필드와 setter 메소드에 @Inject 기능 추가 실습

실습 환경

- https://github.com/slipp/jwp-basic 저장소를 자신의 계정으로 fork한다.
- Fork한 저장소를 https://youtu.be/xid_GG8kL_w 동영상 참고해 로컬 개발 환경을 구축한다.
- jwp-basic 저장소의 **step11-1st-di-framework** 브랜치로 변경한다.
    - 브랜치를 변경하는 방법은 https://youtu.be/VeTjDYI7UVs 동영상을 참고한다.
- src/test/java 디렉토리의 next.WebServerLauncher를 실행한 후 브라우저에서 http://localhost:8080으로 접속해 질문/답변 게시판 서비스 화면이 나타나는지 확인한다.

요구사항

@Inject를 활용해 DI하는 방법은 빈 간의 의존관계를 쉽게 연결할 수 있어 유용하겠다. 단, 현재 DI 프레임워크에서 아쉬운 점이라면 생성자를 통해서만 DI가 가능하다는 것이다. 생성자 이외에 필드(field), setter 메소드를 통해서도 DI를 할 수 있다면 좋겠다. 예를 들어 다음과 같은 방식으로 DI가 가능하도록 기능을 추가하려고 한다.

제약사항: 빈은 한 가지 방식의 Injection만 사용할 수 있다. 예를 들어 생성자 기반으로 Injection을 하는 경우 필드와 setter 메소드 기반 Injection을 사용할 수 없다.

필드 Injection 활용한 DI 예제

```
@Service
public class MyUserService {
    @Inject
    private UserRepository userRepository;

    public void setUserRepository(UserRepository userRepository) {
        this.userRepository = userRepository;
    }
}
```

Setter injection을 활용한 DI 예제

```
@Controller
public class MyUserController {
    private MyUserService userService;

    @Inject
    public void setUserService(MyUserService userService) {
        this.userService = userService;
    }
}
```

DI 프레임워크를 활용한 DI 예제

```
@Repository
public class JdbcQuestionRepository implements QuestionRepository {
}

@Repository
public class JdbcUserRepository implements UserRepository {
}
```

1단계 힌트

—

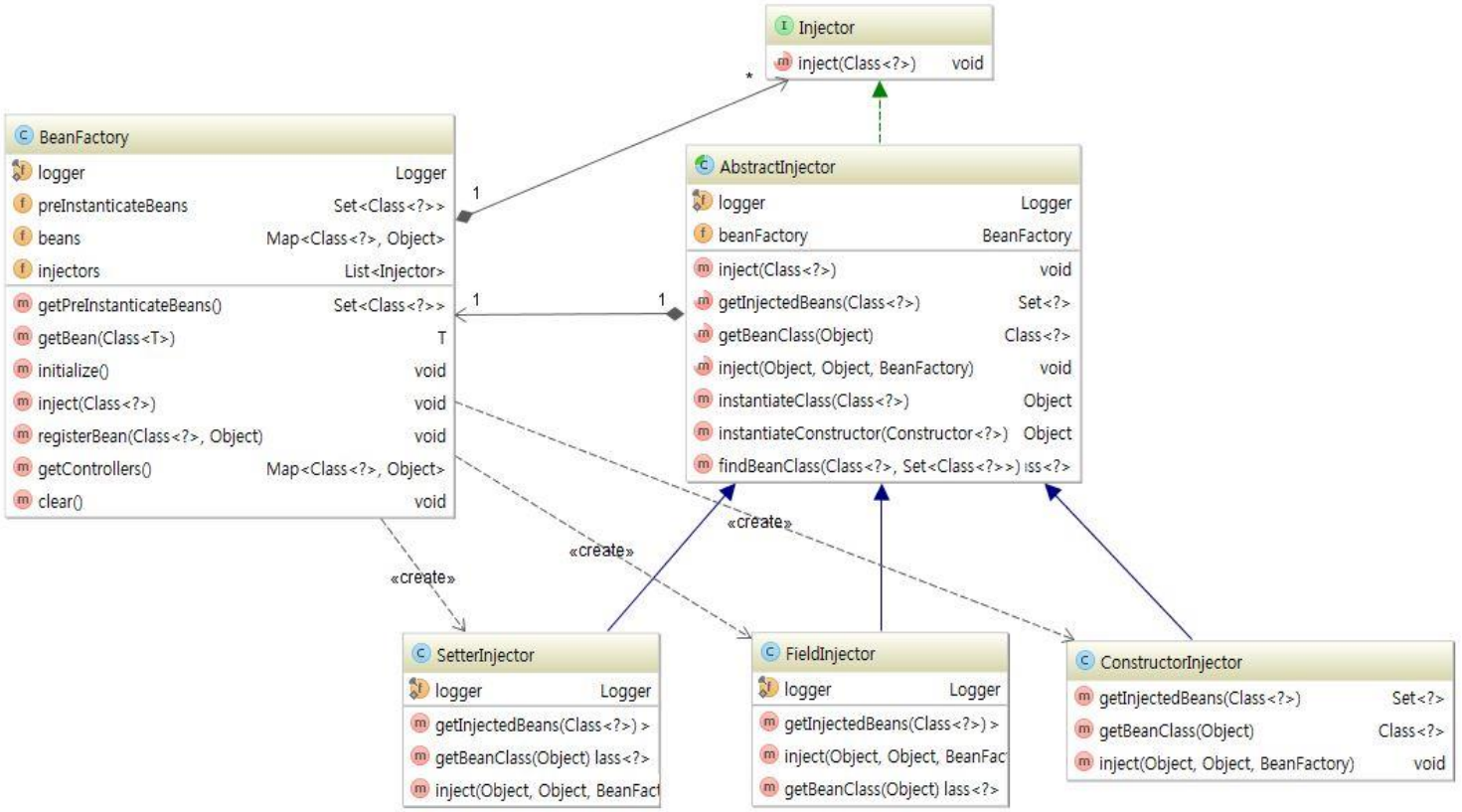
실습

3가지 방식의 Injection을 추상화한 인터페이스를 추가한다.

힌트

```
public interface Injector {
    void inject(Class<?> clazz);
}
```

3개의 Injector 구현체를 구현한 클래스 다이어그램



2단계 힌트

—

실습

생성자 구현체(ConstructorInjector)는 @Inject가 설정되어 있는 생성자를 가지는 클래스를 찾는다. 생성자의 인자에 해당하는 빈이 BeanFactory에 등록되어 있는지 확인해 보고 빈이 등록되어 있지 않으면 빈 인스턴스를 생성한 후 DI를 한다.

실습

필드 구현체(FieldInjector)와 setter 메소드 구현체(SetterInjector)도 같은 방식으로 구현하면 된다. 필드 구현체의 경우 클래스에 @Inject가 설정되어 있는 모든 필드는 찾는다. 필드를 순환하면서 필드 타입(클래스)에 해당하는 빈이 BeanFactory에 등록되어 있으면 이 빈을 활용하고, 등록되어 있지 않으면 빈 인스턴스를 생성한 후 DI를 한다.

힌트

```
public class FieldInjector implements Injector {
    [...]

    @Override
    public void inject(Class<?> clazz) {
        instantiateClass(clazz);
        Set<Field> injectedFields = BeanFactoryUtils.getInjectedFields(clazz);
        for (Field field : injectedFields) {
            Class<?> concreteClazz = …// 인터페이스인 경우 구현 클래스 찾아야 함.
            Object bean = beanFactory.getBean(concreteClazz);
            if (bean == null) {
                bean = instantiateClass(concreteClazz);
            }
            try {
                field.setAccessible(true);
                field.set(beanFactory.getBean(field.getDeclaringClass()), bean);
            } catch (IllegalAccessException ¦ IllegalArgumentException e) {
                logger.error(e.getMessage());
            }
        }
    }
}
```

힌트

```
public class SetterInjector implements Injector {
    [...]

    @Override
    public void inject(Class<?> clazz) {
        instantiateClass(clazz);
        Set<Method> injectedMethods = BeanFactoryUtils.getInjectedMethods(clazz);
        for (Method method : injectedMethods) {
            logger.debug("invoke method : {}", method);
            Class<?>[] paramTypes = method.getParameterTypes();
            if (paramTypes.length != 1) {
                throw new IllegalStateException("DI할 메소드 인자는 하나여야 합니다.");
            }

            Class<?> concreteClazz = ...// 인터페이스인 경우 구현 클래스 찾아야 함.
            Object bean = beanFactory.getBean(concreteClazz);
            if (bean == null) {
                bean = instantiateClass(concreteClazz);
            }
            try {
                method.invoke(beanFactory.getBean(method.getDeclaringClass()), bean);
            } catch (IllegalAccessException | IllegalArgumentException |
InvocationTargetException e) {
                logger.error(e.getMessage());
            }
        }
    }
}
```

실습

구현 과정에서 Injector에 중복 코드가 발생한다. 중복 코드는 AbstractInjector와 같은 추상 클래스를 만들어 제거한다.

힌트

```
public abstract class AbstractInjector implements Injector {
    private BeanFactory beanFactory;

    public AbstractInjector(BeanFactory beanFactory) {
        this.beanFactory = beanFactory;
    }

    @Override
    public void inject(Class<?> clazz) {
        instantiateClass(clazz);
        Set<?> injectedBeans = getInjectedBeans(clazz);
        for (Object injectedBean : injectedBeans) {
            Class<?> beanClass = getBeanClass(injectedBean);
            inject(injectedBean, instantiateClass(beanClass), beanFactory);
        }
    }

    abstract Set<?> getInjectedBeans(Class<?> clazz);

    abstract Class<?> getBeanClass(Object injectedBean);

    abstract void inject(Object injectedBean, Object bean, BeanFactory beanFactory);

    private Object instantiateClass(Class<?> clazz) {
        [...]
    }

    private Object instantiateConstructor(Constructor<?> constructor) {
        [...]
    }
}
```

추가 요구사항

—

요구사항

지금까지의 과정을 통해 DI 프레임워크를 완료했다면 다음 단계는 앞에서 구현한 MVC 프레임워크와의 통합이 필요하다. 여기서 구현한 DI 프레임워크를 활용할 경우 앞에서 @Controller이 설정되어 있는 클래스를 찾는 ControllerScanner를 DI 프레임워크가 있는 패키지로 이동해 @Controller, @Service, @Repository에 대한 지원이 가능하도록 개선한다.

클래스 이름도 @Controller 애노테이션만 찾던 역할에서 @Service, @Repository 애노테이션까지 확대 되었으니 BeanScanner로 이름을 리팩토링한다.

MVC 프레임워크의AnnotationHandlerMapping이 BeanFactory와 BeanScanner를 활용해 동작하도록 리팩토링한다.

힌트

이 리팩토링 과정은 생각보다 간단하다. BeanScanner는@Controller, @Service, @Repository이 설정되어 있는 모든 클래스를 찾아 Set에 저장하도록 하면 된다. 기존의 ControllerScanner와 같이 이 단계에서 빈 생성을 담당하지 않아도 된다.

AnnotationHandlerMapping은 @Controller가 설정되어 있는 빈만 관심이 있다. 따라서 BeanFactory에 getControllers() 메소드를 추가해 @Controller가 설정되어 있는 빈 목록을 Map<Class<?>, Object>으로 제공하면 된다.

```
Class<?> clazz = QnaController.class;
Annotation annotation = clazz.getAnnotation(Controller.class);
```