Spring Frame Work Introduction

By
Apparao G

Spring Frame work :

- Spring is light weight open source frame work to develop the java,j2ee based applications.
- spring is lightweight in terms of both size and overhead. The bulk of the spring framework can be distributed in a single JAR file that weight in at just over 2.5 MB. And the processing overhead required by spring is negligible.
- Which is invented by Rod Jhonson, in the year 2003.

Spring Framework Benefits:

- Allows pojo object,poji model programming(light weight programming).
- Allows to develop all kind of applications like standalone, web, and distributed-tier applications.
- It provides loose coupling using Dependency Injection.
- It provides middle ware services like transaction management, logging, and security service.(with help of Aspect Oriented Programming (AOP))
- Spring can eliminate the creation of singletons and factory classes seen on many projects.
- Spring provides own web framework to develop web applications.(with help of Spring web MVC)
- Spring provides light weight containers. So we can work with spring without using web, application server software's.
- Spring eliminates boilerplate code such as JNDI lookup, opening and closing JDBC connections and JDBC, exception handling etc.

Spring Key Benefits:

1. Modularity

Plain old Java Objects keep your code concise, simple and modular

2. Productivity

Over 70% of developers report productivity gains and reduction in time to deploy with Spring

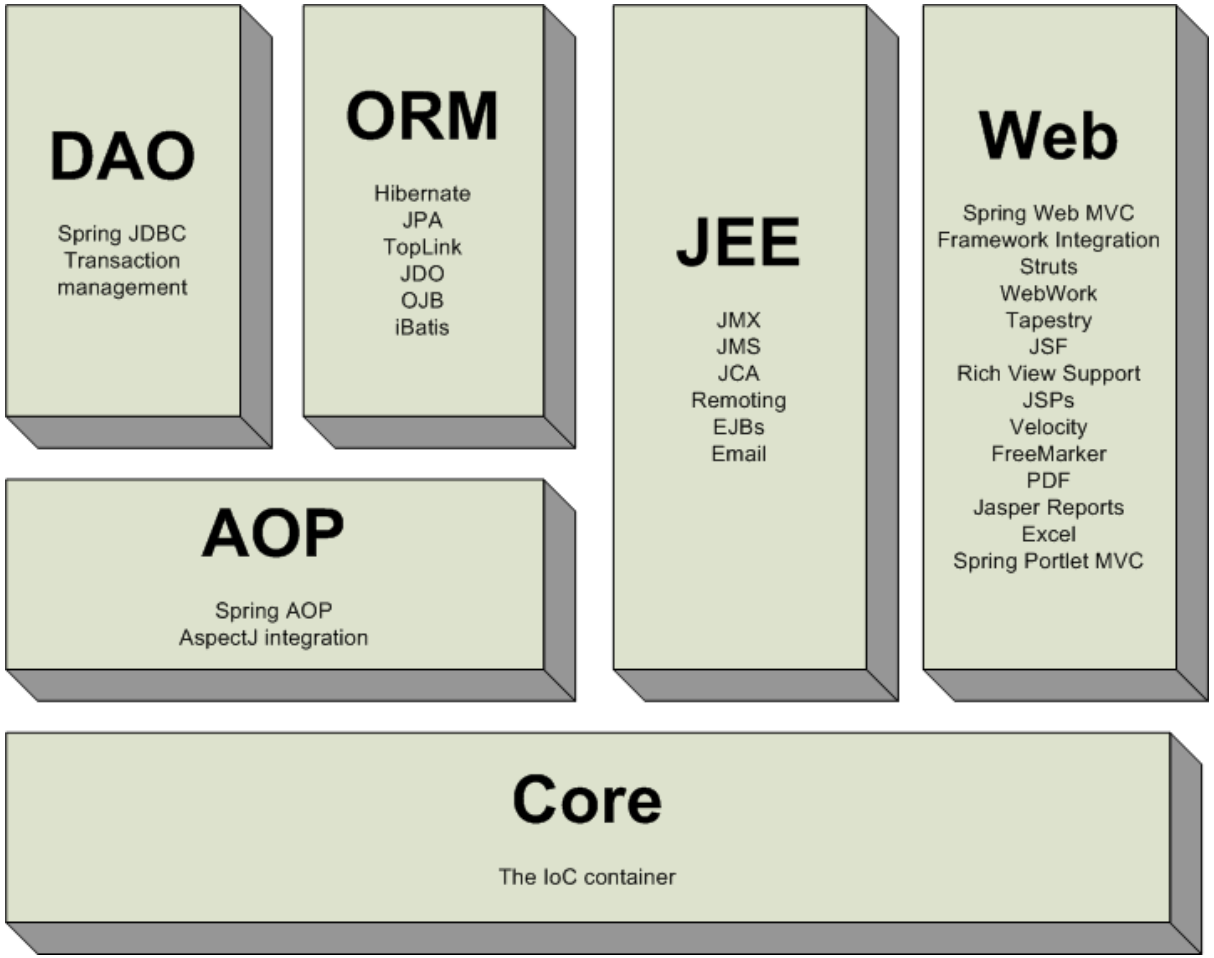
3. Portability

Applications run on Tomcat, all Java EE servers as well as cloud platforms

4. Testability

Cleanly expressed dependencies make unit and integration testing easier

Spring Modules:



When to use Spring / Ejb:

- The EJB specification clearly defines the contract between EJB components and EJB containers.
- By running in an EJB container, EJB components can get the benefits of life cycle management, transaction management, and security services.
- However, in EJB versions prior to 3.0, a single EJB component requires a remote/local interface, a home interface, and a bean implementation class. These EJBs are called heavyweight components due to their complexity.
- Moreover, in those EJB versions, an EJB component can only run within an EJB container and must look up other EJBs with JNDI (Java Naming and Directory Interface).
- So EJB components are technology dependent because they cannot be reused and tested outside the scope of an EJB container.

- The Spring Framework's declarative transaction management is similar to EJB CMT in that you can specify transaction behavior (or lack of it) down to individual method level. It is possible to make a setRollbackOnly() call within a transaction context if necessary.
- The differences between the two types of transaction management are:
- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JTA transactions or local transactions using JDBC, JPA, Hibernate or JDO by simply adjusting the configuration files.
- You can apply the Spring Framework declarative transaction management to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*, a feature with no EJB equivalent. Both programmatic and declarative support for rollback rules is provided.
- The Spring Framework enables you to customize transactional behavior, by using AOP. For example, you can insert custom behavior in the case of transaction rollback. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you cannot influence the container's transaction management except with setRollbackOnly().
- The Spring Framework does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, consider carefully before using such a feature, because normally, one does not want transactions to span remote calls.

Inversion of Control (IOC)

- Inversion of control is an architectural pattern describing an external entity (that is container) used to wire objects at creation time by injecting there dependencies, that is, connecting the objects so that they can work together in a system.
- In other words the IOC describes that a dependency injection needs to be done by an external entity instead of creating the dependencies by the component itself.

Why Dependency Injection (DI)

Example: Observe the below use case in figure(1)

```
public class ReportService {
      public static void main
            (String args[]) {
      PdfReportGenerator
      pdfReportGenerator = new
            PdfReportGenerator();

      pdfReportGenerator.getAnnual
            Report("2012");
      }
}
```

```
public class PdfReportGenerator {
      public void getAnnualReport
                  (String year){
            System.out.println
      ("Annual report in PDF format
            for year:"+year);
      }
}
```

Here ReportService class depends on the PdfReportGenerator class to complete the given request. Here we are creating the PdfReportGenerator object using "new" operator.
Suppose if we introduce ExcelReportGenerator or HtmlReportGenerator class, ReportService class code will not work.

Generally we are creating the objects in different ways as follows.

1. Using "new" Operator.
2. Factory class and factory methods or from the naming registry.
3. Using clone ().
4. Using Deserialization process.

But these approaches results in some problems which are described below:

1. The complexity of the application increases.
2. The development time –dependency increases.
3. The difficulty for unit testing increases.

Improved version of figure (1) shown below in figure(2).
Here we are creating ReportGenerator interface and declaring the getAnnualReport() method.

- Figure(2) shows improved version of figure(1).

```
public class ReportService {
      public static void main(String
                        args[]) {
      ReportGenerator reportGenerator =
              new PdfReportGenerator();

      reportGenerator.getAnnualReport
                              ("2012");
      }
}
```

```
public interface ReportGenerator {
      public void getAnnualReport
                  (String year);
}
```

```
public class PdfReportGenerator
implements ReportGenerator{
      public void getAnnualReport
            (String year){
      System.out.println("Annual
            report in PDF format
            for year:"+year);
      }
}
```

- But again we have not escape from new PdfReportGenerator() code.If we want Excel report,we can create the ExcelReportGenerate class object using new operator.That means ReportService class tightly coupled with PdfReportGenerator class.
- To solve the above problem spring introduced Inversion of control (IOC) shown in below.

  <bean id=*"pdfGenerator"*
  class=*"com.apparao.report.generator.PdfReportGenerator" />*

```
public class ReportService {

      public static void main(String
            args[]) {
      Resource resource = new
                  ClassPathResource
      ("applicationContext.xml");
      BeanFactory factory = new
            XmlBeanFactory(resource);
      ReportGenerator reportGenerator =
         (ReportGenerator)
      factory.getBean("pdfGenerator");

      reportGenerator.getAnnualReport
                              ("2012");
      }
}
```

```
public interface ReportGenerator {
      public void getAnnualReport
                  (String year);
}
```

```
public class PdfReportGenerator
implements ReportGenerator{
      public void getAnnualReport
            (String year){
      System.out.println("Annual
            report in PDF format
            for year:"+year);
      }
}
```

<bean id="*pdfGenerator*" class="*com.apparao.report.generator.PdfReportGenerator*"></bean>

Spring Containers:

Spring provides two basic core containers.

1. Bean Factory
2. Application Context

Bean Factory:

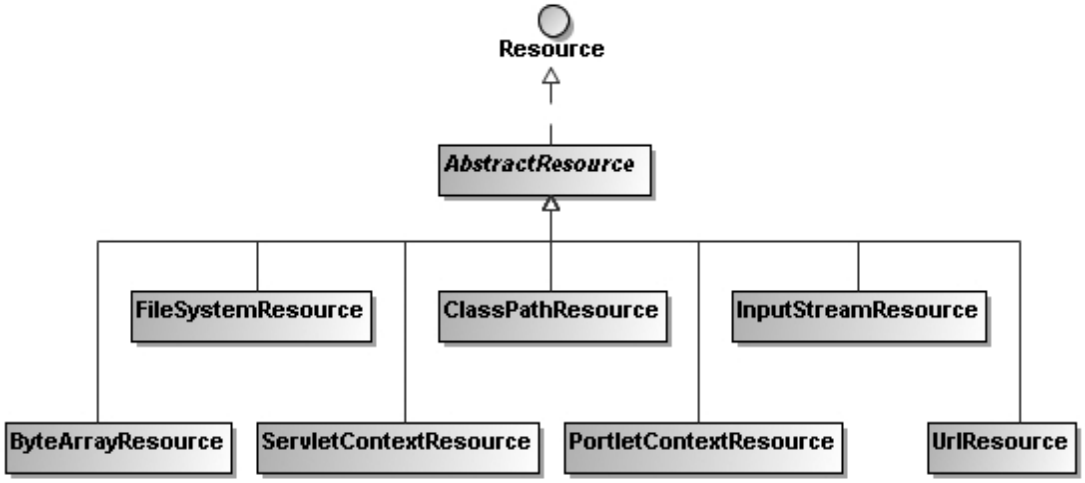
1. A bean factory lazily loads all beans, deferring bean creation until the getBean() method is called.
2. It does not support I18N and distributed applications.
3. It is used for developing the mobile applications.

Step1: **Instantiating a Bean Factory:**

To instantiate a bean factory, you have to load the bean configuration file into a Resource object first. For example, the following statement loads your configuration file from the root of the classpath.

Resource resource = new ClassPathResource("beans.xml");

Resource is only an interface, while ClassPathResource is one of its implementations for loading a resource from the classpath. Other implementations of the Resource interface, such as FileSystemResource, InputStreamResource, and UrlResource, are used to load a resource from other locations. Figure shows the common implementations of the Resource interface in Spring.



Step2: Next, you can use the following statement to instantiate a bean factory by passing in a Resource object with the configuration file loaded:

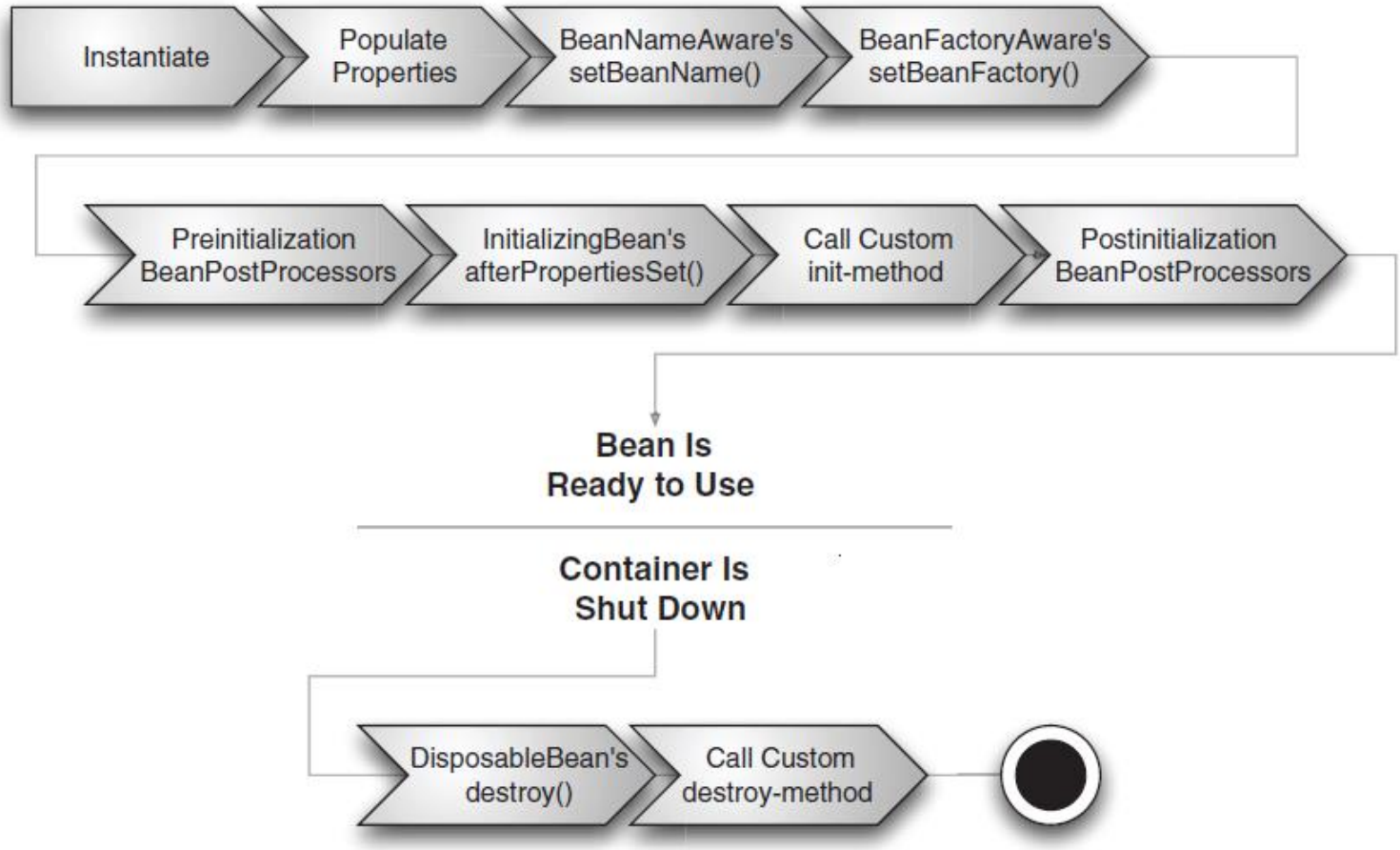
```
BeanFactory factory = new XmlBeanFactory(resource);
```

Step3: **Getting Beans from the IoC Container**

You just make a call to the getBean() method and pass in the unique bean name. The return type of the getBean() method is java.lang.Object, so you have to cast it to its actual type before using it.

```
EmployeeService  employeeService =
                    (EmployeeService)factory.getBean("empService");
```

Bean Factory:(Bean Life Cycle)



Bean Factory performs several setup steps before a bean is ready to use.

1. **Instantiate-** Spring instantiate the beans.
2. **Populate properties-** Spring injects the bean's properties.
3. **Set bean name-** If the bean implements BeanNameAware, spring passes the bean's Id to setBeanName().
4. **Set bean Factory-** If the bean implements BeanFactoryAware, spring passes the bean factory to setBeanFactory().
5. **Post Process** (before initialization)- If there are any BeanPostProcessors, spring calls their postProcessBeforeInitialization() method.
6. **Initialize beans-** If the bean implements InitializingBean, its afterPropertiesSet() method will be called. If the bean has a custom init method declared,the specified initialization method will be called.
7. **Post Process**(after initialization)- If there are any BeanPostProcessors, spring calls their postProcessAfterInitialization() method.
8. **Bean is ready to use-** At this point the bean is ready to be used by the application and will remain in the bean factory until it is no longer needed.
9. **Destroy Bean** – If the bean implements DisposableBean(), its destroy() method will be called.If the bean has a custom destroy-method declared, the specified method will be called.

Application Context:

Application Context:

1. An Application Context is a bit smarter and preloads all singleton beans upon context start up. By preloading singleton beans, you ensure that they will be ready to use when your application needed won't have to wait for them to be created.
2. It support I18N .
3. It is used to develop the distributed applications.

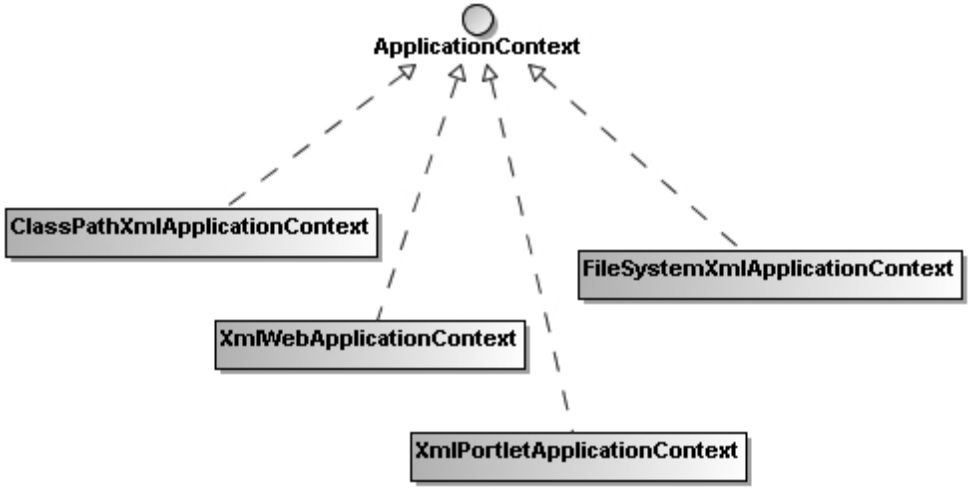
Step1: **Instantiating an Application Context:**

Like BeanFactory, ApplicationContext is an interface only. You have to instantiate an implementation of it. The ClassPathXmlApplicationContext implementation builds an application context by loading an XML configuration file from the classpath. You can also specify multiple configuration files for it.

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Besides ClassPathXmlApplicationContext, there are several other ApplicationContext implementations provided by Spring. FileSystemXmlApplicationContext is used to load XML configuration files from the file system, while XmlWebApplicationContext and XmlPortletApplicationContext can be used in web and portal applications only.

Figure shows the common implementations of the ApplicationContext interface in Spring.



Step2: **Getting Beans from the IoC Container:**

You just make a call to the getBean() method and pass in the unique bean name. The return type of the getBean() method is java.lang.Object, so you have to cast it to its actual type before using it.

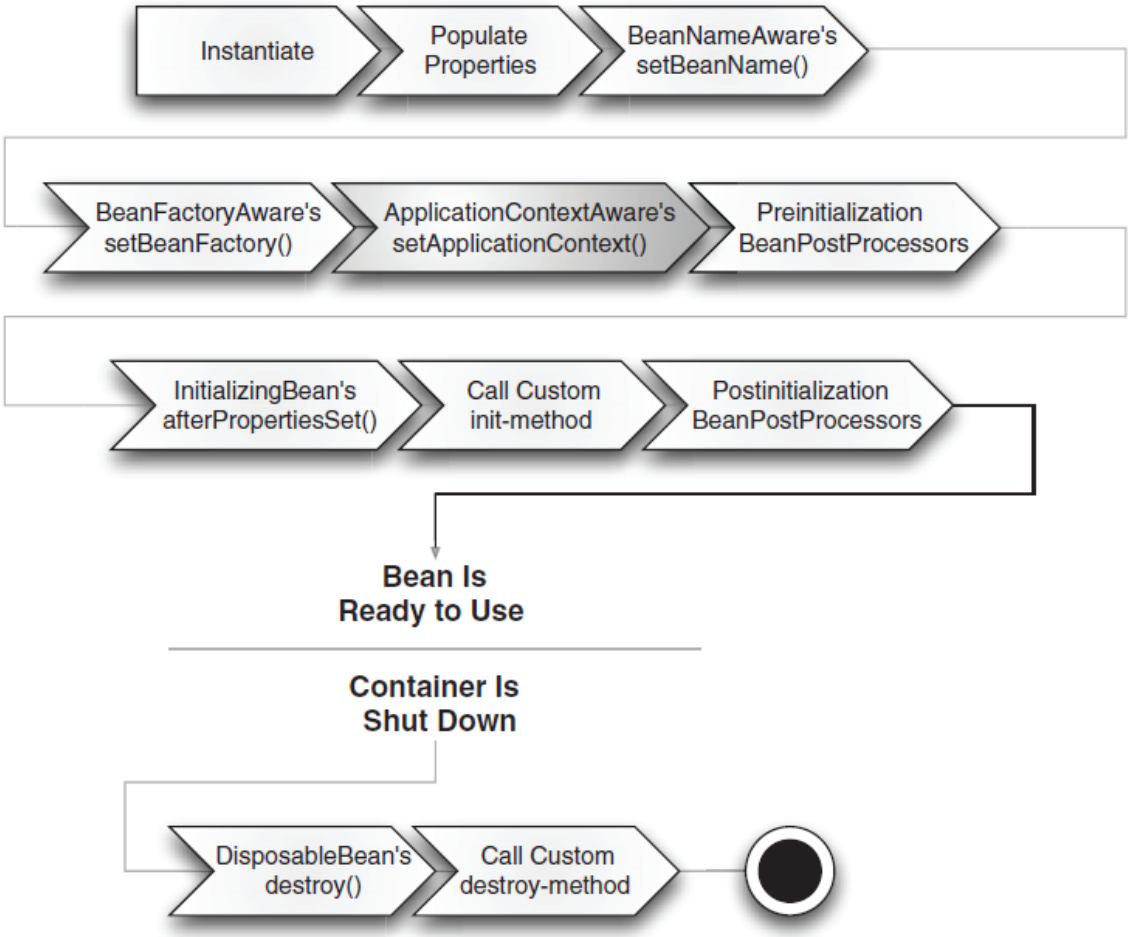
```
EmployeeService  employeeService = (EmployeeService)
                                    context.getBean("empService");
```

Similarly FileSystemXmlApplicationContext- loads a context definition from an xml file in the file system.

```
ApplicationContext container = new
        FileSystemXmlApplicationContext("application-context.xml");
```

Application Context Container:(Bean Life Cycle)

The only difference here is that if the bean implements the ApplicationContextAware interface, the setApplicationContext() method is called.


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Preinitialization BeanPostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Postinitialization BeanPostProcessors
Bean Is Ready to Use
Container Is Shut Down
DisposableBean's destroy()
Call Custom destroy-method

Dependency Injection:

The process of injecting (Pushing) the dependencies into an object is known as dependency injection(DI).

DI gives the following benefits:

- The application development will become faster.
- Dependency will be reduced.
- DI provides a proper test environment for the application as it is much easier to isolate the code under test if we need not worry about the code necessary for instantiating and initializing lot of dependencies.

Types of Dependency Injection:

- Setter Injection.
- Constructor Injection.
- Interface injection (Spring is not supported.)

Constructor Injection :

- Constructor dependency injection is the method of injecting the dependencies of an object through its constructor arguments.
- In this mechanism the dependencies are pushed into the object through the constructor arguments at the time of instantiating it.

Example:

```
public interface EmployeeService {
        public String   getEmployeeType(int
                                        empNo);
 }
```

```
public interface EmployeeDao {
            public String getEmployeeType(int
                                    empNo);
}
```

```
public class EmployeeServiceImpl implements
EmployeeService{
  EmployeeDao employeeDao;
  public EmployeeServiceImpl(EmployeeDao
employeeDao){
        this.employeeDao = employeeDao;
  }
 public String getEmployeeType(int empNo){
       return
employeeDao.getEmployeeType(empNo);
            }
}
```

```
public class EmployeeDaoImpl implements
EmployeeDao {
public String getEmployeeType(int empNo) {
 String status = null;
  if (empNo != 0) {
        if (empNo < 1500) {
            status = "Permanent";
        } else {
            status = "Contract";
         }
    }
  return status;
  }
}
```

```
<bean id="employeeService" class="com.apparao.service.EmployeeServiceImpl">
            <constructor-arg>
              <ref bean="employeeDao" />
            </constructor-arg>
</bean>
<bean id="employeeDao" class="com.apparao.dao.EmployeeDaoImpl" />
```

Example 2: Constructor Argument Resolution example.

```
public class Employee {
	private String empNo;
	public Employee(String empNo){
		System.out.println("String version Constructor called");
		this.empNo = empNo;
	}
	public Employee(int empNo){
		System.out.println("int version constructor called");
		this.empNo = "Number:"+Integer.toString(empNo);
	}
	public String toString(){
		return empNo;
	}
	public static void main(String args[]){
		ApplicationContext container = new
					ClassPathXmlApplicationContext("applicationContext.xml");
		Employee employee = (Employee)container.getBean("employee");
		System.out.println(employee);
	}
}
<bean id="employee" class="com.apparao.constructor.test.Employee">
		<constructor-arg value="100" />
  </bean>
```

Output: String version Constructor called

100

This shows that the constructor with the spring argument was called. This is not the desired effect, sine we want to prefix any integer values passed in using constructor injection with Number: as shown in the int constructor. To get around this, we need to make a small modification to the configuration.

```
<bean id="employee" class="com.apparao.constructor.test.Employee">
        <constructor-arg value="100" type="int"/>
</bean>
```

Output: int version constructor called

Number:100

```
public class Employee {
	private String empNo;
	private String ename;
	private float salary;
	private float comm;
	public Employee(String empNo){
			System.out.println("String version Constructor called");
			this.empNo = empNo;
	}
	public Employee(int empNo){
			System.out.println("int version constructor called");
			this.empNo = "Number:"+Integer.toString(empNo);
	}
	public Employee(String empNo,String ename,float salary,float comm){
			System.out.println("multiple constructor");
			this.empNo = empNo;
			this.ename = ename;
			this.salary = salary;
			this.comm = comm;
	}
	public String toString(){
			return empNo;
	}
	public static void main(String args[]){
			ApplicationContext container = new ClassPathXmlApplicationContext("applicationContext.xml");
			Employee employee = (Employee)container.getBean("employee");
			//System.out.println(employee);
			System.out.println("Empno:"+employee.empNo);
			System.out.println("Emp name:"+employee.ename);
			System.out.println("Emp salary:"+employee.salary);
			System.out.println("Emp comm:"+employee.comm);

	}
}
```

applicationContext.xml

```
<bean id="employee" class="com.apparao.constructor.test.Employee">
                <constructor-arg index="0" value="100" type="java.lang.String"/>
                <constructor-arg index="1" value="Ramu" type="java.lang.String"/>
                <constructor-arg index="2" value="200000" type="float"/>
                <constructor-arg index="3" value="1111" type="float"/>
  </bean>
```

Output: multiple constructor
Empno:100
Emp name:Ramu
Emp salary:200000.0
Emp comm:1111.0

Problems with Constructor injection:

1. If a bean has several dependencies, the constructor's parameter list can be quite lengthy.
2. If there are several ways to construct a valid object, it can be hard to come up with unique constructor, since constructor signatures vary only by the number and type of parameters.
3. If a constructor takes two or more parameter of the same type, it may be difficult to determine what each parameter's purpose is.
4. Constructor injection does not lend itself readily to inheritance. A bean's constructor will have to pass parameters to super() in order to set private properties in the parent object.

Setter Injection :

- 1. Setter injection is the most popular type of DI and is supported by most IOC containers.
- 2. The container injects dependency via a setter method declared in a component.

Example:

```
public interface ReportService {
    public String getReport(String year);
 }
public class ReportServiceImpl implements ReportService {

    private ReportRepository reportRepository;

    public void setReportRepository(ReportRepository reportRepository) {
        this.reportRepository = reportRepository;
    }

    public String getReport(String year) {
        return reportRepository.getReport(year);
    }
}
```

```
public interface ReportRepository {
    public String getReport(String year);
 }
public class PdfRepositoryImpl implements ReportRepository {
    public String getReport(String year){
            return "Pdf report for the year:"+year;
    }
 }
```

applicationContext.xml

```
<bean id="reportService" class="com.apparao.service.ReportServiceImpl">
            <property name="reportRepository">
            <ref bean="pdfRepository"/>
      </property>
  </bean>
  <bean id="pdfRepository" class="com.apparao.dao.PdfRepositoryImpl" />
```

```
public class ReportTest {
   public static void main(String[] args) {
            Resource resource = new ClassPathResource("applicationContext.xml");
            BeanFactory factory = new XmlBeanFactory(resource);
            ReportService reportService =
                                (ReportService)factory.getBean("reportService");
            System.out.println(reportService.getReport("2012"));
     }
 }
```

Out put: Pdf report for the year:2012

Problems with Setter injection:

The first is that, as a component designer, you cannot be sure that a dependency will be injected via the settermethod. If a component user forgets to inject a required dependency, the evil NullPointerException will be thrown and it will be hard to debug.

Second shortcoming of setter injection has to do with code security. After the first injection, a dependency may still be modified by calling the setter method again, unless you have implemented your own security measures to prevent this. The careless modification of dependencies may cause unexpected results that can be very hard to debug.

Bean Name Aliasing:

Spring allows a bean to have more than one name. You can achieve this by specifying a comma or semicolon – separated list of names in the “name” attribute of the bean’s <bean> tag.

<beans -->

<bean id=”name1” name = “name2, name3, name4” class=”
com.apparao.alias.Employee” />

</beans>

We have defined four names: one using the “id” attribute, and the other three as a comma-separated list in the “name” attribute.

Alternatively, we can use the <alias> tag, specifying one or more aliases for any given bean name.

<beans -- >

<bean id=”name1” name=”name2,name3,name4” class=”
com.apparao.alias.Employee” />

<alias name=”name2” alias=”namex1” />

<alias name=”name1” alias=”namex2” />

</beans>

Example:

```
public class Employee {
    private String ename;
    public String getEname() {
        return ename;
    }
    public void setEname(String ename) {
        this.ename = ename;
    }
 }
```

applicationContext.xml

```
<bean id="name1" name="name2,name3,name4" class="com.apparao.alias.Employee"
                                                                        >
        <property name="ename" value="satyanarayana" />
  </bean>
    <alias name="name2" alias="namex1" />
    <alias name="name1" alias="namex2" />
```

```
public class AliasDemo {
	public static void main(String[] args) {
		ApplicationContext container = new ClassPathXmlApplicationContext("applicationContext.xml");
		Employee e1 = (Employee)container.getBean("name1");
		Employee e2 = (Employee)container.getBean("name2");
		Employee e3 = (Employee)container.getBean("name3");
		Employee e4 = (Employee)container.getBean("name4");
		Employee e5= (Employee)container.getBean("namex1");
		Employee e6= (Employee)container.getBean("namex2");
		System.out.println("hash code e1:"+e1.hashCode());
		System.out.println("hash code e2:"+e2.hashCode());
		System.out.println("hash code e3:"+e3.hashCode());
		System.out.println("hash code e4:"+e4.hashCode());
		System.out.println("hash code e5:"+e5.hashCode());
		System.out.println("hash code e6:"+e6.hashCode());
	}
 }
```

Output: hash code e1:28541929
hash code e2:28541929
hash code e3:28541929
hash code e4:28541929
hash code e5:28541929
hash code e6:28541929
All the objects hash codes are same that means all are pointing to same bean.

Bean Scopes

Spring provides five types of bean scopes. They are

1. singleton
2. prototype
3. request
4. session
5. global session

Singleton :

singleton:

- singleton is default scope of the spring container.
- Scopes a single bean definition to a single object instance per Spring IoC container.

We can configure the singleton beans following ways:

<bean id="*employee*" class="*com.apparao.beanscopes.Employee*"/>

<!-- the following is equivalent, though redundant (singleton scope is the default); using spring-beans-2.0.dtd -->

<bean id="*employee*" class="*com.apparao.beanscopes.Employee*"
scope="*singleton*"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="*employee*" class="*com.apparao.beanscopes.Employee*"
singleton="*true*"/>

Example:

```
public class Employee {
    private String ename;
    public String getEname() {
        return ename;
    }
    public void setEname(String ename) {
        this.ename = ename;
    }
 }
```

applicationContext.xml

```
<bean id="employee"  class="com.apparao.beanscopes.Employee"
                                                    scope="singleton">
        <property name="ename" value="satyanarayana" />
 </bean>
```

```
public class SingletonTest {

    public static void main(String[] args) {
        ApplicationContext container = new
                ClassPathXmlApplicationContext("applicationContext.xml");
        Employee e1 = (Employee)container.getBean("employee");
        Employee e2 = (Employee)container.getBean("employee");

        System.out.println("hash code e1:"+e1.hashCode());
        System.out.println("hash code e2:"+e2.hashCode());
        if(e1.hashCode() == e2.hashCode()){
                System.out.println("Both objects hash code is same hence only one
                        object is creating per container.");
        }
    }
 }
```

Output: hash code e1:32392776

hash code e2:32392776

Both objects hash code is same hence only one object is creating per container .

Prototype:

- Every call to the getBean() method returns a new instance of the bean.

We can configure the beans following ways:

<!-- using spring-beans-2.0.dtd -->

<bean id="*employee*" class="*com.apparao.beanscopes.Employee*"
scope="*prototype*"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->

<bean id="*employee*" class="*com.apparao.beanscopes.Employee*"
singleton="*false*"/>

Example:

```
public class Employee {
    private String ename;

    public String getEname() {
        return ename;
    }

    public void setEname(String ename) {
        this.ename = ename;
    }

}
```

applicationContext.xml

```
    <bean id="employee"  class="com.apparao.beanscopes.Employee"
                                                    scope="prototype">
        <property name="ename" value="satyanarayana" />
    </bean>
```

```
public class PrototypeTest {

    public static void main(String[] args) {
        ApplicationContext container = new ClassPathXmlApplicationContext("applicationContext.xml");
        Employee e1 = (Employee)container.getBean("employee");
        Employee e2 = (Employee)container.getBean("employee");

        System.out.println("hash code e1:"+e1.hashCode());
        System.out.println("hash code e2:"+e2.hashCode());
        if(e1.hashCode() == e2.hashCode()){
                System.out.println("Both objects hash code is same hence only one object is " +
                                "creating per container");
        }else{
                System.out.println("Both the hash codes not equals,hence spring container" +
                                "creates two diffrent objects.");
        }
    }
}
```

Output: hash code e1:2145913

hash code e2:28910606

Both the hash codes not equals,hence spring containercreates two diffrent objects.

request:

Every call to the getBean() method in a web application will return a unique instance of the bean for every HTTP request. This behaviour is only implemented in the WebApplicationContext and its subinterfaces.

<bean id="loginAction" class="com.apparao.user.LoginAction" scope="request"/>

session:

Calls to the getBean() method will return a unique instance of the bean for every HTTP session. Just like request, this scope is only available in WebApplicationContext and its subinterfaces.

<bean id="userPreferences" class="com.apparao.user.UserPreferences" scope="session"/>

global session:

Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

<bean id="userPreferences" class="com. apparao.user.UserPreferences" scope="globalSession"/>

Lazy Loading

1. ApplicationContext implementations eagerly create and configure all singleton beans as part of the initialization process.
2. If we are not interested to create a bean objects eagerly, we need to add **lazy-init =”true”** inside the bean definition.

Example:

```
<bean id="employee" class="com.apparao.test.Employee"
                    lazy-init="true"/>
```

A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.

Beans Auto-wiring

The Spring container can *autowire* relationships between collaborating beans. You can allow Spring to resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext.

Advantages:

1. Autowiring can significantly reduce the need to specify properties or constructor arguments.
2. Autowiring can update a configuration as your objects evolve. For example, if you need to add a dependency to a class, that dependency can be satisfied automatically without you needing to modify the configuration

Spring provides four types of Autowiring.

1. no : (Default) No autowiring. Bean references must be defined via a ref element.
   Changing the default setting is not recommended for larger deployments, because specifying collaborators explicitly gives greater control and clarity.

2. byName
3. byType
4. constructor

byname:

Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired.

byName

```java
package com.apparao.autowire;

public class Employee {
      private Address address;
}
```

```java
package
com.apparao.autowire;

public class Address {
      private String city;
}
```

```xml
<bean id="employee" class="com.apparao.autowire.Employee"
                                        autowire="byName"/>
 <bean id="address" class="com.apparao.autowire.Address" >
      <property name="city" value="Bangalore" />
 </bean>
```

byType :

Allows a property to be autowired if exactly one bean of the property type exists in the container. If more than one exists, a fatal exception is thrown, which indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set.

byType

```
package com.apparao.autowire;

public class Student {
      private Course course;

}
```

```
package com.apparao.autowire;

public class Course {
      private String name;
}
```

```
<bean id="student" class="com.apparao.autowire.Student"
                                        autowire="byType" />
 <bean id="course" class="com.apparao.autowire.Course" >
      <property name="name" value="Java" />
  </bean>
```

Note: The main problem of auto-wiring by type is that sometimes there will be more than one bean in the IoC container compatible with the target type. In this case, Spring will not be able to decide which bean is most suitable for the property, and hence cannot perform autowiring.

constructor:

Analogous to *byType*, but applies to constructor arguments. If there is not exactly one bean of the constructor argument type in the container, a fatal error is raised.

constructor

```
package com.apparao.autowire;

public class College {
      private Student student;
      public College(Student
                        student){
            this.student = student;
      }

}
```

```
package
com.apparao.autowire;

public class Student {
      private String
            studentName;
}
```

```
<bean id="college" class="com.apparao.autowire.College"
      autowire="constructor" />
  <bean id="student" class="com.apparao.autowire.Student" >
      <property name="studentName" value="Sunil Kumar" />
  </bean>
```

Limitations and disadvantages of autowiring:

- Explicit dependencies in property and constructor-arg settings always override autowiring. You cannot autowire so-called *simple* properties such as primitives, Strings, and Classes (and arrays of such simple properties). This limitation is by-design.
- Autowiring is less exact than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity that might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.
- Multiple bean definitions within the container may match the type specified by the setter method or constructor argument to be autowired. For arrays, collections, or Maps, this is not necessarily a problem. However for dependencies that expect a single value, this ambiguity is not arbitrarily resolved. If no unique bean definition is available, an exception is thrown.

Annotation-based configuration

Spring 2.5 introduced @AutoWired annotation for injecting the dependencies to objects.

Example:

```
package com.apparao.service;
 public interface ReportService {
     public String getReport(String year);
}
package com.apparao.service;
@Service("reportService")
public class ReportServiceImpl implements ReportService {
     @Autowired
     private ReportRepository reportRepository;
     public String getReport(String year) {
          return reportRepository.getReport(year);
     }
}
```

```
package com.apparao.dao;
 public interface ReportRepository {
       public String getReport(String year);
 }
package com.apparao.dao;
 import org.springframework.stereotype.Repository;
 @Repository
public class PdfRepositoryImpl implements ReportRepository {
            public String getReport(String year){
                        return "Pdf report for the year:"+year;
     }
 }
applicationContext:
<beans --- >
<context:component-scan base-package="com.apparao.service,com.apparao.dao" />
</beans>
```

```
public class ReportTest {

    public static void main(String[] args) {
        ApplicationContext container = new
        ClassPathXmlApplicationContext("applicationContext.xml");
        ReportService reportService =
        (ReportService)container.getBean("reportService");
        System.out.println(reportService.getReport("2012"));
    }

}
```

Output : Pdf report for the year :2012

Bean Definition Inheritance

Problem:

When configuring beans in the Spring IoC container, you may have more than one bean sharing some common configurations, such as bean properties and attributes in the <bean> element. You often have to repeat these configurations for multiple beans.

Solution:

Spring allows you to extract the common bean configurations to form a ***parent bean***. The beans that inherit from this parent bean are called *child beans*. The child beans will inherit the bean configurations, including bean properties and attributes in the <bean> element, from the parent bean to avoid duplicate configurations. The child beans can also override the inherited configurations when necessary.

The parent bean can act as a configuration template and also as a bean instance at the same time. However, if you want the parent bean to act only as a template that cannot be retrieved, you must set the abstract attribute to true, asking Spring not to instantiate this bean.

Example:

```
package com.apparao.bean.inheritance;
 public class Employee {
	private int empNo;
	private String ename;
	public int getEmpNo() {
		return empNo;
	}
	public void setEmpNo(int empNo) {
		this.empNo = empNo;
	}
	public String getEname() {
		return ename;
	}
	public void setEname(String ename) {
		this.ename = ename;
	}
 }
```

```
package com.apparao.bean.inheritance;

public class Address extends Employee{
	private String street;
	private String city;

	public String getStreet() {
		return street;
	}

	public void setStreet(String street) {
		this.street = street;
	}

	public String getCity() {
		return city;
	}

	public void setCity(String city) {
		this.city = city;
	}

}
```

applicationContext.xml

```
<bean id="employee" class="com.apparao.bean.inheritance.Employee" abstract="true">
                        <property name="empNo" value="100" />
                        <property name="ename" value="Suresh" />
    </bean>
  <bean id="address" class="com.apparao.bean.inheritance.Address" parent="employee">
                        <property name="street" value="M.G Road" />
                        <property name="city" value="Bangalore" />
                        <!-- the empNo property value of 100 will be inherited from parent -->
    </bean>
```

```
public class BeanInheritanceTest {
    public static void main(String[] args) {
            ApplicationContext container = new
                                ClassPathXmlApplicationContext("applicationContext.xml");
            Address address = (Address)container.getBean("address");
            System.out.println("Empno:"+address.getEmpNo());
            System.out.println("Street:"+address.getStreet());
            System.out.println("City:"+address.getCity());
    }
}
```

Output: Empno:100

Street:M.G Road

City:Bangalore

Method Injection

When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, the typical and common approach of handling this dependency by defining one bean to be a property of the other is quite adequate. There is a problem when the bean lifecycles are different. Consider a singleton bean A which needs to use a non-singleton (prototype) bean B, perhaps on each method invocation on A. The container will only create the singleton bean A once, and thus only get the opportunity to set the properties once. There is no opportunity for the container to provide bean A with a new instance of bean B every time one is needed.

Spring supports two forms of method injection:

Method replacement: Enables existing methods (abstract or concrete) to be replaced at runtime with new implementations.

Getter injection (Lookup method injection**):** Enables existing methods (abstract or concrete) to be replaced at runtime with a new implementation that returns a specific bean from the spring context.

Method Injection Example:

```
package com.apparao.method.injection;
public class MobileStore {
	public String buyMobile(){
		return "Bought a Mobile Phone";
	}
}
public class MobileStoreReplacer implements MethodReplacer{
	public Object reimplement(Object arg0, Method arg1, Object[] arg2)
				throws Throwable {
		return "Bought an iPhone";
	}
 }
```

applicationContext.xml

```
<beans --- >
<bean name ="mobileStore" class ="com.apparao.method.injection.MobileStore">
		<replaced-method name="buyMobile" replacer="mobileStoreReplacer"/>
	</bean>
	<bean name ="mobileStoreReplacer" class  "com.apparao.method.injection.MobileStoreReplacer"/>
</beans>
```

```
public class ReplaceMethodDemo {
    public static void main(String[] args) {
    ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");

    //Method Replacer
    MobileStore mobileStore = (MobileStore)context.getBean("mobileStore");
    System.out.println(mobileStore.buyMobile());

    }
}
```

Output: Bought an iPhone

Lookup Method Injection

```
public  abstract class BookStore {

public abstract Book orderBook();
}
```

```
public interface Book {
public String bookTitle();
}
```

```
public class StoryBook implements
Book{
public String bookTitle() {
return "HarryPotter";  }
}
```

Managed by Spring

```
public class ProgrammingBook
   implements Book{
public String bookTitle() {
return "spring programming"; }
}
```

- The ability of the container to override methods on *container managed beans,* to return the lookup result for another named bean in the container.

```
package com.apparao.method.injection;
public abstract class BookStore {
	public abstract Book orderBook();

}
 package com.apparao.method.injection;
public interface Book {
	public String bookTitle();
}
 package com.apparao.method.injection;
public class ProgrammingBook implements Book{
	public String bookTitle() {
		return "spring programming";
	}
}
 package com.apparao.method.injection;
public class StoryBook implements Book{
	public String bookTitle() {
		return "HarryPotter";
	}
}
```

```
<!-- Look up method -->
    <bean name ="springBook" class ="com.apparao.method.injection.ProgrammingBook"/>

    <bean name ="book" class ="com.apparao.method.injection.BookStore">
    <lookup-method name="orderBook" bean ="springBook"/>
    </bean>

public class LookupMethodDemo {
    public static void main(String[] args) {
            ApplicationContext context = new
    ClassPathXmlApplicationContext("applicationContext.xml");

            //Look up Method
            BookStore book = (BookStore)context.getBean("book");

            System.out.println(book.orderBook().bookTitle());
    }
}
```

Output : spring programming

Limits:

Spring not supporting method injection feature following situations.

- Classes having final methods.
- Class is final class.