

# Spring Framework

# What is Spring Framework?

- The Spring Framework is an open source application framework for the Java platform.
- By design, the framework offers a lot of freedom to Java developers yet provides well-documented and easy to use solutions for common practices in the industry.
- Spring is a glue framework that gives an easy way of configuring and resolving dependencies throughout the J2EE stack
- It has IoC (Inversion of Control) to achieve the same apart from many more features.

# Core Package

- Core package is the most fundamental part of the framework and provides the IoC and Dependency Injection features
- The basic concept here is the BeanFactory, which --
  - provides a sophisticated implementation of the factory pattern
  - removes the need for programmatic singletons
  - allows to decouple the configuration and specification of dependencies from actual program logic

# Inversion of Control

```
public class FileReader { }  
public class Application {  
    // inflexible concrete dependency  
    private FileReader fileReader = new FileReader();  
}
```

# Inversion of Control

```
public interface Reader { }
public class FileReader implements Reader { }
public class Application {
    // flexible abstract dependency injected in
    private Reader reader;
    public Application(Reader reader) { // constructor injection
        this.reader = reader;
    }
    public void setReader(Reader reader) { // setter injection
        this.reader = reader;
    }
}
```

# Spring IOC Containers

- Spring container is at the core of the Spring Framework
- The container creates the objects, wire them together, configure them, and manage their complete life cycle
- The Spring container uses dependency injection (DI) to manage the components of an application (Beans)
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided
- The configuration metadata can be represented either by XML or Java annotations
- The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application

# Spring IOC Containers

Container & Description
<u>BeanFactory</u> This is the simplest container providing basic support for DI and defined by BeanFactory interface
<u>ApplicationContext</u> This container adds more functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners This container is defined by the ApplicationContext interface.

# Spring Bean Definition

- The objects that form the backbone of the application and that are managed by the Spring IoC container are called **beans**
- A bean is an object that is instantiated, assembled, and otherwise managed by a Spring IoC container
- These beans are created with the configuration metadata (XML or annotations)
- The bean definition contains the information which is needed for the container to know the followings:
  - How to create a bean
  - Bean's lifecycle details
  - Bean's dependencies



# Bean Configuration XML

<beans >

<!-- A simple bean definition -->

<bean id="..." class="..." scope="...">

<!-- collaborators and configuration for this bean go here -->

</bean>

<!-- A bean definition with lazy init set on -->

<bean id="..." class="..." lazy-init="true">

<!-- collaborators and configuration for this bean go here -->

</bean>

</beans>

# Bean Configuration XML contd

**<!-- A bean definition with initialization method -->**

**<bean id="..." class="..." init-method="...">**

**<!-- collaborators and configuration for this bean go here -->**

**</bean>**

**<!-- A bean definition with destruction method -->**

**<bean id="..." class="..." destroy-method="...">**

**<!-- collaborators and configuration for this bean go here -->**

**</bean>**

**</beans>**

# BeanFactory Example

- BeanFactory is an implementation of a Factory Design Pattern (Responsible for creation and dispensing of beans)
- Responsible for creation of many types of beans rather than a single type of bean
- Has knowledge about the objects instantiated within an application and is able to create association between the collaborating objects.
- `org.springframework.beans.factory.xml.XmlBeanFactory` is the most commonly used implementation of Bean Factory in Spring.

*BeanFactory factory =*

*new XmlBeanFactory (new FileInputStream("beans.xml"));*

*MyBean myBean = (MyBean) factory.getBean("myBean");*

# ApplicationContext

- To take full advantage of Spring framework, you may use ApplicationContext which provides wide range of application services.
- Apart from all BeanFactory services, it provides
  - Resolving Text Messages from properties file.
  - Generic way to load file resources and images
  - Publish events to bean that are registered as listeners.
- Common implementation of ApplicationContext are
  - **ClassPathXmlApplicationContext**: - Load context definition from xml located in ClassPath
  - **FileSystemXmlApplicationContext**: - Load context definition from xml located in File System.
  - **XmlWebApplicationContext**: - Load context definition from xml located in Web Application archive.

# ApplicationContext contd.

- Loading ApplicationContext from FileSystem

```
ApplicationContext applicationContext =  
new FileSystemXmlApplicationContext ("C:\foo.xml");
```

- Loading ApplicationContext from ClassPath

```
ApplicationContext applicationContext =  
new ClassPathXmlApplicationContext ("foo.xml");
```

- Loading multiple contexts

```
ApplicationContext context =  
new ClassPathXmlApplicationContext(  
    new String[] {"services.xml", "daos.xml"} );
```

# Beans

- The term “bean” is used to refer any component managed by the BeanFactory / ApplicationContext
- The “beans” are in the form of JavaBeans (in most cases)
  - public class
  - public no arg constructor
  - Optionally we can have other constructor
  - getter and setter methods for the properties
- Beans are singletons by default
- Properties of the beans may be simple values or references to other beans

# Bean creation

- Container finds the bean definition and instantiates the bean
- Using Dependency Injection Spring populates all of properties of bean specified in bean definition
- At this point bean is ready to be used by the application.

# Configuring Beans

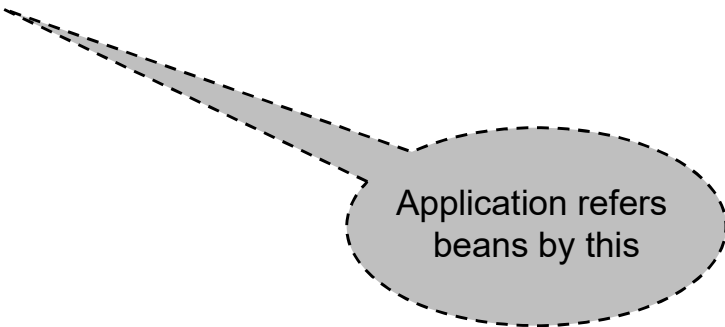
- Piecing together all the beans in the Spring Container is called **wiring**.
- Wiring is most preferably done through xml.
- Various BeanFactories and ApplicationContext objects which support wiring are
  - XmlBeanFactory
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - XmlWebApplicationContext



# Configuring Beans contd.

- Example

```
<? Xml version="1.0" encoding="UTF-8"?>  
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN// EN"  
    "http://www.springframework.org/dtd/spring-beans.dtd"?>  
<beans>  
    <bean id="customerBean" class="com.example.Customer"/>  
    <bean id="emp" class="com.springapp.Employee"/>  
</beans>
```



Application refers  
beans by this

# Example

- Steps to create first spring enabled application:
  1. Create an interface having declarations of the business methods.
  2. Create an implementation file which implements the interface created in step 1.
  3. Create the spring configuration file.
  4. Create the client.

# Beans wiring example

```
<beans>
```

```
  <bean id="weatherService" class="WeatherService">
```

```
    <property name="weatherDao">
```

```
      <ref bean="weatherDao"/>
```

```
    </property>
```

```
    <property name="provider" value="google"/>
```

```
  </bean>
```

```
  <bean id="weatherDao" class="WeatherDao">
```

```
  </bean>
```

```
</beans>
```

# Beans wiring example (contd)

```
public class WeatherService {  
    private WeatherDAO weatherDao;  
    private String provider;  
  
    public void setWeatherDao(WeatherDAO weatherDao) {  
        this.weatherDao = weatherDao;  
    }  
    public void setProvider(String pr) {  
        provider = pr;  
    }  
    .....  
}
```

# Types of Dependency Injection

- Setter Injection

In Setter injection, the injection is done via a setter method. IoC uses setter methods to get the dependent classes it needs.

- Constructor Injection.

In Constructor injection, IOC implementing class defines a constructor to get all its dependants. The dependent classes are defined in the constructor arguments.

# Setter Dependency Injection

```
public class SetterInjection {  
    private Dependency department;  
    public void setMyDepartment(Dependency dep) {  
        this.department = dep;  
    }  
}
```

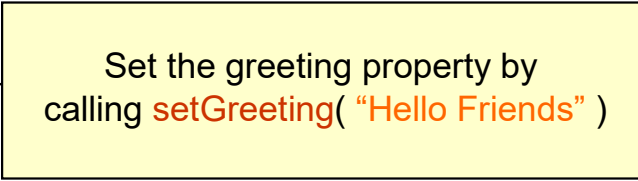
# Constructor Dependency Injection

```
public class ConstructorInjection {  
    private Dependency dep;  
    public ConstructorInjection(Dependency dep) {  
        this.dep = dep;  
    }  
}
```

# Setter Injection

- Configuration

```
<bean id="test" class="com.jp.TestBean">  
  <property name="greeting">  
    <value>Hello friends</value>  
  </property>  
</bean>
```



Set the greeting property by  
calling `setGreeting( "Hello Friends" )`



# Setter Injection

- Referencing other beans

```
<beans>
```

```
  <bean id="test" class ="com.jp.TestBean">
```

```
    <property name="greeting">
```

```
      <ref bean="greetBean"/>
```

```
    </property>
```

```
  </bean>
```

```
<bean id="greetBean" class ="com.jp.GreetBean" />
```

```
<beans>
```

# Setter Injection - example

```
<bean id="exampleBean" class="examples.ExampleBean">  
    <property name="beanOne"><ref bean="anotherBean"/></property>  
    <property name="beanTwo" ref="yetAnotherBean"/>  
    <property name="number" value="1"/>  
</bean>
```

```
<bean id="anotherBean" class="examples.AnotherBean"/>  
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

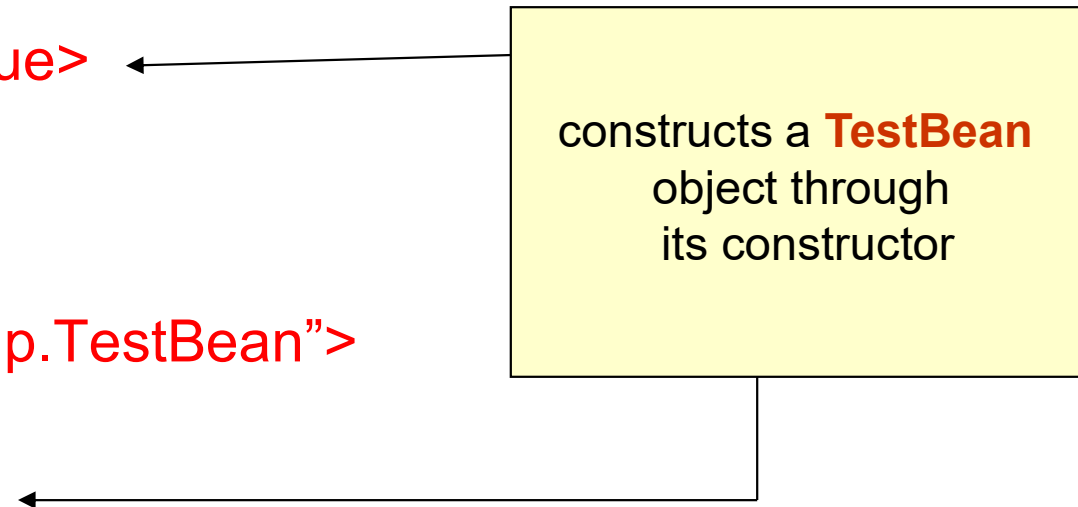
```
package examples;  
public class ExampleBean {  
    private AnotherBean beanOne;  
    private YetAnotherBean beanTwo;  
    private int val;  
    public void setBeanOne(AnotherBean beanOne) { this.beanOne = beanOne; }  
    public void setBeanTwo(YetAnotherBean beanTwo) { this.beanTwo = beanTwo; }  
    public void setNumber(int i) { this.val = i; }  
}
```

# Constructor Injection

- Configuration

```
<bean id="test" class="com.jp.TestBean">  
  <constructor-arg>  
    <value>Hello friends</value>  
  </constructor-arg>  
</bean>
```

```
<bean id="test" class="com.jp.TestBean">  
  <constructor-arg>  
    <ref bean="greetBean"/>  
  </constructor-arg>  
</bean>
```



constructs a **TestBean** object through its constructor

# Constructor Injection

- Constructor Injection (variations)

<beans>

<bean id="foo" class="x.y.One">

<constructor-arg ref="two"/>

<constructor-arg ref="three"/>

</bean>

<bean id="two" class="x.y.Two"/>

<bean id="three" class="x.y.Three"/>

</beans>

Simple  
Injection through  
its constructor

# Constructor Injection

- Constructor Injection (variations)

Injection through  
its constructor  
With data types of  
arguments

```
<beans>
```

```
<bean id="exampleBean" class="examples.ExampleBean">
```

```
  <constructor-arg type="int" value="7500"/>
```

```
  <constructor-arg type="java.lang.String" value="42"/>
```

```
</bean>
```

```
</beans>
```

# Constructor Injection

- Constructor Injection (variations)

Indexing helps  
If arguments are of  
same type

```
<beans>
```

```
  <bean id="exampleBean" class="examples.ExampleBean">
```

```
    <constructor-arg index="0" value="7500"/>
```

```
    <constructor-arg index="1" value="42"/>
```

```
  </bean>
```

```
</beans>
```

# Constructor Injection

- Constructor Injection (variations)

Argument names  
can also be used

```
<beans>
```

```
  <bean id="exampleBean" class="examples.ExampleBean">
```

```
    <constructor-arg name="years" value="34"/>
```

```
    <constructor-arg name="ultimateAnswer" value="42"/>
```

```
  </bean>
```

```
</beans>
```

# Wiring Collections

- Spring supports Many types of Collections as bean properties
- Supported types are:

XML	Types
<list>	java.util.List, arrays
<set>	java.util.Set
<map>	java.util.Map
<props>	java.util.Properties



# Wiring Lists and Arrays

```
<bean id="exampleSessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource"><ref bean="exampleDataSource"/>
    <property name="hibernateProperties">
      <ref bean="exampleHibernateProperties" />
    </property>
    <property name="mappingResources">
      <list>
        <value>Customer.hbm.xml</value>
        <value>Account.hbm.xml</value>
        <ref bean="bar2"/>
      </list>
    </property>
</bean>
```

# Wring Set, Map and Properties

```
<property name="testSet">
  <set>
    <value>value1</value>
    <ref bean="bar2"/>
  </set>
</property>
```

```
<property name="barMap">
  <map>
    <entry key="key1">
      <value>value1</value>
    </entry>
    <entry key="key2">
      <ref bean="bar2"/>
    </entry>
  </map>
</property>
```

```
<property name="barProperty">
  <props>
    <prop key="key1">value1</prop>
    <prop key="key2">value2</prop>
    <prop key="key3">value3</prop>
  </props>
</property>
```

# Bean Scopes

- **singleton**
  - Scopes the bean definition to a single instance per Spring container (default).
- **prototype**
  - Allows a bean to be instantiated any number of times
- **request**
  - Scopes a bean definition to an HTTP request. Only valid when used with a webcapable Spring context (Spring MVC).
- **session**
  - Scopes a bean definition to an HTTP session. Only valid when used with a webcapable Spring context (Spring MVC).
- **global-session**
  - Scopes a bean definition to a global HTTP session. Used in portlet application (each portlet has its own session)

# singleton VS prototype

- all spring beans are singleton
- prototype beans can also be defined
- Prototype beans are instantiated with each `getBean()` method call
- scope attribute can also be used to specify scope

# Bean lifecycle

- When a bean is instantiated, it may be required to perform some initialization to get it into a usable state
- Similarly, when the bean is no longer required and is removed from the container, some cleanup may be required
- To define setup and teardown for a bean, we simply declare the **init-method** and **destroy-method** parameters
- The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation
- Similarly, destroy-method specifies a method that is called just before a bean is removed from the container (only singletons)

# Initialization callbacks

- class can implement *org.springframework.beans.factory.InitializingBean* interface and specify `afterPropertiesSet()` method

```
public class ExampleBean implements InitializingBean {  
    public void afterPropertiesSet() {  
        // do some initialization work  
    }  
}
```

- In the case of XML-based configuration , you can use the **init-method** attribute to specify the name of the method that has a void no-argument signature

```
<bean id="exampleBean" class="examples.ExampleBean"  
    init-method="init"/>
```

# Destruction callbacks

- class can implement *org.springframework.beans.factory.DisposableBean* interface and specify `destroy()` method

```
public class ExampleBean implements DisposableBean {  
    public void destroy() {  
        // do some cleanup  
    }  
}
```

- In the case of XML-based configuration , you can use the **destroy-method** attribute to specify the name of the method that has a void no-argument signature

```
<bean id="exampleBean" class="examples.ExampleBean"  
    destroy-method="cleanup"/>
```

# Default callback methods

- If you have many beans having initialization and or destroy methods with the same name, you don't need to declare **init-method** and **destroy-method** on each individual bean
- framework provides the flexibility to configure such situation using **default-init-method** and **default-destroy-method** attributes on the <beans> element as follows:

```
<beans  
  default-init-method="init"  
  default-destroy-method="destroy">
```



# Lazy-initialized beans

- By default, all singleton beans created as part of the initialization process
- Generally, this pre-instantiation is desirable, because errors in the configuration are discovered immediately
- When this behaviour is *not* desirable, you can prevent pre-instantiation of a singleton bean by marking the bean definition as lazy-initialized
- A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup

# Lazy initialized beans - example

```
<bean id="lazy" class="com.foo.SlowBean" lazy-init="true"/>
```

```
<bean name="notLazy" class="com.foo.FastBean"/>
```

- the bean named `lazy` is not eagerly pre-instantiated when the `ApplicationContext` is starting up
- the `notLazy` bean is eagerly pre-instantiated

# Autowiring

- Beans may be auto-wired (rather than using <ref>)
  - Specify autowire attribute
- autowire="byName"
  - Bean identifier matches property name
- autowire="byType"
  - Type matches other defined bean
- autowire="constructor"
  - Match constructor argument types
- autowire="autodetect"
  - Attempt by constructor, otherwise "type"

# Autowiring - example

```
package common;
public class Customer {
    private Person person;
    public void setPerson(Person person) {
        this.person = person;
    }
    //...
}
public class Person {
    .....
}
```

```
<bean id="customer" class="common.Customer" autowire="byName" />
```

```
<bean id="person" class="common.Person" />
```

[illegible]

# Annotation based configuration

- An alternative to XML setups is provided by annotation-based configuration which rely on the bytecode metadata for wiring up components instead of XML declarations
- Instead of using XML to describe a bean wiring, the developer moves the configuration into the component class itself by using annotations on the relevant class, method, or field declaration
- Annotation injection is performed *before XML injection*, thus *the XML configuration will* override the annotations for properties wired through both approaches

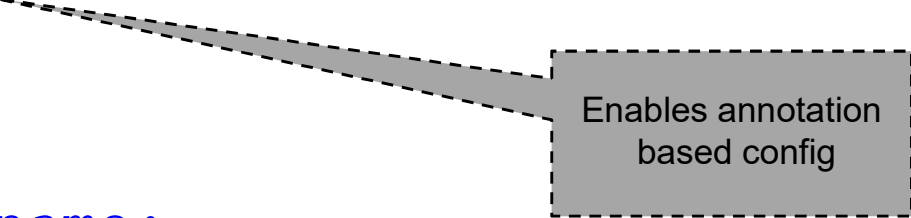
# Annotation based configuration

<u>Annotation</u>	<u>Use</u>	<u>Description</u>
@Autowired	Constructor, Field, Method	Declares a constructor, field, setter method, or configuration method to be autowired by type
@Qualifier	Field, Parameter	Guides autowiring to be performed by means other than by type
@Scope	Type	Specifies the scope of a bean, either singleton, prototype, request, session or some custom scope
@Required	Method (setters)	Specifies that a particular property must be injected or else the configuration will fail.

# Annotation @Autowired

```
<bean id="pirate" class="Pirate">  
    <constructor-arg value="Sameer Dhawan" />  
</bean>  
<context:annotation-config />
```

```
public class Pirate {  
    private String name;  
    private TreasureMap treasureMap;  
    public Pirate(String name){ this.name = name; }  
    @Autowired  
    public void setTreasureMap(TreasureMap treasureMap)  
    {  
        this.treasureMap = treasureMap;  
    }  
}
```



Enables annotation based config



# Annotation @Autowired

- @Autowired can be used on any method (not just setter methods)
- The wiring can be done through any method or member variables

@Autowired

```
public void directionsToTreasure(TreasureMap treasureMap) {  
    this.treasureMap = treasureMap;  
}
```

@Autowired

```
private TreasureMap treasureMap;
```

# Annotation @Qualifier

- To resolve any autowiring ambiguity, use the @Qualifier attribute with @Autowired.

```
public class MovieRecommender {  
    @Autowired  
    @Qualifier("main")  
    private MovieCatalog movieCatalog;  
    // ... }  
}
```

```
<bean id="main" class="example.SimpleMovieCatalog">
```

```
</bean>
```

```
<bean id="action" class="example.SimpleMovieCatalog">
```

```
</bean>
```

# Annotation @Required

- To ensure that a property is injected with a value, use the @Required annotation:

@Required

```
public void setTreasureMap(TreasureMap treasureMap) {  
    this.treasureMap = treasureMap;  
}
```

- In this case, the “treasureMap” property must be injected or else Spring will throw a BeanInitializationException and context creation will fail

# Annotations @PostConstruct @PreDestroy

- Specify the lifecycle methods

```
public class CachingMovieLister {  
  
    @PostConstruct  
    public void populateMovieCache() {  
  
        .....  
    }  
  
    @PreDestroy  
    public void clearMovieCache() {  
  
        .....  
    }  
}
```

# Annotation @Resource

- Similar to @Autowired but java EE standard

Example :

@Resource

```
public void createTemplate(DataSource myDataSource) {  
  
}
```

@Resource(name="myDataSource") for specific bean by name

# Annotations @Configuration

- @Configuration-annotated classes consist principally of @Bean-annotated methods that define instantiation, configuration, and initialization logic for objects that will be managed by the Spring IoC container
- Annotating a class with the @Configuration indicates that the class may be used by the Spring IoC container as a source of bean definitions

@Configuration

```
public class AppConfig {  
    @Bean  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

- the configuration above is exactly equivalent to the following Spring XML

```
<beans>
```

```
<bean name="transferService" class="com.acme.TransferServiceImpl"/>
```

```
</beans>
```

# Annotations @Configuration

- Default strategy for determining the name of a bean is to use the name of the @Bean method
- If explicit naming is desired, the name attribute may be used

@Configuration

```
public class AppConfig {  
    @Bean(name="service")    OR    @Bean(name={"transfer","bean2"})  
    public TransferService transferService() {  
        return new TransferServiceImpl();  
    }  
}
```

- the bean can be referred as transfer and bean2

# Annotation @Component

- Indicates that the annotated class is a component.
- Such classes are considered for auto-detection in classpath scanning
- Class name with first char converted to lowercase is bean ID
- Example :

@Component

```
public class StudentDao { }
```

```
public class StudentService {
```

```
    @Autowired
```

```
    StudentDao    studentDao;
```

```
}
```



# Annotation @Service

- Similar to @Component but indicates a service based on interface
- Example :

@Service

```
public interface EmpService { ----- }
```

@Service("empService")

```
public class EmpServiceImpl implements EmpService { ----- }
```

```
BeanFactory factory = .....
```

```
EmpService test = (EmpService) factory.getBean("empService");
```

# Annotation @Repository

- Similar to @Component.
- But used to identify beans in persistence layer
- Example :

```
public interface EmployeeDAO
{
    public EmployeeDTO createNewEmployee();
}
```

```
@Repository ("employeeDao")
public class EmployeeDAOImpl implements EmployeeDAO
{
    .....
}
```

# Component Scan

- Spring is able to auto scan, detect and instantiate beans from pre-defined project package based on the following annotations
  - @Component
  - @Service
  - @Configuration
  - @Repository
- Put “component-scan” in bean configuration file indicating base package for searching
- Example :

```
<context:component-scan base-package="com />
```
- *This will enable searching all packages under com*

# Annotation @Value

- you can also use the @Value annotation to inject values from a property file into a bean's attributes.
- Property file will be placed in classpath
- Example :

test.properties file

user.name = sony

user.age = 35

# Annotation @Value (contd)

- Refer these properties with @Value annotation
- Example :

```
@Component
public class User {

    @Value("${user.name}")
    private String name;

    @Value("${user.age}")
    private int age;

    .....

}
```

# Annotation @Value (contd)

- Register properties file with PropertyPlaceholderConfigurer in XML file
- Example :

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">  
  <property name="locations">  
    <list>  
      <value>test.properties</value>  
      <!-- List other property files here -->  
    </list>  
  </property>  
</bean>
```