="userName" value="${db.un}"/>
        <property name="password" value="${db.pwd}"/>
</bean>
```

If you create a BeanFactory with the above configuration, the ConnectionManager bean will gets created with url, username and password with ${} token values. Instead of this, if we want to replace the ${} values with property file key values, we need to configure PropertyPlaceHolderConfigurer. This post processor after loading the configuration by BeanFactory and before the factory creates the ConnectionManager bean, will replaces the ${} values with property key values as shown below.

db.properties

```
db.url=jdbc:odbc:thin@1521:XE
db.un=weblogic
db.pwd=welcome1
```

Adding to the above configuration, you need to configure the post processor and register this with the BeanFactory.

```
<bean id="pphc"
class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location" value="classpath:db.properties"/>
</bean>
```

BFPPTest.java

```
package com.bfpp.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.ConfigurableListableBeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.ClassPathResource;

import com.bfpp.beans.ConnectionManager;

public class BFPPTest {
        public static void main(String[] args) {
                BeanFactory factory = new XmlBeanFactory(new ClassPathResource(
                                "com/bfpp/common/application-context.xml"));
                BeanFactoryPostProcessor bfpp = factory.getBean("pphc",
BeanFactoryPostProcessor.class);
                bfpp.postProcessBeanFactory((ConfigurableListableBeanFactory)factory);

                ConnectionManager cm = factory.getBean("connectionManager",
ConnectionManager.class);
                System.out.println("cm.url : " + cm.getUrl());
        }

}
```

3.15 Event Processing

Spring provides Event handling capabilities using the ApplicationContext. For any event based processing technic we have four actors, 1) source 2) event 3) event listener and 4) event handler.

Source is the actor who raises an Event, Event is the class which contains the data representing the purpose of the event. Listener is the person who will listens for the event, and upon raising the event by source, listener will catches it and raises a method call on the handler to process that event.

So, Event handling is used for asynchronous processing, in spring an Event is represented with ApplicationEvent, Listener is created using ApplicationListener, and the application listener will contains the method onApplicationEvent() which will be fired upon raising the Event. In order to publish an event, the source needs an ApplicationEventPublisher. The below example shows the same.

RefreshEvent.java

```
package com.ep.beans;

import org.springframework.context.ApplicationEvent;

public class RefreshEvent extends ApplicationEvent {
	private String tableName;

	public RefreshEvent(Object source, String tableName) {
		super(source);
		this.tableName = tableName;
	}

	public String getTableName() {
		return tableName;
	}

}
```

RefreshEventListener.java

```
package com.ep.beans;

import org.springframework.context.ApplicationListener;

public class RefreshEventListener implements ApplicationListener<RefreshEvent> {

	@Override
	public void onApplicationEvent(RefreshEvent event) {
		System.out.println("Refreshing table : " + event.getTableName());
	}

}
```

RefreshEventSource.java

```
package com.ep.beans;

import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class RefreshEventSource implements ApplicationEventPublisherAware {
	private ApplicationEventPublisher publisher;

	public void raiseRefresh(String table) {
		publisher.publishEvent(new RefreshEvent(this, table));
	}

	@Override
	public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {
		this.publisher = publisher;
	}

}
```

application-context.xml

```
<bean class="com.ep.beans.RefreshEventListener"/>
<bean id="refreshEventSource" class="com.ep.beans.RefreshEventSource"/>
```

EPTest.java

```
package com.ep.test;

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

import com.ep.beans.RefreshEventSource;

public class EPTest {
	public static void main(String[] args) {
		ApplicationContext context = new ClassPathXmlApplicationContext(
				"com/ep/common/application-context.xml");

		RefreshEventSource res = context.getBean("refreshEventSource",
				RefreshEventSource.class);
		res.raiseRefresh("TBLEMPLOYEE");
	}

}
```

3.16 Bean Factory VS Application Context

Bean Factory	Application Context
• Bean Factory is a lazy initializer, this means after creating the Bean Factory, it will loads the configuration file but will not create any beans for the bean declarations. When you try to fetch the bean by using factory.getBean(""), Bean Factory will creates the bean. • Bean Factory doesn't support Internationalization. • Bean Factory doesn't supports Event Handling and Event processing • Bean Factory will not automatically registers BeanPostProcessors or BeanFactoryPostProcessors, we need to explicitly write the code for registering them.	• ApplicationContext is an eager initializer. When you first create the application context with the configuration, after loading the configuration, it will immediately create all the beans in the configuration. • Application Context supports internationalization. • Application Context supports Event processing. • Application Context up seeing the BeanPostProcessor and BeanFactoryPostProcessor declarations in the configuration, will automatically registers them with the container and applies processing.

DURGA SOFTWARE SOLUTIONS, Plot No: 202, IInd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad-500038., Cell. 9246212143. Phone: 040-64512786

Spring Annotation

4 Spring Annotation Support

4.1 Introduction to J2EE Annotation

Annotation is the code about the code that is metadata about the program itself. In other words information about the code is provided at the source code. Annotations are parsed/processed by compilers, annotation-processing tools and also executed at runtime.

Annotations have been introduced in JDK 1.5. It allows the programmers to specify the information about the code at the source code level itself. There are several ways we can use annotations. Few helps in understanding the source code (like documentation assistance) @override is the annotation that will be used when we override a method from base class. This annotation doesn't have any impact on run-time behavior, rather it helps javadoc complier to generate documentation based on it.

Apart from these annotations we have another way in using annotations which assist source code generators to generate source code using them. If we take example as Web Services, in order to expose a class as web service, we need to mark the class as @WebService. This annotation would be read by a tool and generates class to expose that class as web service.

Along with this there is another way, where when you mark a class with annotation, the run-time engine will executes the class based on that annotation with which it marked. For example, when you mark a class with @EJB, the class would be exposed as Enterprise Java Bean by the container rather than a simple pojo.

By the above we can understand that using annotations, a programmer can specify the various behavioral aspects of your code like documentation, code generation and run-time behavior. This helps in RAPID application development where in instead of specifying the information about your code in xml configuration file, the same can be specified by marking your classes with annotations.

Note: - Always your annotation configuration will be overwritten with the xml configuration if provided.

4.2 Spring Annotation Support

Spring support to annotations is an incremental development effort. Spring 2.0 has a very little support to annotation, when it comes to spring 2.5; it has extended its framework to support various aspects of spring development using annotations. In spring 3.0 it started supporting java config project annotations as well.

The journey to spring annotations has been started in spring 2.0; it has added @Configuration, @Repository and @Required annotations. Along with this it added support to declarative and annotation based aspectj AOP.

In spring 2.5 it has added few more annotations @Autowired, @Qualifier and @Scope. In addition in introduced stereotype annotations @Component, @Controller and @Service.

In spring 3.0 few more annotations has been added like @Lazy, @Bean, @DependsOn etc. In addition to spring based metadata annotation support, it has started adoption JSR - 250 Java Config project annotations like @PostConstruct, @PreDestory, @Resource, @Inject and @Named etc.

The below table list spring supported and Java Config project supported annotations.

		Java Config Annotation Support	
Spring 2.0	• @Configuration • @Required • @Repository		• No Support
Spring 2.5	• @Autowired • @Qualifier • @Scope Stereotyped annotations • @Component • @Service • @Controller		• No Support
Spring 3.0	• @Bean • @DependsOn • @Lazy • @Value		• @PostConstruct • @PreDestroy • @Resource • @Inject • @Named
Spring 3.x (later)	• @ComponentScan • @Profile • @Import • @ContextConfiguration		• Support

Let's explore these annotations with examples.

4.2.1 Working with @Configuration and @Bean

Instead of declaring a class as spring bean in a configuration file, you can declare it in a class as well. The class in which you want to provide the configuration about other beans, that class is called configuration class and you need to annotate with @Configuration. In this class you need to provide methods which are responsible for creating objects of your bean classes, these methods has to be annotated with @Bean.

Now while creating the core container, instead of using XMLBeanFactory, you need to create it using AnnotationConfigApplicationContext by passing the configuration class as input.

AppConfig.java

```
@Configuration
public class AppConfig {
  @Bean
  public MyService myService() {
     return new MyServiceImpl();
  }
}
```

MyService.java

```
public class MyService {
        public void doSomeStuff() {
                // some dummy logic
        }
}
```

ConfigurationTest.java

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);
MyService myService = ctx.getBean(MyService.class);
myService.doSomeStuff();
```

4.2.2 Working with @Required

In spring 1.x onwards we have dependency check. For a bean if you want to detect unresolved dependencies of a bean, you can use dependency check. In this IOC container will check whether all the bean dependencies, which is expressed in its properties are satisfied or not. The problem with dependency check is you can't impose restriction on a certain property, either you need to check the dependencies on all simple or object or all.

In 2.0, it has introduced an annotation @Required and dependency check has been completely removed in 2.5. Using @Required annotation you can make, a particular property has been set with value or not.

Engine.java

```
package com.annotation.beans;

public class Engine {
	private Integer id;
	private String type;

	public Integer getId() {
		return id;
	}

	@Required
	public void setId(Integer id) {
		this.id = id;
	}

	public String getType() {
		return type;
	}

	public void setType(String type) {
		this.type = type;
	}
}
```

You can use @Required on a setter level to mark it as mandatory. Simply using @Required annotation will not enforce the property checking, you also need to register an RequiredAnnotationBeanPostProcessor in the configuration file as shown below.

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
	<property name="id" value="22"/>
	<! - -if you don't provide value for id, raises error - ->
	<property name="type" value="T1"/>
	<qualifier value="myengine"/>
</bean>
(<context:annotation-config/>
```

(OR)

<bean class=
"org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"
/>)

4.2.3 Working with @Autowire

You can enable autowiring using @Autowire annotation rather than configuration approach. Unlike your declarative <bean autowire="byName/byType/constructor"/> etc, in annotation driven we don't have any modes.

In annotation based approach you may mark an attribute of a target class or a setter or a constructor or any arbitrary method for injecting the dependent object. In all the cases it will performs autowiring byType.

@Autowired annotation has an attribute required=true (@Autowired(required=true)). By default required is true. When you use required=true, while creating the target class object it will try to find the appropriate dependent class object in IOC container, if it couldn't find one the container will throws exception without creating core container.

At any point of time out of available number of constructors, you can mark only one constructor with @Autowired(required=true) and remaining should be set to false.

Below examples shows various ways of using @Autowired

Attribute Level

```
package com.annotation.beans;

import org.springframework.beans.factory.annotation.Autowired;

public class Motor {
	@Autowired(required=false)
	private Engine engine;

	public void run() {
		System.out.println("running with engine id : " + engine.getId());
	}
}
```

Setter Level

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

Constructor Level

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public Motor(Engine engine) {
                this.engine = engine;
        }
....
}
```

Arbitrary method

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void newEngine(Engine engine) {
                this.engine = engine;
        }
....
```

```
}
```

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:annotation-config/>
```

In order to detect @Autowire either you need to use <context:annotation-config/> or need to declare a bean whose class is org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.

4.2.4 Working with @Qualifier

If you have more than one beans of type Engine in the configuration, as @Autowire will perform injection byType, it will not be able to make the decision of which bean has to be injected and will raise ambiguity error. To resolve this we need to perform byName, this can be done with @Qualifier as shown below.

Using Qualifier

```
(ignored few sections for clarity)

public class Motor {
        private Engine engine;

        @Autowired(required=false)
        @Qualifier("engine2")
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<bean id="maruthiengine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="engine2"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:annotation-config/>
```

Only the bean whose qualifier value is "engine2" will be injected into Motor class engine attribute, so indirectly using @Qualifier we are able to perform byName injection.

4.2.5 Working with stereotype annotations @Component, @Repository, @Service and @Controller

In addition to @Repository annotation in spring 2.0, spring 2.5 has added stereotype annotations @Component, @Service and @Controller to make your classes as spring beans.

When you mark you class with any of the above annotations those classes will be exposed as spring beans by the container. Based on the type of class you are trying to expose you need to use appropriate annotations.

@Component – This acts as a more generic stereotype annotation to manage any component by spring, whereas the other annotations are specialized for the specific use cases.

@Repository – This is used for exposing a DAO class as a spring bean. Even though database specific semantics are not imposed by using it, it helps you in applying exception translations on these @Repository classes using AOP.

@Service – The service layer class are annotated with @Service, even though the current release of the spring doesn't has any impact on using it (other than exposing that class as spring bean), but future releases of spring might add some specializations on those classes.

@Controller – When you mark a class with @Controller, it will be exposed as spring MVC controller to handle form submissions.

In a spring core/spring jdbc applications you can use @Component, @Service or @Repository whereas @Controller can be used only in a Spring MVC application.

The below examples shows how to use @Component, but you can mark that class with @Service or @Repository as well, where the end effect would be same.

Motor.java

```
(ignored few sections for clarity)

@Component("motor")
public class Motor {
        private Engine engine;

        @Autowired(required=false)
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

With the above configuration, the class will be exposed as spring bean with id "motor". In order to detect the @Component or stereotype annotations you need to use the tag <context:component-scan base-package="PKG NAME OF THE CLASSES"/> as shown below.

application-context.xml

```
<context:component-scan base-package="com.annotation.*"/>
```

You can retrieve the bean with context.getBean("motor").

4.3 Spring Java Config annotations

As explained earlier from spring 3.x it has added support to Java Config annotation support, this means when you mark you classes with any of the above discussed annotations like @Component or @Autowired, you classes will be tightly coupled with spring framework and will lose invasive feature of spring.

In order to retain loosely coupled feature, it spring has added support to java annotations, so when we use these annotations in spring applications, those will be

read by spring container and add spring specific behavior to your classes. If you use in J2EE environment, those will behave based on container specification.

4.3.1 Working with @Inject

@Inject is a j2ee specific annotation, used for injecting/autowiring one class into another. This is more similar to @Autowired spring annotation. But the difference between them is @Autowire supports required attribute where @Inject doesn't has it.

@Inject also injects a bean into another bean byType as similar to @Autowired. The advantage of @Inject is it is from java (javax.inject package) which means even you separate your application from spring, still your classes can work with java rather than bounded to spring.

Motor.java

```
(ignored few sections for clarity)

@Component("motor")
public class Motor {
        private Engine engine;

        @Inject
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

Along with this you need to use <context:component-scan base-package=""/> to detect it by core container.

4.3.2 Working with @Named

There are two usages of @Named annotation

1) When you try to inject a bean using @Inject annotation it will performs injection byType, if you have more than one beans of that type in the application, the container will throw ambiguity error. In order to resolve it you need to use @Named annotation.

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<bean id="maruthiengine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
</bean>

<bean id="motor" class="com.annotation.beans.Motor"/>
<context:component-scan base-package="com.annotation.*"/>
```

In the above configuration you have two beans of type Engine, so when you try to inject the Engine into Motor by using @Inject, it will not be able to detect which bean has to be injected. So you need to mark the Engine attribute with @Named along with @Inject indicating the bean you want to inject show below.

Motor.java

(ignored few sections for clarity)

```
@Component("motor")
public class Motor {
        private Engine engine;

        @Inject
        @Named("maruthiengine")
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

2) Another use of @Named is instead of using the stereotype annotations of spring to expose your classes as spring beans, you can mark your class with @Named to expose it as spring bean show below.

Motor.java

```
(ignored few sections for clarity)

@Named("motor")
public class Motor {
        private Engine engine;

        @Inject
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

4.3.3 Working with @Resource

Instead of using @Inject, you can also use @Resource, the difference between @Inject and @Resource it @Inject will perform the injection byType where as @Resource will perform the injection byName.

Motor.java

```
(ignored few sections for clarity)

@Named("motor")
public class Motor {
        private Engine engine;

        @Resource
        public void setEngine(Engine engine) {
                this.engine = engine;
        }
....
}
```

So, we need to have a bean whose id is "engine" in the configuration or with the component declaration with this name.

application-context.xml

```
<bean id="engine" class="com.annotation.beans.Engine">
        <property name="id" value="22"/>
        <property name="type" value="T1"/>
        <qualifier value="myengine"/>
</bean>
<context:component-scan base-package="com.annotation.*"/>
```

4.3.4 Working with @PostConstruct and @PreDestroy

As we discussed, Bean Lifecycle you can perform Initialization on a bean after injecting the dependent objects using init-method declaration or InitializingBean interface afterPropertiesSet and Destruction on a bean while removing a bean from core container using destroy-method or DisposableBean interface destroy() method.

Along with the above two ways, you can mark any arbitrary method on a class with @PostContruct and @PreDestory annotations. When you mark a method with @PostConstruct annotation, this method will be invoked by core container after injecting the dependent objects into the bean. When you mark a method with @PreDestory annotation, this method will be invoked as part of bean destruction process.

So, the @PostContruct and @PreDestory or annotation based lifecycle methods.

Motor.java

```
@Named("motor")
public class Motor {
        private Engine engine;

        @PostConstruct
        public void init() {
                // I will be invoked after performing injection
        }

        @Resource
        public void setEngine(Engine engine) {
                this.engine = engine;
        }

        @PreDestory
        public void release() {
                // I will be invoked while I am removing from container
        }
....
}
```

5 Aspect Oriented Programming (AOP)

AOP stands for Aspect Oriented Programming; it is a programming model or paradigm that allows you to apply cross-cutting logic across various components of your application in a decoupled manner.

How we have Object oriented programming, similarly AOP is also a programming paradigm, which defines some set of principles. If you write a program following the AOP principles then it is called AOP style of programming.

In every program there will be two types of logic, one is called primary business logic and other one is helper logic which makes your primary business logic work better. For example calculating the loan amount will be primary business logic, but in addition we may write some logging, here logging is called secondary logic because without logging our functionality will be fulfilled but without loan calculation logic we cannot say our application is complete.

Now logging will be written in one place of the application or will be written across several, as logging will be done across various components of the application so it is called cross-cutting logic.

Generalized examples of cross-cutting concerns are Auditing, Security, Logging, Transactions, Profiling and Caching etc.

In a traditional OOPS application if you want to apply a piece of code across various classes, you need to copy paste the code or wrap the code in a method and call at various places. The problem with this approach is your primary business logic is mixed with the cross-cutting logic and at any point of time if you want to remove your cross-cutting concern; you need to modify the source code. So, in order to overcome this, AOP helps you in separating the business logic from cross-cutting concerns.

By the above, we can understand that AOP compliments OOP but never AOP replaces OOP. The key unit of modularity in OOP is class, whereas in AOP it is Aspect. As how we have various principles of OOP like abstraction, encapsulation etc., AOP also has principles describing the nature of its use. Following are the AOP principles.

5.1 AOP Principles

1) Aspect – Aspect is the piece of code that has to be applied across various classes of the application

2) JoinPoint – The point at which you want to apply the aspect logic, generally in spring you can apply an aspect at method execution
3) Advice – Action taken by an aspect at a particular JoinPoint. Advice indicates how you want to apply the aspect on a joinpoint. There are multiple types of advices like before advice, after returning advice, around advice and throws advice
4) Pointcut – Collection of joinpoint representing on whom you want to advice the aspect
5) Target – The class on which you want to advice the aspect.
6) Weaving – The process of advising a target class with an aspect based on a pointcut to build proxy
7) Proxy – The outcome of weaving is called proxy, where the end class generated out of weaving process contains cross-cutting logic as well.

AOP is not something specific to spring; rather it is a programming technic similar to OOP, so the above principles are generalized principles which can be applied to any framework, supporting AOP programming style.

There are many frameworks in the market which allows you to work with AOP programming few of them are Spring AOP, AspectJ, JAC (Java aspect components), JBossAOP etc. Among which the most popular once are AspectJ and Spring AOP.

Let's try to compare the features between Spring AOP and AspectJ AOP.

Spring AOP	AspectJ AOP
• Only supported joinpoint is method execution • Spring supports run-time weaving; this means the proxy objects will be built on fly at runtime in the memory. • Spring supports static and dynamic pointcuts	• It supports various types of Joinpoints like constructor execution, method execution, field set or field get etc. • AspectJ uses compile-time weaving; this indicates your proxy classes will be available whenever you compile your code. • AspectJ supports only static pointcuts

At the time spring released AOP, already aspectj AOP has acceptance in the market, and seems to be more powerful than spring AOP. Spring developers instead of comparing the strengths of each framework, they have provided integrations to AspectJ to take the advantage of it, so spring 2.x not only supports AOP it allows us to work with AspectJ AOP as well.

In spring 2.x it provided two ways of working with AspectJ integrations, 1) Declarative programming model 2) AspectJ Annotation model. With this we are left with three ways of working with spring AOP as follows.

1) Spring AOP API (alliance API) – Programmatic approach
2) Spring AspectJ declarative approach
3) AspectJ Annotation approach

Even spring has integrated with AspectJ, it supports only few features of AspectJ described as below.

1) Still the supported Joinpoint is only a method execution.
2) Spring doesn't rely on AspectJ weaving capabilities, and uses its own weaving module so the weaving will happens at run-time.
3) As AspectJ doesn't support dynamic pointcut's, integrating with spring doesn't make any difference.

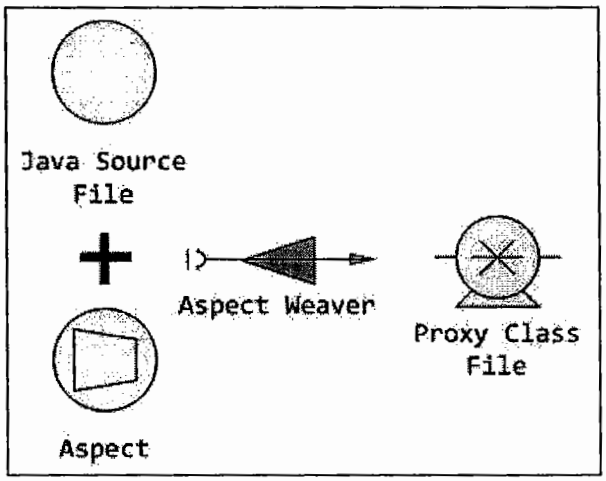


Diagram representing AOP Process

As described the weaving process will happen at run-time, in order to generate a class at run-time, spring uses proxy generation libraries. Spring supports two proxy generation libraries CGLib and JDKProxy run-time bytecode generation libraries to generate proxy classes on fly.

5.2 Types of Advices

Spring AOP implementation supports 4 types of advices, each one is described below.

1) Around advice – This is the advice that executes the aspect around the target class joinpoint, so here the advice method will be called before the target class method execution. So, the advice can control the target method execution.

2) Before advice – In this always the advice method executes before the target class method executes. Once the advice method finishes execution the control will not returned back to the advice method.

3) After Returning advice – Here the advice method executes after the target class method finishes execution, but before it returns the value to the caller.

4) Throws Advice – This advice method will be invoked only when the target class method throws an exception.

Let us try to understand how to work with various advices using different AOP programming styles in the following section.

5.3 Programmatic AOP

We use Spring AOP API's to work with programmatic AOP. As described programmatic aop supports all the types of advices described above. Let us try to work with each advice type in the following section.

5.3.1 Around Advice

Aspect is the cross-cutting concern which has to be applied on a target class; advice represents the action indicating when to execute the aspect on a target class. Based on the advice type we use, the aspect will gets attached to a target class. If it is an around advice, the aspect will be applied before and after, which means around the target class joinpoint execution.

If we have a usecase which demands for applying a cross-cutting concern around the target class joinpoint, we need to use around advice, one of the example for this caching.

Always the around advice method will be executed before your target class method executes by passing the entire target method call information to it. In an around advice we have three control points described below.

1) We have control over arguments; it indicates we can modify the arguments before executing the target class method.
2) We can control the target class method execution in the advice. This means we can even skip the target class method execution.
3) We can even modify the return value being returned by the target class method.

In order to work with any advice, first we need to have a target class, let us build the target class first.

Math.java

```
package com.aa.beans;

public class Math {
	public int add(int a, int b) {
		return a + b;
	}

	public int multiply(int a, int b) {
		return a * b;
	}
```

```
}
```

After creating the target class, we need to code the aspect representing the advice it is using. Create a class for example LoggingAspect which implements from MethodInterceptor and need to override the method invoke; this method accepts a parameter MethodInvocation, into which the entire information about the actual method would be passed.

You can access the original parameters with which the add method was called from MethodInterceptor by calling getArguments method on it. Along with this you can control the method execution by calling methodInterceptor.proceed(); Up on calling the proceed() method, the target method executes, if we don't call proceed() method, the target class method will never get executed. Up on finishing the target method execution the return value being returned by target method will be given to advice method. Here the advice can modify the return value and can return to the caller show below.

LoggingAspect.java

```
package com.aa.beans;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class LoggingAspect implements MethodInterceptor {

	@Override
	public Object invoke(MethodInvocation methodInvocation) throws Throwable {
		Object args[] = methodInvocation.getArguments();
		String methodName = methodInvocation.getMethod().getName();
		// log statement before execution
		System.out.println("entering into method : " + methodName + "("
				+ args[0] + "," + args[1] + ")");

		// modify arguments before calling
		args[0] = (Integer) args[0] + 10;
		args[1] = (Integer) args[1] + 10;
		// proceed() calls the target method add
		Object ret = methodInvocation.proceed();

		// log statement after exection
		System.out.println("exiting the method : " + methodName + "(" +
args[0]
				+ "," + args[1] + ") return value : " + ret);
```

```
            // modify return value
            ret = (Integer) ret + 10;
            return ret;
        }
}
```

In the above code the invoke method is executed when we call add method on the proxy object, by passing the information about the method call to MethodInvocation argument. Before calling the proceed() method, we are modifying the parameters, so that with these modified values proceed will be called. Once the target method execution finishes it returns the value as object in the proceed() method call. Here the advice can modify the return value and can return to the called.

Once the target and advice has been built, we can perform weaving to attach the aspect to the target. In order to perform weaving to build proxy, we need to use ProxyFactory class. To the ProxyFactory we need to add Advice and supply the target class on to which you want to apply that advice. The ProxyFactory will apply the advice on that given target and generates an in-memory proxy class and instantiates and returns that object. As we are working on Programmatic AOP, the weaving process will be done programmatically.

AATest.java

```
package com.aa.test;

import org.springframework.aop.framework.ProxyFactory;
import org.springframework.aop.support.DefaultPointcutAdvisor;

import com.aa.beans.LoggingAspect;
import com.aa.beans.LoggingDynamicPointcut;
import com.aa.beans.LoggingStaticPointCut;
import com.aa.beans.Math;

public class AATest {

        public static void main(String[] args) {
                ProxyFactory pf = new ProxyFactory();
                pf.addAdvice(new LoggingAspect()));
                pf.setTarget(new Math());

                Math math = (Math) pf.getProxy();
                System.out.println("Sum : " + math.add(4, 20));
                //System.out.println("Multiplication : " + math.multiply(1, 2));
        }
}
```

5.3.2 Before Advice

In a Before advice always the advice method executes before your target class joinpoint executes. Once the advice method finishes execution the control will be automatically transferred to the target class method and will not be returned back to the advice. So, in a Before Advice we have only one control point, which is only we can access arguments and can modify them. Below list describes the control points in a Before Advice.

1) Can access and modify the arguments of the original method.
2) Cannot control the target method execution, but we can abort the execution by throwing an exception in the advice method.
3) Capturing and modifying the return value is not applicable as the control doesn't return back to the advice method after finishing you target method.

In order to create a Before Advice, we need to first build a target class. Then we need to create an Aspect class which implements from MethodBeforeAdvice and should override before method. This method has three arguments java.reflect.Method method, Object[] args and Object targetObject. Using these paramters we can access the original method arguments and the method information.

Even we can modify the values in the args array so that with these modified values the target method will gets invoked. Common applications of Before Advice are Security, Authorization etc. Following examples shows the same.

LoanCalculator.java

```
package com.ba.beans;

public class LoanCalculator {
        public float calculateInterest(Long principle, int noOfYears,
                        float rateOfInterest) {
                return (principle * noOfYears * rateOfInterest) / 100;
        }
}
```

LoanCalculator is the target class, before calling the calculateInterest, we want to check whether the caller is logged in and authenticated to call the method or not. We can check this in the calculateInterest() method itself, but if we don't want impose security check for this method, we need to modify source of this method to get rid of cross-cutting security code. Instead we will handle this security check to a MethodBeforeAdvice, where in the advice method we will check for security and if the

security check has been passed then the control will be passed to calculateInterest() method otherwise the advice method will throw exception.

UserInfo.java (Class holding username and password values)

```
package com.ba.beans;

public class UserInfo {
	private String userName;
	private String password;

	public UserInfo(String userName, String password) {
		this.userName = userName;
		this.password = password;
	}

	// setter and getters on userName and password attributes
}
```

AuthenticationManager.java (helper class to perform login, authenticate and logout)

```
package com.ba.beans;

public class AuthenticationManager {
	private static ThreadLocal<UserInfo> threadLocal = new
ThreadLocal<UserInfo>();

	public void login(String un, String pwd) {
		threadLocal.set(new UserInfo(un, pwd));
	}

	public void logout() {
		threadLocal.set(null);
	}

	public boolean isAuthenticated() {
		boolean flag = false;
		UserInfo userInfo = threadLocal.get();
		if (userInfo != null) {
			if (userInfo.getUserName().equals("john")
					&& userInfo.getPassword().equals("welcome1")) {
				flag = true;
			}
		}
		return flag;
```

```
	}
}
```

SecurityAspect.java (Aspect class which executes before target method execution)

```
package com.ba.beans;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class SecurityAspect implements MethodBeforeAdvice {

	@Override
	public void before(Method method, Object[] args, Object target)
			throws Throwable {
		System.out.println("entering into method : " + method);

		// check authenticated
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == false) {
			throw new IllegalAccessException("Invalid username/password");
		}

		// if authentication success
		// modify arguments
		args[0] = (Long) args[0] + 33;
	}
}
```

The code for building the proxy is same as for around advice. In the above the before method in the advice will executes before the calculateInterest() method in LoanCalculator class, here we are checking whether the user is authenticated to access the method or not. If not authenticated, will throw IllegalAccessException and aborting the calculateInterest() method execution.

BATest.java

```
package com.ba.test;

import org.springframework.aop.framework.ProxyFactory;

import com.ba.beans.AuthenticationManager;
import com.ba.beans.LoanCalculator;
import com.ba.beans.SecurityAspect;

public class BATest {

	public static void main(String[] args) {
		ProxyFactory pf = new ProxyFactory();
		pf.addAdvice(new SecurityAspect());
		pf.setTarget(new LoanCalculator());
		LoanCalculator proxy = (LoanCalculator) pf.getProxy();
		AuthenticationManager am = new AuthenticationManager();
		am.login("john", "welcome1");

		System.out.println("Interest : "
				+ proxy.calculateInterest(10L, 1, 12.0f));
	}
}
```

5.3.3 After Returning Advice

In this the advice method will be executed only after the target method finishes execution and before it returns the value to the caller. This indicates that the target method has almost returned the value, but just before returning the value it allows the advice method to see the return value but doesn't allows to modify it. So, in a After returning advice we have the below control points.

1) We can see parameters of the target method, even we modify there is no use, because by the time the advice method is called the target method finished execution, so there is no effect of changing the parameter values.
2) You cannot control the target method execution as the control will enters into advice method only after the target method completes.
3) You can see the return value being returned by the target method, but you cannot modify it, as the target method has almost returned the value. But you can stop/abort the return value by throwing exception in the advice method.

In order to create an After returning advice, after building the target class, you need to write the aspect class implementing the AfterReturningAdvice and should override the method afterReturning, the parameters to this method are Object returnValue (returned by actual method), java.reflect.Method method (original method), Object[] args (target method arguments), Object targetObject (with which the target method has been called).

If you observe in the below example, the return value of afterReturning method is void, which means you cannot modify and return the return value.

KeyGenerator.java

```
package com.ar.beans;

import java.util.Random;

public class KeyGenerator {
	public int generateKey(int size) {
		Random random = new Random(size);
		random.setSeed(5);
		int key = random.nextInt();

		return key;
	}
}
```

WeakKeyCheckerAspect.java (aspect checks whether the key computed by KeyGenerator class is strong or weak key)

```
package com.ar.beans;

import java.lang.reflect.Method;

import org.springframework.aop.AfterReturningAdvice;

public class WeakKeyCheckerAspcect implements AfterReturningAdvice {

	@Override
	public void afterReturning(Object retVal, Method method, Object[] args,
			Object target) throws Throwable {
		if ((Integer) retVal <= 0) {
			throw new IllegalArgumentException("Weak Key Generated");
		}
	}

}
```

5.3.4 Throws Advice

Throws advice will be invoked only when the target class method throws exception. The idea behind having a throws advice is to have a centralized error handling mechanism or some kind of central processing whenever a class throws exception.

As said unlike other advices, throws advice will be called only when the target throws exception. The throws advice will not catch the exception rather it has a chance of seeing the exception and do further processing based on the exception, once the throws advice method finishes execution the control will flow through the normal exception handling hierarchy till a catch handler is available to catch it.

Following are the control points a throws advice has.

1) It can see the parameters that are passed to the original method
2) There is no point in controlling the target method execution, as it would be invoked when the target method rises as exception.
3) When a method throws exception, it cannot return a value, so there is nothing like seeing the return value or modifying the return value.

In order to work with throws advice, after building the target class, you need to write a class implementing the ThrowsAdvice interface. The ThrowsAdvice is a marker interface this means; it doesn't define any method in it. You need to write method to monitor the exception raised by the target class, the method signature should public and void with name afterThrowing, taking the argument as exception class type indicating which type of exception you are interested in monitoring (the word monitoring is used here because we are not catching the exception, but we are just seeing and propagating it to the top hierarchies).

When a target class throws exception spring IOC container will tries to find a method with name afterThrowing in the advice class, having the appropriate argument representing the type of exception class. We can have afterThrowing method with two signatures shown below.

```
afterThrowing([Method, args, target], subclassOfThrowable)
```

In the above signature, Method, args and target is optional and only mandatory parameter is subclassOfThrowable. If a advice class contains afterThrowing method with both the signatures handling the same subclassOfThrowable, the max parameter method will be executed upon throwing the exception by target class.

Below example shows the same.

DURGA SOFTWARE SOLUTIONS, Plot No: 202, IInd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad-500038., Cell. 9246212143. Phone: 040-64512786

Thrower.java

```
package com.ta.beans;

public class Thrower {
	public int willThrow(int i) {
		if (i <= 0) {
			throw new IllegalArgumentException("Invalid parameter i");
		}
		return i + 10;
	}
}
```

ExceptionLoggerAspect.java

```
package com.ta.beans;

import java.lang.reflect.Method;

import org.springframework.aop.ThrowsAdvice;

public class ExceptionLoggerAspect implements ThrowsAdvice {
	public void afterThrowing(IllegalArgumentException ie) {
		System.out.println("thrown : " + ie.getMessage());
	}

	public void afterThrowing(Method method, Object[] args, Object target,
			IllegalArgumentException ie) {
		System.out.println("Exception thrown by : " + method.getName() + "("
				+ args[0] + ") with exception message : " +
ie.getMessage());
	}
}
```

5.3.5 Pointcut

In all the above examples, we haven't specified any pointcut while advising the aspect on a target class; this means the advice will be applied on all the joinpoints of the target class. With this all the methods of the target class will be advised, so if you want to skip execution of the advice logic on a specific method of a target class, in the

advice logic we can check for the method name on which the advice is being called and based on it we can execute the logic.

The problem with the above approach is even you don't want to execute the advice logic for some methods, still the call to those methods will delegate to advice, this has a performance impact.

In order to overcome this you need to attach a pointcut while performing the weaving, so that the proxy would be generated based on the pointcut specified and will do some optimizations in generating proxies.

With this if you call a method that is not specified in the pointcut, spring ioc container will makes a direct call to the method rather than calling the advice for it.

Spring AOP API supports two types of pointcuts as discussed earlier. Those are static and dynamic pointcuts. All the pointcut implementations are derived from org.springframework.aop.Pointcut interface. Spring has provided in-built implementation classes from this Interface. Few of them are described below.

Static Pointcut
1) StaticMethodMatcherPointcut
2) NameMatchMethodPointcut
3) JdkRegexpMethodPointcut

Dynamic Pointcut
1) DynamicMethodMatcherPointcut

5.3.6 Static Pointcut

In order to use a StaticMethodMatcherPointcut, we need to write a class extending from StaticMethodMatcherPointcut and needs to override the method matches. The matches method takes arguments as Class and Method as arguments.

Spring while performing the weaving process, it will determines whether to attach an advice on a method of a target class by calling matches method on the pointcut class we supplied, while calling the method it will passes the current class and method it is checking for, if the matches method returns true it will advise that method, otherwise will skip advicing.

LoggingStaticPointcut.java

```
package com.aa.beans;

import java.lang.reflect.Method;

import org.springframework.aop.support.StaticMethodMatcherPointcut;

public class LoggingStaticPointCut extends StaticMethodMatcherPointcut {

    @Override
    public boolean matches(Method method, Class<?> targetClass) {
```

```
            if (Math.class == targetClass && method.getName().equals("multiply")) {
                return true;
            }
            return false;
        }
}
```

While performing the weaving, we need to supply the pointcut as input to the ProxyFactory as shown below.

BATest.java (Skipped logic for clarity)

```
            ProxyFactory pf = new ProxyFactory();
            pf.addAdvisor(new DefaultPointcutAdvisor(new LoggingDynamicPointcut(),
new LoggingAspect()));
            pf.setTarget(new Math());

            Math math = (Math) pf.getProxy();
```

5.3.7 Dynamic Pointcut

If you observe in static pointcut we have hardcoded the class and method names on whom we need to advice the aspect. So, the decision of advising the aspect on a target would be done at the time of weaving and would not be delayed till the method call. In case of Dynamic Pointcut, the decision of whether to attach an aspect on a target class would be made based on the parameters with which the target method has been called.

LoggingDynamicPointcut.java

```
package com.aa.beans;

import java.lang.reflect.Method;

import org.springframework.aop.support.DynamicMethodMatcherPointcut;

public class LoggingDynamicPointcut extends DynamicMethodMatcherPointcut {
        @Override
        public boolean matches(Method method, Class<?> targetClass, Object[] args) {
            if (Math.class == targetClass && method.getName().equals("add")
                        && (Integer) args[0] > 5) {
                return true;
            }
            return false;
        }
}
```

Table representing the control points for several advices

Advice Type/Control Points	When Advice Method executes	Access to Target method parameters	Control of method execution	Access to Return Value
AroundAdvice	Around Target Method	Can see and Modify	✓	Can see and Modify the return value
MethodBeforeAdvice	Before Target Method executes	Can see and Modify	Don't have control and target method execution, only can abort target method exection	Not applicable (Control will not come back to advice method after finishing the target method execution)
AfterReturningAdvice	After Target method finished execution, but before returning value to the called	Can see the parameters	Not applicable	Can see the return value, but cannot modify them. Can abort returning the value to the called by throwing exception
ThrowsAdvice	Only when the target method throws exception	Can see paramters	Not applicable	Not applicable

5.4 Declarative AOP

Spring 2.x has added support to declarative AspectJ AOP. The main problem with programmatic approach is your application code will tightly couple with spring, so that you cannot detach from spring. If you use declarative approach, your advice or aspect classes are still pojo's, you don't need to implement or extend from any spring specific interface or class. Your advice classes uses AspectJ AOP API classes, in order to declare the pojo as aspect you need to declare it in the configuration file rather than weaving it using programmatic approach.

The declarative approach also supports all the four types of advices and static pointcut. In the following section we will explore through the example how to work with various types of advices declaratively.

5.4.1 Around Advice

In this approach the aspect class is not required to implement from any spring specific class or interface, rather it should be declared as aspect in configuration file. In the aspect class you need to declare any arbitrary method with the following signature.

```
public Object methodName(ProceedingJoinPoint pjp)
```

This acts as an advice method. The return type of the method should be Object as the around advice has control over the return value. The advice method name could be anything but should take the parameter as ProceedingJoinPoint, the method on a target class is called JoinPoint in AOP, as we can control the execution of the Joinpoint in Around advice, so the parameter for this method should be ProceedingJoinPoint.

Using the ProceedingJoinPoint, you can access the actual method information similar to MethodInterceptor. You will declare this method as around advice method in the configuration file using <aop:config> tag, which is discussed in the example below.

In this example we are going to apply the aspect on the same Math.java class which we discussed in Programmatic approach.

LoggingAspect.java

```
package com.aa.beans;

import org.aspectj.lang.ProceedingJoinPoint;

public class LoggingAspect {
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		Object args[] = pjp.getArgs();

		System.out.println("entering into " + methodName + "(" + args[0] + ","
				+ args[1] + ")");
		// modify parameters
		args[0] = (Integer) args[0] + 10;
		Object ret = pjp.proceed(args);

		System.out.println("exiting from " + methodName + "(" + args[0] + ","
				+ args[1] + ") with return value (original) : " + ret);
		// modify ret val
		ret = (Integer) ret + 10;
		System.out.println("exiting from " + methodName + "(" + args[0] + ","
				+ args[1] + ") with return value (modified) : " + ret);
		return ret;
	}
}
```

If you see the above class, we haven't implemented or extended from any interface or class, so in order to make this as aspect class you need to provide this in the spring beans configuration file.

First we need to declare our target and aspect classes as beans. While declaring a class as a aspect, you need to import the "aop" namespace and need to use the tag <aop:config>. Inside the <aop:config> take we need to declare the bean as aspect using <aop:aspect ref="aspectbeanid">. Under this you need to declare which method you want to expose as advice method and how do you want to apply this method like around or before or afterReturning etc. Once this is done you need to supply the pointcut indicating on which target class methods you want to apply the aspect. The configuration has been show in the below code snippet.

application-context.xml

```
<bean id="math" class="com.aa.beans.Math"/>
<bean id="loggingAspect" class="com.aa.beans.LoggingAspect"/>

<aop:config>
	<aop:pointcut expression="execution(* com.aa.beans.Math.*(..))" id="pc1"/>
	<aop:aspect id="la1" ref="loggingAspect">
		<aop:around method="log" pointcut-ref="pc1"/>
	</aop:aspect>
</aop:config>
```

In the above declaration if you observe we have used a pointcut expression rather than referring to a class representing as pointcut. This pointcut expression is a static point cut expression, and it has been written in OGNL expression language. OGNL stands for Object Graph Navigation Language.

Expression starts with execution as a word representing apply the advice to all the class executions and the syntax is described below.

```
execution(<returntype>
<packagename>.<classname>.<methodname>(<arguments>))
```

When you create a core container with the following configuration, spring ioc container while creating will apply the advice on the target class (based on the pointcut) and creates proxy classes and instantiates these classes and host in IOC container.
In the above the "math" bean object will be created on the proxy after applying the advice, so when you request for context.getBean("math") instead of returing the original Math class object, IOC container will instantiates the proxy of Math class as the pointcut is matching to it and returns that object to you.

5.4.2 Before Advice

While working with Before Advice the entire approach is same like creating the aspect class and should declare a method, here the method should be public and the return type should be void as the Before advice cannot control the return value. The parameter to the advice method is JoinPoint rather than ProceedingJoinPoint as we don't have control on method execution.

Once we write a method in the aspect class following these rules, we need to declare that method as before advice method in the configuration shown below.

SecurityAspect.java

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;

public class SecurityAspect {
	public void check(JoinPoint jp, long principal, int noOfYears,
			float rateOfInterest) throws Throwable {
		System.out.println("Principal : " + principal);
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == Boolean.FALSE) {
			throw new IllegalAccessException("User not logged in");
		}

	}
}
```

application-context.xml

```
<aop:config>
	<aop:pointcut expression="execution(* com.ba.beans.*.*(..)) and
args(principal,noOfYears,rateOfInterest))"
		id="pc1" />
	<aop:aspect id="ap1" ref="securityAspect">
		<aop:before method="check" pointcut-ref="pc1" />
	</aop:aspect>
</aop:config>
```

In the above code if you observe the advice method along with JoinPoint, it took the parameters as principle, noOfYears and rateOfInterest as well, these are the parameters which are there on your target class method. If you want to access those parameters straight away in the advice method, you can declare your advice method to accept those values in appropriate arguments, and you need to specify those method parameters as arguments in pointcut expression as args(arg1, arg2, arg3). Here the arg1, arg2 and arg3 are the argument names of the advice method indicating the target method parameters has to be copied to these arguments respectively.

5.4.3 After Returning Advice

In the after returning advice while writing the advice method in the aspect, you need to have declared the method with return type as void as it cannot control the return value and the method takes two parameters. The first parameter is the JoinPoint and the second would be the variable in which you want to receive the return value of the target method execution.

WeakKeyCheckerAspect.java

```
public class WeakKeyCheckerAspect {
	public void checkKey(JoinPoint jp, int generatedKey) {
		// write the code to execute the logic with the return value.
	}
}
```

application-context.xml

```
<aop:config>
	<aop:pointcut expression="execution(* com.ar.beans.*.*(..))"
		id="pc1" />
	<aop:aspect id="ap1" ref="weakKeyCheckerAspect">
		<aop:afterReturning method="checkKey" returning="generatedKey"
pointcut-ref="pc1" />
	</aop:aspect>
</aop:config>
```

5.4.4 Throws Advice

In this we need to write the aspect class with advice method taking the signature as below.

```
public void methodName([Method, args[], targetObject], subClassOfThrowable)
```

and the in declaration we need to declare the variable in which you are receiving the exception as throwing shown below.

application-context.xml

```
<aop:config>
	<aop:pointcut expression="execution(* com.ta.beans.*.*(..))"
		id="pc1" />
	<aop:aspect id="ap1" ref="loggingExceptionAspect">
		<aop:afterReturning method="handle" throwing="ex" pointcut-ref="pc1"
/>
	</aop:aspect>
</aop:config>
```

5.5 AspectJ Annotation AOP

In this approach instead of using declarations to expose the classes as aspect and advice, we will annotate the classes with annotations. In order to make a class as aspect, we need to annotate the class with @Aspect. To expose a method as advice method, we need to annotate the method in aspect class with @Around or @Before or @AfterReturning or @AfterThrowing representing the type of advice. These annotations take the pointcut expression as value in them.

5.5.1 Working with advices

The same rules defined in declarative aop section applies in writing the advice method in annotation approach as well, but only difference is annotate the class as @Aspect and annotate the method with @Around("pointcut expression") shown below.

LoggingSecurityAspect.java

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class LoggingSecurityAspect {

	@Around("execution(* com.ba.beans.Math.*(..))")
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		System.out.println("entering into " + methodName);
		Object ret = pjp.proceed();
		System.out.println("exiting from " + methodName);
		return ret;
	}
}
```

In order to detect the annotations marked on the classes, we need to add <aop:aspect-autoproxy/> tag in the configuration.

If we are using the pointcut expression on the multiple advice methods, instead of re-writing the expression in all the places you can declare a method representing the pointcut expression. Annotate that method with @Pointcut("expression") and in the @Around or @Before or other advice annotations, use the method name as value representing the pointcut expression shows below.

LoggingSecurityAspect.java

```
package com.ba.beans;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class LoggingSecurityAspect {
	@Pointcut("execution(* com.ba.beans.Math.*(..))")
	public void mathPointcut() {
	}
	@Around("mathPointcut()")
	public Object log(ProceedingJoinPoint pjp) throws Throwable {
		String methodName = pjp.getSignature().getName();
		System.out.println("entering into " + methodName);
		Object ret = pjp.proceed();
		System.out.println("exiting from " + methodName);
		return ret;
	}
	@Before("mathPointcut()")
	public void check(JoinPoint jp) throws Throwable {
		AuthenticationManager am = new AuthenticationManager();
		boolean flag = am.isAuthenticated();
		if (flag == false) {
			throw new IllegalAccessException("Invalid user/pwd");
		}
		System.out.println("Authenticated succesfully");
	}
	@AfterThrowing(pointcut = "mathPointcut()", throwing = "ex")
	public void handle(IllegalAccessException ex) {
		System.out.println("I am in handle method");
		System.out.println("Exception Message : " + ex.getMessage());
```

```
    }
    @AfterReturning(pointcut = "mathPointcut()", returning = "sum")
    public void monitor(JoinPoint jp, int sum) {
        System.out.println("In monitor method");
        System.out.println("Returning value : " + sum);
    }

}
```

Spring JDBC

6 Spring JDBC (Java Database connectivity)

Spring JDBC is aimed at making it easy to work with data access technologies like JDBC, Hibernate or JPA in a consistent way. This allows us to switch between the above mentioned technologies fairly easy; along with it, one should not worry about how to handle exceptions in each technology.

Spring JDBC has provided a very strong exception hierarchy to handle any technology specific exception into spring jdbc exception or user-defined exception. It provides automatic translation of Technology specific exception into spring exceptions like for example a SQLException is translated into its own exception DataAccessException which is the root exception.

In a traditional JDBC, user has to write try-catch-finally blocks to handle the checked SQLException's. When it comes to spring JDBC without writing this annoying boiler plate code try-catch-block, spring will takes care of catching and handling these, so your code is freed from most of the glue code and you just need to focus on developing your business/persistence logic.

The main advantage of going with spring framework JDBC is you can avoid lot of boiler plate code that you write in a traditional JDBC, and spring jdbc will provide most of the stuff you do for example, creating a connection, statement etc and managing the resources like closing the connection and statements etc. We can better explain the advantages of using spring jdbc in a tabular fashion describing what spring jdbc offers to you and what you need to do.

Table describing the actions that spring provides and what you need to do apart from those.

Action/Operation	Spring JDBC	You
Declaring connection configuration		✓
Opening the connection	✓	
Supply SQL Statement		✓
Declare parameters and provide values (positional parameters ? or named parameters :paramName)		✓
Prepare statement and execute it	✓	
Loop through the resultset values (if any)	✓	
Wrap or extract the resultset values		✓
Handle and process any exceptions	✓	
Handle Transactions	✓	
Managing and closing connection, statement and resultset	✓	

6.1 Choosing an approach for JDBC Data access

Spring JDBC provides multiple approaches in working with data access logic. It provides the following approaches to work with Data access logic; JdbcTemplate, NamedParameterJdbcTemplate, SimpleJdbcTemplate, SimpleJdbcInsert, Mapping Sql Operations as sub classes.

1) JdbcTemplate - is the classic Spring JDBC approach and the most popular. All the other class in Spring JDBC will extends from this.
2) NamedParamterJdbcTemplate – This is a wrapper on JDBCTemplate and allows you to work with Named Parameters instead of (?) place holders. If you use (?) in you SQL Query, it would be tough to understand and track the values for those, so NamedParamterJdbcTemplate allows you to declare :NamedPrameter in the sql and provides methods to replace those :params while executing the query.
3) SimpleJdbcTemplate – This combines the most frequently used operations of JdbcTemplate and NamedParameterJdbcTemplate.
4) SimpleJdbcInsert and SimpleJdbcCall – In this one it optimizes the amount of coding and only you need to provide the table on which you want to perform in Insert and need to provide Map of values representing the table columns. These classes rely on Database Table meta data in performing the operations.
5) Mapping SQL Operations as Sub classes (Using SqlQuery and SqlUpdate classes) – In this approach you will create reusable sub-classes by declaring the query and their parameter types once and compile it. And you can execute various operations on it by passing different values. This is similar to how you work with JDO.

Spring JDBC Framework classes has been distributed across four packages those are namely core, datasource, object and support. The JdbcTemplate has been declared in core package and all the datasource related classes has been declare in datasource package. Support and Object contains some util and helper classes to make your jdbc work.

6.2 Types of supported JDBC Operations

On a RDBMS, we can perform various types of operations, among which while working with JDBC, we will most work on Data Manipulation and Data Query Language statements rather the other two. Let us detail which type of operations we perform in the below section.

1) Data Definition Language (DDL) – DDL statements are used for creating new schema or tables in a schema, this indicates these statements are used for setting up the database and its tables required for storing the data of your application. When we use JDBC, we try to access the existing data in a database rather creating a new database from Java Language, so we don't need to experiment on DDL.
2) Data Manipulation Language (DML) – These are the statements that supports inserting/manipulating/removing the data from the existing database tables. Here we might insert a single record/a set of records (bulk operations) into the database, spring jdbc supports both the types of operations.

3) Data Query Language (DQL) – This is used for querying information from a database table. There are different types of query operations we can perform on a table as below.
   a. Aggregate operations (COUNT, AVG, SUM etc.)
   b. Select a column in a row
   c. Select a Row as Object
   d. Select list of values/objects
4) Data Control Language (DCL) – Data control language is used for granting or revoking the access to various database objects in a database. This would be generally done by a Database administrator, so it would not be appropriate to perform these operations through JDBC.

So based on above described db operations that can be done on JDBC we would try to perform these operations using JdbcTemplate, NamedParameterJdbcTemplate and with all other approaches as well in the following sections.

6.3 Setting up DataSource

In order to perform anything in a JDBC application, we need a connection object. In a traditional core java jdbc applications, we create the connection to a database using DriverManager, even though every application needs a connection object to work with a database, developer has to write this logic in every application (this kind of code is called boiler plate code). Instead of repeatedly writing it, spring has provided a declarative configuration for creating a connection. In this we declare the parameters that are required to create a connection object and spring would be able to automatically provide the connection to our code.

In order to get a connection, we need to configure DataSource in the configuration. A DataSource is a class which contains configuration information using which it creates connection object. A class which implements javax.sql.DataSource Interface can be configured as a Datasource spring bean. Spring provides one of the implementations of java.sql.DataSource interface as part of its core package which is DriverManagerDataSource. This class is similar to DriverManager class in a traditional JDBC application. For this class we need to set the properties like DriverClass, Url, Usename and Password for creating a connection. The sample configuration has been show in the below fragment.

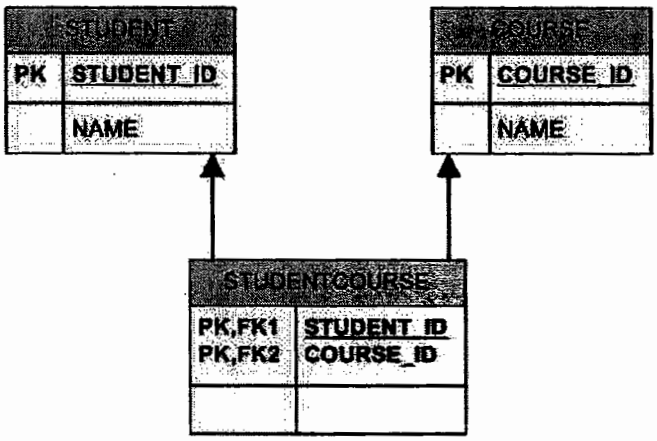
persistence-beans.xml

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="url" value="jdbc:oracle:thin:@//localhost:1521/xe" />
      <property name="username" value="hr" />
      <property name="password" value="hr" />
      <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
</bean>
```

The above dataSource bean is being injected into rest of the Spring Jdbc classes like JdbcTemplate or NamedParamterJdbcTemplate to execute Sql Operations.

6.4 Sample Schema and Table Structure

For explaining the examples on Spring Jdbc, we need some tables to execute operations, you might choose to work on these examples either on an Oracle database or MsSqlServer database, based on the database you are working you need to import the relevant Driver jar. Along with that you need to create some tables to understand and execute the examples which we are going to discuss, so here an E-R representation of the table structure has been given below based on that you should create your tables.



6.5 Using JDBCTemplate

JDBCTemplate is the central class in the JDBC core package. It handles the creation and release of resources, which helps you avoid common errors like forgetting to close a resource etc. Along with this it supports the basic jdbc workflow like creating a statement, executing a statement and iterating over the resultset etc. Here you need to write the business logic for using the resultset values. In this way the core functionality required to execute your sql query will be facilitated by JdbcTemplate.

Apart from this if you want to work with your own logic in executing the statements and iterating the resultset values etc, spring JdbcTemplate allows you to work with PreparedStatements and CallableStatements by providing classes like PreparedStatementCreator, PreparedStatementCallback and CallableStatementCreator, CallableStatementCallback etc. When working with those classes, these classes has two phases of execution 1) Prepare or create phase 2) Execute and Callback to handle the result phase. In the prepare phase you need to create the prepared statement by using the connection object passed to you by JdbcTemplate and in the CallBack phase you need to execute the statement and need to iterator over the resultset values. Here it seems like you are not getting most out spring jdbc, but the creation and closing of the resources has been taken care by JdbcTemplate itself.

By this we understood that working with JdbcTemplate involves two ways, one is straight away using the JdbcTemplate methods to get the resultset values and another is creating your own statements and call back handlers. We will discuss further these approaches using example.

6.5.1 Working with PreparedStatements using Jdbc Template

Spring JdbcTemplate provides lot of methods which supports executing an SQL statement and returning the results in pojo object. If you want to write your own logic of handling the resultset values rather than wrapping into a pojo object, then you need to create a statement and iterator over the resultset to perform business logic. JdbcTemplate exposes convenient methods which allow you to execute a PreparedStatement.

Executing a PreparedStatement involves two phases, creating or preparing the PreparedStatement using the PreparedStatementCreator class and executing and wrapping the resultset values in callback phase using PreparedStatementCallback.

Before creating the PreparedStatementCreator or Callback class to execute the statement, we need to setup the JdbcTemplate with a Datasource object. In order to execute any Sql Query we needs connection object and this would be provided by the DataSource bean which we discussed in the previous section. So, while JdbcTemplate performs any operation it will try to get the connection from the datasource with which we created it. The below code fragment shows how to configure a JdbcTemplate to take DataSource as dependent object.

persistence-beans.xml

```
<bean id="jdbcTempalate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dataSource" />
</bean>
```

Once you create a JdbcTemplate bean with the above configuration, you need to inject this bean into the Dao classes to perform Sql Operations.

EmployeeDao.java

```
package com.emp.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCallback;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.RowMapper;

import com.emp.business.Employee;

public class EmployeeDao {
	private JdbcTemplate jdbcTemplate;

	// insert datasource and create jdbcTemplate
	public EmployeeDao(DataSource dataSource) {
			jdbcTemplate = new JdbcTemplate(dataSource);
	}


	public List<Employee> getEmployeesByName(final String empName) {
		PreparedStatementCreator creator = new PreparedStatementCreator() {
```

```
                public PreparedStatement createPreparedStatement(
                        Connection connection) throws SQLException {
                    PreparedStatement ps = connection
                            .prepareStatement("select * from tblemp where
emp_name like ?");
                    ps.setString(1, empName);
                    return ps;
                }
            };
```

In contd...

```
            PreparedStatementCallback<List<Employee>> callBack = new
PreparedStatementCallback<List<Employee>>() {

                public List<Employee> doInPreparedStatement(
                        PreparedStatement preparedStatement) throws
SQLException,
                        DataAccessException {
                    List<Employee> employees = new ArrayList<Employee>();
                    ResultSet rs = preparedStatement.executeQuery();
                    while (rs.next()) {
                        Employee e = new Employee(rs.getInt("emp_id"),
                                rs.getString("emp_name"),
rs.getFloat("salary"));
                        employees.add(e);
                    }

                    return employees;
                }
            };
            return jdbcTemplate.execute(creator, callBack);
        }
}
```

persistence-beans.xml

```
<!—either you can inject jdbcTemplate bean or inject dataSource and create
JdbcTemplate object by passing dataSource to its constructor →

<bean id="employeeDao" class=" com.emp.dao.EmployeeDao">
        <constructor-arg ref="dataSource" />
</bean>
```

6.5.1 Using JdbcTemplate Operations

As we discussed earlier let us try to explore how to perform various DML and DQL operations using the methods of JdbcTemplate class in the following sections.

6.5.1.1 Aggregate Operations

The aggregate operations that could be performed on a database are SUM, AVG, COUNT etc. All these functions will result in single columned results. So in order to execute an aggregate query using JdbcTemplate class, there is a method queryForInt(), to this method you need to pass the SQL where the result of execution of that query should yield an integer single columned value. Let us try to understand with an example how to work with it.

StudentDao.java

```
package com.jdbctemplate.dao;

import org.springframework.jdbc.core.JdbcTemplate;
import com.jdbctemplate.business.StudentBO;

public class StudentDao {
	private final String SQL_COUNT_OF_STUDENT = "SELECT COUNT(*) FROM
STUDENT";
	private JdbcTemplate jdbcTemplate;
	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}
	public int getCountOfStudents() {
		return jdbcTemplate.queryForInt(SQL_COUNT_OF_STUDENT);
	}
}
```

In the above code we called the method queryForInt to execute our sql query and give the result of execution as Integer rather than ResultSet. In further detail to it, we haven't created the Statement or ResultSet objects or even the connection rather we just configure the JdbcTemplate class as a bean in the configuration and inject into StudentDao as contructor argument, and JdbcTemplate is going to barrow the connection object and going to execute the sql query you passed to it.

Note: - In the above code the SQL Query has been declared as an constant String variable SQL_COUNT_OF_STUDENT. It is recommended to declare your queries as constants and pass those variables to your methods where you want to execute that SQL. This is not a Spring JDBC constraint rather this is J2ee Best Pratice so that you application code maintenance would be easy.

6.5.1.2 Query Single column value

You may query one value from a column for example finding a Student Name by Student Id; here the Student Id is a primary key so that always the result of query execution against a primary key column would yield only one value. As we are trying to find student name by primary key student id the result is one student name.

In order to query a single column value as Object, we need to use queryForObject() method. It has lot of variants, out of which one of it takes parameter as the SQL query and the type of value returned out of execution and Object array, as we are selecting only NAME by STUDENT_ID, the result of execution is a String value. In the WHERE clause of the query we need to specify the condition on STUDENT_ID =?. In this (?) is the place holder or substitution positional parameter for which we are going to provide the value while executing the query, which is show below.

StudentDao.java

```
package com.jdbctemplate.dao;

import org.springframework.jdbc.core.JdbcTemplate;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_FIND_NAME_BY_ID = "SELECT NAME FROM STUDENT
WHERE STUDENT_ID = ?";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public String getStudentNameById(int studentID) {
		return jdbcTemplate.queryForObject(SQL_FIND_NAME_BY_ID,
String.class,
				new Object[] { studentID });
	}
}
```

In the above example the query contains a (?) which is a place holder going to be replaced while executing it. If you observe while calling queryForObject method the first parameter is SQL query, second one is the type of outcome we are expecting after executing the query and third argument is Object array containing studentID value, this value will be replaced with the (?) in the query while executing. Let's say if your query contains two (?)'s, then you need to pass two values as part of your Object array so that the first (?) will be replaced with first value of the Object array and the second one with second value of the array respectively.

6.5.1.3 Query Single Row as Object

In order to query a record from a table, we need to use queryForObject() method, as we said there are multiple signatures of the queryForObject() method, the one we use here take parameters as queryForObject(SQL,Object[],RowMapper); If you observe the method signature, the third argument to the method is RowMapper.

The JdbcTemplate can execute the query which you have provided to it by replacing the (?) with Object[] values. But after executing the query, it cannot wrap the result values into your pojo. So the RowMapper is an Interface which contains a method mapRow(ResultSet, Row Index), you need to pass the object of RowMapper implementation class to the JdbcTemplate's, queryForObject method.

Once the method has finished executing the query it will passes the result set object to the mapRow method of the RowMapper object to convert the resultset values into Object.

So, you need to write the code for mapping the resultset values to object in the mapRow method and should provide it to JdbcTemplate, using which the JdbcTemplate will executes the query and build the object for you. This is shown in the below example.

StudentDao.java

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_FIND_STUDENT_BY_ID = "SELECT STUDENT_ID, NAME
FROM STUDENT WHERE STUDENT_ID = ?";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public StudentBO findStudent(int id) {
		return jdbcTemplate.queryForObject(SQL_FIND_STUDENT_BY_ID, new
StudentRowMapper(), new Object[] { id });
	}

	private static final class StudentRowMapper implements
RowMapper<StudentBO> {
```

```
        @Override
        public StudentBO mapRow(ResultSet rs, int rowIndex)
                throws SQLException {
            return new StudentBO(rs.getInt("STUDENT_ID"), rs
                    .getString("NAME"));
        }
    }

}
```

StudentBO.java

```
package com.jdbctemplate.business;

public class StudentBO {
    private int student_id;
    private String name;

    public StudentBO(int student_id, String name) {
        this.student_id = student_id;
        this.name = name;
    }

    // setters and getters on the attributes

    @Override
    public String toString() {
        return ("Student ID : " + this.getStudent_id() + " Name : " + this
                .getName());
    }
}
```

6.5.1.4 Query Multiple Rows as List of Objects

This is similar to working with querying one row as object but the only difference is instead of calling queryForObject(), you need to call the simple query() method. Same is shown below

StudentDao.java

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_SELECT_ALL_STUDENTS = "SELECT STUDENT_ID,
NAME FROM STUDENT";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public List<StudentBO> getAllStudents() {
		return jdbcTemplate.query(SQL_SELECT_ALL_STUDENTS, new
StudentRowMapper());
	}
```

```
	private static final class StudentRowMapper implements
RowMapper<StudentBO> {
		@Override
		public StudentBO mapRow(ResultSet rs, int rowIndex)
				throws SQLException {
			return new StudentBO(rs.getInt("STUDENT_ID"), rs
```

```
                                .getString("NAME"));
            }
        }

}
```

6.5.1.5 *Insert/Update/Deleting a record*

In order to perform Insert or Update or Delete operations on a table, we need to use the update method provided in JdbcTemplate class. This method will perform the Insert/Update/Delete based on the query you passed to it. The result of executing this method is an Integer indicating the number of rows affected due to the execution.

The update method takes arguments as update(SQL, Object[]), where the values in the Object[] would be replaced with the (?) place holders.

StudentDao.java

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDao {

        private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
        private JdbcTemplate jdbcTemplate;

        public StudentDao(JdbcTemplate jdbcTemplate) {
                this.jdbcTemplate = jdbcTemplate;
        }

        public int insert(int student_id, String name) {
                return jdbcTemplate.update(SQL_INSERT_STUDENT, new Object[] {
                                student_id, name });
        }
}
```

In the above example instead of passing INSERT query to the update method if you pass UPDATE SQL query it will performs update operation the same applies to DELETE as well.

6.5.1.6 *Batch Operations*

If we want to perform bulk insert or update operations on a database, we need to use batchUpdate method, this method takes parameter as BatchPreparedStatementSetter Interface implementation object.

The BatchPreparedStatementSetter interface has methods setValues() and getBatchSize(). The getBatchSize() method should return a Integer value indicating number of records you want to insert as part of operation. The setValues() method takes argument as PreparedStatement, the batchUpdate method while inserting the records it would call the setValues() method for BatchSize no of times by passing PreparedStatement object as parameter to you, in this method you need to set the values which you want to insert as part of this batch. The same is shown below.

StudentDao.java

```
package com.jdbctemplate.dao;

import java.sql.ResultSet;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.jdbctemplate.business.StudentBO;

public class StudentDao {

	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public int[] insert(final List<StudentBO> students) {
		return jdbcTemplate.batchUpdate(SQL_INSERT_STUDENT, new
StudentBatchSetter());
	}

	private static final class StudentBatchSetter implements
BatchPreparedStatementSetter {
```

```
            @Override
            public void setValues(PreparedStatement preparedStatement,
                        int index) throws SQLException {
                  preparedStatement.setInt(1, students.get(index)
                              .getStudent_id());
                  preparedStatement.setString(2, students.get(index)
                              .getName());
            }
            @Override
            public int getBatchSize() {
                  return students.size();
            }
      }
}
```

6.6 Using NamedParameterJDBCTemplate

While working with JdbcTemplate, if we want to pass any values to the queries while executing, we need to declare the (?) place holders in the query and need to pass the values for these place holders in an Object array. It would be tough to correlate the values you are passing the object array to a (?) field, as we completely dependent on order of passing, this approach is easily error prone and future changes involves lot of effort.

In order to overcome the above dis-advantage spring has introduces a class NamedParamterJdbcTemplate, it a wrapper on JdbcTemplate class, which means it exactly contains similar set of features that JdbcTemplate. In addition, NamedParameterJdbcTemplate allows you to declare named parameters in the query instead of (?) as ":varName".

So, while executing the query you can replace the ":varName" with the value. So that you will be sure that for which named parameter what is the value being passed and you don't need to rely on order of passing.

Basically you need to pass values for Named Parameters declare in a query. One way of passing the values is Map<String,Object> where string represents the named Paramter name and Object represents the value you want to pass for it.

Apart from this you can create an object of SqlParameterSource implementation class object; one of the implementation is MapSqlParameterSource, which represents data in terms of Map of Key and values. Below example shows how to pass values for named parameters declared in the query.

Another implementation of SqlParameterSource is BeanPropertyParamterSource this takes input as Object and creates Map of Key and value where key is the Object attribute name and value as its value.

The below example depicts all these scenarios.

StudentDao.java

```
package com.npjt.dao;

import com.npjt.business.StudentBO;

public class StudentDao {
	private final String SQL_SEARCH_STUDENT_BY_NAME = "SELECT STUDENT_ID,
NAME FROM STUDENT WHERE NAME LIKE :STUDENT_NAME";
	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(:studentId, :studentName)";

	private NamedParameterJdbcTemplate npJdbcTemplate;

	public StudentDao(NamedParameterJdbcTemplate npJdbcTemplate) {
		this.npJdbcTemplate = npJdbcTemplate;
	}

	public List<StudentBO> searchStudentByName(String name) {
		SqlParameterSource paramMap = new
			MapSqlParameterSource("STUDENT_NAME", name);

		return npJdbcTemplate.query(SQL_SEARCH_STUDENT_BY_NAME,
				paramMap, new StudentRowMapper());
	}

	public int insert(StudentBO student) {
```

```
            SqlParameterSource paramSource = new
                        BeanPropertySqlParameterSource(student);
            return npJdbcTemplate.update(SQL_INSERT_STUDENT, paramSource);
      }

      private static final class StudentRowMapper implements
                                          RowMapper<StudentBO> {
            @Override
            public StudentBO mapRow(ResultSet resultSet, int rowNum)
                        throws SQLException {
                  return new StudentBO(resultSet.getInt("STUDENT_ID"),
                              resultSet.getString("NAME"));
            }
      }
}
```

6.7 Mapping SQL Operations as Sub classes

As mentioned earlier there are multiple ways of working with spring jdbc, one of which is mapping the Sql operations as sub classes. This is similar to the concept of JDO where you define the query, declare the parameters and compile the query. Once you do that you can execute the query with various parameters.

In order to map sql operations as sub classes, spring jdbc has provided SqlQuery, SqlUpdate classes from which we need to extend our class. We need to provide the query and datasource as parameters while creating the sub class and need to compile the query.

6.7.1 Using SqlQuery

If you want to perform Select operations, you need to create a sub-class inside your DAO class which extends from SqlQuery class, but SqlQuery class is the base class, instead of using it, there is a another sub-class of SqlQuery which is MappingSqlQuery which has sophisticated method mapRow where you map a resultset values to an Object.

Once you create a sub-class from MappingSqlQuery, you need to pass the dataSource and Sql to be used as part of this class. Once you set up datasource and Sql Query for execution, you need to call super.compile() which will compiles the query only once after creating the object of it. Then there are two types of methods on the sub class available 1) executor methods and 2) finder methods. The executor method returns

List of Objects and Finder methods returns only one object. Based on the nature of operation you perform you need to use appropriate methods to query the data.

StudentDao.java

```
package com.mso.dao;

public class StudentDao {
        private DataSource dataSource;

        public MapStudentDao(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public List<StudentBO> searchStudentsByName(String name) {
                return new SelectStudent(dataSource).searchStudentByName(name);
        }
```

Inner class in contd....

```
        private final class SelectStudent extends MappingSqlQuery<StudentBO> {

                public SelectStudent(DataSource ds) {
                        super(ds, "SELECT STUDENT_ID, NAME FROM STUDENT WHERE
                                                                NAME = ?");
                        super.declareParameter(new SqlParameter("NAME",
                                                                Types.VARCHAR));
                        super.compile();
                }

                @Override
                protected StudentBO mapRow(ResultSet resultSet, int rowNum)
                                throws SQLException {
                        return new StudentBO(resultSet.getInt("STUDENT_ID"),
                                        resultSet.getString("Name"));
                }

                public List<StudentBO> searchStudentByName(String name) {
                        return execute("%N%");
                }
        }
}
```

6.7.1 Using SqlUpdate

SqlUpdate class encapsulates a Sql Upate. Like similar to SqlQuery, the SqlUpdate is also a reusable class. You can call the update() operations on this class with various parameters once the query is compiled. Unlike SqlQuery, this is a concrete class but if you want to provide customized update, you can create a inner class which extends from SqlUpdate and can execute the operations on it.

StudentDao.java

```
package com.mso.dao;

public class StudentDao {
        private DataSource dataSource;

        public MapStudentDao(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public int update(int studentId, String name) {
                return new UpdateStudent(studentId, name);
        }
```

Inner class contd....

```
        private final class UpdateStudent extends SqlUpdate {

                public UpdateStudent(DataSource ds, String sql) {
                        super(ds, "UPDATE STUDENT SET NAME=? WHERE STUDENT_ID =
                                                                        ?");
                        declareParameter(new SqlParameter(Types.VARCHAR));
                        declareParameter(new SqlParameter(Types.INTEGER));
                        compile();
                }

                public int update(int id, String name) {
                        return update(new Object[] { name, id });
                }
        }
}
```

6.8 SimpleJdbcInsert

SimpleJdbcInsert allows you to working with insert operations with minimal or no configuration. Here you don't need to provide the SQL query, rather you will pass the table name and the data map containing key as column name and value as Object type.

When you call execute method by passing the data map of values, it would automatically query the metadata of the table into his it has to perform insert operation and builds a query and inserts the data.

StudentDao.java

```
public class SimpleStudentInsertDao {
	private SimpleJdbcInsert simpleJdbcInsert;

	public SimpleStudentInsertDao(DataSource dataSource) {
		this.simpleJdbcInsert = new SimpleJdbcInsert(dataSource);
	}

	public void insert(StudentBO student) {
		simpleJdbcInsert.setTableName("STUDENT");
		SqlParameterSource paramSource = new
BeanPropertySqlParameterSource(student);
		simpleJdbcInsert.execute(paramSource);
	}

	public void insert(int id, String name) {
		simpleJdbcInsert.setTableName("STUDENT");
		Map<String, Object> data = new HashMap<String, Object>();
		data.put("STUDENT_ID", student.getStudentId());
		data.put("NAME", student.getStudentName());
		simpleJdbcInsert.execute(data);
	}
}
```

Spring

DURGA SOFTWARE SOLUTIONS, Plot No: 202, IInd Floor, HUDA Maitrivanam, Ameerpet, Hyderabad-500038., Cell. 9246212143. Phone: 040-64512786

Transaction

7 Spring Transaction Support

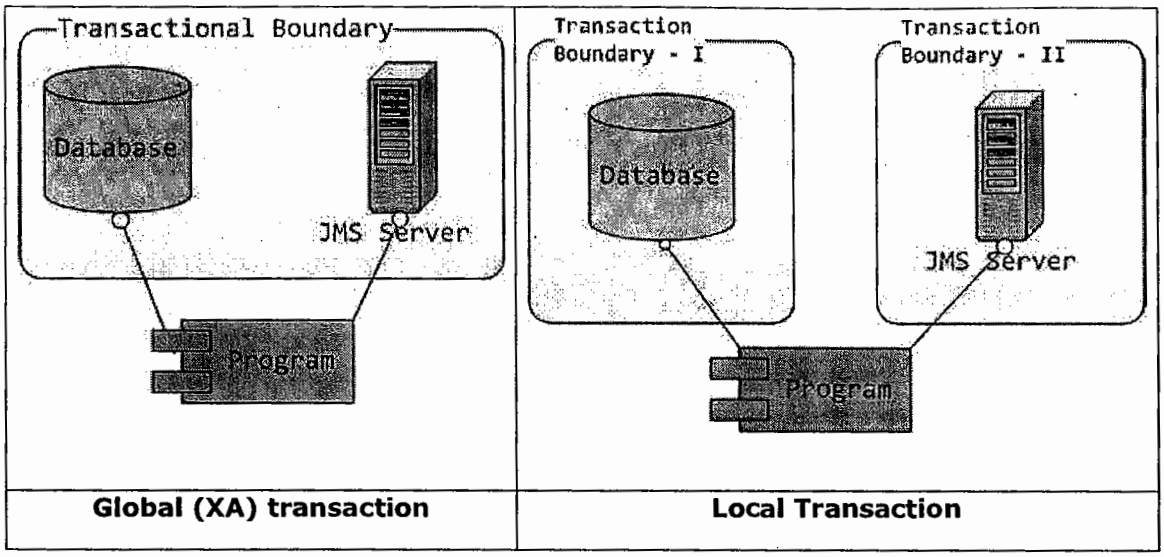
Spring transaction framework provides a consistent abstraction across transaction management API's. Any application extensively requires transaction management capabilities to handle its business operations consistently.

In a traditional Java or J2ee programming, developer has two choices for Transaction management. Those are Global or Local transactions; each of them has specific use cases where they have to be used.

7.1 Global Transaction

Global transactions are also called as XA Transactions, a transaction is said to be Global or XA only when multiple resources are involved in it. Typically the resources would be a database and a message queues etc. When we say these two resources are involved in a transaction, it means performing the operation on a database and on a messaging queue should be part of single transaction and if a roll back happens it should roll back the operations on both the resources rather than one.

The below figure shows the transactional boundaries in a Global transaction and Local Transaction.



The participating resources of a global transaction must be also XA resources rather than local resources. For example while configuring a Datasource to get the connection, the application server will give you an option of selecting the Datasource as XA or non-XA, this indicates whether the connections that are created out of that datasource will participate in global transactions or not.

7.1.1 Two-Phase commit

Global transaction will use the two phase commit process. This indicates the global transaction will be managed by a Transaction coordinator (also called as TransactionManager). Every resource in the transaction is maintained by the ResourceManager and executes the transaction as described below.

Phase – I
a) Each ResourceManager coordinates the local operations and writes them to log
b) If successful operations, response is OK

Phase – II -- If all ResourceManager's responds OK
a) Coordinator Issues commit on all resource managers
b) Participants complete the commit process and writes to log file.

Otherwise:
a) Coordinator issues roll back to all the resourcemanagers.
b) Participants undo's all the operations locally.

In order to work with Global Transactions or Distributed transactions, you need a Transaction Coordinator which will be maintained by a J2EE Server. To work with XA Transactions, J2EE has provided api's like JTA or EJB etc.

In JTA you need to programmatically manage transactions, and the API complex to work with and need to perform JNDI lookup to get the TransactionCordinator from J2EE server.

You can use declarative transaction management using EJB's, but EJB's are heavy wait components and may not be appropriate for whom who are looking for only Global Transaction solution (as EJB are distributed components).

7.2 Local Transaction

Local Transaction is transaction which involves only one resource as part of transactional boundary. If you are working with database and message queue each and every operation that has been done are committed separately, rather than part of single transaction. The above figure depicts the same.

In a local transaction, the commit of a transaction will be done on the resource directly for example while you are working on JDBC, you will issue a commit on directly the connection using con.commit().

7.3 Benefit of Spring Transaction

By the above we understood that working with global and local transactions needs two different API's and need to learn two different API's. Your code is tightly coupled with one transaction implementation, and if you want to switch between local and global or with in global you need to modify your code. Instead spring transaction management capability has provided a unified approach that allows you to work with local or global transactions transparently.

Spring has provided three ways of working with transactions.

1) Programmatic transaction management
2) Declarative transaction management
3) Annotation driven transaction management

Out of which developers extensively uses declarative transaction management.

7.4 Declarative Transaction Management

In a declarative transaction management approach, the developer will declare the transactional semantics in a configuration file rather than programming it using code. The main advantage with this is instead of writing the logic to manage, your application classes are free from transaction related logic and if you want to switch or don't want to use transactions you don't need to modify your logic.

As transactions is a cross-cutting functionality, to implement transactionality we use declarative AOP configuration. In order to work with AOP based transactions, we need to import AOP and Tx Namespaces.

We begin and commit or rollback transactions on an operation, we will begin the transaction while entering a method and while returning from the method we will

commit and if the method throws exception we will roll back the transaction. So, if you observe it is the combination of Around and Throws advice.

In order to manage it we need to declare an transactional advice which will manages the above said logic. In order to configure an transactional advice we need to use Tx namespace to declare it as shown below.

```
<tx:advice id="txAdvice" transaction-manager="transactionManager">
	<tx:attributes>
		<tx:method name="insert*" read-only="false"/>
	</tx:attributes>
</tx:advice>
```

The transaction advice will issue a commit or rollback on the transaction manager, so it needs a transaction manager to perform this, to declare a transaction manager; you need to pass the datasource as a reference to manage all the connections created by the datasource. read-only = "false" indicates you are performing an updatable operation. If you are performing only select type of operation you need to use read-only="true". Along with that you can specify the propagation attribute as well.

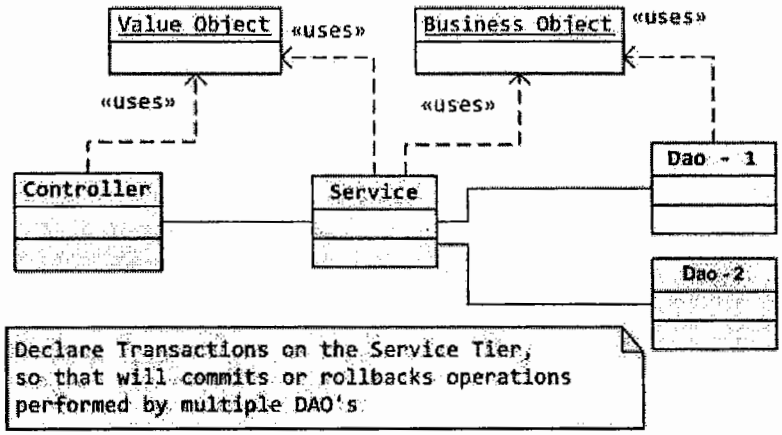
```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
	<property name="dataSource" ref="dataSource"/>
</bean>
```

Once the advice has been declared, you need to declare the classes on which you want to apply this advice using the <aop:confg> tag as shown below.

```
<aop:config>
	<aop:pointcut expression="execution(* com.dtx.service.*.*(..))" id="txpc"/>
	<aop:advisor advice-ref="txAdvice" pointcut-ref="txpc"/>
</aop:config>
```

7.5 Typical Spring Project Flow

In a typical spring application, the transactionality will be imposed on Service tier classes, the same has been depicted in the below architecture.



Let us work on an example which follows the above architecture. In the below example we are trying to insert a Student, inserting a student involves inserting the Student information into student table as well as inserting into student course table. These two insertions has to be performed as part of single transactions. So the service will perform operations by using two dao's and the transaction is imposed on Service class.

EducationalController.java

```
package com.dtx.controller;

import com.dtx.service.DurgaEducationalService;

public class EducationalController {
        private DurgaEducationalService durgaEducationalService;

        public void insert(int studentId, String name, int courseId) {
                int outcome = 0;
```

```
            outcome = durgaEducationalService.insert(studentId, name, courseId);
            if (outcome > 0) {
                  System.out.println("Student Inserted Successfully");
            }
      }

      public void setDurgaEducationalService(
                  DurgaEducationalService durgaEducationalService) {
            this.durgaEducationalService = durgaEducationalService;
      }
}
```

DurgaEducationalService.java

```
package com.dtx.service;

public interface DurgaEducationalService {
      public int insert(int studentId, String name, int courseId);
}
```

DurgaEducationalServiceImpl.java

```
package com.dtx.service;

import org.springframework.transaction.annotation.Transactional;

import com.dtx.dao.StudentCourseDao;
import com.dtx.dao.StudentDao;

public class DurgaEducationalServiceImpl implements DurgaEducationalService {
      private StudentDao studentDao;
      private StudentCourseDao studentCourseDao;
```

```
	@Override
	public int insert(int studentId, String name, int courseId) {
		int outcome = 0;
		outcome = studentDao.insert(studentId, name);
		if (outcome > 0) {
			// re-intialize
			outcome = 0;
			outcome = studentCourseDao.insert(studentId, courseId);
		}
		return outcome;
	}

	public void setStudentDao(StudentDao studentDao) {
		this.studentDao = studentDao;
	}

	public void setStudentCourseDao(StudentCourseDao studentCourseDao) {
		this.studentCourseDao = studentCourseDao;
	}
}
```

StudentDao.java

```
package com.dtx.dao;

public interface StudentDao {
	public int insert(int studentId, String name);
}
```

StudentDaoImpl.java

```
package com.dtx.dao;

import org.springframework.jdbc.core.JdbcTemplate;

public class StudentDaoImpl implements StudentDao {
	private final String SQL_INSERT_STUDENT = "INSERT INTO
STUDENT(STUDENT_ID, NAME) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentDaoImpl(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
```

```
	}

	@Override
	public int insert(int studentId, String name) {
		return jdbcTemplate.update(SQL_INSERT_STUDENT, new Object[] {
				studentId, name });
	}
}
```

StudentCourseDao.java

```
package com.dtx.dao;
public interface StudentCourseDao {
	public int insert(int studentId, int courseId);
}
```

StudentCourseDaoImpl.java

```
package com.dtx.dao;
import org.springframework.jdbc.core.JdbcTemplate;

public class StudentCourseDaoImpl implements StudentCourseDao {
	private final String SQL_INSERT_STUDENTCOURSE = "INSERT INTO
STUDENTCOURSE(STUDENT_ID, COURSE_ID) VALUES(?,?)";
	private JdbcTemplate jdbcTemplate;

	public StudentCourseDaoImpl(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	@Override
	public int insert(int studentId, int courseId) {
		return jdbcTemplate.update(SQL_INSERT_STUDENTCOURSE, new
Object[]{studentId,courseId});
	}
}
```

application-context.xml

```
<bean id="educationalController" class="com.dtx.controller.EducationalController">
	<property name="durgaEducationalService" ref="durgaEducationalService" />
</bean>

<bean id="durgaEducationalService"
class="com.dtx.service.DurgaEducationalServiceImpl">
	<property name="studentDao" ref="studentDao" />
	<property name="studentCourseDao" ref="studentCourseDao" />
</bean>

<bean id="dataSource"
```

```
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
        <property name="url" value="jdbc:oracle:thin:@//localhost:1521/xe" />
        <property name="username" value="hr" />
        <property name="password" value="hr" />
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <constructor-arg ref="dataSource" />
</bean>

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="studentDao" class="com.dtx.dao.StudentDaoImpl">
        <constructor-arg ref="jdbcTemplate" />
</bean>

<bean id="studentCourseDao" class="com.dtx.dao.StudentCourseDaoImpl">
        <constructor-arg ref="jdbcTemplate" />
</bean>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
                <tx:method name="insert*" read-only="false"/>
        </tx:attributes>
</tx:advice>

 <aop:config>
        <aop:pointcut expression="execution(* com.dtx.service.*.*(..))" id="txpc"/>
        <aop:advisor advice-ref="txAdvice" pointcut-ref="txpc"/>
</aop:config>
```

7.6 Annotation approach Transaction Management

In this instead of declaring the transaction semantics in the declaration file we will directly annotate the class methods with @Transactional annotation. You can specify the readOnly and Propagation attributes in it.

In the earlier example instead of declaring the transactional semantics in the declaration, you can directly mark the service class methods with related to transaction annotation as shown below.

DurgaEducationalServiceImpl.java

```
package com.dtx.service;
```

```
import org.springframework.transaction.annotation.Transactional;

import com.dtx.dao.StudentCourseDao;
import com.dtx.dao.StudentDao;

public class DurgaEducationalServiceImpl implements DurgaEducationalService {
	private StudentDao studentDao;
	private StudentCourseDao studentCourseDao;

	@Override
	@Transactional(readOnly="false")
	public int insert(int studentId, String name, int courseId) {
		int outcome = 0;
		outcome = studentDao.insert(studentId, name);
		if (outcome > 0) {
			// re-intialize
			outcome = 0;
			outcome = studentCourseDao.insert(studentId, courseId);
		}
		return outcome;
	}

	public void setStudentDao(StudentDao studentDao) {
		this.studentDao = studentDao;
	}

	public void setStudentCourseDao(StudentCourseDao studentCourseDao) {
		this.studentCourseDao = studentCourseDao;
	}
}
```

In order to detect your annotations that has been marked at class level, you need to declare a tag in spring beans configuration file as

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Spring MVC

8 Spring Web MVC (Model View and Controller)

Spring Web MVC Framework allows you to build web applications, similar to struts and J2ee servlets etc. When compared with any other frameworks, spring offers comprehensive list of features that makes you develop web applications with greater speed and flexibility.

8.1 Advantages of Spring Web MVC

1) Form Object: - In spring mvc you can use any object as command or form object (similar to Struts Action Form) to capture user submitted form values. Your Form object need not implement any spring specific classes or interface.

2) Flexible data binding: - Unlike other frameworks, spring form object can contain non-string typed as attributes, and spring will take care of binding the form submitted values to your form object attributes. In frameworks like struts, ActionForm should hold only string-typed attributes and need to perform the validation to capture type-casting errors. But spring will take care of these conversions and any in-compatible conversions will results into validation errors rather than runtime or type-casting errors.

3) Reusable business code: – no need of duplication, you can use your existing business objects as command or form objects.

4) Clear separation of roles: - controller, validator, command, handler mappings, dispatcher servlet, view resolvers, in this way spring provides multiple components to handle each role by a specialized object.

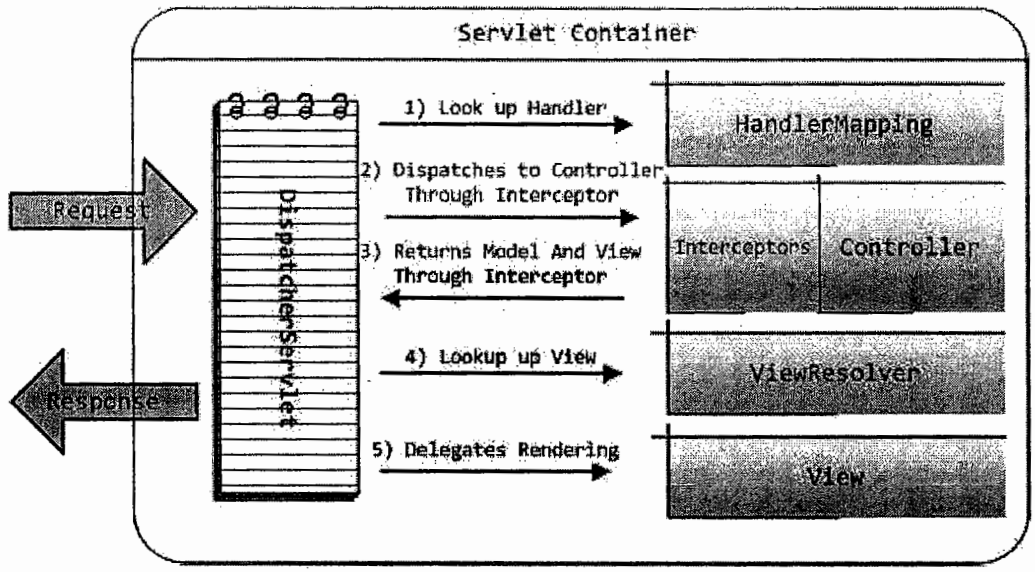
5) Locale and Theme resolvers: - spring provides jsp tag library that provides support for features such as data binding and themes.

8.2 Dispatcher Servlet

Spring Web MVC framework is designed around a DispatcherServlet that dispatches request to handlers, configurable handler mappings, view resolvers and locale and theme resolvers. Handler is the controller class in spring that will process the request and performs business logic to display the next view to the user. Handler mappings will allows you to map the incoming request to a handler. View resolvers will resolve for a view name, the physical view that should be rendered.

Spring Web MVC framework is, like any other web mvc frameworks, request-driven designed around a central servlet which is the DispatcherServlet. DispatcherServlet will listens for the incoming request and dispatches to handler, along with offers other functionalities facilitating the development of web application. In addition it completely integrates with IOC container making it to use every other feature of spring framework.

The typical request handling flow of the spring DispatcherServlet is shown below.



Dispatcher Servlet Request Handling Flow

Few points about Dispatcher Servlet as follows

1) DispatcherServlet acts as a FrontController – entry point for all the spring MVC requests
2) Loads XmlWebApplicationContext
3) Controls workflow and mediates between various MVC components
4) Loads sensible default components if none are configured

As every other servlet, even the DispatcherServlet has to be configured in the web.xml mapped to an URL to handle the incoming request, the mapping declaration is shown below.

web.xml

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

In the above configuration, all request's ending .htm will be handled by the DispatcherServlet. Each DispatcherServlet in spring has its own WebApplicationContext, a typical web application contains two types of components 1) Web components (controllers, command objects, view resolvers and views etc.) and 2) Business components (which performs business logic like Delegate, Service, Dao etc.), So the context that is getting created by DispatcherServlet will contains Web Components and Business components are declared separately which is going to be discussed shortly.

Spring while initializing the DispatcherServlet, will looks for the configuration file [servletname]-servlet.xml for the Web component bean declarations. Based on the above configuration, the file name it looks for is dispatcher-servlet.xml. Now loads all the bean declarations and creates a WebApplicationContainer with those beans.

Note: - You can customize the [servlet-name]-servlet.xml pattern by configuring a init-param at the DispatcherServlet where the param-name is namespace, value is the name of the configuration file shown below.

```
<servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <init-param>
                <param-name>namespace</param-name>
                <param-value>ui</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
</servlet>
```

DispatcherServlet configuration with namespace

With the above configuration, DispatcherServlet will look for the file ui.xml under WEB-INF directory.

8.3 Configuring ApplicationContext

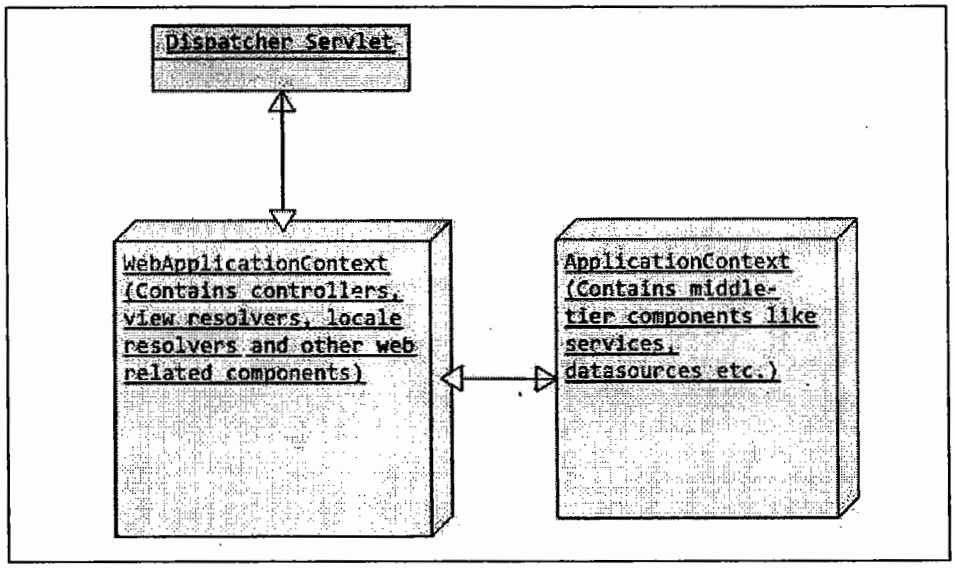
Apart from the WebApplicationContext created by the DispatcherServlet, you can configure one more container ApplicationContext which holds Business component bean declarations. In order to create this, you need to configure a Listener as shown below.

web.xml

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/root-application-context.xml</param-value>
</context-param>
<listener>
        <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

With the above configuration, the ContentLoaderListener will reads the context-param whose param-name is contextConfigLocation, and reads the value representing the configuration file path and creates the IOC container which contains the business component classes as beans.

Note: - The ApplicationContext created by ContextLoaderListener will act as parent factory to the WebApplicationContext which means the Web components can reference your business components whereas business components cannot refer your web components. If you recollect this is the concept of Nested Bean Factories, which we discussed in Spring Core. The main advantage of having two Containers is the web components and declarations are separated from business components, so that if you want to quit from Spring Web MVC, you need to remove the DispatcherServlet configuration and your Business components still can be injected into other mvc framework components through spring integration project.



Diagrammatic representation of WebApplicationContext referencing ApplicationContext

In order to handle the Web application requests, we need to use special components that are provided by mvc framework which are instantiated and mediated by DispatcherServlet (in WebApplicationComponents). In the following sections let's understand the components that must be written or configured to handle the MVC flow.

8.4 Controller

In a JEE application all the requests from the client/view will be handled by the servlet. In case of Struts, "Action" class will handle the request and performs the processing. Similarly in spring "Controller" will acts as a Servlet/Action in handling the request and displaying the view to the user.

In case of struts if we see there are several types of Action classes like Action, DispatchAction, LookupDispatchAction etc. But if you observe carefully all these actions classes will take ActionForm as parameter, this itself tells that always struts assumes the request always comes from a form submission.

But in a web application there are several ways of sending the request to the application. Not only using form submission, we can click on a hyperlink or by entering the url directly in the address bar of the browser etc. If you observe apart from form submission other two ways will not send any data as part of the request. If there is not data being sent as part of the request then why I should take ActionForm as parameter to the Action class or what I should do with this?

This tells struts is poor in addressing several usecases related to web application. Unlike struts, spring has provided a rich set of controller classes which addresses every other usecase in a typical web application.

This list is un-ending and for our reference we list here few of them.

1) Controller
2) ParameterizableViewController
3) UrlFileNameViewController
4) AbstractController
5) AbstractCommandController
6) SimpleFormController
7) AbstractWizardController

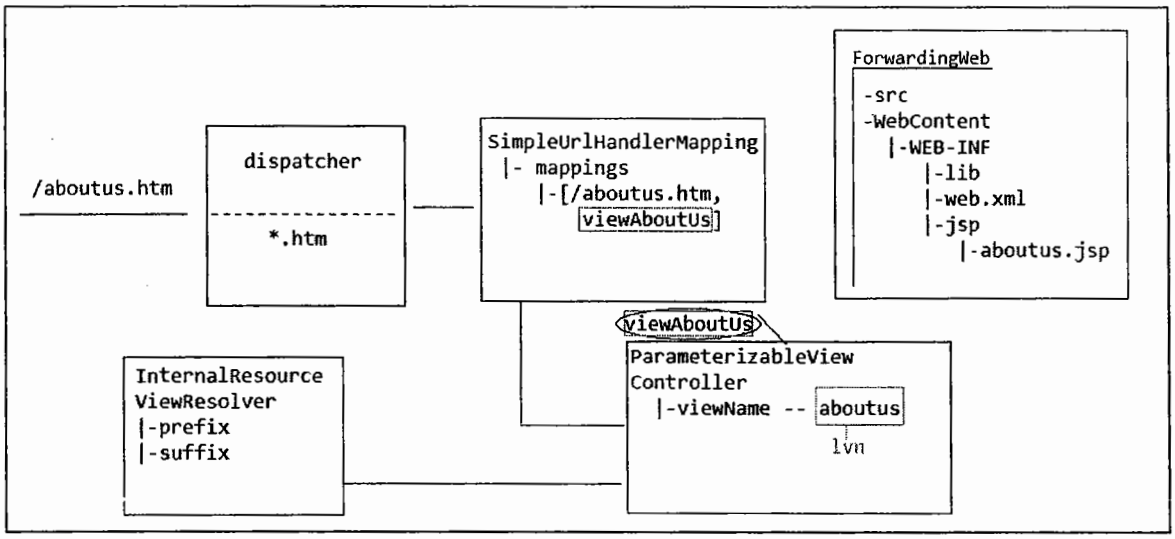
Spring basic controller architecture is org.springframework.web.servlet.mvc.Controller interface. The Controller interface has a single method as shown below.

```
ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse
response) throws Exception;
```

This method is responsible for handling the request and returns an appropriate model and view. The ModelAndView and Controller are basic concepts of spring MVC, based on this rest of the controllers has been designed.

8.4.1 ParameterizableViewController

This is also called as forwarding controller; it is used for forwarding an incoming request to a jsp page based on the parameter we configured, without performing any processing. Here is the example depicts the same.



aboutus.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
            <title>AboutUs</title>
      </head>
      <body>
            Nothing about us....
      </body>
</html>
```

dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
		<property name="mappings">
			<props>
				<prop key="/contactus.htm">
					contactuscontroller
				</prop>
			</props>
		</property>
	</bean>

	<bean id="contactuscontroller"
class="org.springframework.web.servlet.mvc.ParameterizableViewController">
		<property name="viewName" value="contactus"/>
	</bean>

	<bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="prefix" value="/WEB-INF/jsp/"/>
		<property name="suffix" value=".jsp"/>
	</bean>
</beans>
```

root-application-context.xml

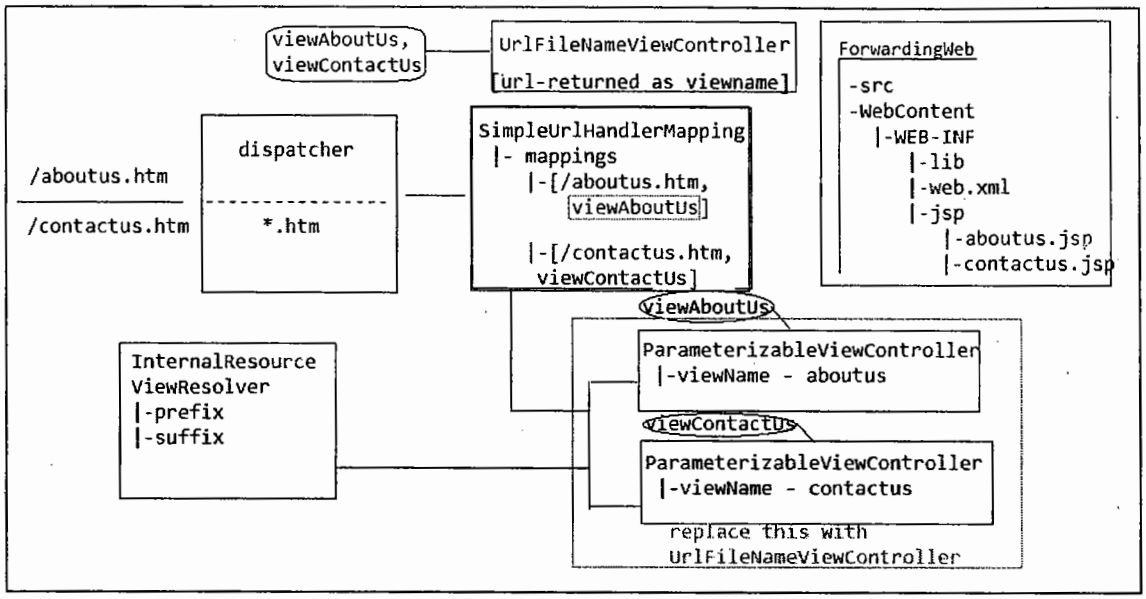
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<!--No business logic beans -->
</beans>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	id="WebApp_ID" version="2.5">
	<display-name>ForwardWeb</display-name>
	<listener>
		<listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
	</listener>
	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/root-application-context.xml</param-value>
	</context-param>
	<servlet>
		<servlet-name>dispatcher</servlet-name>
		<servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>2</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>dispatcher</servlet-name>
		<url-pattern>*.htm</url-pattern>
	</servlet-mapping>
</web-app>
```

8.4.1 UrlFilenameViewController

It is also similar to ParameterizableViewController used for forwarding the request to a jsp page. But here we don't configure any parameter instead it takes the incoming url itself as a view name to display the view. As it uses convention over configuration, we don't need to configure different controllers to forward the request to multiple views rather one UrlFilenameViewController is enough to forward the request to any number of jsp's as shown below.



aboutus.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
            <title>AboutUs</title>
      </head>
      <body>
            Nothing about us....
      </body>
</html>
```

contactus.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
            <title>Contact Us</title>
      </head>
      <body>
            Contact Us @ - 1-232-222-2242
      </body>
</html>
```

dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

      <bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
            <property name="mappings">
                  <props>
                        <prop key="/contactus.htm">
                              fc
                        </prop>
                        <prop key="/aboutus.htm">
                              fc
                        </prop>
                  </props>
            </property>
      </bean>

      <bean id="fc"
class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

      <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
            <property name="prefix" value="/WEB-INF/jsp/"/>
            <property name="suffix" value=".jsp"/>
      </bean>
</beans>
```

root-application-context.xml

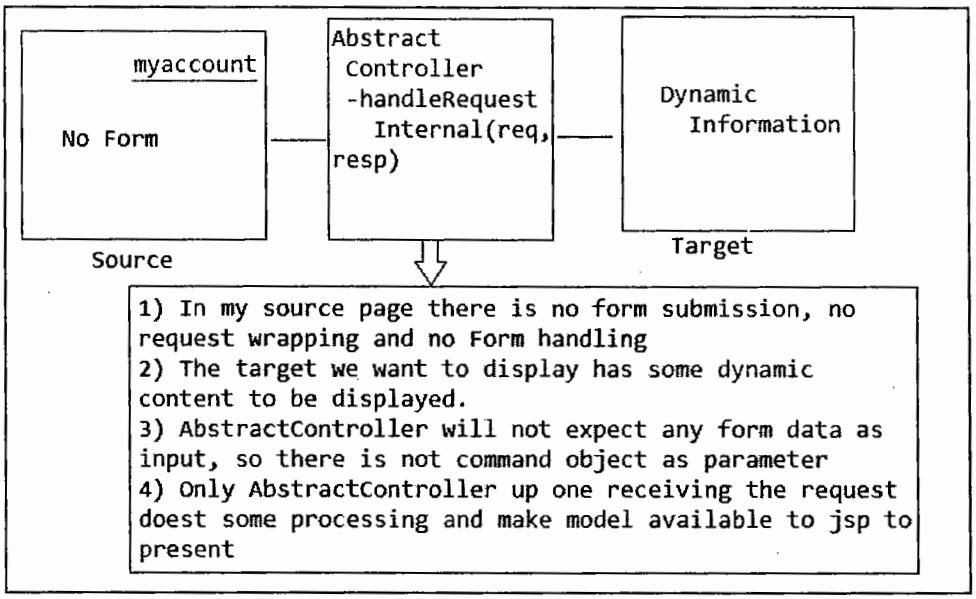
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<!-- no business tier beans, so empty-->
</beans>
```

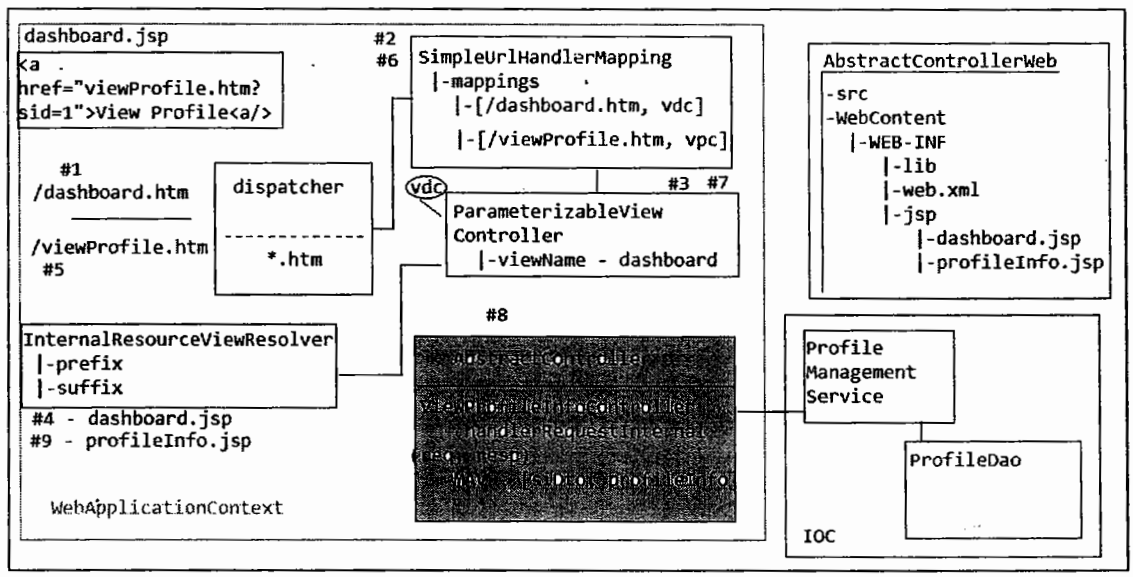
web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	id="WebApp_ID" version="2.5">
	<display-name>ForwardWeb</display-name>
	<listener>
		<listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
	</listener>
	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/root-application-context.xml</param-value>
	</context-param>
	<servlet>
		<servlet-name>dispatcher</servlet-name>
		<servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>2</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>dispatcher</servlet-name>
		<url-pattern>*.htm</url-pattern>
	</servlet-mapping>
</web-app>
```

8.4.1 AbstractController

When we don't have form submission, we just only need to perform processing and should display dynamic data in the jsp as part of that request then we should go for AbstractController.





home.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3:org/TR/html4/loose.dtd">
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
            <title>Home</title>
      </head>
      <body>
            <a href="liststuds.htm">List students</a>
      </body>
</html>
```

liststuds.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
      <head>
            <meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
            <title>List Students</title>
      </head>
      <body>
            <table>
                  <tr>
                        <th>Student Id</th>
                        <th>Name</th>
                  </tr>
                  <c:forEach items="${students}" var="stud">
                        <tr>
                              <td><c:out value="${stud.id}"/></td>
                              <td><c:out value="${stud.name}"/></td>
                        </tr>
                  </c:forEach>
            </table>
      </body>
</html>
```

```
package com.acw.beans;

public class StudentInfo {
	private int id;
	private String name;

	//setters & getters
}
```

```
package com.acw.bo;

import java.io.Serializable;

public class StudentBO implements Serializable {
	private int id;
	private String name;

	//setter & getter
}
```

```
package com.acw.controller;

import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.acw.beans.StudentInfo;
import com.acw.service.StudentService;

public class ListStudentsController extends AbstractController {
	private StudentService studentService;

	@Override
	protected ModelAndView handleRequestInternal(HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		ModelAndView mav = null;
		List<StudentInfo> students = null;
		students = studentService.getAllStudents();
		mav = new ModelAndView();
		mav.addObject("students", students);
		mav.setViewName("liststuds");
		return mav;
	}
	public void setStudentService(StudentService studentService) {
		this.studentService = studentService;
	}
}
```

```
package com.acw.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import com.acw.bo.StudentBO;

public class StudentDao {
	private final String SQL_GET_ALL_STUDENTS = "SELECT STUDENT_ID, NAME
FROM STUDENT";
	private JdbcTemplate jdbcTemplate;

	public StudentDao(JdbcTemplate jdbcTemplate) {
		this.jdbcTemplate = jdbcTemplate;
	}

	public List<StudentBO> getAllStudents() {
		return jdbcTemplate.query(SQL_GET_ALL_STUDENTS, new
StudentRowMapper());
	}

	private final class StudentRowMapper implements RowMapper<StudentBO> {
		@Override
		public StudentBO mapRow(ResultSet rs, int rowNum) throws
SQLException {
			return new StudentBO(rs.getInt("STUDENT_ID"),
rs.getString("NAME"));
		}
	}
}
```

```
package com.acw.service;

import java.util.ArrayList;
import java.util.List;

import com.acw.beans.StudentInfo;
import com.acw.bo.StudentBO;
import com.acw.dao.StudentDao;

public class StudentService {
	private StudentDao studentDao;

	public List<StudentInfo> getAllStudents() {
		List<StudentInfo> studInfos = null;
		List<StudentBO> studBos = null;

		studBos = studentDao.getAllStudents();
		if (studBos != null && studBos.size() > 0) {
			studInfos = new ArrayList<StudentInfo>();
			for (StudentBO stud : studBos) {
				StudentInfo si = new StudentInfo();
				si.setId(stud.getId());
				si.setName(stud.getName());
				studInfos.add(si);
			}
		}
		return studInfos;
	}
	public void setStudentDao(StudentDao studentDao) {
		this.studentDao = studentDao;
	}
}
```

application-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
	<import resource="persistence-beans.xml" />
	<import resource="service-beans.xml" />
</beans>
```

dispatcher-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
		<property name="mappings">
			<props>
				<prop key="/home.htm">homeController</prop>
				<prop key="/liststuds.htm">listStudController</prop>
			</props>
		</property>
	</bean>

	<bean id="homeController"

	class="org.springframework.web.servlet.mvc.ParameterizableViewController">
		<property name="viewName" value="home" />
	</bean>

	<bean id="listStudController"
class="com.acw.controller.ListStudentsController">
		<property name="studentService">
			<ref parent="studentService"/>
		</property>
	</bean>

	<bean

	class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name="prefix" value="/WEB-INF/jsp/" />
		<property name="suffix" value=".jsp" />
	</bean>
</beans>
```

persistence-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="dataSource"
		class="org.springframework.jdbc.datasource.DriverManagerDataSource">
		<property name="driverClassName"
value="oracle.jdbc.driver.OracleDriver" />
		<property name="url" value="jdbc:oracle:thin:@localhost:1521:xe" />
		<property name="username" value="hr" />
		<property name="password" value="hr" />
	</bean>

	<bean id="jdbcTemplate"
class="org.springframework.jdbc.core.JdbcTemplate">
		<constructor-arg ref="dataSource" />
	</bean>

	<bean id="studentDao" class="com.acw.dao.StudentDao">
		<constructor-arg ref="jdbcTemplate" />
	</bean>

</beans>
```

service-beans.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

	<bean id="studentService" class="com.acw.service.StudentService">
		<property name="studentDao" ref="studentDao"/>
	</bean>

</beans>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
	id="WebApp_ID" version="2.5">
	<display-name>ACWeb</display-name>
	<listener>
		<listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
	</listener>
	<context-param>
		<param-name>contextConfigLocation</param-name>
		<param-value>/WEB-INF/application-context.xml</param-value>
	</context-param>
	<servlet>
		<servlet-name>dispatcher</servlet-name>
		<servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
		<load-on-startup>2</load-on-startup>
	</servlet>
	<servlet-mapping>
		<servlet-name>dispatcher</servlet-name>
		<url-pattern>*.htm</url-pattern>
	</servlet-mapping>
</web-app>
```

8.4.2 Validator

In order to create a validator you need to write a class which implements Validator interface and override two methods supports and validate method. Unlike struts, the validator component is the separate component, which will validate the command data, and you can attach multiple validators to a form. In order to ensure the validator is being invoked on the right command object, the supports method will perform this check, and returns Boolean. If the supports method returns true, then the validate method will be invoked on the command object otherwise will ignore.

dispatcher-servlet.xml

```
<bean id="studentValidator" class="com.sfw.validator.StudentValidator"/>

<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
	<property name="basename" value="messages"/>
</bean>
```

messages.properties *(Should be placed under src directory or its sub-directories)*

```
StudentId.Blank=Student Id is mandatory
StudentName.Blank=Student name is required
```

StudentValidator.java

```
package com.sfw.validator;

import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.sfw.command.Student;

public class StudentValidator implements Validator {

	@Override
	public boolean supports(Class<?> classType) {
		return Student.class == classType;
	}

	@Override
	public void validate(Object command, Errors errors) {
		Student sCommand = (Student) command;

		if(sCommand.getId() <= 0) {
			errors.reject("StudentId.Blank");
		}

		if(sCommand.getName() == null || sCommand.getName().equals("")) {
			errors.reject("StudentName.Blank");
		}
	}
}
```

If the validator returns any errors, the relevant messages for the keys will be picked from the messageSource bean configured above and redisplays the form back to the user.

In order to re-display the form back to the user with user entered values, we need to write the JSP forms with spring form tag libraries. These libraries are similar to struts form tag libraries which allows you to map the user interface fields to the command

attributes and help in displaying the form back up on errors with user entered values and error messages.

insertStudent.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
<%@ taglib uri="http://www.springframework.org/tags/form" prefix="form" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
	<head>
		<meta http-equiv="Content-Type" content="text/html; charset=ISO-
8859-1">
		<title>Insert Student</title>
	</head>
	<body>
		<form:form method="POST">
			<table>
				<tr>
					<td colspan="2">
						<p style="color: red">
							<form:errors/>
						</p>
					</td>
				</tr>
				<tr>
					<td>Id</td>
					<td><form:input path="id"/></td>
				</tr>
				<tr>
					<td>Name</td>
					<td><form:input path="name"/></td>
				</tr>
				<tr>
					<td>
						<input type="submit" value="Insert"/>
					</td>
				</tr>
			</table>
```

```
            </form:form>
        </body>
</html>
```

8.5 Handler Mappings

Using handler mappings you can map an incoming web requests to appropriate handler. You can configure more than one handler mappings, based on the order (priority) will maps the request to a Handler (Controller). Basically it works in this way, the DispatcherServlet once receives the request will handover it to HandlerMapping to let it inspect the request and come up with HandlerExecutionChain. Then the DispatcherServlet will execute the handler in the chain.

HandlerMapping can optionally contain HandlerInterceptors, the concept of HandlerInterceptor will be discusses later, but the HandlerExcecutionChain returned will contain list of HandlerInterceptors and Handler for execution.

Two of the most commonly used Handler Mappings's in spring are discussed in the below section, but they both extend from a class AbstractHandlerMapping and share the properties as below.

1) Interceptors: - the list of interceptors to use. HandlerInterceptor will perform pre and post processing of request.

2) Order:- If multiple handler mappings has been configured then spring will sort all the handler mappings in the chain and will apply the first one in the chain.

8.5.1 BeanNameUrlHandlerMapping

This will maps the incoming request URL to the beanName to identify the controller for execution. Let's say for example we have an appropriate form controller which will handle the insert of an employee. When an incoming request likes /insertEmployee.htm come to the DispatcherServlet, it will forwards it to configured BeanNameUrlHanlderMapping to map it to the Controller, it will pick the controller bean whose name is "/insertEmployee.htm" as show below.

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean  name="/insertEmployee.htm"
class="com.emp.controller.InsertEmployeeController">
        <property name="formView" value="insertEmployee"/>
```

```
        <property name="successView" value="showEmployeeDetail"/>
        <property name="commandClass" value="com.emp.command.Employee"/>
</bean>
```

If you don't configure any handler mapping, by default spring will provide BeanNameUrlHandlerMapping as the default one.

8.5.1 SimpleUrlHandlerMapping

Much more powerful and most used handler mapping is SimpleUrlHandlerMapping. In this you will configure the mapping between the incoming requests to a handler in the configuration file. It allows you to map your request using ant-style path matching capabilities, the below configuration will lets you point to the InsertEmployeeController by using the url insert.htm.

```
<bean id="handlerMapping"
class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
                <props>
                        <prop key="/insert.htm">insertEmployeeController</prop>
                </props>
        </property>
</bean>

<bean id="insertEmployeeController"
class="com.emp.controller.InsertEmployeeController">
        <!—left for clarity →
</bean>
```

8.6 Handler Interceptors

In the HandlerMappings section we come across the HandlerIterceptors, which means the HandlerMappings along with mapping URL to a Handler will also configured with HandlerInterceptors. The main purpose of HandlerInterceptors is to handle pre and post processing of incoming request; this is similar to the concept of filters in J2EE. But the main difference between filters and interceptors is, filters are applied to all the requests of the web application, whereas interceptors are applied to certain group of handlers. Secondly you have three states of interceptor execution like before processing request, before rendering the view and after the view has rendered to the user.

In order to configure a class as HandlerInterceptor, you need to write a class which implements HandlerInterceptor. Instead you can override your class from HandlerInterceptorAdapter and decide to override the method of your choice as shown below.

StoreTimeInterceptor.java

```
package com.sw.handlerinterceptor;

import java.util.Calendar;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class StoreTimeInterceptor extends HandlerInterceptorAdapter {

	@Override
	public boolean preHandle(HttpServletRequest request,
			HttpServletResponse response, Object handler) throws Exception {
		Calendar c = Calendar.getInstance();
		if (c.get(Calendar.HOUR) >= 5) {
			response.sendRedirect("timeout.jsp");
		}
		return true;
	}
}
```

dispatcher-servlet.xml

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
	<property name="mappings">
		<props>
			<prop key="/search.htm">SearchController</prop>
		</props>
	</property>
	<property name="interceptors">
		<list>
			<ref local="timeoutInterceptor"/>
```

```
            </list>
        </property>
</bean>

<bean id="timeoutInterceptor"
class="com.sw.handlerinterceptor.StoreTimeInterceptor"/>
```

8.7 ViewResolver

Every spring MVC framework will one way or in another provides a mechanism to dispatch the request to render a view. In Spring MVC it ViewResolvers are used for mapping and dispatching the request to an appropriate view.

When discussing about the controller, we observed that all the controllers will returns an ModelAndView as response after execution, the ModelAndView contains the logical view name that will be returned to the DispatcherServlet for rendering, the DispatcherServlet will maps the logical viewName to a View by using ViewResolver.

The two fundamentals of spring view handling is org.springframework.web.servlet.ViewResolver and org.springframework.web.servlet.View interfaces one will maps the view name to view and other prepares the request and handovers the request to render the View.

Spring has a rich set of pre-defined ViewResolvers provided out of box, which you can use straight forward.

ViewResolver	Description
XmlViewResolver	An implementation of ViewResolver interface which accepts the XML file as View Configurations to resolve the views.
ResourceBundleViewResolver	Instead of using XML file for View configurations, you will use the properties file, the key is the logical view name and the value is the resource you want to render
UrlBasedViewResolver	A simple implementation of the ViewResolver interface which maps

	directly the symbolic view names to the URL's without explicit mapping definition
InternalResourceViewResolver	This is a sub class of UrlBasedViewResolver which supports direct rendering of InternalResourceViews like Servlets and JSP's. and other view Sub classes like JSTLView and TilesView

An example of the ViewResolver configurations for each one is discussed in the following section.

8.7.1 UrlBasedViewResolver

This ViewResolver will maps the view name to a URL and handover's it to the DispatcherServlet to render the view.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
</bean>
```

8.7.1 ResourceBundleViewResolver

When working with different view technologies in a web application, you can use ResourceBundleViewResolver.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
        <property name="baseName" value="views"/>
</bean>
```

With the above declaration it means all the view definitions have been declared in the file views.properties under your classpath (src directory). The ResourceBundleViewResolver inspects the ResourceBundle and identifies the baseName for each view it is supposed to resolve.

The mapping information in the properties file would be like [viewname].class and [viewname].url, where [viewname].class represents the class that acts as view class to render the view and the [viewname].url represents the path to the view located in the application. Sample views.properties has been given below.

views.properties

```
insertStudent.class=org.springframework.web.servlet.view.JstlView
insertStudent.url=/WEB-INF/jsp/insertStudent.jsp
```

8.7.1 XmlViewResolver

Instead of providing the mappings in a properties file here we declare an xml file which contains bean definitions of the views that has to be render. The following configuration shows the same.

```
<bean id="viewResolver"
class="org.springframework.web.servlet.view.XmlViewResolver">
        <property name="location" value="/WEB-INF/views.xml"/>
</bean>
```

In the above configuration, we indicated the view declarations are declared in views.xml

views.xml

```
<bean id="home" class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/home.jsp"/>
</bean>
```

Spring ORM

9 Spring ORM (Object Relational Mapping)

Spring ORM framework allows you to integrate with Hibernate, Java Persistence API (JPA), Java Data Objects (JDO) and iBATIS for resource management and data access object (DAO) implementations and other transaction strategies.

Benefits of using Spring Framework to create your ORM DAOs include:

a) Easier testing: Spring IOC approach allows you to swap easily the implementations and configuration locations of Hibernate SessionFactory instances etc, so that you can point your configurations to various environments without modifiying the source code.

b) Common data acess exception: Spring instead of exposing ORM specific checked exceptions to the top level tier's of the application, it will wraps the technology specific exceptions to a common runtime DataAccessException hierarchy.

c) Integrated Transaction management: Instead of dealing with ORM technology related transactional code, it allows you to declaratively manage the transactionality using AOP Declarative transaction management tags <tx:advice> or annotation driven @Transactional annotation.

9.1 Integrating with Hibernate

In order to use spring ORM with hibernate, you need to declare your Hibernate Language objects (HLO) using declarative mapping.hbm files are annotate your classes with hibernate annotations. In order to perform the operations using these classes, you need to declare an HibernateSessionFactory and then injects it into HibernateTemplate, recall the concept of Template JDBCTemplate which allows you to perform the JDBC operations using Template approach.

Sample code is shown below.

EmployeeHLO.java

```
package com.ew.hlo;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "TBLEMP")
public class EmployeeHLO {
	private long id;
	private String name;
	private double salary;

	@Id
	@Column(name="EMP_ID")
	public long getId() {
		return id;
	}

	public void setId(long id) {
		this.id = id;
	}

	@Column(name="EMP_NM")
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
```

```
	}

	@Column(name="SALARY")
	public double getSalary() {
		return salary;
	}

	public void setSalary(double salary) {
		this.salary = salary;
	}
}
```

EmployeeDao.java

```
package com.ew.dao;

import org.springframework.orm.hibernate3.HibernateTemplate;

import com.ew.hlo.EmployeeHLO;

public class EmployeeDao {
	private HibernateTemplate hibernateTemplate;

	public EmployeeDao(HibernateTemplate hibernateTemplate) {
		this.hibernateTemplate = hibernateTemplate;
	}

	public void insert(EmployeeHLO employeeHLO) {
		hibernateTemplate.save(employeeHLO);
	}
}
```

EmployeeService.java

```
package com.ew.service;

import com.ew.command.Employee;
import com.ew.dao.EmployeeDao;
import com.ew.hlo.EmployeeHLO;

public class EmployeeService {
	private EmployeeDao employeeDao;
```

```
    public void setEmployeeDao(EmployeeDao employeeDao) {
        this.employeeDao = employeeDao;
    }

    public void insert(Employee employee) {
        EmployeeHLO employeeHLO = new EmployeeHLO();
        employeeHLO.setId(employee.getId());
        employeeHLO.setName(employee.getName());
        employeeHLO.setSalary(employee.getSalary());
        employeeDao.insert(employeeHLO);
    }
}
```

persistence-beans.xml

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
value="com.microsoft.sqlserver.jdbc.SQLServerDriver" />
    <property name="url"
        value="jdbc:sqlserver://localhost:1433;databaseName=spdb" />
    <property name="username" value="sa" />
    <property name="password" value="welcome1" />
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactor
yBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.ew.hlo.EmployeeHLO</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop
key="hibernate.dialect">org.hibernate.dialect.SQLServerDialect</prop>
            <prop key="show_sql">true</prop>
        </props>
    </property>
</bean>

<bean id="hibernateTemplate"
class="org.springframework.orm.hibernate3.HibernateTemplate">
```

```
        <property name="sessionFactory" ref="sessionFactory" />
</bean>
<bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="employeeDao" class="com.ew.dao.EmployeeDao">
        <constructor-arg ref="hibernateTemplate"/>
</bean>
```

aop-beans.xml

```
<aop:config>
        <aop:pointcut expression="execution(* com.ew.service.EmployeeService.*(..))"
                id="empServicePC" />
        <aop:advisor advice-ref="empTxAdvice" pointcut-ref="empServicePC" />
</aop:config>
<tx:advice id="empTxAdvice" transaction-manager="transactionManager">
        <tx:attributes>
                <tx:method name="insert*" read-only="false" propagation="REQUIRED"
/>
        </tx:attributes>
</tx:advice>
```