콩

인스턴스화 2가지

1. 컨테이너가 콩을 반사적으로 호출해 콩을 직접 생성하는경우

```
<bean id="helloWorld" class="com.example.HelloWorld">
        <constructor-arg value="Hello, Spring!"/>
    </bean>
```

```
public class MainApp {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml");
        HelloWorld helloWorld = (HelloWorld)
context.getBean("helloWorld");
        helloWorld.printMessage();
    }
}
```

2. 정적 팩토리 메소드를 통해 콩을 생성하는 경우

    - 정적 팩토리 메소드로 인스턴스화하기
    - 정적 팩토리 메소드로 빈을 생성할 때, class 속성을 사용하여 정적 팩토리 메소드를 포함하는 클래스를 지정하고 factory-method라는 속성을 사용하여 팩토리 메소드 자체의 이름을 지정합니다. 이 메소드를 호출하여 (나중에 설명할 옵션 인수와 함께) 실제 객체를 반환하고 생성자를 통해 생성된 것처럼 처리해야 합니다. 이러한 빈 정의의 한 가지 용도는 레거시 코드에서 정적 팩토리를 호출하는 것입니다.
    - 다음 빈 정의는 빈이 팩토리 메소드를 호출하여 생성된다고 지정합니다. 정의는 반환되는 객체의 유형(클래스)을 지정하지 않고 팩토리 메소드가 포함된 클래스를 지정합니다. 이 예에서는 createInstance() 메소드가 정적 메소드여야 합니다. 다음 예는 팩토리 메소드를 지정하는 방법을 보여줍니다:

```
public class ClientService {
private static ClientService clientService = new ClientService();
private ClientService() {}

public static ClientService createInstance() {
    return clientService;
}
```

```
<bean id="clientService"class="examples.ClientService"factory-
method="createInstance"/>
```

3. 인스턴스 팩토리 메소드를 사용한 인스턴스화

    - 정적 팩토리 메소드를 통한 인스턴스화와 비슷하게, 인스턴스 팩토리 메소드를 사용한 인스턴스화는 컨테이너의 기존 빈에서 비정적(non-static) 메소드를 호출하여 새로운 빈을 생성합니다. 이 메커니즘을 사용하려면 class 속성을 비워 두고, factory-bean 속성에서 객체를 생성하기 위해 호출되는 인스턴스 메소드를 포함하는 현재 컨테이너(또는 부모 또는 조상 컨테이너)의 빈 이름을 지정합니다. factory-method 속성을 사용하여 팩토리 메소드 자체의 이름을 설정합니다. 다음 예제는 이러한 빈을 구성하는 방법을 보여줍니다.

```
<!-- 팩토리 빈, createInstance()라는 메소드를 포함 -->
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 이 로케이터 빈에 필요한 모든 종속성 주입 -->
</bean>

<!-- 팩토리 빈을 통해 생성될 빈 -->
<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new
ClientServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }
}
```

    - 하나의 팩토리 클래스는 두개 이상의 팩토리 메소드를 가질수 있음

```
<bean id="serviceLocator" class="examples.DefaultServiceLocator">
    <!-- 이 로케이터 빈에 필요한 모든 종속성 주입 -->
</bean>

<bean id="clientService"
    factory-bean="serviceLocator"
    factory-method="createClientServiceInstance"/>

<bean id="accountService"
    factory-bean="serviceLocator"
    factory-method="createAccountServiceInstance"/>
```

```
public class DefaultServiceLocator {

    private static ClientService clientService = new
ClientServiceImpl();

    private static AccountService accountService = new
AccountServiceImpl();

    public ClientService createClientServiceInstance() {
        return clientService;
    }

    public AccountService createAccountServiceInstance() {
        return accountService;
    }
}
```

콩의 런타임 타입

- 런타임 타입 : IoC 컨테이너에서 객체가 생성되고 실행되는 동안 해당 객체의 실제 타입
- BeanFactory의 getType()

```
public interface Service{};
public class TellService implements Service{};

public class Telecom{
    private Service service;
    public void service(){
        try(ClassPathXmlApplicationContext context = new
ClassPathXmlApplicationContext("applicationContext.xml")){
            //이때의 service의 런타임 타입은 TellService이다.
            service = context.getBean(TellService.class);

        }
    }
}
```

BeanScope

- 싱글톤

```
<bean id="accountService" class="com.something.DefaultAccountService"/>
```

```
<!-- the following is equivalent, though redundant (singleton scope is the
default) -->
<bean id="accountService" class="com.something.DefaultAccountService"
scope="singleton"/>
```

- 프로토타입
- 싱글톤과 프로토타입의 의존성