XML

1 XML

XML stands for extensible markup language. Markup language is the language using which you can build other languages like, HTML, XML.

XML is defined and governed by W3Org. The first and final version of XML is XML 1.0. XML is the document which represents data. Unlike C, C++, Java etc. XML is not a programing language. It is the defacto standard for carrying information between computer systems.

Every language has keywords. If you take example as C, it has keywords like (if, for, while, do, break, continue etc.) but when it comes to XML there are no keywords or reserved words. What you write will become the element of XML language.

1.1 XML Element

Element in an XML is written in angular braces for e.g. <beans>. In XML there are two types of elements as follows

1) Start Element/Opening Tag: - Start element is the element which is written in <elementname> indicating the start of a block.
2) End Element/End Tag: - End element is the element which is written in </elementname> indicating the end of a block.

As everything is written in terms of start and end elements, XML is said to be more structured in nature. An XML Element may contain content or may contain other elements under it. So, XML elements which contain other elements in it are called as Compound elements or XML Containers.

1.2 XML Attribute

If we want to have supplementary information attach to an element, instead of having it as content or another element, we can write it as an Attribute of the element.

Example:-

```
<bean name="pilot" class="Pilot">
  <constructor-arg ref="plane"/>
</bean>
```

In the above example "bean" is an element which contains two attributes name and class which acts as an supplementary information. Constructor-arg is a sub-element under "bean" element.

1.3 Well-formness

As how any programing language has syntax in writing its code, Well-formness of XML document talks about how to write an XML document. Well-formness indicates the readability nature of an XML document. In other way if an XML document is said to be well-formed then it is readable in nature.

Following are the rules that describe the Well-formness of an XML Document.

1) Every XML document must start with **PROLOG**: - prolog stands for processing instruction, and typically used for understanding about the version of XML used and the data encoding used.

   Example: - <?xml version="1.0" encoding="utf-8"?>

2) Root Element: - XML document must contain a root element, and should be the only one root element. All the other elements should be the children of root element.
3) Level constraint: - Every start element must have an end element and the level at which you open a start element, the same level you need to close your end element as well.

Example:-

```
<?xml version="1.0" encoding="utf-8"?>
<student>
        <info>
                <rollno>42</rollno>
                <name>John</info>  (-- info is closed in-correctly --)
        </name>
<student>
```

If any XML is said to follow the above defined rules then it is termed as well-formed.

1.4 XML Usage

An XML document is used in two scenarios

1) **Used for transferring information**:- As said earlier XML is a document representing data and is used for carrying information between two computer systems in an Interoperable manner.
2) **Configurations**: - In J2EE world every component/resource you develop like Servlet or EJB, it has to be deployed into a Web Server. For e.g. in a Servlet application, the path with which the servlet has to be accessible should be specified to the Servlet container so that it can map the incoming request to the Servlet. This is done in a web.xml. When we provide the configuration

information in a XML file the main advantage is, the same xml document can be used in different platforms without changing anything.

1.5 Validatity

Every XML in-order to parse should be well-formed in nature. As said earlier well-formness of an XML document indicates whether it is readable or not, it doesn't talks about wheather the data contained in it is valid or not.

Validity of the XML document would be defined by the application which is going to process your XML document. Let's consider a scenario as follows.

You want to get driving license. In order to get a driving license you need to follow certain process like filling the RTA forms and signing them and submitting to the RTA department.

Instead of this can you write your own format of letter requesting the driving license from an RTA department, which seems to be not relevant because driving license is something that would be issued by the RTA department. So, the whole and sole authority of defining what should a person has to provide data to get a driving license will lies in the hands of RTA department rather than you.

In the same way when an application is going to process your xml document, the authority of defining what should be there as part of that xml is lies in the hands of the application which is going to process your document.

For example let's consider the below xml fragment.

Example:- PurchaseOrder xml document

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
 <orderItems>
        <item>
                <itemCode>IC323</itemCode>
                <quantity>24</quantity>
        </item>
        <item>
                <itemCode>IC324</itemCode>
                <quantity>abc</quantity>
        </item>
 </orderItems>
<purchaseOrder>
```

In the above xml even though it confirmes to all the well-formness rules, it cannot be used for business transaction, as the 2nd <quantity> element carries the data as "abc" which doesn't makes any sense.

So, in order to check for data validity we need to define the validation criteria of an XML document in either a DTD or XSD document.

2 DTD

DTD stands for Document Type Definition. It is the document which defines the structure of an XML document.

As we have two types of XML elements Simple and Compound elements we need to represent in DTD, my xml contains these elements which are of either simple or compound.

So, we need to understand the syntax of how to declare a simple element of an xml in dtd as well as compound element.

Syntax for Simple Element of an XML

<!Element elementname (#PCDATA)>

Here we declared that my xml should contain an element whose name is elementname which contains parse able character data.

Syntax for Compound elemtent of an XML

<!Element elementname (sub-elem1, sub-elem2...)

Here we declared I have an element "elementname" which contains sub elements under it sub-elem1, sub-elem2 etc.

For example the sample DTD document for an xml is shown below.

XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<purchaseOrder>
 <orderItems>
        <item>
                <itemCode>IC323</itemCode>
                <quantity>24</quantity>
        </item>
        <item>
                <itemCode>IC324</itemCode>
                <quantity>abc</quantity>
        </item>
 </orderItems>
<purchaseOrder>
```

For the above xml the DTD looks as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<!Element purchaseOrder (orderItems)>
<!Element orderItems (item+)>
<!Element item (itemCode, quantity)>
<!Element itemCode (#PCDATA)>
<!Element quantity (#PCDATA)>
```

In the above xml if you observe the orderItems element can contain any number of item elements in it, but atleast one item element must be there for an purchaseOrder. This is called occurance of an element under another element, to indicate this we use three symbols.

? – Represents the sub element under a parent element can appear zero or one-time (0/1).

+ - indicates the sub element must appear at least once and can repeat any number of times (1 – N)

* - indicates the sub element is optional and can repeat any number of times (0 - N)

You will mark the occurrence of an element under another element as follows.

<!Element elementname (sub-elem1 (?/+/*), sub-elem2(?/+/*)>

Leaving any element without any symbol indicates it is mandatory and at max can repeat only once.

2.1 Drawback with DTD's.

DTD are not typesafe, which means when we declare simple elements we indicate it should contain data of type (#PCDATA). #PCDATA means parsable character data means any data that is computer represented format. So it indicates an element can contain any type of data irrespective of whether it is int or float or string. You cannot impose stating my element should contain int type data or float. This is the limitation with DTD documents.

3 XML Schema Document (XSD)

XSD stands for XML schema document. XSD is also an XML document. It is owned by W3Org. The latest version of XSD is 1.1.

Even though we can use DTD's for defining the validity of an XML document, it is not type safe and is not flexibile. XSD is more powerful and is type strict in nature.

XSD document is used for defining the structure of an XML document; it declares elements and the type of data that elements carry.

As XSD is also an XML document so, it also starts with prolog and has one and only one root element. In case of XSD document the root element is <schema>. All the subsequent elements will be declared under this root element.

An XML contains two types of elements. A) Simple elements (these contains content as data) B) Compound or Container's (these contains sub-elements under it).

So, while writing an XSD documents we need to represent two types of elements simple or compound elements. The syntax's for representing a simple and compound elements of XML in a XSD document are shown below.

Syntax for representing simple elements of XML document

```
<xs:element name="elementname" type="datatype"/>
```

Syntax for representing compound element of XML document

In order to represent compound element of an XML document in an XSD, first we need to create a type declaration representing structure of that xml element. Then we need to declare element of that user-defined type, shown below.

```
<xs:complexType name="typeName">
  <xs:sequence> or <xs:all>
        <xs:element name=".." type=".."/>
        <xs:element name=".." type=".."/>
        <xs:element name=".." type=".."/>
  </xs:sequence> or </xs:all>
</xs:complexType>
```

Declaring a complex type is equal to declaring a class in java. In java we define user-defined data types using class declaration. When we declare a class it represents the structure of our data type, but it doesn't allocates any memory. Once a class has been declared you can create any number of objects of that class.

The same can be applied in case of complex type declaration in an XSD document. In order to create user-defined data type in XSD document you need to declare a

complex type, creating a complex indicates you just declared the class structure. Once you create your own data type, now you can define as many elements you need of that type. Let's say for example I declared a complex type whose name is AddressType as follows

```
<xs:complexType name="AddressType">
   <xs:sequence>
         <xs:element name="addressLine1" type="xs:string"/>
         <xs:element name="addressLine2" type="xs:string"/>
         <xs:element name="city" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Now the above fragment indicates you created your own user-defined data type whose name is "AddressType". Now you can declare any number of elements representing that type as shown below.

```
<xs:element name="myAddress" type="AddressType"/>
```

Let's take an XML document for which we will show how its XSD looks like

XML Document – Courier consignment information

```
<consignment>
   <id>C4242</id>
   <bookedby>durga</bookedby>
   <deliveredTo>Sriman</deliveredTo>
   <shippingAddress>
         <addressLine1>S.R Nagar</addressLine1>
         <addressLine2>Opp Chaitanya</addressLine2>
         <city>Hyderabad</city>
         <state>Andhra Pradesh</state>
         <zip>353</zip>
         <country>India</country>
   </shippingAddress>
</consignment>
```

XSD Document – Courier consignment information

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="consignment" type="consignmentType"/>
   <xs:complexType name="consignmentType">
         <xs:sequence>
               <xs:element name="id" type="xs:string"/>
               <xs:element name="bookedby" type="xs:string"/>
               <xs:element name="deliveredTo" type="xs:string"/>
               <xs:element name="shippingAddress"
type="shippingAddressType"/>
         </xs:sequence>
   </xs:complexType>
   <xs:complexType name="shippingAddressType">
         <xs:sequence>
               <xs:element name="addressLine1" type="xs:string">
               <xs:element name="addressLine2" type="xs:string">
               <xs:element name="city" type="xs:string">
               <xs:element name="state" type="xs:string">
               <xs:element name="zip" type="xs:int">
               <xs:element name="country" type="xs:string">
         </xs:sequence>
   </xs:complexType>
</xs:schema>
```

3.1 Sequence VS All

When declaring a complex type, following the <xs:complexType> tag we see a tag <xs:sequence> or <xs:all>. When we use <xs:sequence> under <xs:complexType> tag, what it indicates is all the elements that are declared in that complex type must appear in the same order in xml document. For example let's take a complex type declaration "ItemType" which uses <xs:sequence> tag in its declaration.

```
<xs:element name="item" type="ItemType"/>
<xs:complexType name="ItemType">
   <xs:sequence>
         <xs:element name="itemCode" type="xs:string"/>
         <xs:element name="quantity" type="xs:int"/>
   </xs:sequence>
</xs:complexType>
```

So while using the item element in the xml we need to declare the itemCode and quantity sub-elements under item in the same order as shown below.

```
<item>
   <itemCode>IC303</itemCode>
   <quantity>35</quantity>
</item>
```

When we use <xs:all>, under item element the itemCode and quantity may not appear in the same order. First quantity might appear then itemCode can present in the xml.

4 XSD Namespace

Every programming language one or in another way allows you to declare user-defined data types. Let's consider the case of "C" language it allows you to declare your own types using Structure. In case of "C++" Class declaration allows you to declare your own type, same in case of Java as well.

When it comes to XSD you can declare your element type using XSD complex type declaration.

As how any language allows you to create your own types, they allow you to resolve the naming collision between their types. Let's consider the case of Java; it allows you to declare your own types by class declarations. But when a programmer is given a choice of declaring their own types, language has to provide a means to resolve type names collision declared by several programmers. Java allows resolving those type naming conflicts using packages.

Packages are the means of grouping together the related types. It will allow you to uniquely identify a type by prefixing the package name. In the same way XSD also allows you to declare your own data type by using Complex Type declaration and in the same way it allows you to resolve the type naming conflicts by means of Namespace declaration.

XSD Namespaces has two faces,

1) Declaring the namespace in the XSD document using Targetnamespace declaration

2) Using the elements that are declared under a namespace in xml document.

4.1 XSD Targetnamespace

Targetnamespace declaration is similar to a package declaration in java. You will bind your classes to a package so, that while using them you will refer with fully qualified name. Similarly when you create a complexType or an Element you will bind them to a namespace, so that while referring them you need to use qName.

In order to declare a package we use package keyword followed by name of the package. To declare a namespace in XSD we need to use targetNameSpace attribute at the Schema level followed by targetnamespace label as shown below.

Example:-

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace="http://durgasoft.com/training/calendar/types">
  <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
  </xs:complexType>
</xs:schema>
```

The courseType by default is binded to the namespace http://durgasoft.com/training/calendar/types

So, while creating an element "course" of type courseType you should use the qName of the courseType rather simple name. (qName means namespace:element/type name).

But the namespace labels could be any of the characters in length, so if we prefix the entire namespace label to element/type names it would become tough to read. So, to avoid this problem XSD has introduced a concept called short name. Instead of referring to namespace labels you can define a short name for that namespace label using xmlns declaration at the <schema> level and you can use the short name as prefix instead of the complete namespace label as shown below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetnamespace=http://durgasoft.com/training/calendar/types
xmlns:dt="http://durgasoft.com/training/calendar/types">
  <xs:element name="course" type="dt:courseType"/>
  <xs:complexType name="courseType">
        <xs:element name="courseId" type="xs:string"/>
        <xs:element name="courseName" type="xs:string"/>
        <xs:element name="duration" type="xs:datetime"/>
  </xs:complexType>
</xs:schema>
```

In java we can have only one package declaration at a class level. In the same way we can have only one targetNamespace declaration at an XSD document.

4.2 Using elements from an xml namespace (xmlns)

- While writing an XML document, you need to link it to XSD to indicate it is following the structure defined in that XSD. In order to link an XML to an XSD we use an attribute "schemaLocation".

schemaLocation attribute declaration has two pieces of information. First part is representing he namespace from which you are using the elements. Second is the document which contains this namespace, shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

In the above xml we stated this xml is using xsd document courseInfo.xsd whose namespace is http://durgasoft.com/training/calendar/types

If you want to include two XSD documents in the xml then you need to declare in schemaLocation tag <namespace1> <schemalocation1> <namespace2> <schemalocation2>.
For example:-

```
<?xml version="1.0" encoded="utf-8"?>

<course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-Instance">

</course>
```

Now in the above xml we are using two XSD documents courseInfo.xsd and vacation.xsd. With this declaration we will not be able to find whether the course element is used from courseInfo.xsd or vacation.xsd. To indicate it we should prefix the namespace to the course element.

But the namespace labels can be arbitrary string of characters in any length. So, we need to define short name, so that we can prefix shortname while referring the elements as shown below.

```
<?xml version="1.0" encoded="utf-8"?>

<dc:course xsi:schemaLocation="http://durgasoft.com/training/calendar/types
file:///c:\folder1\folder2\courseInfo.xsd
http://durgasoft.com/training/vacation/types
file:///c:\folder1\folder2\vacation.xsd"
xmlns:xsi=http://www.w3.org/2001/XMLSchema-Instance
xmlns:dc="http://durgasoft.com/training/calendar/types">

</dc:course>
```

4.3 Difference between DTD and XSD

DTD	XSD
• DTD stands for document type definition • DTD's are not XML Type documents • DTD's are tough to learn as those are not XML documents. So, an XML programmer has to under new syntaxes in coding DTD • DTD's are not type safe, these will represents all the elements with data type as (#PCDATA). • DTD's don't allow you to create use-defined types.	• XSD stands for XML Schema Documents • XSD's are XML Type documents • As XSD's are XML type documents it is easy for any programmer to work with XSD. • XSD's are strictly typed, where the language has list of pre-defined data types in it. While declaring the element you need tell whether this element is of what type. • XSD's allows you to create user-defined data types using complexType declaration and using that type you can create any number of elements.

Spring Framework

5 Spring Framework

5.1 Introduction to Spring Framework

Spring Framework is an open source, light weight, loosely coupled Java Framework that allows you to build enterprise applications. Spring has been started in year 2003, it is not a one kind of framework which supports development of certain areas of J2EE, and it is a complete framework.

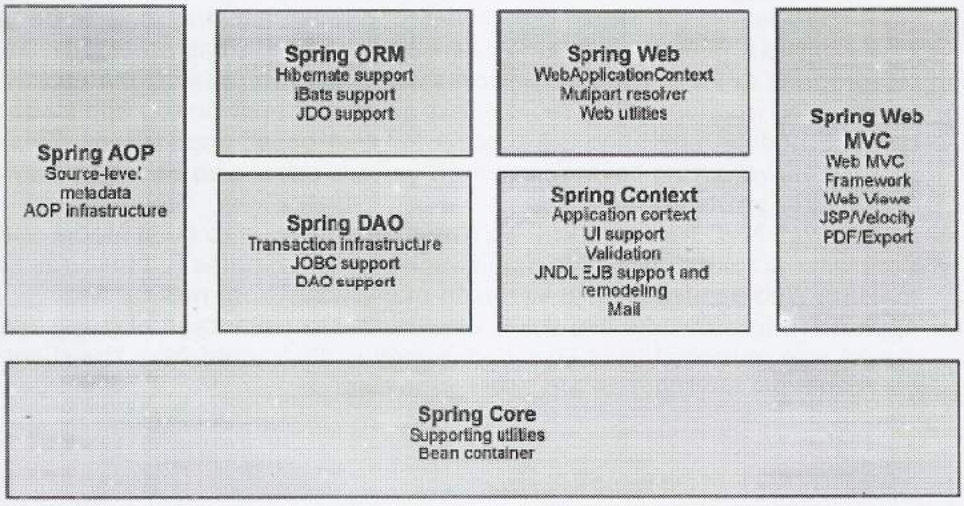
Using spring you can develop various types of application (in par with J2EE) like core java, remoting, web applications etc. So, you can think spring as a layer above your J2EE, but never spring is a replacement for the J2EE, it always compliments your J2EE in facilitation in quicker development.

Spring handles the infrastructure so you can focus on your application. Spring enables you to build applications from POJO (Plan old java objects), by providing enterprise services (like logging, transaction etc) non-invasively. This indicates your application business logic is free from spring specific classes or interfaces, at any point of time you can easily move out of spring without making any code changes.

Few of the examples of spring platform advantages

1) Pojo methods can work within database transaction without dealing with Transaction API's
2) Pojo methods can be exposed as remote methods without dealing with RMI API's
3) Pojo methods can behave like a message handler without dealing with JMS API's

Main advantage of spring is its layered architecture. Even its supports various types of application development, it has been designed keeping in mind to be light weight by dividing it into various modules as shown below.



Based on your application needs you can select and pick the module of your choice for development, even you can use combination of multiple modules. 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.

By the above we understood that spring core is the foundation module on top of which other modules has been designed, so it is important to understand 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.

6 Spring Core (Basic)

Spring core basic part is foundation which contains essential features of spring framework. The heart of spring framework is IOC (Inversion of Control), which manages dependencies between objects and their lifecycle. In order to benefit out of IOC spring recommends every application to follow certain design guidelines or principles (strategy pattern) to make their applications loosely coupled.

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

6.1 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
- Code should be open for extension and closed for modification
- Always design to Interfaces never code to Implementation

Let us examine all the three principles by taking an example

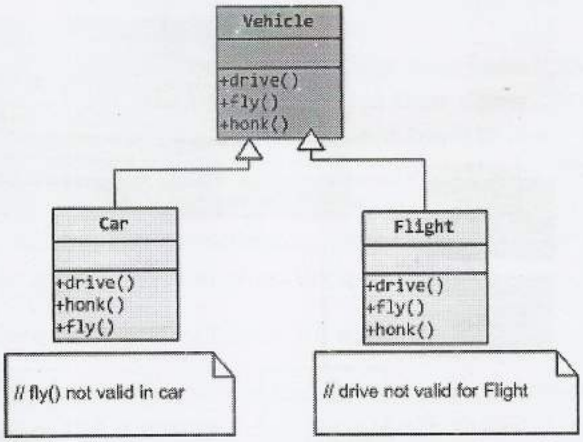
A core java application contains multiple classes. Each class contains logic for performing some kind of task. A pojo class which contains logic for performing operation is called pojo based business logic class.

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 once class from another class to get its functionalities.
2) Composition – A class will hold reference of other classes as attributes inside it.

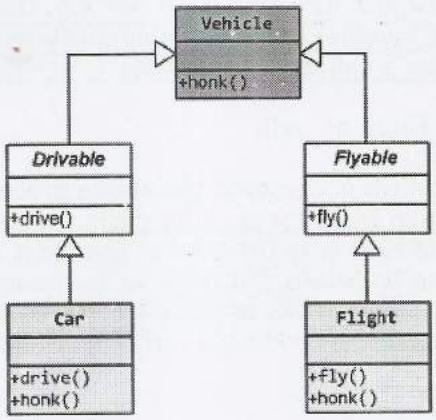
Using Inheritance you always express relation between two classes as IS-A relationship. But always not every relationship can be expressed in IS-A relationship.

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 shared across derived classes. Refer to the below example explaining the same.



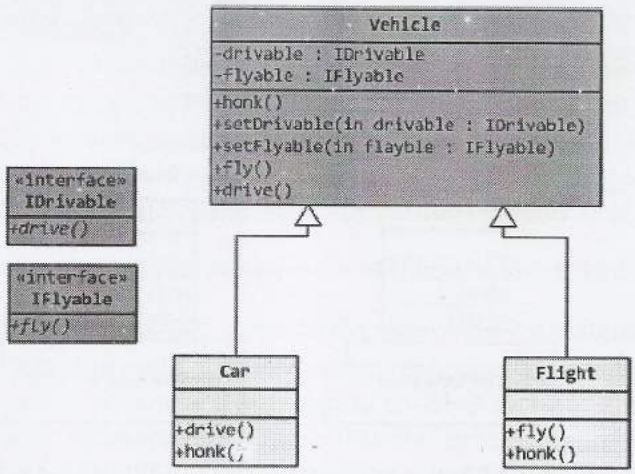
In the above diagram I have a vehicle which contain drive (), fly () and blow horn (honk ()). So I want to represent a Car and a Flight which is of Vehicle type. If I try to express this with Inheritance I will end up in implementing fly method in Car class as empty and drive method in flight class as 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 design, Vehicle is an abstract class which contains IDrivable and IFlyable as attributes. Along with this vehicle contains honk () as implemented method as it shares across all its derived classes.
Now Vehicle declares two protected methods drive and fly which in-turn calls the methods on drivable and flyable attributes.

Now when we create a Car, Car will override the drive method where it will sets the implementation Object IDrivable class (for e.g. DefaultDrivable is implementing from IDrivable) to the drivable attribute of super class through its setter and then makes a call to super.drive().

The same applies to Flight as well.

With the above design we understood the above problem can be solved using composition rather than Inheritance. Your design is so flexible that instead of having your vehicle of type only Drivable or Flyable it is now more open to create both the types. As the IDrivable and IFlyable are interfaces, you can switch between various implementations of those interfaces. So, your vehicle is not talking to particular Drivable or Flyable rather it is talking to interfaces.

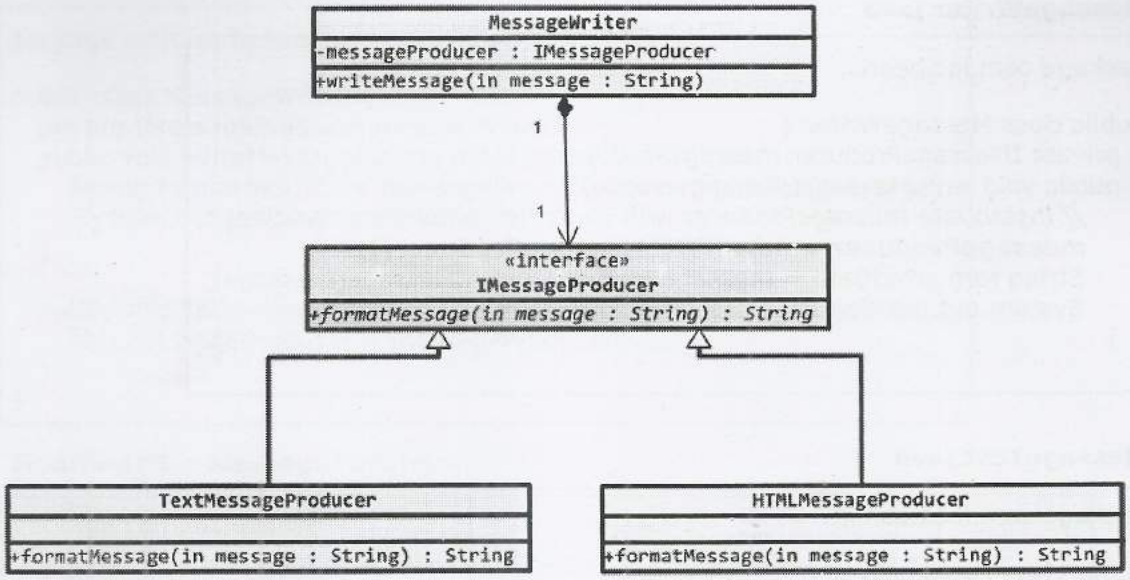
6.2 Spring Inversion of Control (IOC)

IOC stands for Inversion of control, which is a principle also known as Dependency Injection. It is the process where every object defines their dependency with whom they work with in a configuration file. These dependent objects are injected through Constructor arguments or method parameters or factory methods. The container injects the declared dependencies while creating the beans. As the process is inverse where the bean is not acquiring the dependency rather the container is pushing the dependencies hence it is called inversion of control.

Let's understand the above principle by taking an example.

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 just wants to use the functionality of IMessageProducer, but still MessageWriter has to know the instantiation process of IMessageProducer implementation class's. This means how to create IMessageProducer implementation class whether to call new operation to create or any factory method should be called to get the object etc.

To avoid the above problems instead of MessageWriter creating the Object of IMessageProducer implementation class, it should externalize this functionality to someone else who is going to create the IMessageProducer objects and injects into MessageWriter. Below code fragment shows the same.

Modified#1 - 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. Secondly 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.

6.3 Types of IOC

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

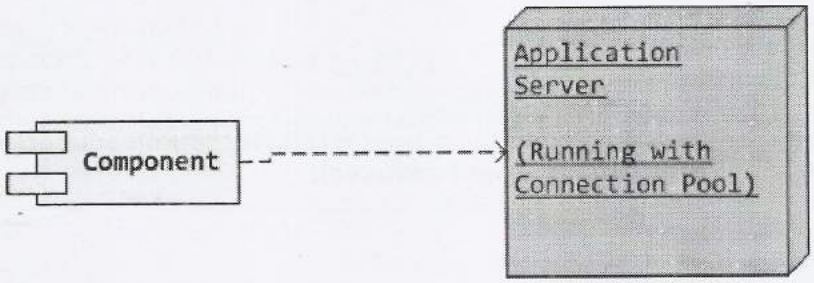
1) Dependency Lookup
    a. Dependency Pull
    b. Contextual Dependency lookup
2) Dependency Injection
    a. Setter Injection
    b. Constructor Injection

6.3.1 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. As the process is pull, it is called dependency pull.

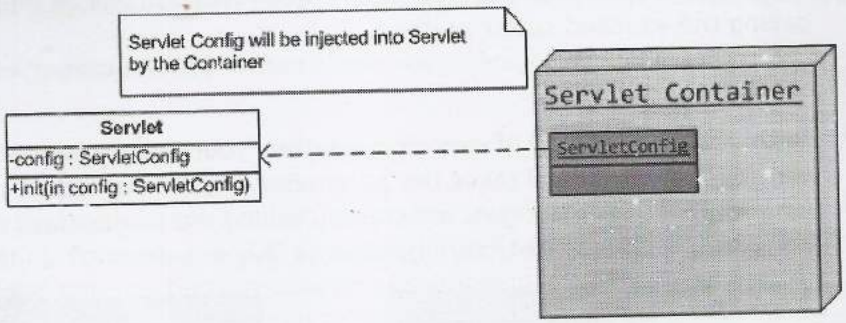
Pseudo code

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

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.



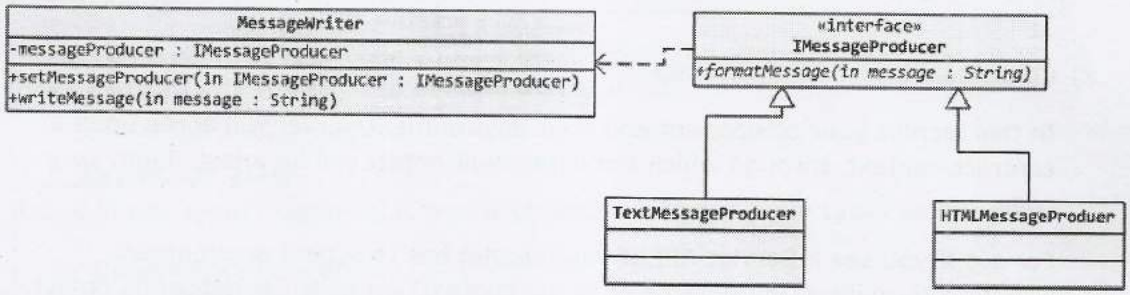
Then the container will pushes the config object by calling the init method of the servlet. Until your code implements from servlet interface, container will not pushes the config object, this indicates servlet interface acts as a contract between you and your servlet so, it is called Context Dependency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

6.3.2 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

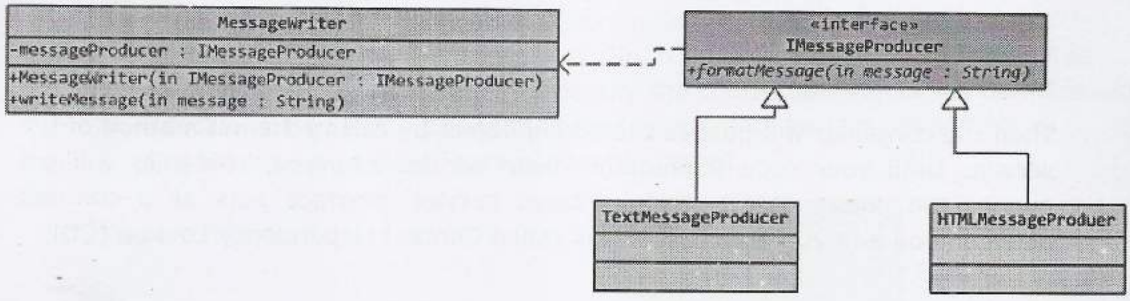
In setter injection if an object depends on another object then the dependent object will be injected into the target class through the setter method that has been exposed on the target classes as shown below.



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 takes 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">
       </bean>
</beans>
```

The org.springframework.beans and org.springframework.context packages are the basis for the Spring Framework's IOC Container. The BeanFactory interface provides an advanced configuration mechanism capable of managing any type of Object. So, the above configuration file has to be given to BeanFactory to manage the Objects that are declared in that configuration.

BeanFactory has multiple implementations one of which is XMLBeanFactory which will reads the xml configuration file as resource and creates objects and holds in Memory. This memory representation of objects is called IOC Container. Spring IOC container is an non-physical in-memory object which knows how to instantiate the beans that are declared in the "Spring Bean configuration" and manages their dependencies. The below snippet shows you how to create IOC Container.

Creating IOC Container

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

Example showing IMessageProducer injected through setter injection into MessageWriter. In our earlier example we have already exposed setter on IMessageProducer class messageProducer attribute. But we are creating both the objects and injecting the messageProducer by explicitly calling the setter in main method.

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.

Then create IOC Container from which we can get the MessageWriter object and use it. Sample modified #2 version of the code is show below.

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 dependent 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.

Modified#2 – 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");
        }
}
```

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

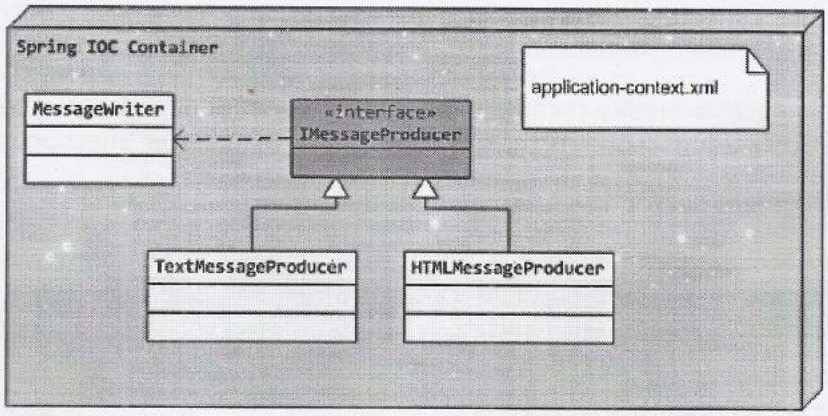
```
package com.ioc.beans;

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



6.4 Constructor VS Setter Injection

As said above we have two types of Dependency Injection's, constructor injection and setter injection. We need to under what is the difference between constructor and setter before proceeding.

Constructor Injection	Setter Injection
• At the time of creating your target class object, the dependent objects are available (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 will not available while creating the target classes. After the target class has been instantiated, then the setter on target will be called to inject the dependent objects. • 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.

6.5 Resolving Constructor Confusion

If a class contains overloaded constructors, and if you are trying to perform constructor injection, your constructor arguments resolution matching occurs using argument's type. If no potential ambiguity exists in the constructor arguments then the order in which those were declared in the configuration file will be mapped to the constructor arguments.

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

Robot.java

```
package com.cc.beans;

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

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

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

	public void showRobot() {
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
	}
}
```

In the above example the Robot class has two constructors one will take int and another takes string as parameter, if you configure this class as spring bean, you need to pass the argument for either of these constructors to instantiate as shown 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"/>
	</bean>

</beans>
```

The above configuration will makes a call to String argument constructor even you passed an integer value to it. This ambiguity rises as the "10" is compatible to String. To resolve this, you need to specify the type of value you are passing to it, so that it will detect the appropriate constructor in performing inject. Code 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"/>
	</bean>

</beans>
```

Let us consider one more scenario where the same robot class contains two argument constructors as well as shown below.

Modified - #2 Robot.java

```
package com.cc.beans;

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

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

	public void showRobot() {
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
	}
}
```

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>
```

6.6 Collection Injection

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 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;
    }
    public void showSubjects() {
        System.out.println("Subjects :");
        for (String s : subjects) {
            System.out.println(s);
        }
    }
}
```

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>
```

6.7 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 you all of the functionalities of your base class.

When it comes to spring Bean inheritance, it talks about how to re-use the existing bean configuration instead of re-defining again. Let's consider a scenario where your class contains 10 attributes, if you want to configure it as spring bean, you need to inject values for 10 attributes via constructor or setter injection.

If we want to create 10 beans of that class, we need to configure for all the 10 beans the setter or constructor injection. In case if most of the attributes has same value, even then also we need to re-write the configuration. This leads to duplicate configuration declaration results in high amount of maintenance.

In order to avoid this you can declare the configuration in one bean which acts as parent bean. And all the remaining 9 bean declarations can inherit their declaration values from the parent bean, so that we don't need to repeatedly write the same configuration in all the child beans. In this way if we modify the attribute value in parent bean, it will automatically reflects in all its 9 child beans.

The child bean can override the inherited value of parent by re-declaring at the child level. Refer to the below snippet for the same.

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;
	}

	public void showCar() {
		System.out.println("Car Info");
		System.out.println("Id : " + id);
		System.out.println("Name : " + name);
		System.out.println("Engine Type : " + engineType);
		System.out.println("Engine Model : " + engineModel);
		System.out.println("Class Type : " + classType);
	}

}
```

application-context.xml

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

<bean id="swift" class="com.bi.beans.Car" parent="baseCar">
	<property name="id" value="23"/>
	<property name="name" value="Swift"/>
</bean>
```

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.

6.8 Collection Merging

In spring 2.0, the container supports collection merging. In this your parent bean can declare a list as parent list. In your child bean you can declare a list with values; you can inherit the values of your parent list values into your child bean list values. As the list is merged with parent list values, this is called Collection Merging.

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;
	}

	public void showCourse() {
		System.out.println("Subjects :");
		for (String s : subjects) {
			System.out.println(s);
		}
	}
}
```

application-context.xml

```
<bean id="bTech1Yr1Sem" class="com.cm.beans.Course" abstract="true">
	<property name="subjects">
		<list>
			<value>c</value>
			<value>DMS</value>
		</list>
	</property>
</bean>

<bean id="bTechCS" class="com.cm.beans.Course" parent="bTech1Yr1Sem">
	<property name="subjects">
		<list merge="true">
			<value>S.E</value>
		</list>
	</property>
</bean>
```

6.9 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.

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());
	}
}
```

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;
        }
}
```

application-context.xml

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

6.10 Using IDRef

The idref element is simply an error-proof way of passing the id of a bean into another bean. Let's take a scenario where a bean wants to retrieve another bean instead of injection.

If a bean wants to fetch another bean from factory, we need to pass id of the bean via constructor or setter injection into your target bean. If we pass the id as value, if your dependent bean id changes it has to be reflected in all the beans where it has been injected. If not your target bean tries to fetch the dependent bean with wrong id, which results in Null.

In order to avoid these configuration mis-matches, to pass the id of another bean we need to pass it as idref rather than simple value as shown below.

BiCycle.java

```
package com.idref.beans;

import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;
import org.springframework.beans.factory.BeanNameAware;

public class BiCycle {
	private String chainName;

	public BiCycle() {
		System.out.println("Constructor");
	}

	public void run() {
		BeanFactory factory = new XmlBeanFactory(new
ClassPathResource("com/idref/common/application-context.xml"));
		Chain c = factory.getBean(chainName, Chain.class);
		System.out.println("I am running with BiCycle reference : " + bName);
		System.out.println("Running with Chain : " + c.getType());
	}

	public void setChainName(String chainName) {
		System.out.println("Setter");
		this.chainName = chainName;
	}
}
```

Chain.java

```
package com.idref.beans;

public class Chain {
	private String type;

	public String getType() {
		return type;
	}

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

}
```

application-context.xml

```
<bean id="biCycle" class="com.idref.beans.BiCycle">
        <property name="chainName">
                <idref bean="ct1"/>
        </property>
</bean>

<bean id="ct1" class="com.idref.beans.Chain">
        <property name="type" value="t1"/>
</bean>
```

6.11 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 beans names separated with ",".

Following code snippet shows the same.

application-context.xml

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

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" bean="robot"/>
<alias name="machine" bean="robot"/>
```

6.12 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;
	}
	public void showMotor() {
		System.out.println("Id : " + id);
	}
}
```

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/>.

6.13 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 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 example 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

```
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.

6.14 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.

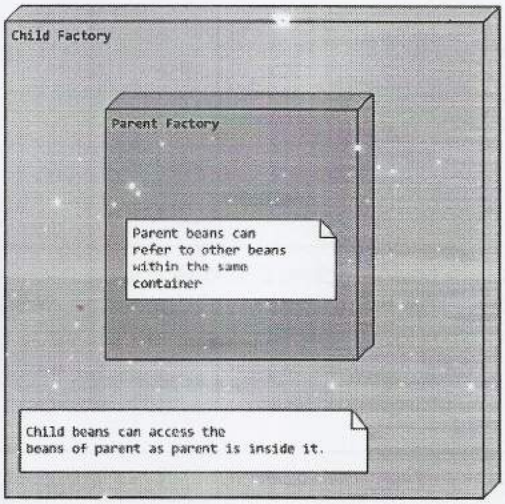
Drawback with autowiring – The problem with autowiring is we don't have control over which beans has to get injected into what, so it is least recommended to use autowiring for a large applications. For pilot projects where RAPID application development is needed we use autowiring.

6.15 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.

7 Spring Core (Advanced)

7.1 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>
```

7.2 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 and is available prior to 2.5. From spring 2.5 it has been replaced with @Required annotation, we will discuss about it 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.

7.3 Depends-On

Till now we are talking about how to manage dependencies between beans using another bean. When it comes to depends-on it doesn't talks about managing the dependencies between dependent beans, instead it talks about the creational dependencies of the beans.

If we have two beans say Employee and CacheManager, Employee uses some data in the Cache which has been loaded by CacheManager. Now Employee is dependent on Cache, but not on CacheManager. But only the CacheManager knows how to create the Cache. This indicates before the Employee gets instantiated first your cache manager should get created then only your employee bean can use the cache that has been built by the CacheManager. So in order to express such creational dependencies we need to use depends-on attribute at the <bean> tag as shown below.

Cache.java

```
package com.dpo.cache;

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

public class Cache {
	private static Cache _instance;
	private Map<String, String> data;

	private Cache(Map<String, String> data) {
		this.data = data;
	}

	private Cache() {

	}

	public synchronized static void load(Map<String, String> data) {
		if (_instance == null) {
			_instance = new Cache(data);
		}
	}

	public synchronized static Cache getInstance()
			throws IllegalAccessException {
		if (_instance == null) {
			throw new IllegalAccessException("Cache has not been initialized");
		}
		return _instance;
	}

	public synchronized String get(String key) {
		return data.get(key);
	}

	public synchronized void put(String key, String value) {
		data.put(key, value);
	}
}
```

CacheManager.java

```
package com.dpo.cache;

import com.dpo.accessor.DBDataAccessor;

public class CacheManager {

	public CacheManager() {
		// load the cache
		initialize();
	}

	public void initialize() {
		DBDataAccessor accessor = new DBDataAccessor();
		Cache.load(accessor.getData());
	}
}
```

DBAccessor.java

```
package com.dpo.accessor;

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

public class DBDataAccessor {
	public Map<String, String> getData() {
		Map<String, String> data = new HashMap<String, String>();

		// write db logic
		data.put("IND", "1");
		data.put("US", "2");

		return data;
	}
}
```

SearchEmployee.java

```
package com.dpo.beans;

import com.dpo.cache.Cache;

public class SearchEmployee {

        public void search(String countryNm) throws IllegalAccessException {
                Cache cache = Cache.getInstance();
                System.out.println("Country ID : " + cache.get(countryNm));
        }
}
```

application-context.xml

```
<bean id="searchEmployee" class="com.dpo.beans.SearchEmployee" depends-
on="cacheManager"/>
<bean id="cacheManager" class="com.dpo.cache.CacheManager"/>
```

In the above configuration, irrespective of order in which you declared searchEmployee and cacheManager beans first your cacheManager will be created first and then only your searchEmployee will be created.

7.4 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.

In the same way, for spring beans also, the core container provides extension hooks to initialize its state and should facilitate to release the resources held be spring bean. Unlike your normal pojo or servlet lifecycle, spring bean lifecycle would be slightly different. In spring you can consider a bean has been completely constructor only after the constructor, setter or any other injection or performed and all its dependents are acquired. So if we write the initialization logic in constructor for a spring bean, only the dependents that are injected via constructor are available but the dependents that are injected via setter will not be available in the constructor.

So, we need a unified place where after all the dependents are available we want to perform initialization logic and before the container removes the bean, we want to perform destruction process.

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

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

7.4.1 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;

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

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;
  }

}
```