

제02장

IoC

Spring

```
each: function(e, t, n) {  
  var r, i = 0,  
      o = e.length,  
      a = M(e);  
  if (n) {  
    if (a) {  
      for (; o > i; i++)  
        if (r = t.apply(e[i], n), r ===  
    } else  
      for (i in e)  
        if (r = t.apply(e[i], n), r ===  
  } else if (a) {  
    for (; o > i; i++)  
      if (r = t.call(e[i], i, e[i]))  
    } else  
      for (i in e)  
        if (r = t.call(e[i], i, e[i]))  
  return e  
},  
trim: b && !b.call("\uffff\u00a0") ?  
  return null == e ? "" : b.call(  
} : function(e) {  
  return null == e ? "" : (e + "  
},  
makeArray: function(e, t) {  
  var n = t || [];  
  return null != e && (M(Ob  
},  
isArray: function(e, t, n) {  
  var r;  
  if (t) {  
    if (n) return n.c  
    for (r = t.length;  
    if (n in t)  
  }  
}
```

# 학습목표

1. IoC 개념에 대해서 알 수 있다.
2. DI 방식에 대해서 알 수 있다.

```
each: function(e, t, n) {  
  var r, i = 0,  
      o = e.length,  
      a = M(e);  
  if (n) {  
    if (a) {  
      for (; o > i; i++)  
        if (r = t.apply(e[i], n), r !==  
    } else  
      for (i in e)  
        if (r = t.apply(e[i], n), r !==  
  } else if (a) {  
    for (; o > i; i++)  
      if (r = t.call(e[i], i, e[i])  
  } else  
    for (i in e)  
      if (r = t.call(e[i], i, e[i])  
  return e  
},  
trim: b && !b.call("\uffff\u00a0") ?  
  return null == e ? "" : b.call(  
} : function(e) {  
  return null == e ? "" : (e + "  
},  
makeArray: function(e, t) {  
  var n = t || [];  
  return null != e && (M(Ob  
},  
isArray: function(e, t, n) {  
  var r;  
  if (t) {  
    if (n) return n.c  
    for (n = t.length  
      if (n in t  
  }  
}
```

# 목차

1. IoC
2. DI

```

each: function(e, t, n) {
  var r, i = 0,
      o = e.length,
      a = M(e);
  if (n) {
    if (a) {
      for (; o > i; i++)
        if (r = t.apply(e[i], n), r === !1) break
    } else
      for (i in e)
        if (r = t.apply(e[i], n), r === !1) break
  } else if (a) {
    for (; o > i; i++)
      if (r = t.call(e[i], e[i]), r === !1) break
  } else
    for (i in e)
      if (r = t.call(e[i], e[i]), r === !1) break;
  return e
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
  return null == e ? "" : b.call(e)
} : function(e) {
  return null == e ? "" : (e + "").replace(C, "")
},
makeArray: function(e, t) {
  var n = t || [];
  return null != e && (M(Object(e)) ? x.merge(n, "string"
),
isArray: function(e, t, n) {
  var r;
  if (t) {
    if (n) return m.call(t, e, n);
    for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : r; r--;)
      if (n in t && t[r] === e) return r
  }
}

```

# 1. IoC

## ■ IoC

- Inversion of Control
- 제어의 역전
- 예전부터 프로그램의 제어권은 개발자가 가지고 있었음
- 개발자의 제어권을 스프링 프레임워크가 가져감
- Bean 생성, 의존관계 설정(Dependency), 생명주기(Lifecycle) 등을 Framework가 직접 관리하는 것을 말함

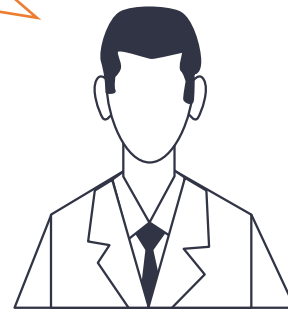
# Bean

## ■ Bean

- IoC 컨테이너에 의해서 관리되는 객체(인스턴스)를 의미함
- 기존의 프레임워크에 종속된 무거운 객체가 아닌 보통의 객체를 의미함
  - POJO(Plain Old Java Object)

### Bean

Spring Container에 의해서 관리되는  
POJO(Plain Old Java Object)를  
의미한다.  
쉽게 말하면 자바 인스턴스가  
곧 Bean이다.

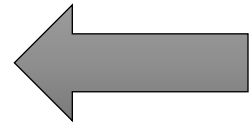


# IoC 컨테이너

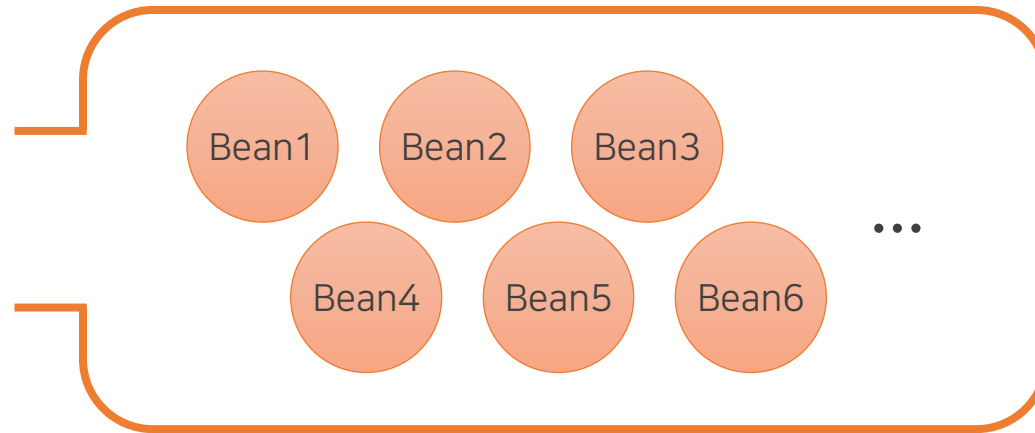
## ■ IoC 컨테이너

- 컨테이너 : Container. 객체의 생명주기를 관리하고 생성된 인스턴스를 관리함
- Spring Framework에서 객체를 생성과 소멸을 담당하고 의존성을 관리하는 컨테이너를 IoC 컨테이너라고 함
- IoC 컨테이너 = 스프링 컨테이너

개발할 때 컨테이너에 있는  
Bean을 받아서 사용

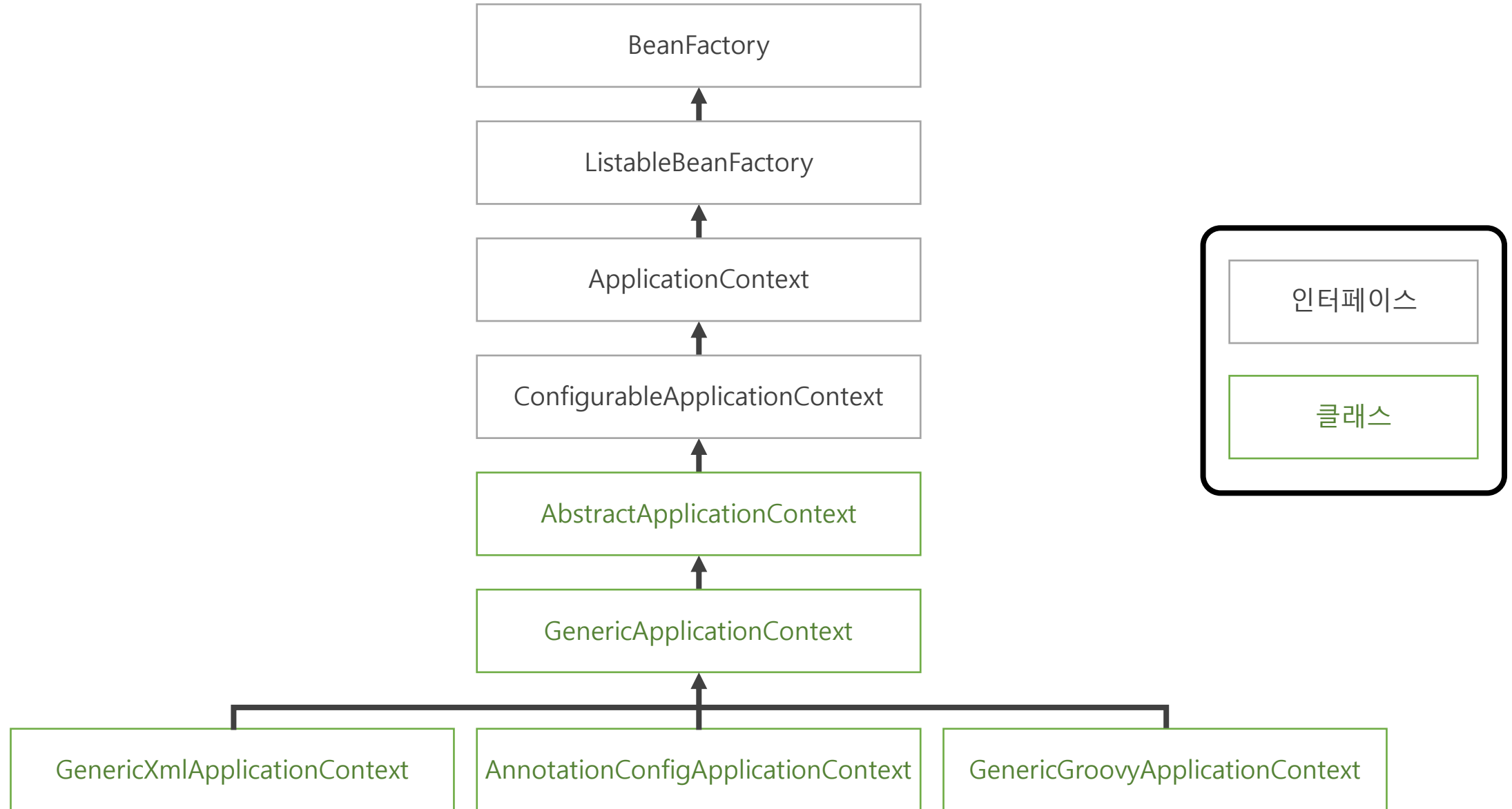


getBean()  
@Inject  
@Autowired  
...



IoC Container (ApplicationContext)

# Spring IoC 컨테이너





# Spring IoC 컨테이너

## ■ Spring IoC 컨테이너 종류

컨테이너	의미
BeanFactory	<ul style="list-style-type: none"><li>스프링 빈 설정 파일(Spring Bean Configuration File)에 등록된 bean을 생성하고 관리하는 가장 기본적인 컨테이너</li><li>클라이언트 요청에 의해서 bean을 생성함</li></ul>
ApplicationContext	<ul style="list-style-type: none"><li>트랜잭션 관리, 메시지 기반 다국어 처리 등 추가 기능을 제공</li><li>bean으로 등록된 클래스들을 객체 생성 즉시 로딩시키는 방식으로 동작</li></ul>
GenericXmlApplicationContext	<ul style="list-style-type: none"><li>파일 시스템 또는 클래스 경로에 있는 XML 설정 파일을 로딩하여 &lt;bean&gt; 태그로 등록된 bean을 생성하는 컨테이너</li></ul>
AnnotationConfigApplicationContext	<ul style="list-style-type: none"><li>자바 애너테이션(Java Annotation)에 의해서 bean으로 등록된 bean을 생성하는 컨테이너</li><li>@Configuration, @Bean 애너테이션 등이 필요함</li></ul>

# WebApplicationContext

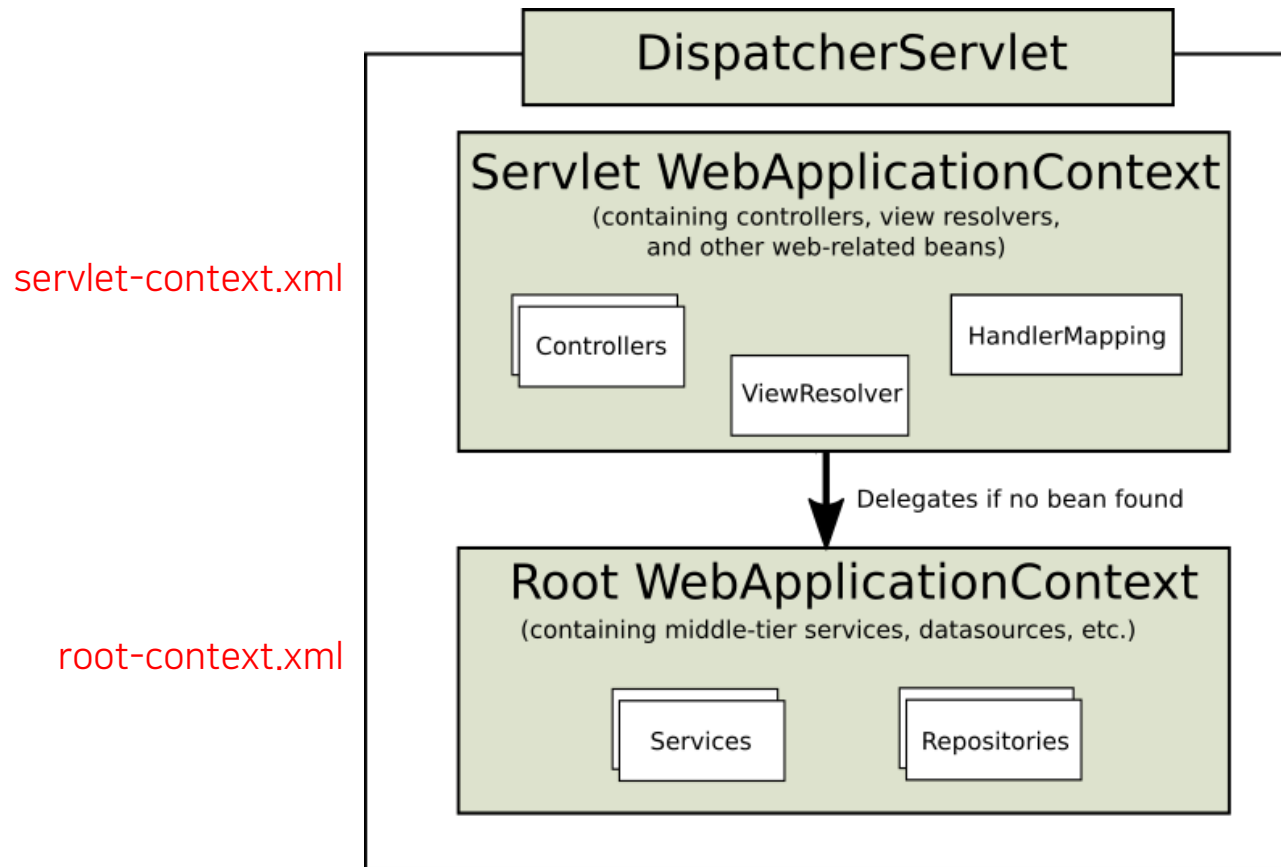
## ■ root-context.xml

- View와 관련이 없는 Bean을 정의하는 파일
- Service, Repository(DAO) 등과 관련된 Bean을 등록함
- <bean> 태그를 이용해서 Bean을 등록

## ■ servlet-context.xml

- 요청에 관련된 Bean을 정의하는 파일
- Controller, ViewResolver, Interceptor 등과 관련된 Bean을 등록함
- <beans:bean> 태그를 이용해서 Bean을 등록함

# WebApplicationContext



찾는 bean 이  
servlet-context.xml 에 없으면  
root-context.xml 에서 검색한다.

(출처 : <https://docs.spring.io/>)

# <property>

## ■ <property> 태그

- Setter를 이용해서 값을 전달하고 저장하는 태그

## ■ 형식1

- <property> 태그와 <value> 태그

```
<property name="필드">  
  <value>값</value>  
</property>
```

- <property> 태그와 value 속성

```
<property name="필드" value="값" />
```

## ■ 형식2

- <property> 태그와 <ref> 태그

```
<property name="필드">  
  <ref bean="bean이름" />  
</property>
```

- <property> 태그와 ref 속성

```
<property name="필드" ref="bean이름" />
```

# <property>

```
<bean id="contact" class="com.min.app.Contact">  
  <property name="mobile" value="010-1111-1111" />  
  <property name="email" value="user@min.com" />  
</bean>
```

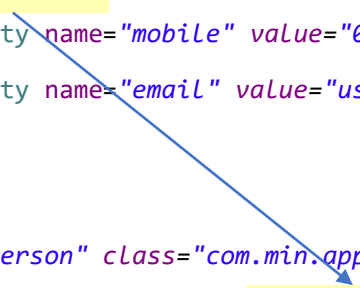
Setter를 이용한 값의 전달

```
package com.min.app;  
  
public class Contact {  
  
    private String mobile;  
    private String email;  
  
    public String getMobile() {  
        return mobile;  
    }  
  
    public void setMobile(String mobile) {  
        this.mobile = mobile;  
    }  
  
    public String getEmail() {  
        return email;  
    }  
  
    public void setEmail(String email) {  
        this.email = email;  
    }  
  
}
```

# <property>

```
<bean id="contact" class="com.min.app.Contact">
  <property name="mobile" value="010-1111-1111" />
  <property name="email" value="user@min.com" />
</bean>

<bean id="person" class="com.min.app.Person">
  <property name="contact" ref="contact" />
</bean>
```



다른 bean을 참조할 땐 ref 태그나 ref 속성을 사용함

```
package com.min.app;

public class Person {

    private Contact contact;

    public Contact getContact() {
        return contact;
    }

    public void setContact(Contact contact) {
        this.contact = contact;
    }

}
```

# <constructor-arg>

## ■ <constructor-arg> 태그

- Constructor를 이용해서 값을 전달하고 저장하는 태그
- Constructor에 정의된 매개변수 순서대로 값을 전달해야만 함

## ■ 형식1

- <constructor-arg> 태그와 <value> 태그

```
<constructor-arg>  
  <value>값</value>  
</constructor-arg>
```

- <constructor-arg> 태그와 value 속성

```
<constructor-arg value="값" />
```

## ■ 형식2

- <constructor-arg> 태그와 <ref> 태그

```
<constructor-arg>  
  <ref bean="bean이름" />  
</constructor-arg>
```

- <constructor-arg> 태그와 ref 속성

```
<constructor-arg ref="bean이름" />
```

# <constructor-arg>

```
<bean id="contact" class="com.min.app.Contact">  
  <constructor-arg value="010-1111-1111" />  
  <constructor-arg value="user@min.com" />  
</bean>
```

생성자를 이용한 방식  
매개변수 순서대로 값이 전달됨

```
package com.min.app;  
  
public class Contact {  
  
  private String mobile;  
  private String email;  
  
  public Contact(String mobile, String email) {  
    this.mobile = mobile;  
    this.email = email;  
  }  
  
}
```



# XML 방식의 한계

## ■ XML을 이용한 bean 생성의 한계

- 프로젝트의 규모가 커지면서 XML의 관리가 어려워짐
- XML에서는 자바 데이터를 사용하기가 불편함
- 그로 인해 Java Annotation을 이용하는 새로운 bean 생성 방식 도입
- 기존 XML을 이용하는 방식은 Java Annotation을 이용하는 방식으로 대체되는 중임
- Spring Boot 프로젝트는 bean 생성을 위한 root-context.xml이나 servlet-context.xml 파일을 지원하지 않고 있음

# @Configuration

## ■ @Configuration

- 클래스 레벨의 Annotation
- @Configuration이 명시된 클래스를 bean으로 등록함
- @Configuration으로 등록한 클래스는 메소드 단위로 bean을 만들 수 있음

## ■ @Bean

- 메소드 레벨의 Annotation
- @Bean이 명시된 메소드가 반환하는 값을 bean으로 등록함
- 기본적으로 메소드 이름을 bean의 이름으로 등록함
- @Bean을 사용하는 메소드가 포함된 클래스는 반드시 @Configuration을 사용해야 함

# @Configuration

```
package com.min.app;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {

    @Bean
    Contact contact() {
        Contact contact = new Contact();
        contact.setMobile("010-1111-1111");
        contact.setEmail("user@min.com");
        return contact;
    }
}
```

```
package com.min.app;

public class Contact {

    private String mobile;
    private String email;

    public String getMobile() {
        return mobile;
    }

    public void setMobile(String mobile) {
        this.mobile = mobile;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

# @Component

## ■ @Component

- 클래스 레벨의 Annotation
- @Component가 명시된 클래스는 자동으로 bean으로 등록됨
- @Component가 명시된 클래스를 찾기 위해서는 반드시 미리 컴포넌트를 찾을 위치를 등록해야 하는데, 이를 Component Scan이라고 함
- Spring MVC Project는 servlet-context.xml에 Component Scan이 등록되어 있어 별도의 설정이 필요 없고, Spring Boot Project는 @SpringBootApplication에 Component Scan이 포함되어 있어 별도의 설정이 필요 없음
- 스프링에서 가장 권장하는 bean 생성 방식임

# @Component

## ■ @Component 사용 방법

→ Contact 타입에 이름이 contact 인 bean 이 Spring Container 에 저장된다.

```
package com.min.app;

import org.springframework.stereotype.Component;

@Component
public class Contact {

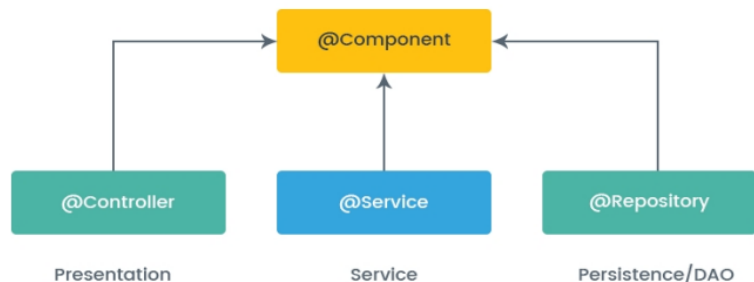
    private String mobile = "010-1111-1111";
    private String email = "user@min.com";

    ...

}
```

# @Component 계층 구조

## ■ @Component 의 계층 구조



## ■ 주요 @Component

애너테이션	의미
@Component	<ul style="list-style-type: none"><li>스tere오 타입의 최상의 객체</li></ul>
@Controller	<ul style="list-style-type: none"><li>요청과 응답을 처리하는 Controller 클래스에서 사용</li><li>Spring MVC 아키텍처에서 자동으로 Controller로 인식됨</li></ul>
@Service	<ul style="list-style-type: none"><li>비즈니스 로직을 처리하는 Service 클래스에서 사용</li><li>Service Interface를 구현하는 ServiceImpl 클래스에서 사용</li></ul>
@Repository	<ul style="list-style-type: none"><li>데이터베이스 접근 객체(DAO)에서 사용</li><li>데이터베이스 처리 과정에서 발생하는 예외를 변환해주는 기능을 포함함</li></ul>

# Component Scan

## ■ Component Scan

- @Component 를 찾아 볼 패키지를 스프링에게 알려 주는 방식
- 방식
  - XML 태그로 등록하기
  - Java Annotation 으로 등록하기

## ■ XML 태그로 등록하기

```
<context:component-scan base-package="com.min.app" />
```

## ■ Java Annotation 으로 등록하기

```
@ComponentScan(basePackages = "com.min.app")
```

```

each: function(e, t, n) {
    var r, i = 0,
        o = e.length,
        a = M(e);
    if (n) {
        if (a) {
            for (; o > i; i++)
                if (r = t.apply(e[i], n), r === !1) break;
        } else
            for (i in e)
                if (r = t.apply(e[i], n), r === !1) break;
    } else if (a) {
        for (; o > i; i++)
            if (r = t.call(e[i], e[i]), r === !1) break;
    } else
        for (i in e)
            if (r = t.call(e[i], e[i]), r === !1) break;
    return e;
},
trim: b && !b.call("\uffff\u00a0") ? function(e) {
    return null == e ? "" : b.call(e);
} : function(e) {
    return null == e ? "" : (e + "").replace(C, "");
},
makeArray: function(e, t) {
    var n = t || [];
    return null != e && (M(Object(e)) ? x.merge(n, "string"
    ),
inArray: function(e, t, n) {
    var r;
    if (t) {
        if (m) return m.call(t, e, n);
        for (r = t.length, r = r ? 0 > n ? Math.max(0, r + n) : n; r < t.length; r++)
            if (n in t && t[r] === e) return r;
    }
}

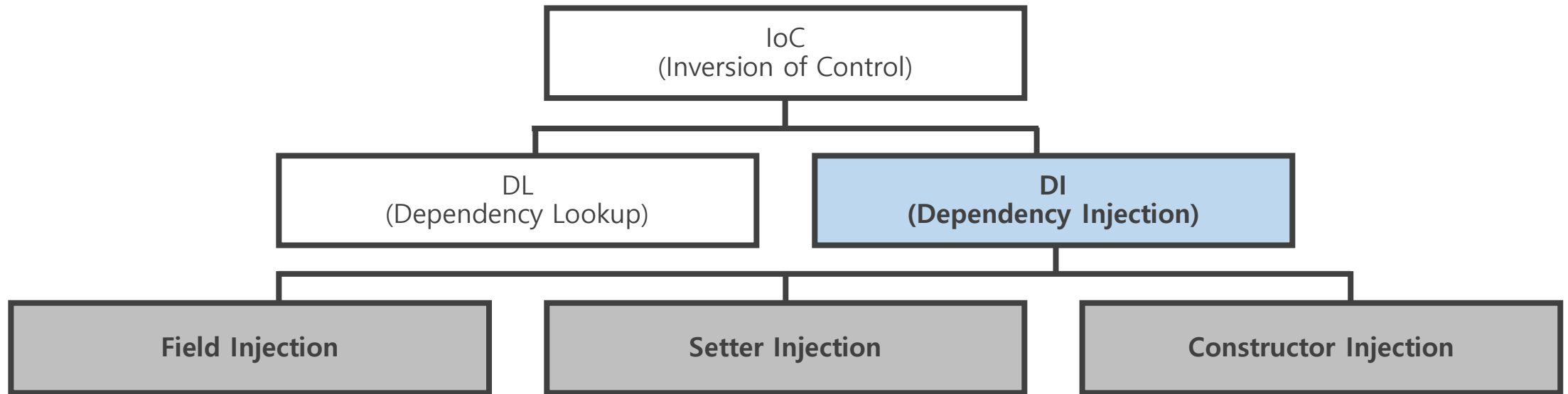
```

## 2. DI



# IoC 구현 방식

- IoC 구현 방식



## ■ Dependency Lookup

- 컨테이너가 애플리케이션 운용에 필요한 객체를 생성하면 클라이언트는 컨테이너가 생성한 객체를 검색(Lookup)해서 사용하는 방식
- 실제 애플리케이션 개발에서 사용하지 않음

## ■ Dependency Injection

- 객체 사이의 의존 관계를 컨테이너가 직접 설정하는 방식
- 스프링 빈 설정 파일에 등록된 정보를 바탕으로 컨테이너가 객체를 처리하는 방식으로 Spring Framework에서 주로 사용하는 방식

# Dependency Injection

## ■ DI

- DI, Dependency Injection
- 컨테이너에 등록된 Bean을 가져오는 방식
- Field Injection, Setter Injection, Constructor Injection 방식이 있음
- @Inject, @Autowired 등의 애노테이션을 이용해서 처리할 수 있음

## ■ DI Annotation

애너테이션	의미
@Autowired	<ul style="list-style-type: none"><li>• bean의 타입이 일치하면 가져옴</li><li>• 동일한 타입의 bean이 여러 개 있는 경우 bean의 이름이 일치하면 가져옴</li><li>• org.springframework.beans.factory.annotation.Autowired</li></ul>
@Qualifier	<ul style="list-style-type: none"><li>• bean의 이름이 일치하면 가져옴</li><li>• org.springframework.beans.factory.annotation.Qualifier</li></ul>
@Inject	<ul style="list-style-type: none"><li>• bean의 타입이 일치하면 가져옴</li><li>• 동일한 타입의 bean이 여러 개 있는 경우 bean의 이름이 일치하면 가져옴</li><li>• javax.inject.Inject</li></ul>

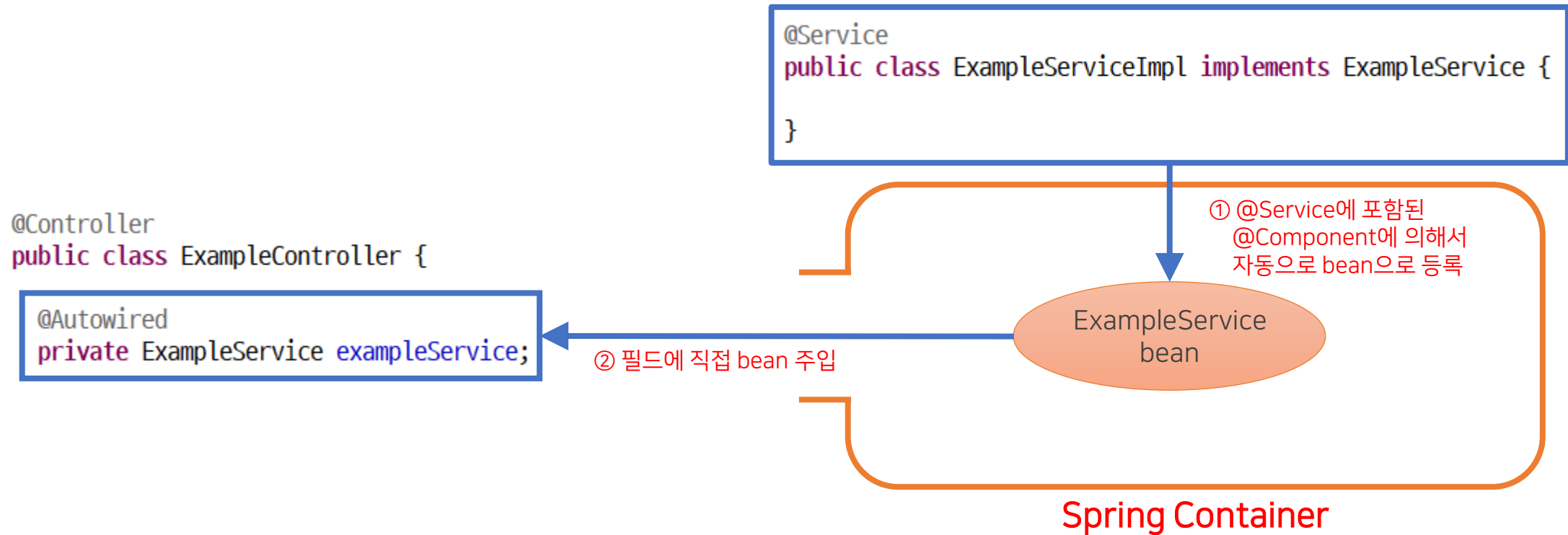
# Field Injection

## ■ Field Injection

- @Autowired가 명시된 필드로 bean을 가져옴
- 모든 필드마다 @Autowired를 명시해야 함
- 코드 작성 시 NULL 여부를 체크하지 않아서 Spring Container에 생성된 bean이 없는 경우 위험할 수 있음
- 코드 작성 시 순환 참조 여부를 미리 확인할 수 없음

# Field Injection

## Field Injection 예시



# Setter Injection

## ■ Setter Injection

- Setter 를 만들 필요는 없음
  - Setter 처럼 매개변수를 필드에 전달하는 형식의 메소드면 가능함
- @Autowired가 명시된 메소드의 매개변수로 bean을 가져옴
- 메소드에 @Autowired를 한 번만 명시하면 모든 매개변수로 bean을 가져오기 때문에 편리함
- 코드 작성 시 NULL 여부를 체크하지 않아서 Spring Container에 생성된 bean이 없는 경우 위험할 수 있음
- 코드 작성 시 순환 참조 여부를 미리 확인할 수 없음

# Setter Injection

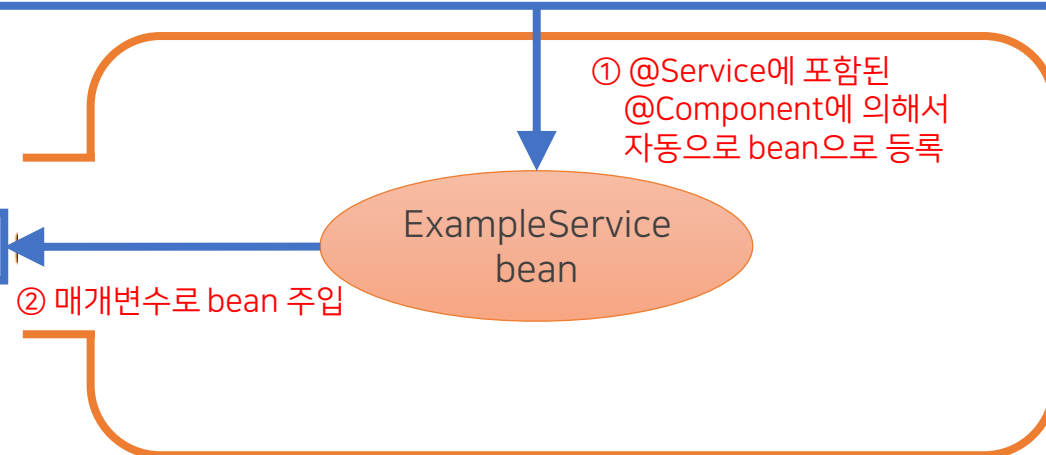
## ■ Setter Injection 예시

```
@Controller
public class ExampleController {

    private ExampleService exampleService;

    @Autowired
    public void setExampleService(ExampleService exampleService) {
        this.exampleService = exampleService;
    }
}
```

```
@Service
public class ExampleServiceImpl implements ExampleService {
}
```



Spring Container

# Constructor Injection

## ■ Constructor Injection

- @Autowired가 명시된 생성자의 매개변수로 bean을 가져옴
- Spring Framework 4.3 이후로 @Autowired 는 생략 가능함
- @Autowired 생략이 가능하므로 Lombok 사용 시 @AllArgsConstructor 또는 @RequiredArgsConstructor 같은 Annotation 사용 가능
- 생성자 주입을 이용하면 필드에 final 처리가 가능해져서 보다 안전한 코드 작성이 가능함
- 스프링에서 가장 추천하는 방식임



# Constructor Injection

## ■ Constructor Injection 예시

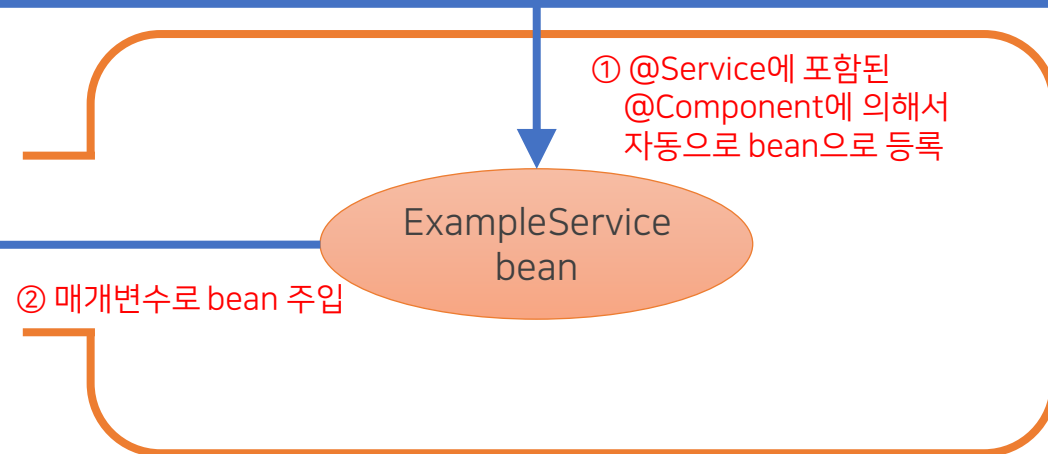
```
@Controller
public class ExampleController {

    private ExampleService exampleService;

    public ExampleController(ExampleService exampleService) {
        this.exampleService = exampleService;
    }
}
```

```
@Service
public class ExampleServiceImpl implements ExampleService {

}
```



Spring Container

# Constructor Injection 사용 권장

## ■ Constructor Injection 이 좋은 점

