Bean Management

By Rahul Barve

Objectives

- Working with ApplicationContext
- Bean Life Cycle
- Understanding Dependency Injections
- Bean Scopes
- Auto-wiring

ApplicationContext

- In a Spring-based application, the application objects live within the Spring Container.
  - The container creates the objects, wire them together, configure them, and manage their lifecycle.

ApplicationContext

- The *ApplicationContext* interface standardizes the Spring bean container behavior.
  - Its implementations form the simple container, providing basic support for DI.
  - Supports I18N.
  - Supports Event Management.

ApplicationContext

- Spring comes with several flavors of *ApplicationContext*.
  - *FileSystemXmlApplicationContext*
  - *ClassPathXmlApplicationContext*
  - *XmlWebApplicationContext*

FileSystemXmlApplicationContext

- Loads a context definition from an XML file located in the file system.
- E.g.

```
ApplicationContext ctx;
String file = “c:/config.xml”;
ctx =
new
FileSystemXmlApplicationContext(file);
```

ClassPathXmlApplicationContext

- Loads a context definition from an XML file located in the classpath.
- E.g.

```
ApplicationContext ctx;
String file = “config.xml”;
ctx =
new ClassPathXmlApplicationContext(file);
```

XmlWebApplicationContext

- Loads a context definition from an XML file contained within a web application.
- Used in Spring MVC environment.

Bean Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <bean id="emp"
  class="com.emp.Employee">
  </bean>
</beans>
```

Bean Configuration File

- **bean:** The Element which is the most basic configuration unit in Spring. It tells Spring Container to create an Object.
- **id**: The Attribute which gives the bean a unique name by which it can be accessed.
- **class**: The Attribute which tells Spring the type of a Bean.

Accessing Bean

```
ApplicationContext ctx;
String file = “c:/config.xml”;
ctx                    =                    new
FileSystemXmlApplicationContext(file);

GreetingService service;
service                                        =
(GreetingService)ctx.getBean(“greet”);
```

How DI/IoC Container works

- In Inversion Of Control (IoC), control is inverted back to the Container to support the object dependencies.
- IoC container creates the POJO objects and provides dependencies to them.
  - These POJO objects are not tied to any framework.
- The declarative configuration for POJO objects is defined with unique identities (id) in XML.
  - These are known as bean definitions.

How DI/IoC Container works

- The IoC container at runtime identifies POJO bean definitions, creates bean objects and returns them to the Application.
- IoC container manages dependencies of the objects.

Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.

Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.
  - Create the POJO objects by using parameterized constructors and injecting the dependent properties through the constructor.

Injecting the Dependencies

- The IoC container will Inject the dependencies in three ways
  - Create the POJO objects by using no-argument constructor and injecting the dependent properties by calling the setter methods.
  - Create the POJO objects by using parameterized constructors and injecting the dependent properties through the constructor.
  - Implements the method Injection.

Injecting Properties by calling Setters

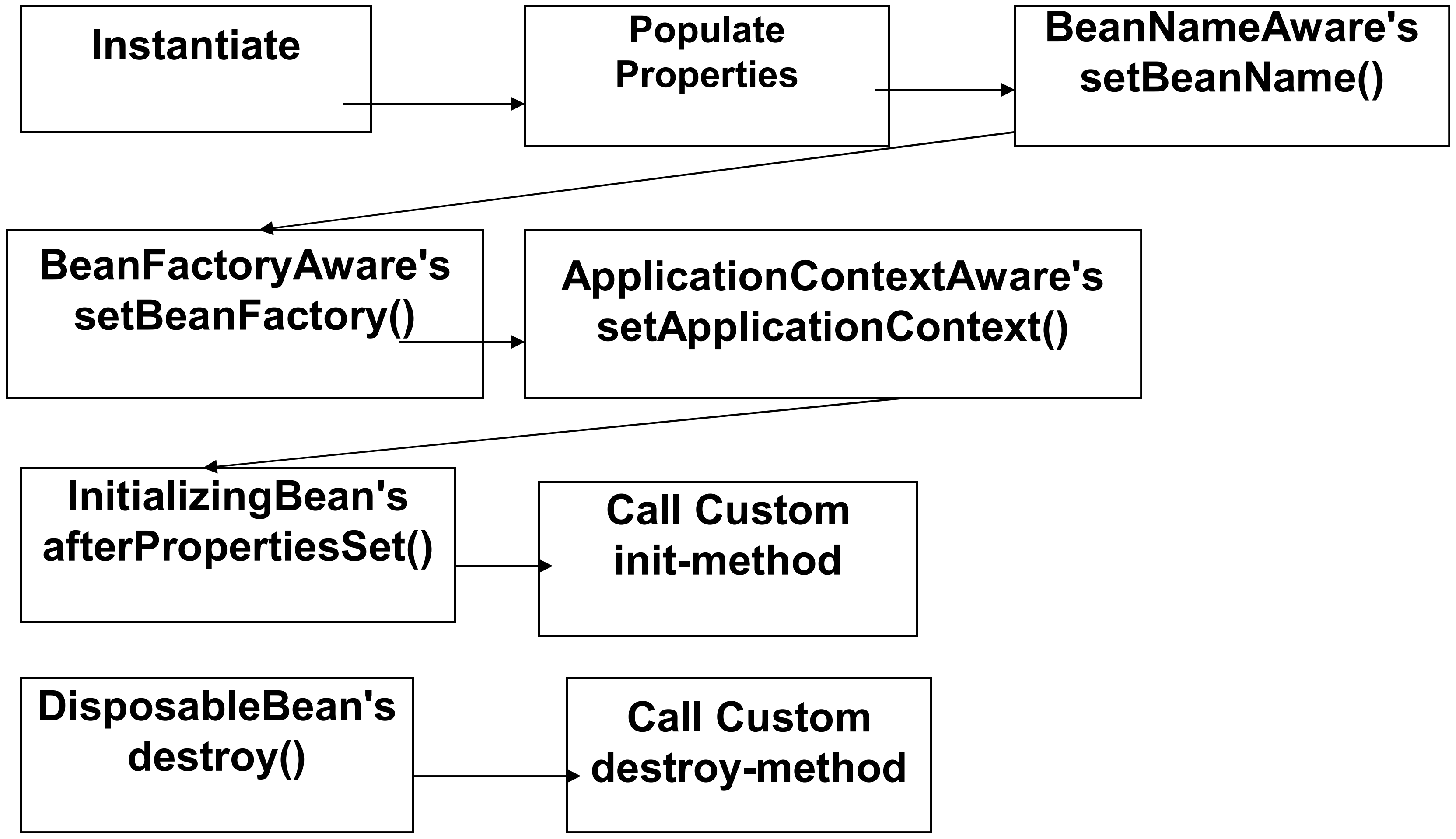
```
public Class Employee {
     private String fname, lname;
  public Employee(){}

  public void setFname(String f) {
              fname = f;
   }
  public void setLname(String l) {
              lname = l;
    }
}
```

Injecting Properties by calling Setters

```
<bean id="emp"
class="com.emp.Employee">
   <property name="fname" value="a"/>
  <property name="lname" value="b"/>
</bean>
```

Bean LifeCycle

Bean LifeCycle


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
InitializingBean's afterPropertiesSet()
Call Custom init-method
DisposableBean's destroy()
Call Custom destroy-method

Bean Life Cycle

- Instantiate: Spring instantiates the bean.
- Populate properties: Spring injects the bean’s properties.
- Set bean name: If the bean implements `BeanNameAware`, Spring passes the bean’s ID to `setBeanName()`.

Bean Life Cycle

- Set Application Context: If the bean implements `ApplicationContextAware`, Spring passes the application context to `setApplicationContext()`.
- Postprocessor (before initialization): If there are any BeanPostProcessors, Spring calls their `postProcessBeforeInitialization()` method.

Bean Life Cycle

- 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.
- Postprocessor (after initialization): If there are any BeanPostProcessors, Spring calls their `postProcessAfterInitialization()` method.

Bean Life Cycle

- 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.
- 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.

Injecting Properties through Constructors

```
public Class Employee {
  private String fname, lname;
  public Employee(String f,String l ){
       fname = f;
        lname = l;
   }
}
```

Injecting Properties through Constructors

```
<bean id="emp"
    class="com.emp.Employee">
  <constructor-arg value="Joe"/>
  <constructor-arg value="Thomas"/>
</bean>
```

Dependent Beans

Dependent Beans

- A bean is a dependency of another bean, is expressed by the fact that, one bean is set as a property of another.

Dependent Beans

- It is achieved with the `<ref/>` element or `ref` attribute in XML based configuration metadata of beans.