DI(Dependency Injection)

안 화 수

제어의 역행 (Inversion of Control, IoC)

- **IoC (Inversion of Control)**

- 기존에 개발자들이 빈 객체를 관리해 오던 개념에서 빈 관리를 컨테이너에서 처리한 다는 의미

- 개발자가 코드의 제어 흐름을 처리하지 않고, 스프링 프레임워크가 객체의 라이프사이클 및 실행 흐름을 관리한다.

제어의 역행 (Inversion of Control, IoC)

❖ 기존 시스템

- 애플리케이션 에서 개발자가 직접 객체를 생성해서 사용함.
- 애플리 케이션에서 직접 MessageBeanKr 클래스 객체를 생성해서 메소드를 사용하기 때문에 의존성이 너무 강함(tight coupling)

제어의 역행 (Inversion of Control, IoC)

- 기존 시스템

```
package sample01;

public class Ex01 {
	public static void main(String[] args) {
		MessageBeanKr mb = new MessageBeanKr();
		mb.sayHello("Spring");
	}
}
```

제어의 역행 (Inversion of Control, IoC)

- 기존 시스템

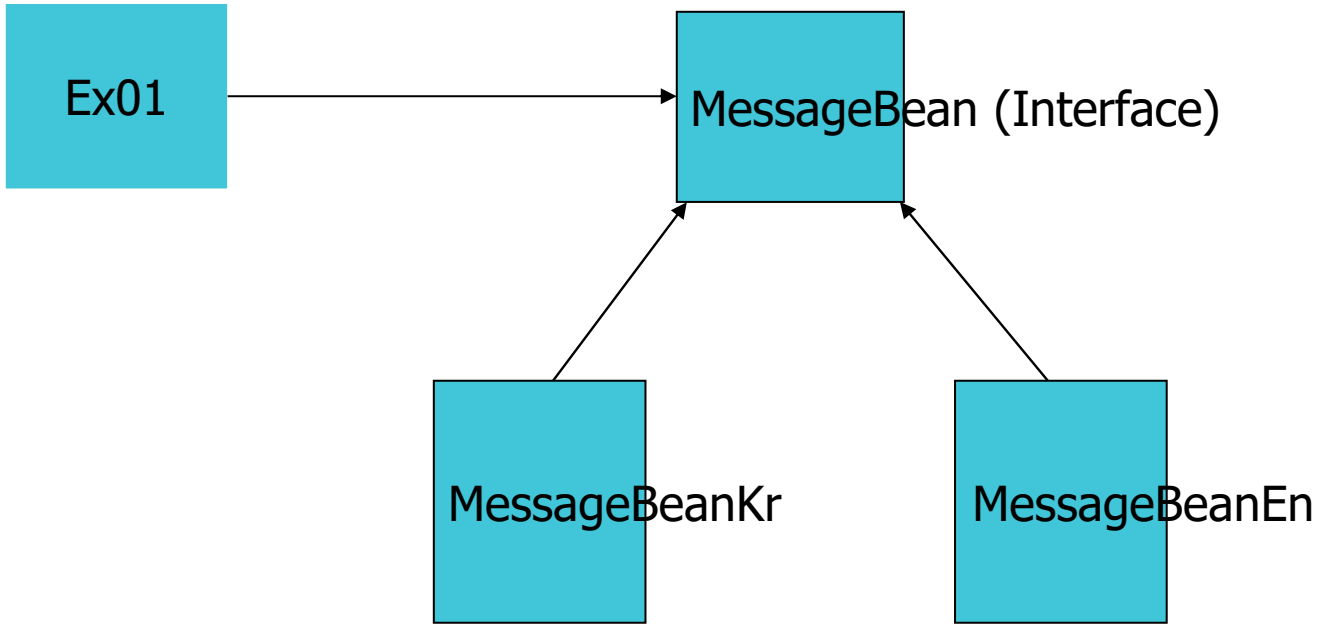
```
package sample01;

public class MessageBeanKr {
	void sayHello(String name) {
		System.out.println("안녕하세요! " + name);
	}
}
```

제어의 역행 (Inversion of Control, IoC)

기존 시스템

- 인터페이스를 사용함으로써 조금 loose coupling 으로 변환되어서 유연성을 제공 할 수 있음.
- 인터페이스를 사용함으로써 조금 느슨한 결합을 하여도 여전히 MessageBean인터페이스와 객체에 의존적임.



제어의 역행 (Inversion of Control, IoC)

- 기존 시스템

```
package sample02;

public class Ex01 {
	public static void main(String[] args) {
		MessageBean mb = new MessageBeanKr();
		mb.sayHello("Spring");
	}
}
```

제어의 역행 (Inversion of Control, IoC)

- 기존 시스템

```
package sample02;

public interface MessageBean {
	void sayHello(String name);
}
```

제어의 역행 (Inversion of Control, IoC)

- 기존 시스템

```
package sample02;

public class MessageBeanKr implements MessageBean {
	public void sayHello(String name) {
		System.out.println("안녕하세요! " + name);
	}
}
```

제어의 역행 (Inversion of Control, IoC)

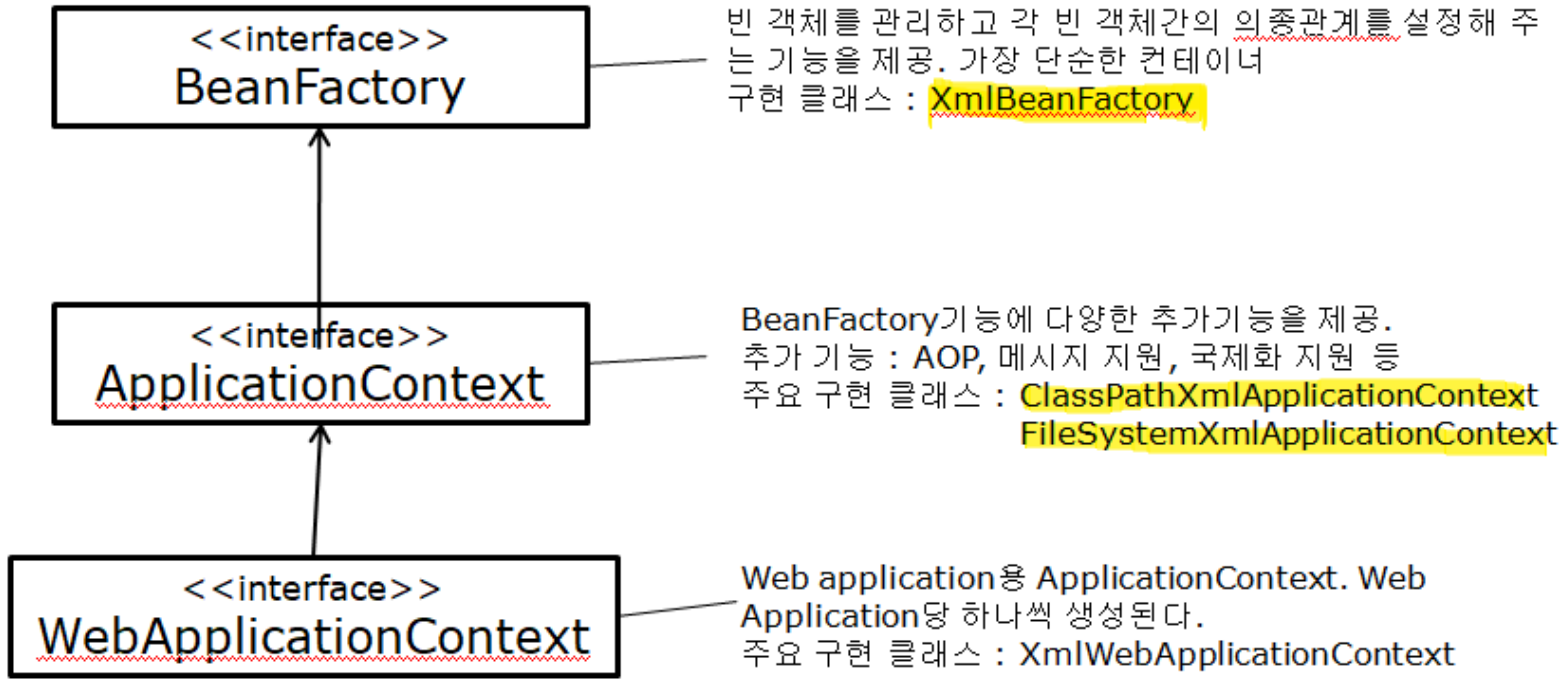
- 기존 시스템

```
package sample02;

public class MessageBeanEn implements MessageBean {
	public void sayHello(String name) {
		System.out.println("Hello! " + name);
	}
}
```

제어의 역행 (Inversion of Control, IoC)

- DI 관련 주요 클래스
  - Spring Container : 객체를 관리하는 컨테이너
    - 다음 아래의 interface들을 구현한다.



제어의 역행 (Inversion of Control, IoC)

- IOC 방식 – DI (Dependency Injection 구현)

```
package sample03;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.core.io.FileSystemResource;

public class Ex01 {
        public static void main(String[] args) {
                BeanFactory bf = new XmlBeanFactory(new FileSystemResource("beans01.xml"));
                MessageBean mb = (MessageBean) bf.getBean("mb");
                mb.sayHello("Spring");
        }
}
```

제어의 역행 (Inversion of Control, IoC)

- IOC 방식 – DI (Dependency Injection 구현)

- Spring의 환경 설정 파일에서 bean객체를 생성함

beans01.xml

```
<!-- MessageBeanKr mb = new MessageBeanKr() -->

<bean id="mb" class="sample03.MessageBeanKr" ></bean>
```

의존성 주입 (Dependency Injection, DI)

- **DI (Dependency Injection)**

- 빈 간의 의존 관계를 컨테이너에서 설정하고 관리 한다는 개념

- **Constructor DI(Dependency Injection)**
  빈 간의 의존 관계를 설정하기 위해 생성자를 이용

- **Setter DI(Dependency Injection)**
  빈 간의 의존 관계를 설정하기 위해 setter 메소드를 이용

의존성 주입 (Dependency Injection, DI)

- **Constructor DI(Dependency Injection)**

```
package sample04;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Ex01 {
	public static void main(String[] args) {
		ApplicationContext ac = new FileSystemXmlApplicationContext("beans01.xml");
		MessageBean mb = (MessageBean) ac.getBean("mb2");
		mb.sayHello();
	}
}
```

의존성 주입 (Dependency Injection, DI)

- **Constructor DI(Dependency Injection)**

```
package sample04;

public class MessageBeanImpl implements MessageBean {
	private String name;
	private String greet;

	public MessageBeanImpl(String name, String greet) {
		this.name = name;			// 도깨비
		this.greet = greet;			// 안뇽
	}

	public void sayHello() {
		System.out.println(name + " ! " + greet);
	}
}
```

의존성 주입 (Dependency Injection, DI)

- **Constructor DI(Dependency Injection)**

beans01.xml

```
<!-- Constructor DI -->
<bean id="mb2" class="sample04.MessageBeanImpl">
        <constructor-arg value="도깨비">
                <!-- <value>박보검</value> -->
        </constructor-arg>
        <constructor-arg value="안뇽">
                <!-- <value>Hello</value> -->
        </constructor-arg>
</bean>
```

의존성 주입 (Dependency Injection, DI)

- **Setter DI(Dependency Injection)**

```
package sample07;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Ex01 {
	public static void main(String[] args) {
	ApplicationContext ac = newClassPathXmlApplicationContext("/sample07/beans07.xml");

		MessageBean mb = (MessageBean) ac.getBean("mb");
		mb.sayHello();
	}
}
```

의존성 주입 (Dependency Injection, DI)

- **Setter DI(Dependency Injection)**

```
package sample07;

public class MesageBeanImpl implements MessageBean {
        private String name;                    // property
        private String greet;
        public void setName(String name) {
                this.name = name;                       // 길동
        }
        public void setGreet(String greet) {
                this.greet = greet;                     // 안녕
        }
        public void sayHello() {
                System.out.println(name + " !! " + greet);
        }
}
```

의존성 주입 (Dependency Injection, DI)

- **Setter DI(Dependency Injection)**

beans07.xml

```
<!-- Setter DI -->
<bean id="mb" class="sample07.MesageBeanImpl">
        <property name="name">
                <value>길동</value>
        </property>
        <property name="greet" value="안녕"/>
</bean>
```

의존성 주입 (Dependency Injection, DI)

- **Anotaion을 이용한 DI(Dependency Injection)**

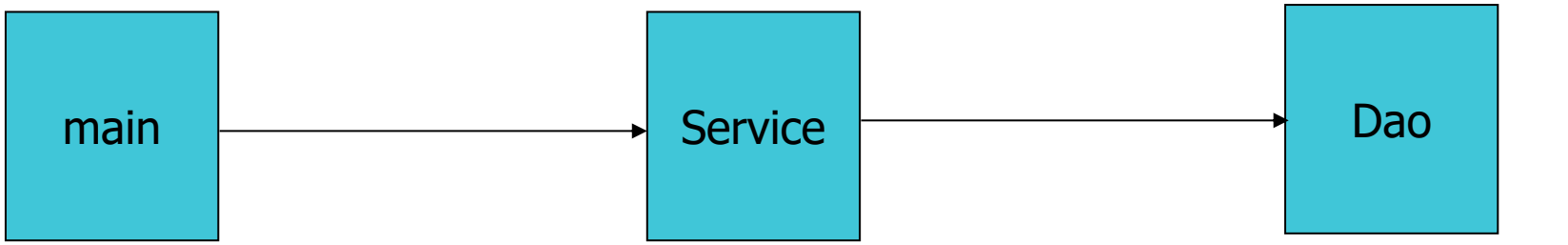
spring의 환경 설정 파일에 더 이상 빈 객체를 생성하지 않고 아래의 코드를 추가한다.

```
<context:component-scan base-package="sample15"/>
```

1. sample15 패키지 하위 클래스를 스캔한다는 의미를 가진다.
2. sample15 패키지 하위 클래스에 @Component, @Controller, @Service, @Repository 어노테이션이 붙어있는 클래스는 @Autowired 어노테이션을 이용해서 필요한 빈 객체를 setter 메소드 없이 자동으로 주입을 받는다.

의존성 주입 (Dependency Injection, DI)

- **Anotation을 이용한 DI(Dependency Injection)**



Ex01

@Service
BookServiceImpl

@Autowired
private BookDao bd;

@Repository
BookDaoImpl

의존성 주입 (Dependency Injection, DI)

- **Anotation을 이용한 DI(Dependency Injection)**

```
package sample15;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.GenericXmlApplicationContext;

public class Ex01 {
	public static void main(String[] args) {
	ApplicationContext ac = new GenericXmlApplicationContext("/sample15/beans15.xml");
		BookService bs = ac.getBean(BookService.class);
		Book book = bs.getBook();
		System.out.println(book);
	}
}
```

의존성 주입 (Dependency Injection, DI)

- **Anotation을 이용한 DI(Dependency Injection)**

beans15.xml

```
<beans>
<!--
        1. sample15 패키지 하위 클래스를 스캔한다는 의미를 가진다.
        2. @Component, @Controller, @Service, @Repository 어노테이션이 붙어있는 클래스는
           @Autowired 어노테이션을 이용해서 필요한 빈 객체를 setter 메소드 없이 주입을 받는다.
 -->

        <context:component-scan base-package="sample15"/>

</beans>
```

의존성 주입 (Dependency Injection, DI)

- **Anotation을 이용한 DI(Dependency Injection)**

```
package sample15;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.stereotype.Service;

@Service
public class BookServiceImpl implements BookService {
	@Autowired
	private BookDao bd;

	public Book getBook() {
		return bd.getBook("바람과 함께 사라지다");
	}
}
```

의존성 주입 (Dependency Injection, DI)

- **Anotation을 이용한 DI(Dependency Injection)**

```
package sample15;

import org.springframework.stereotype.Component;
import org.springframework.stereotype.Repository;

@Repository
public class BookDaoImpl implements BookDao {
        public Book getBook(String title) {          //title="바람과 함께 사라지다"
                return new Book(title, 25000);
        }
}
```