Chap04. DI Annotation

1. @Autowired

> 💡 `@Autowired` 어노테이션은 `Type`을 통한 DI를 할 때 사용한다. 스프링 컨테이너가 알아서 해당 타입의 Bean을 찾아서 주입해준다.

아래 코드는 테스트에 공통적으로 사용 할 `BookDTO`, `BookDAO`, `BookDAOImpl` 클래스이다.

1. BookDTO

```
@Getter @Setter @ToString
@AllArgsConstructor
public class BookDTO {

    private int sequence;        //도서번호
    private int isbn;            //isbn
    private String title;        //제목
    private String author;       //저자
    private String publisher;    //출판사
    private Date createdDate;    //출판일

}
```

2. BookDAO

```
public interface BookDAO {

    /* 도서 목록 전체 조회 */
    List<BookDTO> selectBookList();

    /* 도서 번호로 도서 조회 */
    BookDTO selectOneBook(int sequence);
}
```

3. BookDAOImpl

```
/* @Repository : @Component의 세분화 어노테이션의 한 종류로 DAO 타입의 객체에 사용한다. */
@Repository("bookDAO")
public class BookDAOImpl implements BookDAO {

    private Map<Integer, BookDTO> bookList;

    public BookDAOImpl() {
        bookList = new HashMap<>();
        bookList.put(1, new BookDTO(1, 123456, "자바의 정석", "남궁성", "도우출판", new Date()));
        bookList.put(2,
          new BookDTO(2, 654321, "칭찬은 고래도 춤추게 한다", "고래", "고래출판", new Date()));
    }
```

```
    @Override
    public List<BookDTO> selectBookList() {
        return new ArrayList<>(bookList.values());
    }

    @Override
    public BookDTO selectOneBook(int sequence) {
        return bookList.get(sequence);
    }
}
```

1-1. 필드(field) 주입

```
/* @Service : @Component의 세분화 어노테이션의 한 종류로 Service 계층에서 사용한다. */
@Service("bookServiceField")
public class BookService {

    /* BookDAO 타입의 빈 객체를 이 프로퍼티에 자동으로 주입해준다. */
    @Autowired
    private BookDAO bookDAO;

    /* 도서 목록 전체 조회 */
    public List<BookDTO> selectAllBooks(){

        return bookDAO.selectBookList();
    }

    /* 도서 번호로 도서 조회 */
    public BookDTO searchBookBySequence(int sequence) {

        return bookDAO.selectOneBook(sequence);
    }
}
```

`private BookDAO bookDAO = new BookDAOImpl();` 와 같이 필드를 선언한다면 `BookService` 클래스는 `BookDAOImpl` 클래스의 변경에 직접적으로 영향을 받는 강한 결합을 가지게 된다. 객체간의 결합을 느슨하게 하기 위해 `new BookDAOImpl()` 와 같은 직접적으로 객체를 생성하는 생성자 구문을 제거하고 `필드`에 `@Autowired` 어노테이션을 작성할 수 있다. 그러면 스프링 컨테이너는 BookService 빈 객체 생성 시 `BookDAO` 타입의 빈 객체를 찾아 의존성을 주입해준다.

스프링 컨테이너를 생성하여 `@Repository`, `@Service` 등의 어노테이션이 작성 된 클래스가 빈 스캐닝을 통해 잘 등록 되었는지, 또한 객체의 의존 관계에 따라 `@Autowired` 어노테이션을 통해 의존성 주입이 되었는지를 테스트한다.

```
/* AnnotationConfigApplicationContext 생성자에 basePackages 문자열을 전달하며 ApplicationContext 생성한다. */
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section01");

BookService bookService = context.getBean("bookServiceField", BookService.class);

/* 전체 도서 목록 조회 후 출력 확인 */
bookService.selectAllBooks().forEach(System.out::println);

/* 도서번호로 검색 후 출력 확인*/
```

```
System.out.println(bookService.searchBookBySequence(1));
System.out.println(bookService.searchBookBySequence(2));
```

▼ 실행 결과

```
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
```

1-2. 생성자(constoructor) 주입

```
/* @Service : @Component의 세분화 어노테이션의 한 종류로 Service 계층에서 사용한다. */
@Service("bookServiceConstructor")
public class BookService {

    private final BookDAO bookDAO;

    /* BookDAO 타입의 빈 객체를 생성자에 자동으로 주입해준다. */
    @Autowired
    public BookService(BookDAO bookDAO) {
        this.bookDAO = bookDAO;
    }

    public List<BookDTO> selectAllBooks(){

        return bookDAO.selectBookList();
    }

    public BookDTO searchBookBySequence(int sequence) {

        return bookDAO.selectOneBook(sequence);
    }

}
```

`생성자`에도 `@Autowired` 어노테이션을 작성할 수 있다. 그러면 스프링 컨테이너는 BookService 빈 객체 생성 시 `BookDAO` 타입의 빈 객체를 찾아 의존성을 주입해준다.

`Spring 4.3 버전` 이후로는 생성자가 한 개 뿐이라면 `@Autowired` 어노테이션을 생략해도 자동으로 생성자 주입이 동작한다. 단, 생성자가 1개 이상일 경우에는 명시적으로 작성을 해주어야 한다. 위의 코드에 기본 생성자를 추가로 작성하고 매개변수 생성자에 `@Autowired` 어노테이션을 생략하게 되면 생성자 주입이 동작하지 않아 오류가 발생한다.

생성자 주입의 `장점`

- 객체가 생성 될 때 모든 의존성이 주입 되므로 의존성을 보장할 수 있다.

    - 필드 주입/세터 주입은 의존성이 있는 객체가 생성되지 않아도 객체 생성은 가능하여 메소드가 호출 되면(런타임) 오류가 발생한다.
    - 생성자 주입은 의존성이 있는 객체가 생성되지 않으면 객체 생성이 불가능하여 어플리케이션 실행 시점에 오류가 발생한다.
- 객체의 불변성을 보장할 수 있다.
    - 필드에 final 키워드를 사용 할 수 있고 객체 생성 이후 의존성을 변경할 수 없어 안정성이 보장된다.
- 코드 가독성이 좋다.
    - 해당 객체가 어떤 의존성을 가지고 있는지 명확히 알 수 있다.
- DI 컨테이너와의 결합도가 낮기 때문에 테스트 하기 좋다.
    - 스프링 컨테이너 없이 테스트를 할 수 있다.

스프링 컨테이너를 생성하여 `@Repository`, `@Service` 등의 어노테이션이 작성 된 클래스가 빈 스캐닝을 통해 잘 등록 되었는지, 또한 객체의 의존 관계에 따라 `@Autowired` 어노테이션을 통해 의존성 주입이 되었는지를 테스트한다.

```
/* AnnotationConfigApplicationContext 생성자에 basePackages 문자열을 전달하며 ApplicationContext 생성한다. */
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section01");

BookService bookService = context.getBean("bookServiceConstructor", BookService.class);

/* 전체 도서 목록 조회 후 출력 확인 */
bookService.selectAllBooks().forEach(System.out::println);

/* 도서번호로 검색 후 출력 확인*/
System.out.println(bookService.searchBookBySequence(1));
System.out.println(bookService.searchBookBySequence(2));
```

▼ 실행 결과

```
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
```

1-3. 세터(setter) 주입

```
/* @Service : @Component의 세분화 어노테이션의 한 종류로 Service 계층에서 사용한다. */
@Service("bookServiceSetter")
public class BookService {
```

```
    private BookDAO bookDAO;

    /* BookDAO 타입의 빈 객체를 setter에 자동으로 주입해준다. */
    @Autowired
    public void setBookDAO(BookDAO bookDAO) {
        this.bookDAO = bookDAO;
    }

    public List<BookDTO> selectAllBooks(){

        return bookDAO.selectBookList();
    }

    public BookDTO searchBookBySequence(int sequence) {

        return bookDAO.selectOneBook(sequence);
    }
}
```

`setter` 메소드에도 `@Autowired` 어노테이션을 작성할 수 있다. 그러면 스프링 컨테이너는 BookService 빈 객체 생성 시 `BookDAO` 타입의 빈 객체를 찾아 의존성을 주입해준다.

스프링 컨테이너를 생성하여 `@Repository`, `@Service` 등의 어노테이션이 작성 된 클래스가 빈 스캐닝을 통해 잘 등록 되었는지, 또한 객체의 의존 관계에 따라 `@Autowired` 어노테이션을 통해 의존성 주입이 되었는지를 테스트한다.

```
/* AnnotationConfigApplicationContext 생성자에 basePackages 문자열을 전달하며 ApplicationContext 생성한다. */
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section01");

BookService bookService = context.getBean("bookServiceSetter", BookService.class);

/* 전체 도서 목록 조회 후 출력 확인 */
bookService.selectAllBooks().forEach(System.out::println);

/* 도서번호로 검색 후 출력 확인*/
System.out.println(bookService.searchBookBySequence(1));
System.out.println(bookService.searchBookBySequence(2));
```

▼ 실행 결과

```
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=1, isbn=123456, title=자바의 정석, author=남궁성, publisher=도우출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
BookDTO(sequence=2, isbn=654321, title=칭찬은 고래도 춤추게 한다, author=고래, publisher=고래출판,
  createdDate=Sun May 28 20:19:12 KST 2023)
```

2. 더 많은 DI Annotation

`@Autowired` 어노테이션은 가장 보편적으로 사용 되는 의존성 주입 Annotation이다. `@Autowired` 와 함께 사용하거나 또는 대체해서 사용할 수 있는 어노테이션을 학습한다.

아래 코드는 테스트에 공통적으로 사용 할 `Pokemon`, `Charmander`, `Pikachu`, `Squirtle` 클래스이다.

1. Pokemon

```
public interface Pokemon {

  /* 공격하다 */
  void attack();
}
```

2. Charmander

```
@Component
public class Charmander implements Pokemon {

  @Override
  public void attack() {
    System.out.println("파이리 불꽃 공격🔥");
  }
}
```

3. Pikachu

```
@Component
public class Pikachu implements Pokemon {

  @Override
  public void attack() {
    System.out.println("피카츄 백만볼트⚡");
  }
}
```

4. Squirtle

```
@Component
public class Squirtle implements Pokemon {

  @Override
  public void attack() {
    System.out.println("꼬부기 물대포 발사🌊");
  }
}
```

2-1. @Primary

> 💡 `@Primary` 어노테이션은 여러 개의 빈 객체 중에서 우선순위가 가장 높은 빈 객체를 지정하는 어노테이션이다.

생성자로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다.

```
@Service("pokemonServicePrimary")
public class PokemonService {

  private Pokemon pokemon;

  @Autowired
  public PokemonService(Pokemon pokemon) {
    this.pokemon = pokemon;
  }

  public void pokemonAttack() {
    pokemon.attack();
  }

}
```

`Charmander`, `Pikachu`, `Squirtle`, `PokemonService` 를 빈 스캐닝 할 수 있는 basePackages를 설정하여 스프링 컨테이너를 생성한다.

```
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section02");

PokemonService pokemonService = context.getBean("pokemonServicePrimary", PokemonService.class);

pokemonService.pokemonAttack();
```

▼ 실행 결과

```
Exception in thread "main" org.springframework.beans.factory.UnsatisfiedDependencyException:
Error creating bean with name 'pokemonServicePrimary' defined in file 파일 경로 :
Unsatisfied dependency expressed through constructor parameter 0;
nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No qualifying bean of type 'com.ohgiraffers.section02.common.Pokemon' available:
expected single matching bean but found 3: charmander,pikachu,squirtle
...생략
```

스프링 컨테이너 내부에 `Pokemon` 타입의 빈 객체가 charmander,pikachu,squirtle 3개가 있어 1개의 객체를 `PokemonService`의 생성자로 전달할 수 없어 오류가 발생했음을 확인할 수 있다.

`Charmander`, `Pikachu`, `Squirtle` 중에서 `Charmander` 빈 객체를 우선적으로 주입받도록 `@Primary` 어노테이션을 설정한다.

```
@Component
@Primary
public class Charmander implements Pokemon {

    @Override
    public void attack() {
        System.out.println("파이리 불꽃 공격🔥");
    }
}
```

`@Primary` 어노테이션을 설정하면 `@Autowired`로 동일한 타입의 여러 빈을 찾게 되는 경우 자동으로 연결 우선 시 할 타입으로 설정 된다.

동일한 타입의 클래스 중 한 개만 `@Primary` 어노테이션을 사용할 수 있다.

`Charmander` 빈 객체에 `@Primary` 어노테이션이 설정되어 있으므로, `PokemonService`의 생성자로 `Pokemon` 객체를 주입받으면 `Charmander` 빈 객체가 우선적으로 주입된다.

▼ 실행 결과

```
파이리 불꽃 공격🔥
```

2-2. @Qualifier

> 💡 `@Qualifier` 어노테이션은 여러 개의 빈 객체 중에서 특정 빈 객체를 이름으로 지정하는 어노테이션이다.

2-2-1. 필드 주입

필드로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다. `@Autowired` 어노테이션과 함께 `@Qualifier` 어노테이션을 사용하여 빈 이름을 통해 주입할 빈 객체를 지정한다.

```
@Service("pokemonServiceQualifier")
public class PokemonService {

  /* @Qualifier 어노테이션을 사용하여 pikachu 빈 객체를 지정한다. */
  @Autowired
  @Qualifier("pikachu")
  private Pokemon pokemon;

  public void pokemonAttack() {
    pokemon.attack();

}
```

`Charmander`, `Pikachu`, `Squirtle`, `PokemonService` 를 빈 스캐닝 할 수 있는 basePackages를 설정하여 스프링 컨테이너를 생성한다.

```
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section02");

PokemonService pokemonService = context.getBean("pokemonServiceQualifier", PokemonService.class);

pokemonService.pokemonAttack();
```

▼ 실행 결과

```
피카츄 백만볼트⚡
```

`@Primary` 어노테이션과 `@Qualifier` 어노테이션이 함께 쓰였을 때 `@Qualifier` 가 우선한다는 것도 결과를 통해 확인할 수 있다.

2-2-2. 생성자 주입

생성자 주입의 경우 `@Qualifier` 어노테이션은 메소드의 파라미터 앞에 기재한다. 역시 빈 이름을 통해 주입할 빈 객체를 지정한다.

```
@Service("pokemonServiceQualifier")
public class PokemonService {

  private Pokemon pokemon;

  /* @Qualifier 어노테이션을 사용하여 squirtle 빈 객체를 지정한다. */
  @Autowired
  public PokemonService(@Qualifier("squirtle") Pokemon pokemon) {
    this.pokemon = pokemon;
  }

  public void pokemonAttack() {
    pokemon.attack();
  }
}
```

▼ 실행 결과

```
꼬부기 물대포 발사🌊
```

2-3. Collection

같은 타입의 빈을 여러 개 주입 받고 싶다면 `Collection` 타입을 활용할 수 있다.

2-3-1. List 타입

`List<Pokemon>` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다.

```
@Service("pokemonServiceCollection")
public class PokemonService {

  /* 1. List 타입으로 주입 */
  private List<Pokemon> pokemonList;

  @Autowired
  public PokemonService(List<Pokemon> pokemonList) {
    this.pokemonList = pokemonList;
  }

  public void pokemonAttack() {
    pokemonList.forEach(Pokemon::attack);
  }
}
```

`Charmander`, `Pikachu`, `Squirtle`, `PokemonService` 를 빈 스캐닝 할 수 있는 basePackages를 설정하여 스프링 컨테이너를 생성한다.

```
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section02");

PokemonService pokemonService = context.getBean("pokemonServiceCollection", PokemonService.class);

pokemonService.pokemonAttack();
```

▼ 실행 결과

```
파이리 불꽃 공격🔥
피카츄 백만볼트⚡
꼬부기 물대포 발사🌊
```

bean 이름의 사전순으로 List에 추가 되어 모든 Pokemon 타입의 빈이 주입 된다.

2-3-2. Map 타입

또는 `Map<String, Pokemon>` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다.

```
@Service("pokemonServiceCollection")
public class PokemonService {

  /* 2. Map 타입으로 주입 */
  private Map<String, Pokemon> pokemonMap;

  @Autowired
  public PokemonService(Map<String, Pokemon> pokemonMap) {
    this.pokemonMap = pokemonMap;
  }

  public void pokemonAttack() {
      pokemonMap.forEach((k, v) -> {
          System.out.println("key : " + k);
          System.out.print("공격 : ");
          v.attack();
      });
  }
}
```

▼ 실행 결과

```
key : charmander
공격 : 파이리 불꽃 공격🔥
key : pikachu
공격 : 피카츄 백만볼트⚡
key : squirtle
공격 : 꼬부기 물대포 발사🌊
```

bean 이름의 사전순으로 Map에 추가 되어 모든 Pokemon 타입의 빈이 주입 된다.

2-4. @Resource

> 💡 `@Resource` 어노테이션은 자바에서 제공하는 기본 어노테이션이다. `@Autowired`와 같은 스프링 어노테이션과 다르게 `name` 속성 값으로 의존성 주입을 할 수 있다.

해당 어노테이션은 사용하기 전 라이브러리 의존성 추가가 필요하므로 Maven Repository에서 `javax annoataion`을 검색하여 `build.gradle.kts` 파일에 아래와 같은 구문을 추가한다.

```
dependencies {
  implementation 'javax.annotation:javax.annotation-api:1.3.2'
  ...생략
}
```

2-4-1. 이름으로 주입

필드로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다. `@Resource` 어노테이션의 `name` 속성에 주입할 빈 객체의 이름을 지정한다.

```
@Service("pokemonServiceResource")
public class PokemonService {

  /* pikachu 이름의 빈 지정 */
  @Resource(name = "pikachu")
  private Pokemon pokemon;

  public void pokemonAttack() {
    pokemon.attack();
  }
}
```

`Charmander`, `Pikachu`, `Squirtle`, `PokemonService`를 빈 스캐닝 할 수 있는 basePackages를 설정하여 스프링 컨테이너를 생성한다.

```
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section02");

PokemonService pokemonService = context.getBean("pokemonServiceResource", PokemonService.class);

pokemonService.pokemonAttack();
```

▼ 실행 결과

```
피카츄 백만볼트⚡
```

2-4-2. 타입으로 주입

`List<Pokemon>` 타입으로 변경한 뒤 `name` 속성을 따로 기재하지 않고 동작시킬 수 있다. 기본적으로는 `name` 속성을 통해 주입하지만 `name` 속성이 없을 경우 `Type`을 통해 의존성 주입한다.

```
@Service("pokemonServiceResource")
public class PokemonService {

  @Resource
  private List<Pokemon> pokemonList;

  public void pokemonAttack() {
    pokemonList.forEach(Pokemon::attack);
  }
}
```

▼ 실행 결과

```
파이리 불꽃 공격🔥
피카츄 백만볼트⚡
꼬부기 물대포 발사🌊
```

bean 이름의 사전순으로 List에 추가 되어 모든 Pokemon 타입의 빈이 주입 된다.

사용 시 유의할 점은 `필드 주입` 과 `세터 주입` 은 가능하지만 `생성자 주입` 은 불가능하다는 점이다.

2-5. @Inject

> 💡 `@Inject` 어노테이션은 자바에서 제공하는 기본 어노테이션이다. `@Autowired` 어노테이션과 같이 `Type` 으로 빈을 의존성 주입한다.

해당 어노테이션은 사용하기 전 라이브러리 의존성 추가가 필요하므로 Maven Repository에서 `javax inject` 을 검색하여 `build.gradle.kts` 파일에 아래와 같은 구문을 추가한다.

```
dependencies {
  implementation 'javax.inject:javax.inject:1'
  ...생략
}
```

2-5-1. 필드 주입

필드로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다. `@Inject` 어노테이션은 `Type` 으로 의존성 주입하므로 3개의 동일한 타입의 빈이 있는 현재 상황에서는 오류가 발생한다. 따라서 `@Named` 어노테이션을 함께 사용해서 빈의 이름을 지정하면 해당 빈을 의존성 주입할 수 있다.

```
@Service("pokemonServiceInject")
public class PokemonService {

  /* 1. 필드 주입 */
  @Inject
  @Named("pikachu")
  private Pokemon pokemon;
```

```
  public void pokemonAttack() {
    pokemon.attack();
  }
}
```

`Charmander`, `Pikachu`, `Squirtle`, `PokemonService` 를 빈 스캐닝 할 수 있는 basePackages를 설정하여 스프링 컨테이너를 생성한다.

```
ApplicationContext context = new AnnotationConfigApplicationContext("com.ohgiraffers.section02");

PokemonService pokemonService = context.getBean("pokemonServiceInject", PokemonService.class);

pokemonService.pokemonAttack();
```

▼ 실행 결과

```
피카츄 백만볼트⚡
```

2-5-2. 생성자 주입

생성자로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다.

```
@Service("pokemonServiceInject")
public class PokemonService {

  private Pokemon pokemon;

  /* 2. 생성자 주입 */
  @Inject
  public PokemonService(@Named("pikachu") Pokemon pokemon) {
    this.pokemon = pokemon;
  }

  public void pokemonAttack() {
    pokemon.attack();
  }
}
```

`@Named` 어노테이션의 경우 메소드 레벨, 파라미터 레벨에서 둘 다 사용 가능하다.

▼ 실행 결과

```
피카츄 백만볼트⚡
```

2-5-3. 세터 주입

세터로 `Pokemon` 타입의 객체를 의존성 주입 받는 `PokemonService` 클래스를 선언한다.

```
@Service("pokemonServiceInject")
public class PokemonService {

  private Pokemon pokemon;

  /* 3. 세터 주입 */
  @Inject
  public void setPokemon(@Named("pikachu") Pokemon pokemon) {
    this.pokemon = pokemon;
  }

  public void pokemonAttack() {
    pokemon.attack();
  }
}
```

▼ 실행 결과

```
피카츄 백만볼트⚡
```

`@Inject`는 `필드 주입`, `생성자 주입`, `세터 주입` 이 모두 가능하다.

정리

DI는 스프링 프레임워크에서 매우 중요한 개념 중 하나로, 개발자는 객체 간의 의존성을 직접 관리하지 않고 스프링 컨테이너가 객체 간의 의존성을 주입해주는 방식으로 관리할 수 있다.

다양한 DI 어노테이션이 있는데 각각의 특징과 사용 방식이 다르다.

	@Autowried	@Resource	@Inject
제공	Spring	Java	Java
지원 방식	필드, 생성자, 세터	필드, 세터	필드, 생성자, 세터
빈 검색 우선 순위	타입 → 이름	이름 → 타입	타입 → 이름
빈 지정 문법	@Autowired @Qualifier(”name”)	@Resource(name=”name”)	@Inject @Named(”name”)