SPRING IOC CONTAINER

NEELAM AGARWAL

OUTLINE

- Inversion of Control explained
- Spring IOC Container
- Spring Bean
- Container Overview
- Spring Bean Configuration
  - XML Based Spring Bean Configuration
  - Annotation Based Spring Bean Configuration
  - Java Based Spring Bean Configuration

INVERSION OF CONTROL

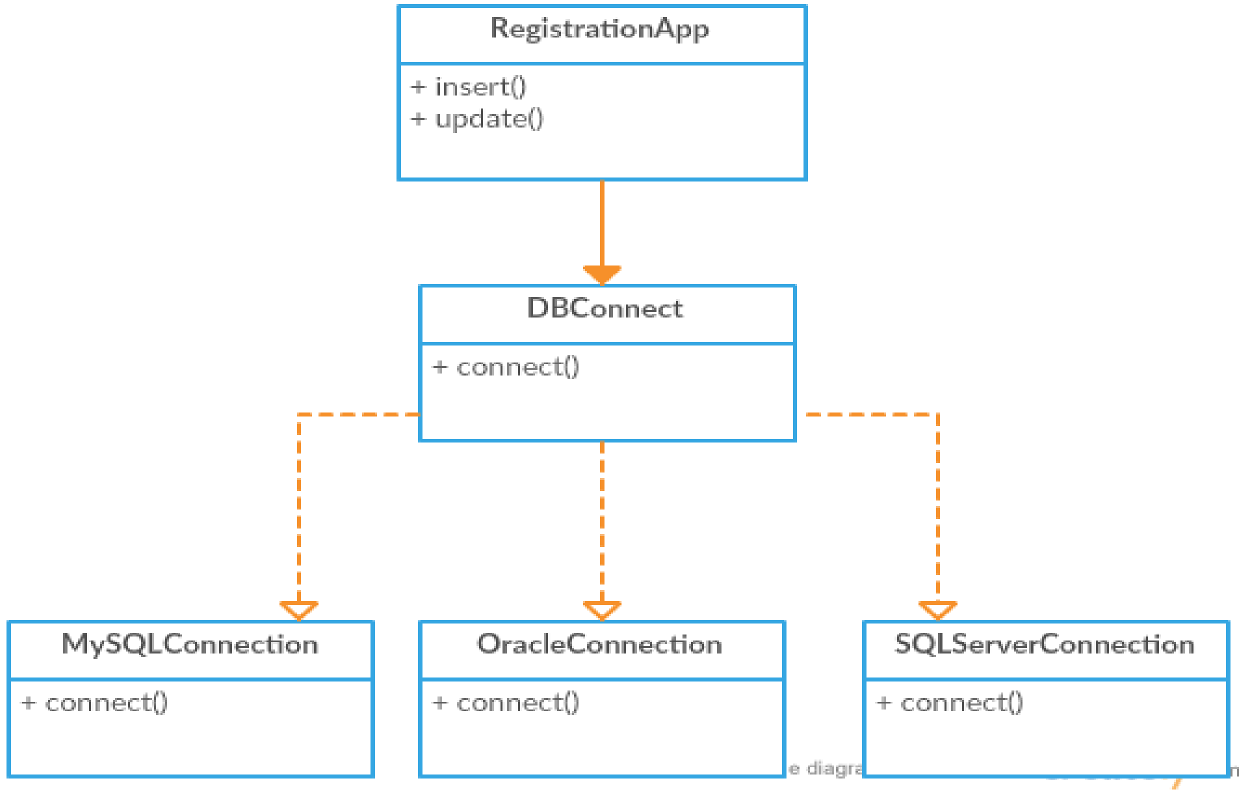
NEELAM AGARWAL

INVERSION OF CONTROL

- Object management inverted from Application Code to the Container
- A Design Pattern that says you do not create your objects but describe how they should be created.
  - You don't directly connect your components and services together in code but describe which services are needed by which components in a configuration file.
  - Help achieve loose coupling between Object Dependencies
- Dependency Injection is a specialized form of Inversion of Control.
- Object Dependencies are injected by other assembler objects.
- Example of IOC is explained in the upcoming slides

INVERSION OF CONTROL

Class Diagram for a Registration Application, which may need to connect to different Databases.



INVERSION OF CONTROL (IOC)

```
package com.training.spring;

public class RegistrationApp {
        public static void main(String[] args) {
                DBConnect dbcon = new MySQLConnection();
                dbcon.connect();
        }
}
```

Creating object of corresponding DB class

```
package com.training.spring;

public class RegistrationApp {
        public static void main(String[] args) {
                DBConnect dbcon = new OracleConnection();
                dbcon.connect();
        }
}
```

INVERSION OF CONTROL (IOC)

IOC reverse the process of Object creation. Container is going to provide us with the required class object

```
package com.training.spring;

public class RegistrationApp {
        public static void main(String[] args) {
        DBConnect dbcon = (DBConnect)Container.getComponent(args[0]);
        if(dbcon !=null)
                dbcon.connect();
        }
}
```

INVERSION OF CONTROL (IOC)

Container class Implementation

```java
package com.training.spring;

public class Container {
            private static Map<String, object> container;

            public synchronized static Object getComponent(final String componentName) {
                        if (container == null) {
                                    container = new HashMap<String, Object>();
                        }
                        Object result = container.get(componentName);
                        if (result == null) {
                                    if ("mysql".equals(componentName)) {
                                                result = new MySQLConnection();
                                    } else if ("oracle".equals(componentName)) {
                                                result = new OracleConnection();
                                    } else if ("sqlserver".equals(componentName)) {
                                                result = new SQLServerConnection();
                                    }
                                    if (result != null) {
                                                container.put(componentName, result);
                                    }
                        }
                        return result;
            }
}
```

SUMMEDUP

- DI is a process whereby objects define their dependencies through constructor arguments, setters or arguments to a factory method. The container then injects those dependencies when it creates the bean

- This process is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself controlling the instantiation or location of its dependencies by using direct construction of classes, or a mechanism such as the *Service Locator* pattern.

WHAT DOES IOC DO?

- Create new objects
- Configure/solve dependency among objects and assemble them
- Allow objects to be retrieved by id/name
- Manage object’s lifecycle
- Allow external configuration

WHY DO WE USE IOC?

- Achieve Loose coupling among Object Dependencies
- Reduce the amount of code in your application
- Does the plumbing work for you
- Application is more testable
- No more creating and hooking of objects together
- No more lookup

SPRING IOC CONTAINER

NEELAM AGARWAL

SPRING IOC CONTAINER

- Spring IOC Container is the program that injects dependencies into an object and make it ready for use.

- Packages for Spring IOC container
  - org.springframework.beans
  - org.springframework.context

- 2 types of IoC container implementation
  - BeanFactory
  - ApplicationContext

BEAN FACTORY

- BeanFactory interface provides an advanced configuration mechanism capable of managing any type of objects
- Provides the underlying basis for Spring’s IOC functionality.
- It is the root container that loads all the beans and provide dependency injection to enterprise applications
- Now largely historical in nature for most users of Spring.

APPLICATIONCONTEXT

- ApplicationContext is a subinterface of BeanFactory
- It adds easier integration with Spring AOP features, i18n, event publication, and application-layer specific context

USEFUL APPLICATIONCONTEXT IMPLEMENTATIONS

- **AnnotationConfigApplicationContext**: If we are using Spring in standalone java applications and using annotations for Configuration, then we can use this to initialize the container and get the bean objects.

- **ClassPathXmlApplicationContext**: If we have spring bean configuration xml file in standalone application, then we can use this class to load the file and get the container object.

- **FileSystemXmlApplicationContext**: This is similar to ClassPathXmlApplicationContext except that the xml configuration file can be loaded from anywhere in the file system.

- **AnnotationConfigWebApplicationContext** and **XmlWebApplicationContext** for web applications.

BEANFACTORY OR APPLICATIONCONTEXT?

Use an ApplicationContext unless you have a good reason for not doing so.

Feature	BeanFactory	ApplicationContext
Bean instantiation/wiring	Yes	Yes
Automatic BeanPostProcessor registration	No	Yes
Automatic BeanFactoryPostProcessor regist ration	No	Yes
Convenient MessageSource access (for i18n)	No	Yes
ApplicationEvent publication	No	Yes

SPRING BEANS

NEELAM AGARWAL

SPRING BEANS

- The objects that form the backbone of the application and that are managed by Spring IOC Container are called beans.
- A bean is an object that is instantiated, assembled and otherwise managed by a Spring IOC Container.

BEAN SCOPES

- Singleton
  - Default Scope
  - Only one instance of the bean will be created for each container
- Prototype
  - A new instance will be created every time the bean is requested
- Request
  - Same as prototype scope, but is used in Web Applications.
  - A new instance will be created for each HTTP request
- Session
  - A new bean will be created for each HTTP Session by the container
- Global-session
  - To create global session beans for Portlet applications

SPRING BEAN CONFIGURATION

NEELAM AGARWAL

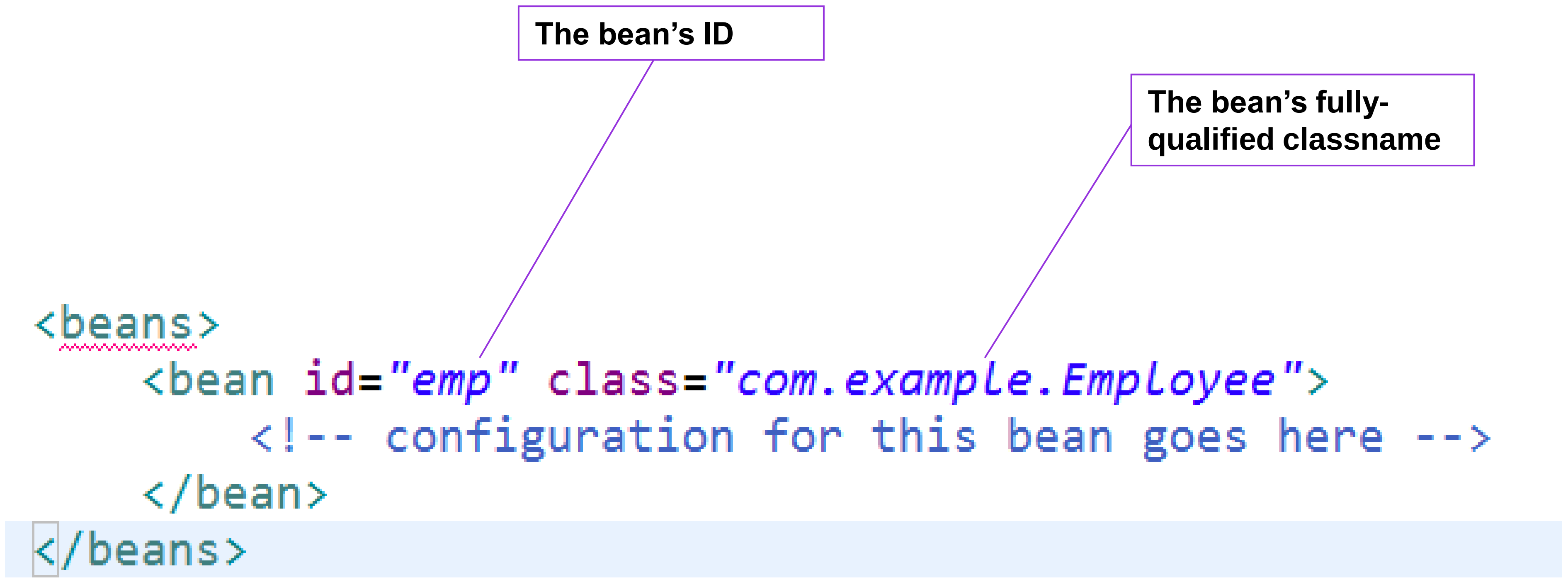
SPRING BEAN CONFIGURATION

- Spring provides three ways to configure beans to be used in applications
  - XML Based Configuration
    - By creating Spring Configuration XML file to configure the beans.
  - Annotation Based Configuration
    - Spring 2.5 introduced support for annotation-based configuration metadata. Base Container is still XML.
  - Java Based Configuration
    - Starting from Spring 3.0, we can configure Spring beans using java programs. Pure Java-based configuration. No need for having XML file for configuration Metadata

XML-BASED CONFIGURATION METADATA

- Root element: <beans>
- The XML contains one or more <bean> elements
  - id (or name) attribute to identify the bean
  - class attribute to specify the fully qualified class
- By default, beans are treated as singletons
- Can also be prototypes (non singletons)

XML-BASED CONFIGURATION METADATA

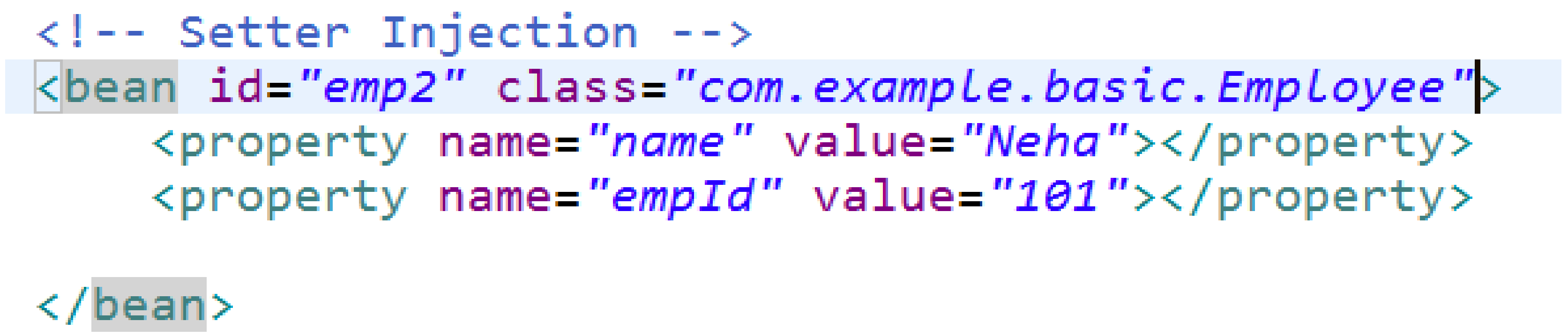


```xml
<beans>
    <bean id="emp" class="com.example.Employee">
        <!-- configuration for this bean goes here -->
    </bean>
</beans>
```

DEPENDENCY INJECTION

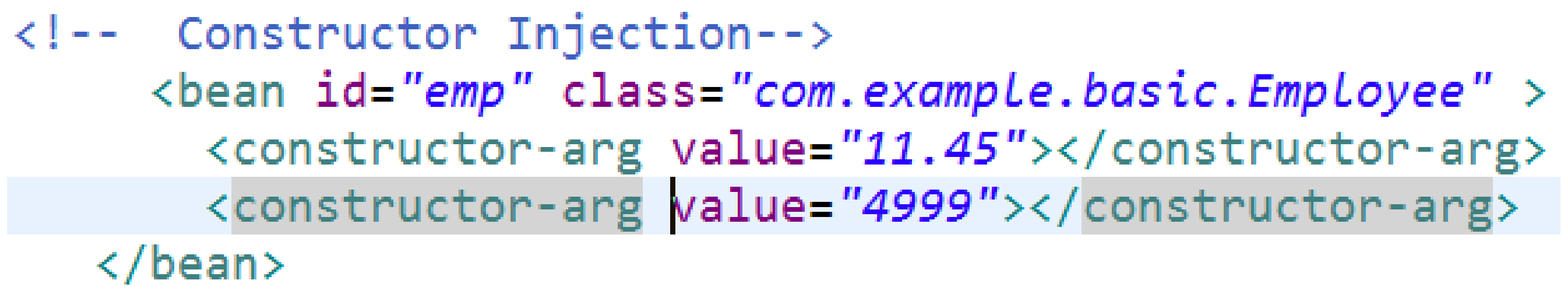
- Setter-Based
  - Dependencies are assigned through **JavaBeans** properties (for example, setter methods)
- Constructor-Based
  - Dependencies are provided as **constructor parameters** and are not exposed as JavaBeans properties
- Method-Based
  - The container is responsible for **implementing methods** at **runtime**

SETTER INJECTION



```
<!-- Setter Injection -->
<bean id="emp2" class="com.example.basic.Employee">
    <property name="name" value="Neha"></property>
    <property name="empId" value="101"></property>

</bean>
```

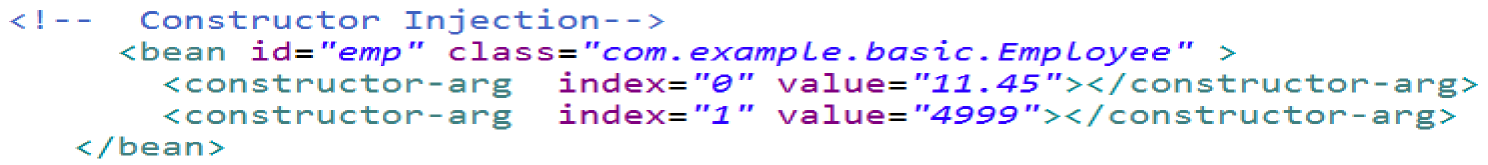
CONSTRUCTOR INJECTION



```
<!--  Constructor Injection-->
     <bean id="emp" class="com.example.basic.Employee" >
       <constructor-arg value="11.45"></constructor-arg>
       <constructor-arg value="4999"></constructor-arg>
   </bean>
```

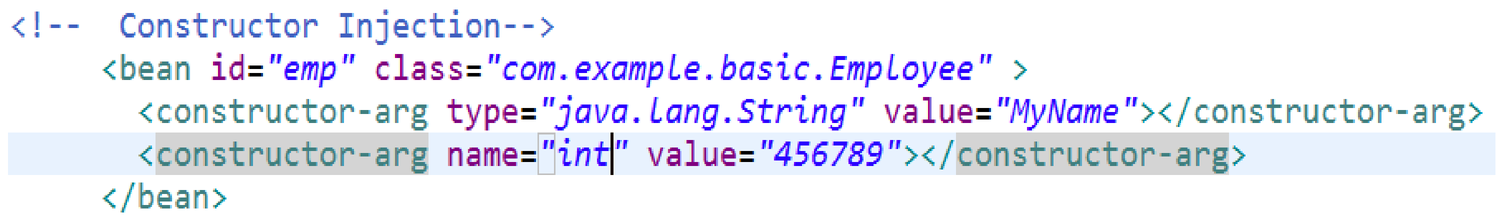
CONSTRUCTOR ARGUMENT RESOLUTION

- Index



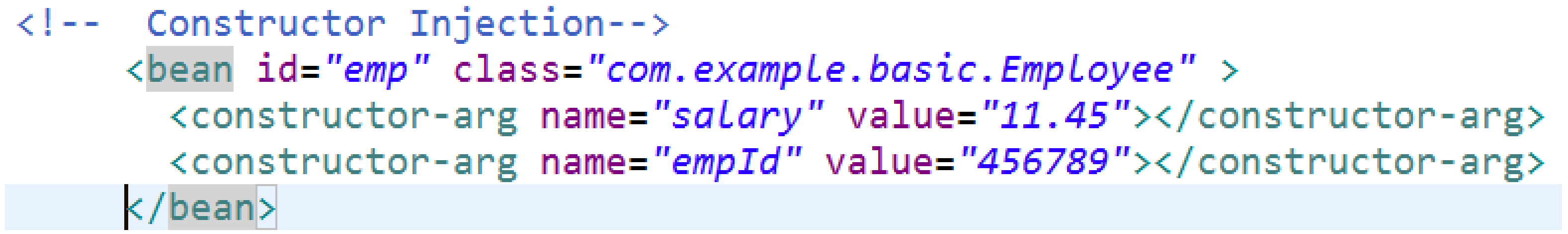
```
<!--  Constructor Injection-->
     <bean id="emp" class="com.example.basic.Employee" >
       <constructor-arg  index="0" value="11.45"></constructor-arg>
       <constructor-arg  index="1" value="4999"></constructor-arg>
   </bean>
```

- Type



```
<!--  Constructor Injection-->
     <bean id="emp" class="com.example.basic.Employee" >
       <constructor-arg type="java.lang.String" value="MyName"></constructor-arg>
       <constructor-arg name="int" value="456789"></constructor-arg>
     </bean>
```

- Name



```
<!--  Constructor Injection-->
     <bean id="emp" class="com.example.basic.Employee" >
       <constructor-arg name="salary" value="11.45"></constructor-arg>
       <constructor-arg name="empId" value="456789"></constructor-arg>
     </bean>
```

WHICH ONE TO CHOOSE?

POINTS IN FAVOR OF CONSTRUCTOR

- Constructor injection enforces a strong dependency contract. In short, a bean cannot be instantiated without being given all of its dependencies. It is perfectly valid and ready to use upon instantiation.
- Because all of the bean's dependencies are set through its constructor, there's no need for superfluous setter methods. This helps keep the lines of code at a minimum.
- By only allowing properties to be set through the constructor, you are, in effect, making those properties immutable.

POINTS IN FAVOR OF SETTER

- If a bean has several dependencies, the constructor's parameter list can be quite lengthy.
- If there are several ways to construct a valid object, it can be hard to come up with unique constructors since constructor signatures vary only by the number and type of parameters.
- If a constructor takes two or more parameters of the same type, it may be difficult to determine what each parameter's purpose is.
- 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.

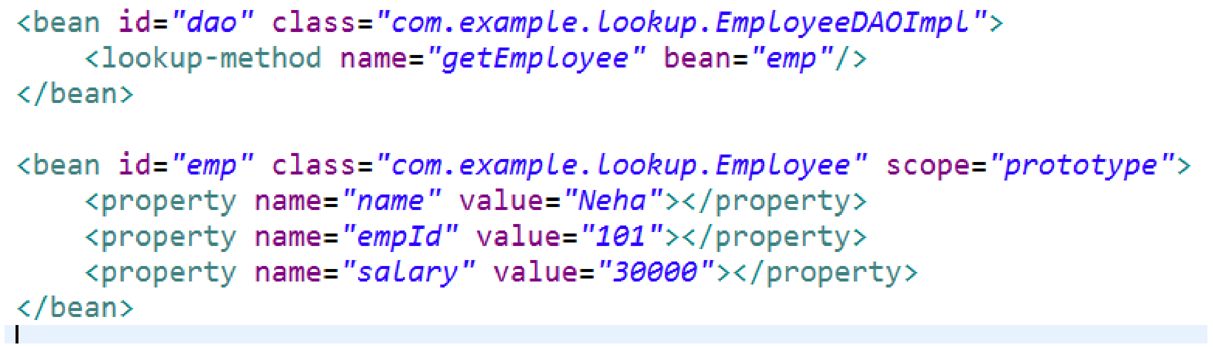
CONSTRUCTOR-BASED VS SETTER-BASED DI

- Tips : Use constructor arguments for mandatory dependencies and setters for optional dependencies
- More properties, more arguments to constructor
- Hence the Spring team generally advocates setter injection

METHOD-BASED INJECTION

- Useful when a singleton bean needs to use a non-singleton bean

- Using CGLIB library, Spring generates dynamically a subclass and overrides the look up method



```xml
<bean id="dao" class="com.example.lookup.EmployeeDAOImpl">
    <lookup-method name="getEmployee" bean="emp"/>
</bean>

<bean id="emp" class="com.example.lookup.Employee" scope="prototype">
    <property name="name" value="Neha"></property>
    <property name="empId" value="101"></property>
    <property name="salary" value="30000"></property>
</bean>
```

- Spring overrides the getEmployee() using lookup-method injection to provide a new instance of a Employee every time that method is called

METHOD BASED INJECTION

- Look-up method must be as follows
  - <public|protected> [abstract] <return-type> theMethodName(no-arguments)

- We need an abstract method which will be configured as a lookup-method in the configuration file.

- Spring will generate a proxy around which will implement the abstract method and return the object of the target bean. Again used only if scopes of both the beans are different

- Also you must have the CGLIB jar(s) in your classpath

AUTOWIRING COLLABORATORS

- Spring can resolve collaborators (other beans) automatically for your bean by inspecting the contents of the ApplicationContext
- Advantages:
  - Reduces the need to specify properties or constructor arguments
  - New dependencies can be added to a class without changing the configuration

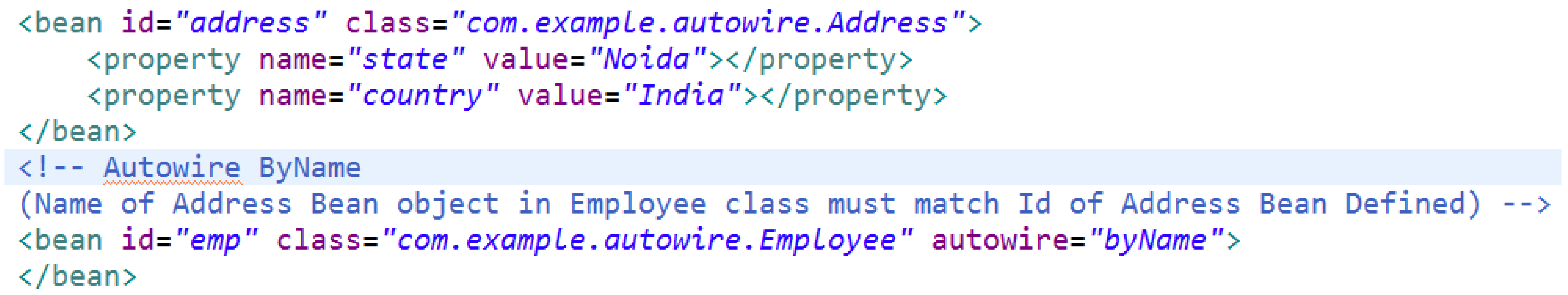
AUTOWIRING MODES

Mode	Explanation
No (Default)	No autowiring
byName	Autowiring by property name. Spring looks for a bean with the same name as the property that needs to be autowired
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.
constructor	Analogous to byType, but applies to constructor arguments.
autodetect	If a default constructor with no argument is found, the dependencies will be auto-wired by type. Otherwise, they will be auto-wired by constructor

- Note :Autowiring works best when it is used consistently across a project

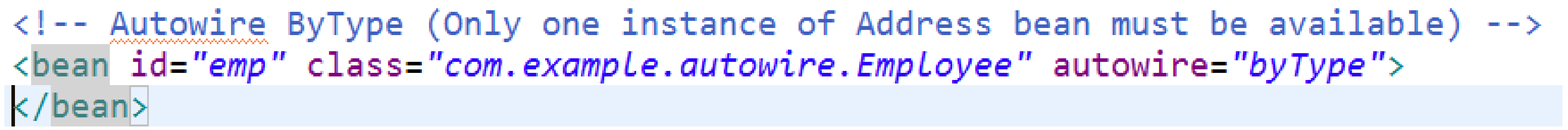
AUTOWIRING EXAMPLE

Autowire by Name



```xml
<bean id="address" class="com.example.autowire.Address">
    <property name="state" value="Noida"></property>
    <property name="country" value="India"></property>
</bean>
<!-- Autowire ByName
(Name of Address Bean object in Employee class must match Id of Address Bean Defined) -->
<bean id="emp" class="com.example.autowire.Employee" autowire="byName">
</bean>
```

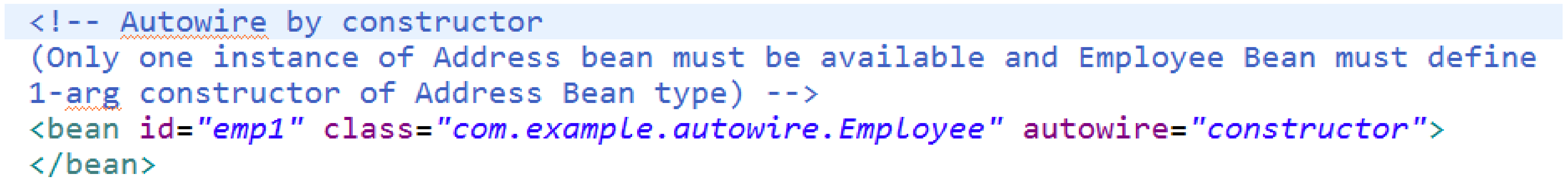
Autowire by Type



```xml
<!-- Autowire ByType (Only one instance of Address bean must be available) -->
<bean id="emp" class="com.example.autowire.Employee" autowire="byType">
</bean>
```

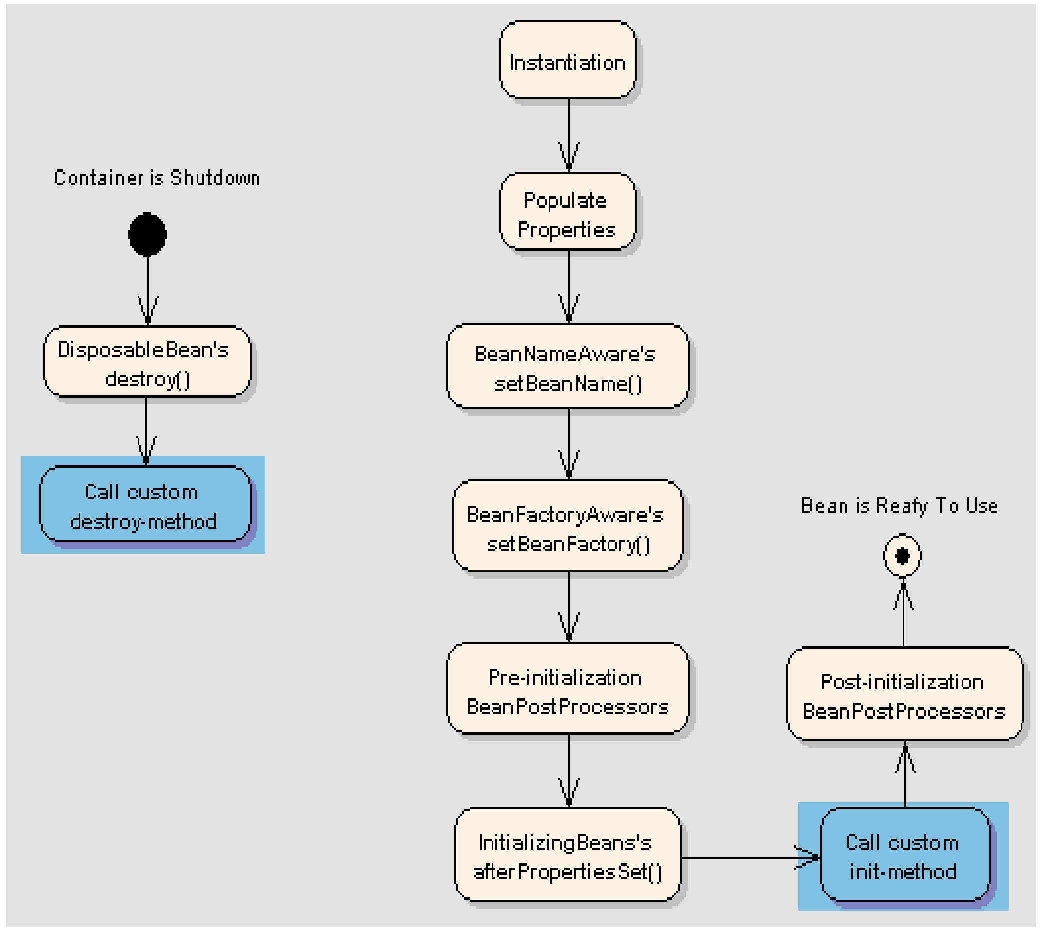
AUTOWIRING EXAMPLE

Autowire Constructor



```xml
<!-- Autowire by constructor
(Only one instance of Address bean must be available and Employee Bean must define
1-arg constructor of Address Bean type) -->
<bean id="emp1" class="com.example.autowire.Employee" autowire="constructor">
</bean>
```

LIFECYCLE OF BEANS


Instantiation
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBeans's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is Reafy To Use
Container is Shutdown
DisposableBean's destroy()
Call custom destroy-method

LIFECYCLE CALLBACKS

- Three ways to interact with the container’s bean lifecycle management

  - By implementing the InitializingBean and DisposableBean interfaces

  - Using init-method and destroy-method attributes in the bean definition

    (if you don’t want your classes coupled to Spring interfaces)

  - @PostConstruct and @PreDestroy annotations (More on this later)

INITIALIZATION CALLBACKS

- Implement the InitializingBean interface and override afterPropertiesSet() method
- Container calls this method upon initialization of your beans

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

- Alternatively specify a POJO initialization using the init-method attribute

```
public class Employee {
public void init() {
// do some initialization work
      }
}
```

```
<bean id=“emp”  class=“com.example.lifecycle.Employee“ init-method="init"/>
```

DESTRUCTION CALLBACKS

- Implement the DisposableBean interface and override destroy() method
- Container calls this method upon destruction of your beans

```
public class Employee implements DisposableBean{
        public void destroy() {
//do some destruction work (like releasing pooled connections)   }
}
```

- Alternatively specify a POJO initialization using destroy-method attribute

```
public class Employee {
        public void cleanup() {
                        // do some destruction work
        }
}
```

```
<bean id="emp" class="com.example.lifecycle.Employee" destroy-method="cleanup"/>
```

ANNOTATION BASED CONTAINER CONFIGURATION

NEELAM AGARWAL

TO CONFIGURE SPRING - ANNOTATION VS XML

- Annotations

  + More concise configuration

  + Wiring is more close to the source

  - Configuration becomes decentralized and harder to control

- XML

  + Wiring done without touching source code or recompiling

  + A centralized location for configuration

  - Can become verbose

- It is up to the developer to decide the strategy that suits better

CAN WE USE BOTH?

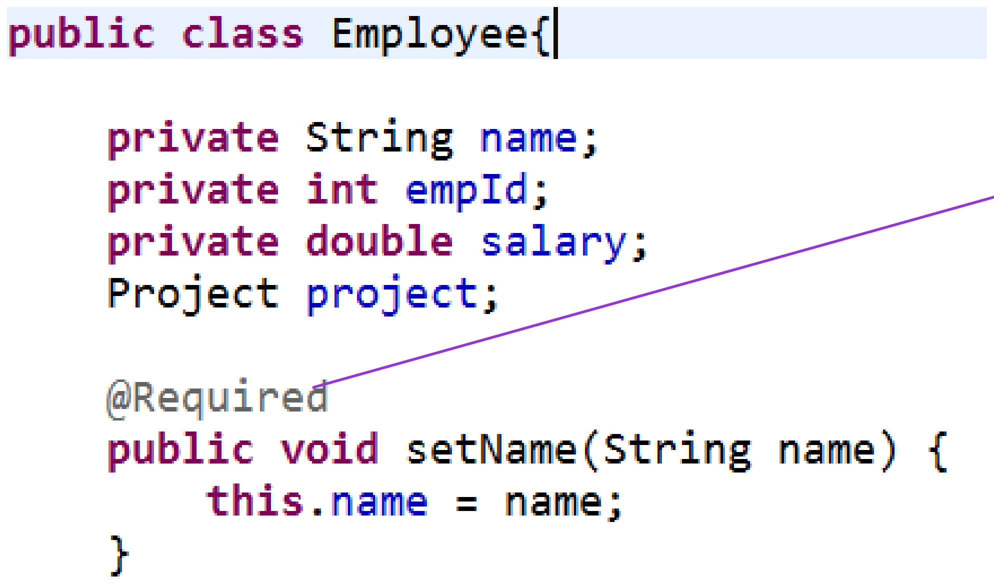
- Spring supports mix of both
- But Annotation injection is performed *before* XML injection
- Hence XML-based injection will override Annotation-based injection for properties wired through both approaches

ANNOTATIONS INTRODUCED IN SPRING

- Spring 2.0
  - *@Required*
- Spring 2.5
  - *@Autowired*
  - *@Resource, @PostConstruct, @PreDestroy*
- Spring 3.0
  - *@Inject, @Qualifier, @Named, and @Provider*

@REQUIRED

- Applies to bean property setter methods
- Indicates that the affected bean property must be populated at configuration time
- Throws an exception if it has not been set



```
public class Employee{

    private String name;
    private int empId;
    private double salary;
    Project project;

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

@Required use. Must use setter injection to set the value

@AUTOWIRED

- Can be used in the Java source code for specifying DI requirement

- Places where @Autowired can be used
  - Fields
  - Setter methods
  - Constructor methods
  - Arbitrary methods

- Need to include the below element in the bean configuration file to use this
  - <context:annotation-config>

- Because autowiring by type may lead to multiple candidates, it is necessary to have more control over the selection process
- One way to accomplish this is with Spring's *@Qualifier annotation.*

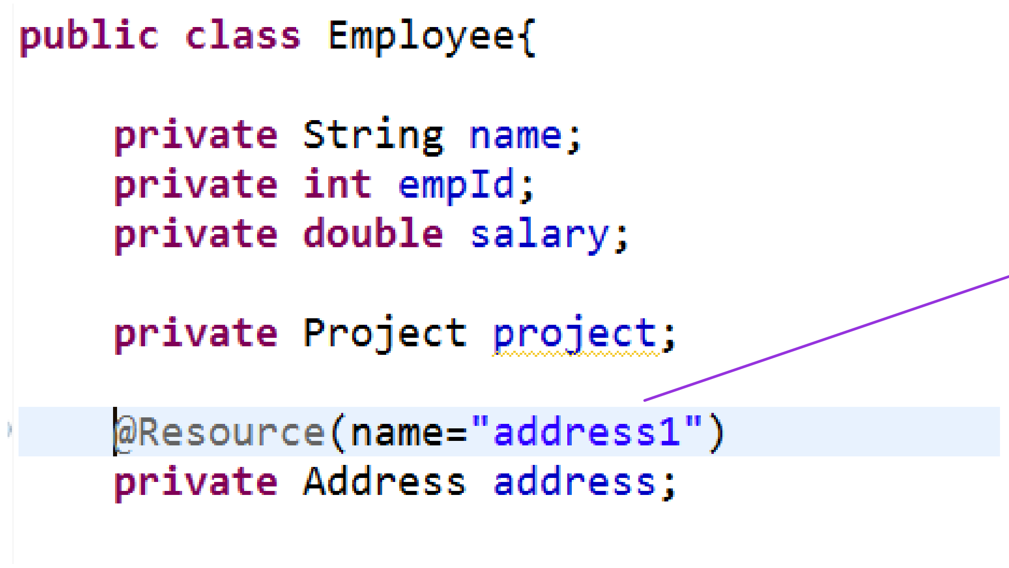
```
@Autowired
@Qualifier("address1")
public void setAddress( Address address) {
    this.address = address;
}

@Autowired
public void assignValues
(@Qualifier("address1") Address address,Project project) {
    this.address = address;
    this.project = project;
}
```

Of all the beans of Address class, it uses one matching the address1 id.

@Qualifier can be used with parameter names as well.

@RESOURCE

- Spring also supports injection using the @Resource on fields or bean property setter methods
- It takes a 'name' attribute, and by default Spring will interpret that value as the bean name to be injected i.e., it follows by-name semantics

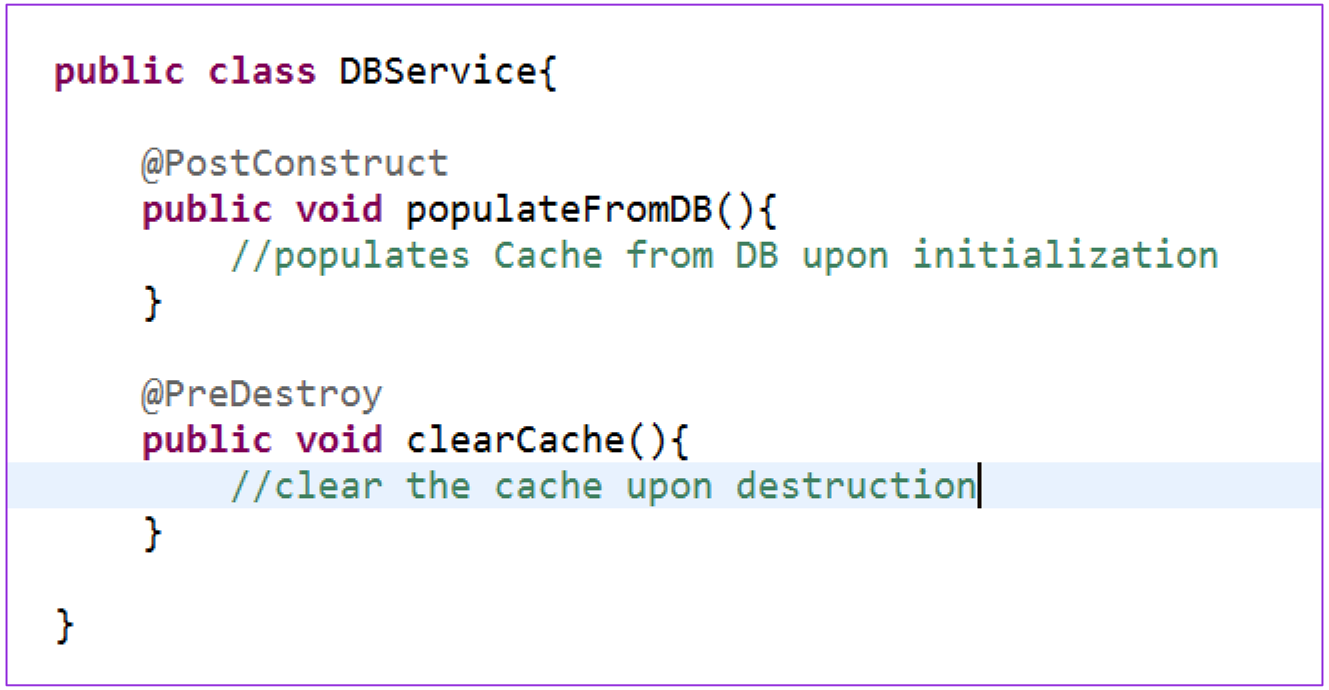


```
public class Employee{

    private String name;
    private int empId;
    private double salary;

    private Project project;

    @Resource(name="address1")
    private Address address;
```

Must have a bean with id – address1 of type Address defined in config file.

@POSTCONSTRUCT AND @PREDESTROY

- An alternative to initialization callbacks and destruction callbacks



```java
public class DBService{

    @PostConstruct
    public void populateFromDB(){
        //populates Cache from DB upon initialization
    }

    @PreDestroy
    public void clearCache(){
        //clear the cache upon destruction
    }

}
```

JAVA BASED CONTAINER CONFIGURATION

NEELAM AGARWAL

@CONFIGURATION AND @BEAN

- Class with *@Configuration* indicates that the class can be used as a source of bean definitions
- @Bean-annotated methods define instantiation, configuration, and initialization logic for objects to be managed by the Spring IoC container

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

- This is equivalent to

```
<beans>
<bean id="myService" class="MyServiceImpl"/>
</beans>
```

ANNOTATIONCONFIGAPPLICATIONCONTEXT

- An ApplicationContext implementation
- Uses @Configuration classes as input

```
public static void main(String[] args) {

ApplicationContext ctx = new
AnnotationConfigApplicationContext(AppConfig.class);

MyService myService = ctx.getBean(MyService.class);

myService.doStuff();
}
```

@CONFIGURATION AND @BEAN

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

This is same as:

```
<bean id = "accountRepository"
class = “InMemoryAccountRepository”></bean>
<bean id = "transferService“ class = “TransferServiceImpl">
<property name="accountRepository" ref="accountRepository"/>
</bean>
```

SCANNING COMPONENTS FROM CLASSPATH

SCANNING COMPONENTS FROM THE CLASSPATH

- So far the “base" bean definitions are explicitly defined in the XML file, while the annotations only drive the dependency injection
- But *Component Scanning avoids* manual configuration
- It can automatically scan, detect, and instantiate your components with particular stereotype annotations from the classpath
- The basic annotation denoting a Spring-managed component is @Component
- Other stereotypes include @Repository, @Service, and @Controller denoting components in the persistence, service, and presentation layers, respectively

AUTOMATICALLY DETECTING CLASSES AND REGISTERING BEAN DEFINITIONS

```java
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private DBImpl dao;
}
```

```java
@Repository
public class DBImpl implements DBInterface {

    @Autowired
    DataSource datasource;

}
```

To Add in Configuration XML File other than DataSource Configuartion

common parent package for the two classes

```xml
<context:component-scan base-package="com.config"></context:component-scan>
```