



3. 스프링 컨테이너와 스프링 빈

🕒 Created	@November 2, 2022 11:14 AM
📌 Progress	In Progress

 **스프링 컨테이너**

- Application Context**
- BeanDefinition** : **설정 메타정보 (추상화)**
 - 스프링** 이 다양한 형태의 설정 정보를 **BeanDefinition** 으로 **추상화** 해서 사용한다.

 **학습 TODO list**

- ☐ 스프링이 내부에서 사용하는 빈
- ☐ 람다 함수

- 3.1. 스프링 컨테이너 생성
 - 3.1.1. 스프링 컨테이너 생성 과정
- 3.2. 컨테이너에 등록된 모든 빈 조회
- 3.3. 스프링 빈 조회 - 기본
- 3.4. 스프링 빈 조회 - 동일한 타입이 둘 이상
- 3.5. 스프링 빈 조회 - 상속 관계
- 3.6. BeanFactory와 ApplicationContext
 - 3.6.1. BeanFactory
 - 3.6.2. ApplicationContext
- 3.7. 다양한 설정 형식 지원 - 자바 코드, XML
 - 3.7.1. 어노테이션 기반 자바 코드 설정
 - 3.7.2. XML 설정
- 3.8. 스프링 빈 설정 메타 정보 - BeanDefinition
 - 3.8.1. 스프링이 다양한 설정 형식을 지원하는 방법
 - 3.8.2. BeanDefinition

3.1. 스프링 컨테이너 생성

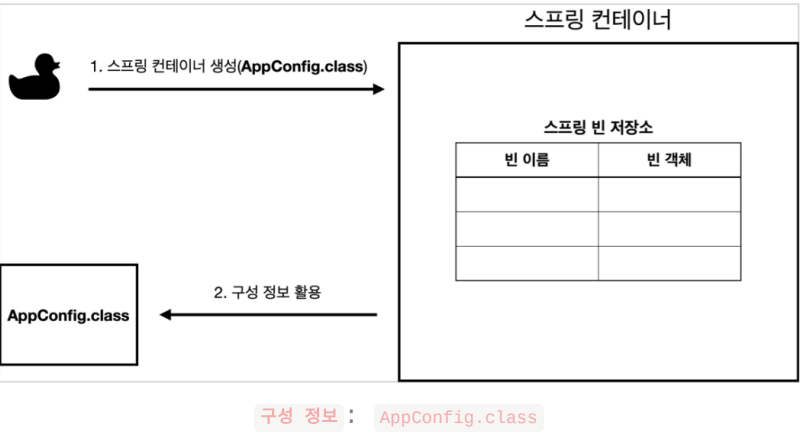
스프링 컨테이너 를 생성하고, **설정 정보(구성 정보)** 를 참고해서 **스프링 빈** 을 등록하고, **의존관계를 설정**해보자.

```
//스프링 컨테이너 생성
ApplicationContext applicationContext = new AnnotationConfigApplicationContext(AppConfig.class);
```

- ApplicationContext** : **스프링 컨테이너** (인터페이스)
 - XML** 기반으로 만들 수 있고, **어노테이션** 기반의 자바 설정 클래스로 만들 수 있다.
 - 2. 스프링 핵심 원리 이해 에서 **AppConfig** 를 사용한 방식은 **어노테이션** 기반 자바 설정 클래스로 스프링 컨테이너를 만든 방법이다.
 - `new AnnotationConfigApplicationContext(AppConfig.class)`
 - AnnotationConfigApplicationContext** 클래스는 **ApplicationContext** 인터페이스의 구현체이다.

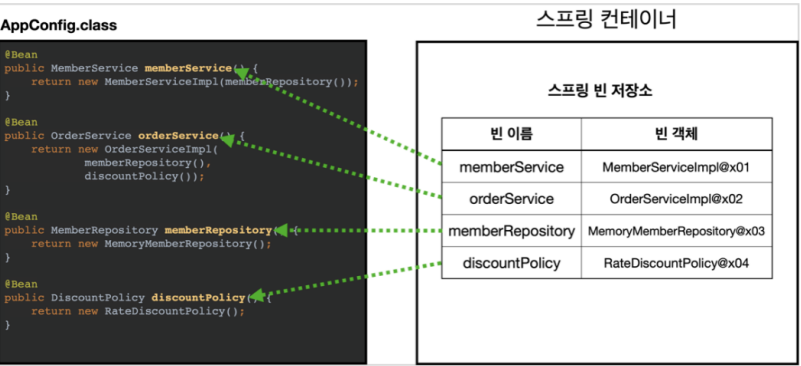
3.1.1. 스프링 컨테이너 생성 과정

- 스프링 컨테이너** 생성



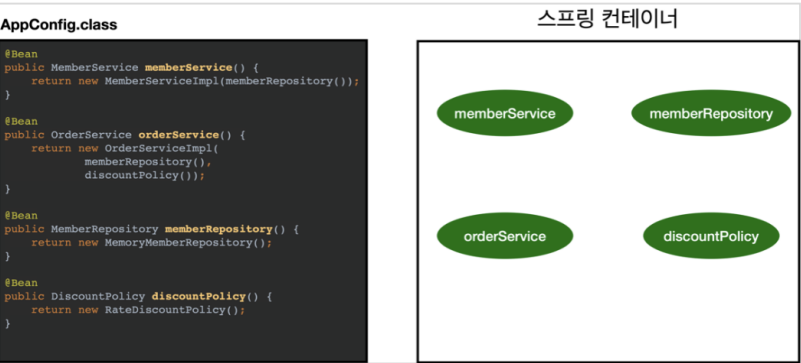
- `new AnnotationConfigApplicationContext(AppConfig.class)`
- **스프링 컨테이너** 를 생성할 때는 **구성 정보** 를 지정해줘야 한다.

2. **스프링 빈** 등록

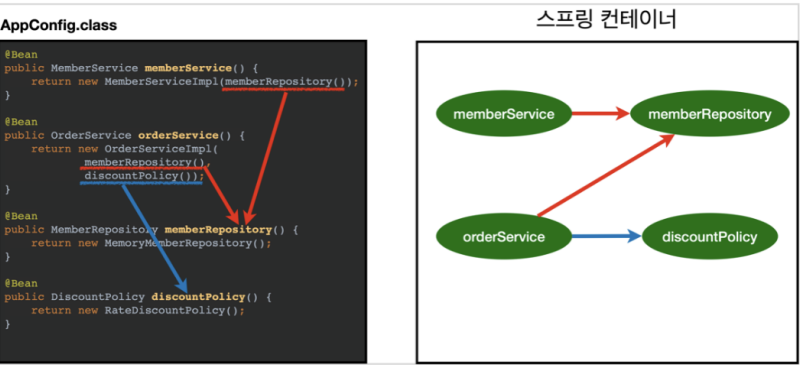


- **스프링 컨테이너** 는 파라미터로 넘어온 설정 클래스 정보(**구성 정보**)를 사용해서 **스프링 빈** 을 등록한다.
 - **빈** 이름은 메서드 이름을 사용한다.
 - **빈** 이름을 직접 부여할 수 있다. (ex. `@Bean(name="memberService2")`)
 - **빈** 이름은 **항상 다른 이름**으로 부여해야 한다. 같은 이름을 부여하면 다른 빈이 무시되거나 기존 빈을 덮어쓰는 등 설정에 따라 오류가 발생한다.

3. **스프링 빈** 의존관계 설정 - 준비



4. **스프링 빈** 의존관계 설정 - 완료



- **스프링 컨테이너** 는 **설정 정보(구성 정보)** 를 참고해서 의존관계를 주입(**DI**)한다.

3.2. 컨테이너에 등록된 모든 빈 조회

| **스프링 컨테이너에 스프링 빈이 잘 등록되었는지 확인해보자.**

- 모든 **빈** 출력하기

```
class ApplicationContextInfoTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);
}
```

```

@Test
@DisplayName("모든 빈 출력하기")
void findAllBean() {
    String[] beanDefinitionNames = ac.getBeanDefinitionNames();
    for (String beanDefinitionName : beanDefinitionNames) {
        Object bean = ac.getBean(beanDefinitionName);
        System.out.println("name=" + beanDefinitionName + " object=" + bean);
    }
}
}

```

- `ac.getBeanDefinitionNames()` : **스프링**에 등록된 모든 **빈**의 이름을 조회한다.
- `ac.getBean()` : **빈** 이름으로 **빈** 객체(인스턴스)를 조회한다.
- 애플리케이션 **빈** 출력하기: **스프링**이 내부에서 사용하는 **빈**은 제외하고 직접 등록한 **빈**만 출력한다.

```

class ApplicationContextInfoTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    @Test
    @DisplayName("애플리케이션 빈 출력하기")
    void findApplicationBean() {
        String[] beanDefinitionNames = ac.getBeanDefinitionNames();
        for (String beanDefinitionName : beanDefinitionNames) {
            BeanDefinition beanDefinition = ac.getBeanDefinition(beanDefinitionName);
            //BeanDefinition.ROLE_APPLICATION: 직접 등록한 애플리케이션 빈
            //BeanDefinition.ROLE_INFRASTRUCTURE: 스프링 내부에서 사용하는 빈
            if (beanDefinition.getRole() == BeanDefinition.ROLE_APPLICATION) {
                Object bean = ac.getBean(beanDefinitionName);
                System.out.println("name=" + beanDefinitionName + " object=" + bean);
            }
        }
    }
}

```

- `ROLE_APPLICATION` : 일반적으로 사용자가 정의한 **빈**
- `ROLE_INFRASTRUCTURE` : **스프링**이 내부에서 사용하는 **빈**

```

name = org.springframework.context.annotation.internalConfigurationAnnotationProcessor object =
    org.springframework.context.annotation.ConfigurationClassPostProcessor@7d42c224
name = org.springframework.context.annotation.internalAutowiredAnnotationProcessor object =
    org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor@56aaaeed
name = org.springframework.context.annotation.internalCommonAnnotationProcessor object = org
    .springframework.context.annotation.CommonAnnotationBeanPostProcessor@522a32b1
name = org.springframework.context.event.internalEventListenerProcessor object = org.springframework
    .context.event.EventListenerMethodProcessor@35390ee3
name = org.springframework.context.event.internalEventListenerFactory object = org.springframework
    .context.event.DefaultEventListenerFactory@5e01a982

```

3.3. 스프링 빈 조회 - 기본

- **스프링 컨테이너**에서 **스프링 빈**을 찾는 가장 기본적인 조회 방법
 - `ac.getBean(빈 이름, 타입)`
 - `ac.getBean(빈 이름)`
- **빈** 이름으로 조회

```

class ApplicationContextBasicFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    @Test
    @DisplayName("빈 이름으로 조회")
    void findBeanByName() {
        MemberService memberService = ac.getBean("memberService", MemberService.class);
        assertThat(memberService).assertInstanceOf(MemberServiceImpl.class);
    }
}

```

- 이름 없이 타입만으로 조회

```

class ApplicationContextBasicFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    @Test

```

```

    @DisplayName("이름 없이 타입만으로 조회")
    void findBeanByType() {
        MemberService memberService = ac.getBean(MemberService.class);
        assertThat(memberService).isInstanceOf(MemberServiceImpl.class);
    }
}

```

- 구체 타입으로 조회

```

class ApplicationContextBasicFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    @Test
    @DisplayName("구체 타입으로 조회")
    void findBeanByName2() {
        MemberService memberService = ac.getBean("MemberService", MemberServiceImpl.class);
        assertThat(memberService).isInstanceOf(MemberServiceImpl.class);
    }
}

```

- 구체 타입으로 조회하면 변경 시 유연성이 떨어진다.

- 없는 빈 이름으로 조회

```

class ApplicationContextBasicFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);

    @Test
    @DisplayName("빈 이름으로 조회X")
    void findBeanByNameX() {
        Assertions.assertThrows(NoSuchBeanDefinitionException.class, () -> ac.getBean("xxxxx", MemberService.class));
    }
}

```

- `NoSuchBeanDefinitionException: No bean named 'xxxxx' available`

3.4. 스프링 빈 조회 - 동일한 타입이 둘 이상

- 타입으로 조회 시 같은 타입의 **스프링 빈**이 둘 이상이면 오류가 발생한다. 이럴 경우, **빈** 이름을 지정해야 한다.

```

class ApplicationContextSameBeanFindTest {
    @Configuration
    static class SameBeanConfig {
        @Bean
        public MemberRepository memberRepository1() {
            return new MemoryMemberRepository();
        }
        @Bean
        public MemberRepository memberRepository2() {
            return new MemoryMemberRepository();
        }
    }
}

```

- 같은 타입의 **스프링 빈**이 둘 이상일 경우 타입으로 조회

```

class ApplicationContextSameBeanFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(SameBeanConfig.class);

    @Test
    @DisplayName("타입으로 조회 시 같은 타입이 둘 이상 있으면, 중복 오류가 발생한다")
    void findBeanByTypeDuplicate() {
        assertThrows(NoUniqueBeanDefinitionException.class, () -> ac.getBean(MemberRepository.class));
    }
    @Test
    @DisplayName("타입으로 조회 시 같은 타입이 둘 이상 있으면, 빈 이름을 지정한다")
    void findBeanByName() {
        MemberRepository memberRepository = ac.getBean("memberRepository1", MemberRepository.class);
        assertThat(memberRepository).isInstanceOf(MemberRepository.class);
    }
}

```

- 특정 타입을 모두 조회하기

```

class ApplicationContextSameBeanFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(SameBeanConfig.class);

    @Test
    @DisplayName("특정 타입을 모두 조회하기")
    void findAllBeanByType() {
        Map<String, MemberRepository> beansOfType = ac.getBeansOfType(MemberRepository.class);
    }
}

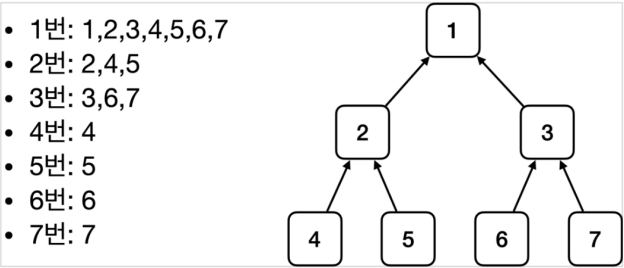
```

```
for (String key : beansOfType.keySet()) {
    System.out.println("key = " + key + " value = " + beansOfType.get(key));
}
System.out.println("beansOfType = " + beansOfType);
assertThat(beansOfType.size()).isEqualTo(2);
}
}
```

- `ac.getBeansOfType()`

3.5. 스프링 빈 조회 - 상속 관계

- 부모 타입으로 조회하면, 자식 타입도 함께 조회한다.



- `Object` 타입(자바 객체의 최상위 부모)으로 조회하면, 모든 `스프링 빈`을 조회한다.

```
class ApplicationContextExtendsFindTest {
    @Configuration
    static class TestConfig {
        @Bean
        public DiscountPolicy rateDiscountPolicy() {
            return new RateDiscountPolicy();
        }
        @Bean
        public DiscountPolicy fixDiscountPolicy() {
            return new FixDiscountPolicy();
        }
    }
}
```

- 부모 타입으로 조회

```
class ApplicationContextExtendsFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(TestConfig.class);

    @Test
    @DisplayName("부모 타입으로 조회 시, 자식이 둘 이상 있으면, 중복 오류가 발생한다")
    void findBeanByParentTypeDuplicate() {
        assertThrows(NoUniqueBeanDefinitionException.class, () -> ac.getBean(DiscountPolicy.class));
    }
    @Test
    @DisplayName("부모 타입으로 조회 시, 자식이 둘 이상 있으면, 빈 이름을 지정해야한다")
    void findBeanByParentTypeBeanName() {
        DiscountPolicy rateDiscountPolicy = ac.getBean("rateDiscountPolicy", DiscountPolicy.class);
        assertThat(rateDiscountPolicy).assertInstanceOf(RateDiscountPolicy.class);
    }
}
```

- 특정 하위 타입으로 조회

```
class ApplicationContextExtendsFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(TestConfig.class);

    @Test
    @DisplayName("특정 하위 타입으로 조회")
    void findBeanBySubType() {
        RateDiscountPolicy bean = ac.getBean(RateDiscountPolicy.class);
        assertThat(bean).assertInstanceOf(RateDiscountPolicy.class);
    }
}
```

- 부모 타입으로 모두 조회

```
class ApplicationContextExtendsFindTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(TestConfig.class);

    @Test
    @DisplayName("부모 타입으로 모두 조회하기")
    void findBeanByParentType() {
        Map<String, DiscountPolicy> beansOfType = ac.getBeansOfType(DiscountPolicy.class);
        assertThat(beansOfType.size()).isEqualTo(2);
    }
}
```

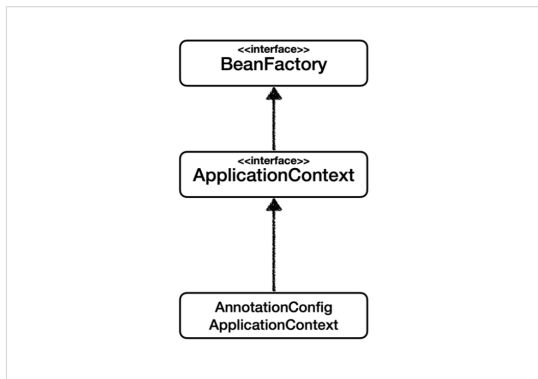
```

        for (String key : beansOfType) {
            System.out.println("key = " + key + " value = " + beansOfType.get(key));
        }
    }
    @Test
    @DisplayName("부모 타입으로 모두 조회하기 - Object")
    void findBeanByObjectType() {
        Map<String, Object> beansOfType = ac.getBeansOfType(Object.class);
        for (String key : beansOfType) {
            System.out.println("key = " + key + "value = " + beansOfType.get(key));
        }
    }
}

```

3.6. BeanFactory와 ApplicationContext

- `BeanFactory` 나 `ApplicationContext` 를 **스프링 컨테이너** 라고 한다.

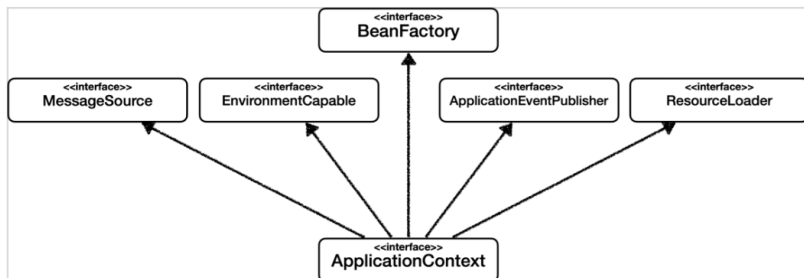


3.6.1. BeanFactory

- `BeanFactory` : **스프링 컨테이너** 의 최상위 인터페이스이다.
 - **스프링 빈** 을 관리하고 조회한다.
 - 대부분의 기능을 `BeanFactory` 에서 제공한다.
 - `getBean()`

3.6.2. ApplicationContext

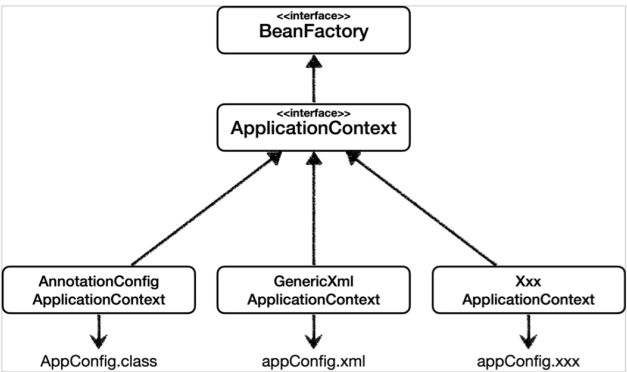
- `ApplicationContext` : `BeanFactory` 기능을 상속받아서 제공한다.
 - 빈 관리 기능
 - 부가기능(`BeanFactory` 를 직접 사용할 일은 거의 없다. 부가기능이 포함된 `ApplicationContext` 를 사용한다.)



- `MessageSource` : 국제화 기능
 - 한국에서 들어오면 한국어, 영어권에서 들어오면 영어로 출력한다.
- `EnvironmentCapable(환경변수)` : 로컬, 개발, 운영 등을 구분해서 처리한다.
- `ApplicationEventPublisher` : 이벤트를 발행하고 구독하는 모델을 편리하게 지원한다.
- `ResourceLoader` : 파일, 클래스패스, 외부 등에서 리소스를 편리하게 조회한다.

3.7. 다양한 설정 형식 지원 - 자바 코드, XML

- **스프링 컨테이너** 는 다양한 형식의 설정 정보를 받을 수 있게 유연하게 설계되어 있다.
 - 자바 코드, `XML` 등



3.7.1. 어노테이션 기반 자바 코드 설정

- `AnnotationConfigApplicationContext` 클래스를 사용하여 자바 코드로 된 설정 정보를 넘긴다.
 - `new AnnotationConfigApplicationContext(AppConfig.class)`

3.7.2. XML 설정

- 최근에는 `스프링 부트` 를 많이 사용하면서 `XML` 기반의 설정을 잘 사용하지 않는다. 하지만 아직 많은 레거시 프로젝트들이 `XML` 로 되어있고, 컴파일 없이 `빈 설정 정보` 를 변경할 수 있는 장점이 있어 알아야 한다.
- `GenericXmlApplicationContext` 클래스를 사용하여 `xml` 설정 파일을 넘긴다.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="memberService" class="com.allyhyeseongkim.core.member.MemberServiceImpl" >
        <constructor-arg name="memberRepository" ref="memberRepository" />
    </bean>

    <bean id="memberRepository" class="com.allyhyeseongkim.core.member.MemoryMemberRepository"/>

    <bean id="orderService" class="com.allyhyeseongkim.core.order.OrderServiceImpl">
        <constructor-arg name="memberRepository" ref="memberRepository" />
        <constructor-arg name="discountPolicy" ref="discountPolicy" />
    </bean>

    <bean id="discountPolicy" class="com.allyhyeseongkim.core.discount.RateDiscountPolicy" />
</beans>
```

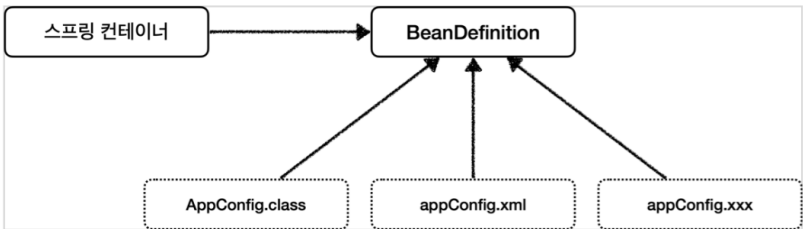
```
public class XmlAppContext {
    @Test
    void xmlAppContext() {
        ApplicationContext ac = new GenericXmlApplicaiontContext("appConfig.xml");

        MemberService memberService = ac.getBean("memberService", MemberService.class);
        assertThat(memberService).isInstanceOf(MemberService.class);
    }
}
```

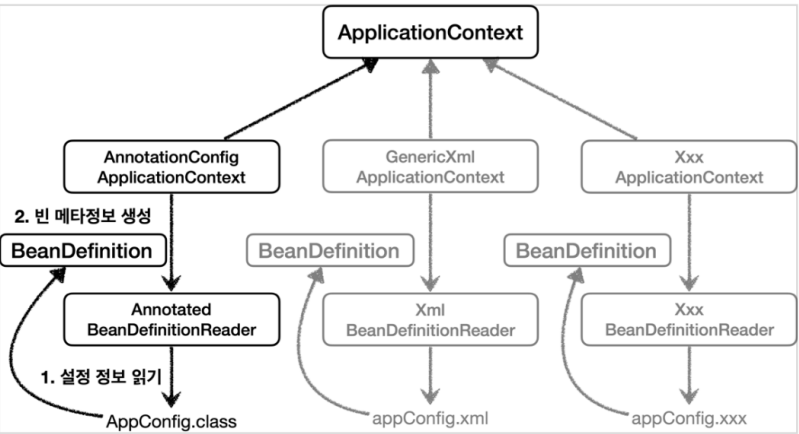
3.8. 스프링 빈 설정 메타 정보 - BeanDefinition

3.8.1. 스프링이 다양한 설정 형식을 지원하는 방법

- `BeanDefinition` : `빈 설정 메타정보` (추상화)
 - 역할과 구현을 개념적으로 나눈다.



- 자바 코드, `XML` 을 읽어서 `BeanDefinition` 으로 만든다.
- `스프링 컨테이너` 는 자바 코드인지, `XML` 일지 몰라도 된다. `BeanDefinition` 만 알고 있다.
- `@Bean` , `<bean>` 당 각각 하나씩 `메타 정보` 가 생성된다.
 - `스프링 컨테이너` 는 `메타 정보` 를 기반으로 `스프링 빈` 을 생성한다.
- `BeanDefinition` 으로 `설정 정보` 를 읽는 방법



- 새로운 형식의 **설정 정보**가 추가되면 형식에 맞는 **BeanDefinitionReader**를 만들어서 **BeanDefinition**을 생성한다.
 - **AnnotationConfigApplicationContext**는 **AnnotatedBeanDefinitionReader**를 사용해서 **AppConfig**를 읽고 **BeanDefinition**(빈 **메타 정보**)을 생성한다.
 - **GenericXmlApplicationContext**는 **XmlBeanDefinitionReader**를 사용해서 **appConfig.xml**을 읽고 **BeanDefinition**을 생성한다.

3.8.2. BeanDefinition

- **BeanDefinition** **메타 정보**

```
public class BeanDefinitionTest {
    AnnotationConfigApplicationContext ac = new AnnotationConfigApplicationContext(AppConfig.class);
    // GenericXmlApplicaionContext ac = new GenericXmlApplicaionContext("appConfig.xml");

    @Test
    @DisplayName("빈 설정 메타정보 확인")
    void findApplicationBean() {
        String[] beanDefinitionNames = ac.getBeanDefinitionNames();
        for (String beanDefinitionName : beanDefinitionNames) {
            BeanDefinition beanDefinition = ac.getBeanDefinition(beanDefinitionName);
            if (beanDefinition.getRole() == BeanDefinition.ROLE_APPLICATION) {
                System.out.println("beanDefinitionName" + beanDefinitionName + " beanDefinition = " + beanDefinition);
            }
        }
    }
}
```

- **BeanClassName** : 생성할 **빈**의 클래스 명
 - **AnnotatedBean**으로 생성할 경우 **null**
- **factoryBeanName** : **설정 정보(구성 정보)**가 있는 클래스로 생성된 **빈**의 이름
 - **XmlBean**으로 생성할 경우 **null**
 - ex. **appConfig**
- **factoryMethodName** : **빈**을 생성할 **팩토리 메서드**
 - ex. **memberService**
- **Scope** : 싱글톤(기본값)
- **lazyInit** : **스프링 컨테이너**를 생성할 때 **빈**을 생성하는 것이 아니라, 실제 **빈**을 사용할 때까지 최대한 생성을 **지연처리**하는지의 여부
- **InitMethodName** : **빈**을 생성하고, 의존관계를 적용한 뒤에 호출되는 초기화 메서드 명
- **DestroyMethodName** : **빈**의 생명주기가 끝나서 제거하기 직전에 호출되는 메서드 명
- **Constructor arguments**, **Properties** : 의존관계 주입에서 사용한다.

```
constructor-argument = org.springframework.beans.factory.config.ConstructorArgumentValues@474de529 constructor-property = PropertyValues: length=0
```

- **AnnotatedBean**으로 생성할 경우 없다.

```
constructor-argument = org.springframework.beans.factory.config.ConstructorArgumentValues@cb constructor-property = PropertyValues: length=0
```




AnnotatedBean : 팩토리 메서드 (AppConfig)를 사용하여 빈을 등록하는 방법

- beanDefinition 이 Root bean

```
beanDefinitionName = memberService beanDefinition = Root bean: class [null];
```

- 내용이 직접적으로 드러나지 않고, factoryBean 에서 factoryMethod 를 통해서 빈이 등록된다.

```
factoryBeanName=appConfig; factoryMethodName=memberService;
```

```
defined in com.allyhyeseongkim.core.AppConfig
```



XmlBean : 직접 스프링 에 빈을 등록하는 방법

- beanDefinition 이 Generic bean

```
beanDefinitionName = memberService beanDefinition = Generic bean: class [com.allyhyeseongkim.core.member.MemberServiceImpl];
```

- 내용이 직접적으로 들어나 있다. 직접 빈을 등록한다.

```
beanDefinition = Generic bean: class [com.allyhyeseongkim.core.member.MemberServiceImpl]
```

```
defined in class path resource [appConfig.xml]
```

- factoryBean 과 factoryMethod 는 사용하지 않는다.

```
factoryBeanName=null; factoryMethodName=null;
```