SPRING INVERSION OF CONTROL (IOC) AND DEPENDENCY INJECTION

1. Inversion of Control (IoC)

- Inversion of Control is a principle in software engineering which transfers the control of objects or portions of a program to a container or framework.
- Inversion of control is a design guideline.
- IoC serves the following purposes:
  - There is a decoupling of the execution of a certain task from implementation.
  - Every module can focus on what it is designed for.
  - Modules make no assumptions about what other systems do but rely on their contracts.
  - Replacing modules has no side effect on other modules.

2. Ways to implement IoC

In object-oriented programming, there are several basic techniques to implement inversion of control. These are:

- Using a factory pattern.
- Using a service locator pattern.
- Using a dependency injection of any given below type:
  - a constructor injection
  - a setter injection
  - an interface injection

3. Inversion of Control in Spring

In the Spring framework, the interfaces BeanFactory and ApplicationContext represents the ***IoC container***. The Spring IoC container is responsible for instantiating, configuring and assembling objects known as beans, as well as managing their life cycles.

4. BeanFactory

- The BeanFactory is the actual representation of the Spring IoC container that is responsible for containing and otherwise managing the aforementioned beans.
- The BeanFactory interface is the central IoC container interface in Spring.
- BeanFactory also takes part in the life cycle of a bean, making calls to custom initialization and destruction methods.
- The BeanFactory interface provides an advanced configuration mechanism capable of managing objects of any nature.
- There are a number of implementations of the BeanFactory interface. The most commonly used BeanFactory implementation is the XmlBeanFactory class.

```
Resource res = new FileSystemResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);

or

ClassPathResource res = new ClassPathResource("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(res);
```

5. ApplicationContext

- The ApplicationContext interface builds on top of the BeanFactory (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, and application-layer specific contexts such as the WebApplicationContext for use in web applications.
- Implementation: ClassPathXmlApplicationContext, FileSystemXmlApplicationContext, XmlWebApplicationContext
  - ClassPathXmlApplicationContext : It Loads context definition from an XML file located in the classpath, treating context definitions as classpath resources. The application context is loaded from the application's classpath by using the code.
  - FileSystemXmlApplicationContext : It loads context definition from an XML file in the filesystem. The application context is loaded from the file system by using the code.
  - XmlWebApplicationContext : It loads context definition from an XML file contained within a web application.

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
new String[] {"applicationContext.xml", "applicationContext-part2.xml"});


Or

ApplicationContext appContext = new ClassPathXmlApplicationContext("bean.xml");

Or

ApplicationContext appContext = new FileSystemXmlApplicationContext("bean.xml");

And then

BeanFactory factory = (BeanFactory) appContext;
```

6. What should be used preferably BeanFactory or ApplicationContext?

- Normally when building most applications in a J2EE-environment, the best option is to use the ApplicationContext, since it offers all the features of the BeanFactory and adds on to it in terms of features, while also allowing a more declarative approach to use of some functionality, which is generally desirable.
- The main usage scenario when you might prefer to use the BeanFactory is when memory usage is the greatest concern (such as in an applet where every last kilobyte counts), and you don't need all the features of the ApplicationContext.

7. Dependency Injection

- Dependency injection is a pattern we can use to implement IoC, where the control being inverted is setting an object's dependencies.
- Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality.
- IoC relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.
- Connecting objects with other objects, or “injecting” objects into other objects, is done by an assembler rather than by the objects themselves.
- The basic principle behind Dependency Injection (DI) is that objects define their dependencies only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually inject those dependencies when it creates the bean.

8. Ways for Dependency Injection

- Setter Injection
  Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

```
public class TestSetterDI {

    DemoBean demoBean = null;

    public void setDemoBean(DemoBean demoBean) {
        this.demoBean = demoBean;
    }
}
```

- Constructor Injection
  Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator.

```
public class ConstructorDI {

    DemoBean demoBean = null;

    public TestSetterDI (DemoBean demoBean) {
        this.demoBean = demoBean;
    }
}
```

- Interface Injection
  In this methodology we implement an interface from the IOC framework. IOC framework will use the interface method to inject the object in the main class. It is much more appropriate to use this approach when you need to have some logic that is not applicable to place in a property. Such as logging support.

```
public void SetLogger(ILogger logger){
  _notificationService.SetLogger(logger);
  _productService.SetLogger(logger);
}
```

- Field Injection
  Through reflection, directly into fields by using @Autowired directly on your field.

```
public void SetLogger(ILogger logger){
   _notificationService.SetLogger(logger);
   _productService.SetLogger(logger);
}
```

9. Injection guidelines

A general guideline, which is recommended by Spring is the following:

- For mandatory dependencies or when aiming for immutability, use constructor injection
- For optional or changeable dependencies, use setter injection
- Avoid field injection in most cases

10. Field injection drawbacks

The reasons why field injection is frowned upon are as follows:

- You cannot create immutable objects, as you can with constructor injection
- Your classes have tight coupling with your DI container and cannot be used outside of it
- Your classes cannot be instantiated (for example in unit tests) without reflection. You need the DI container to instantiate them, which makes your tests more like integration tests
- Your real dependencies are hidden from the outside and are not reflected in your interface (either constructors or methods)
- It is really easy to have like ten dependencies. If you were using constructor injection, you would have a constructor with ten arguments, which would signal that something is fishy. But you can add injected fields using field injection indefinitely. Having too many dependencies is a red flag that the class usually does more than one thing, and that it may violate the Single Responsibility Principle.