1주차 키워드 정리

- **키워드**
  - Java Generic
  - 자바 제어자와 접근 제어자에 대해 알아보기
  - 싱글톤(Singleton)
  - Spring의 각각의 의존성 주입 방법 특징 및 생성자 주입 방식의 장점
  - JAR, WAR의 차이점

Java Generic

Java의 자료형은 크게 primitive type(원시 타입), reference type(참조 타입) 두 가지로 나뉜다.

primitive type에 대응되는 값들을 감싸고 있는 Wrapper class의 장점 중 하나는 `Generic으로 사용될 수 있다.` 는 것이다.

자바에서 제네릭(Generic)은 **클래스 내부에서 사용할 데이터 타입을 외부에서 지정하는 기법**을 의미한다.

Data type을 특정한 하나로 정하지 않고 사용할 때마다 바뀔 수 있게 범용적이고 포괄적으로 지정하는 것이다.

아래 예제를 보자.

```
/**
 * Generic version of the Box class.
 * @param <T> the type of the value being boxed
 */
public class Box<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

위의 예제에서 사용된 'T'를 타입 변수(type variable)라고 하며, 임의의 참조형 타입을 의미한다.

'T'는 객체를 생성할 때 구체적인 타입으로 변경된다.

```
Box<String> box = new Box<String>();
```

```
Box<Integer> box = new Box<Integer>();
```

('T'뿐만 아니라 어떠한 문자를 사용해도 상관없음, 여러 개의 타입 변수는 쉼표(,)로 구분하여 명시)

타입 변수는 클래스에서뿐만 아니라 메소드의 매개변수나 반환값으로도 사용할 수 있다.

```
public <T> List<T> fromArrayToList(T[] a) {
  return Arrays.stream(a)
    .collect(Collectors.toList());
}
```

- **제네릭(Generic)의 장점**
  - 컴파일 시 강한 타입 체크를 할 수 있다.
  - 타입변환(casting)을 제거한다.

제어자와 접근 제어자

제어자(modifier)

- 클래스, 변수, 메소드의 선언부에 사용되어 부가적인 의미를 부여한다.
- 접근 제어자 : public, protected, default, private
- 그 외 : static, final, abstract, native, transient, synchronized, volatile, strictfp
- 하나의 대상에 여러 개의 제어자 조합 가능, but 접근제어자는 단 하나만 사용 가능!

접근 제어자

- `private` **<** `default` **<** `protected` **<** `public` 순으로 보다 많은 접근을 허용한다.

1. **private**

```
public class Sample {
    private String secret;
    private String getSecret() {
        return this.secret;
    }
}
```

- private이 붙은 변수나 메서드는 해당 클래스 안에서만 접근이 가능하다.
- secret 변수와 getSecret 메서드는 오직 Sample 클래스에서만 접근이 가능하고 다른 클래스에서는 접근이 불가능하다.

2. **default**
    - 접근 제어자를 별도로 설정하지 않는다면 변수나 메서드는 default 접근 제어자가 자동으로 설정되어 동일한 패키지 안에서만 접근이 가능하다.

```
package house;  // 패키지가 동일하다.

public class HouseKim {
    String lastname = "kim";  // lastname은 default 접근제어자로 설정된다.
}
```

```
package house;  // 패키지가 동일하다.

public class HousePark {
    String lastname = "park";

    public static void main(String[] args) {
        HouseKim kim = new HouseKim();
        System.out.println(kim.lastname);  // HouseKim 클래스의 lastname 변수를 사용할 수 있다.
    }
}
```

- HouseKim과 HousePark의 패키지는 house로 동일하다. 따라서 HousePark 클래스에서 default 접근 제어자로 설정된 HouseKim의 lastname 변수에 접근이 가능하다.

3. **protected**
    - protected가 붙은 변수나 메서드는 **동일 패키지의 클래스 또는 해당 클래스를 상속받은 클래스**에서만 접근이 가능하다.

```
package house;  // 패키지가 서로 다르다.

public class HousePark {
```

```
    protected String lastname = "park";
}
```

```
package house.person;  // 패키지가 서로 다르다.

import house.HousePark;

public class YoungPark extends HousePark {  // HousePark을 상속했다.
    public static void main(String[] args) {
        YoungPark yp = new YoungPark();
        System.out.println(yp.lastname);  // 상속한 클래스의 protected 변수는 접근이 가능하다.
    }
}
```

4. **public**
    - 접근 제어자가 public으로 설정되었다면 public 접근 제어자가 붙은 변수나 메서드는 어떤 클래스에서도 접근이 가능하다.

```
package house;

public class HousePark {
    protected String lastname = "park";
    public String info = "this is public message.";
}
```

    - HousePark의 info 변수는 public 접근 제어자가 붙어 있으므로 어떤 클래스라도 접근이 가능하다.

```
import house.HousePark;

public class Sample {
    public static void main(String[] args) {
        HousePark housePark = new HousePark();
        System.out.println(housePark.info);
    }
}
```

표로 정리하면 아래와 같다.

제어자	같은 클래스	같은 패키지	자손 클래스	전체
private	O			
default	O	O		
protected	O	O	O	

제어자	같은 클래스	같은 패키지	자손 클래스	전체
public	O	O	O	O

접근 제어자를 사용하는 이유

- 외부로부터 데이터를 보호하기 위해서
- 외부에는 불필요한, 내부적으로만 사용되는 부분을 감추기 위해서

그 외 제어자

1. **static**
   - 인스턴스 변수는 하나의 클래스로부터 생성되어도 각기 다른 값을 유지하지만, 클래스 변수(static 변수)는 인스턴스에 관계없이 같은 값을 갖는다.

```
class Test{
  static int num = 10;              // 클래스(static) 변수

    static {                        // 클래스 초기화 블럭
                                    // static 변수의 복잡한 초기화 수행
    }
    static int max(int a, inr b){   // 클래스 메서드(static 메서드)
      return a > b ? a : b;
    }
}
```

2. **final**
   - 변수에 사용되면 값을 변경할 수 없는 상수가 되며, 메서드에 사용되면 오버라이딩을 할 수 없게 되고 클래스에 사용되면 자신을 확장하는 자식 클래스를 정의하지 못하게 된다.

```
final class Test{               // 조상이 될 수 없는 클래스
  final int num = 10;           // 값을 변경할 수 없는 멤버변수

    final void getMaxNum(){     // 오버라이딩 할 수 없는 메서드
      final int n = Max_size    // 값을 변경할 수 없는 지역변수
        return Max_size;
    }
}
```

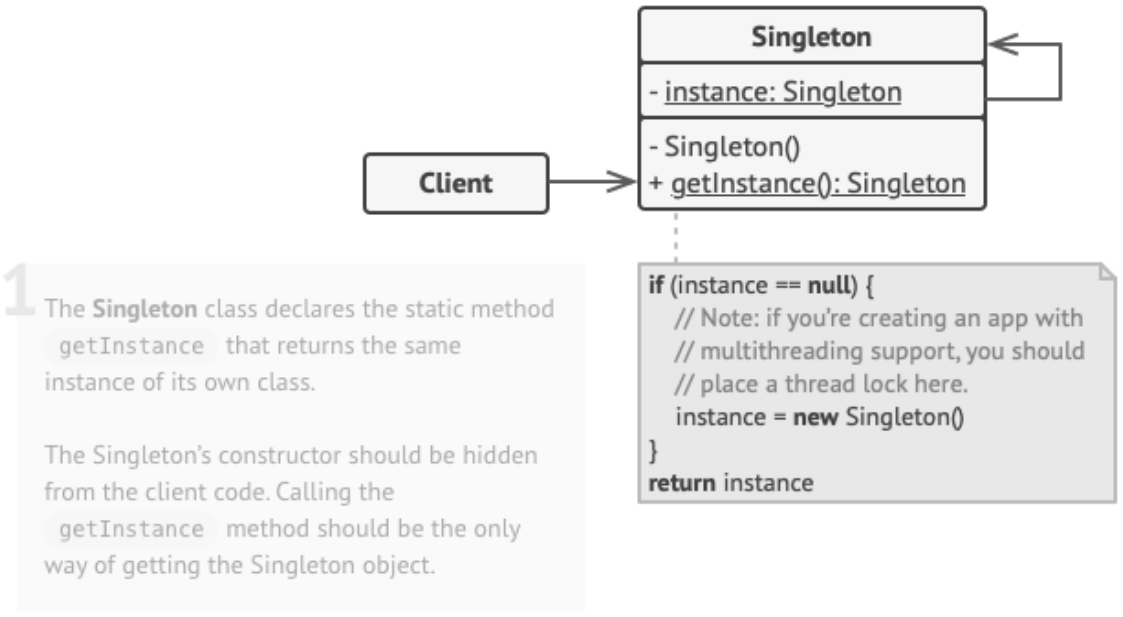
3. **abstract**
   - 메서드의 선언부만 작성하고 실제 수행내용은 구현하지 않은 추상메서드를 선언하는데 사용된다.

```
abstract class Test{        // 추상 클래스(추상 메서드를 포함한 클래스)
  abstract void study();    // 추상 메서드(구현부가 없는 메서드)
}
```

싱글톤(Singleton)

💡 소프트웨어 디자인 패턴에서 싱글턴 패턴(Singleton pattern)을 따르는 클래스는, 생성자가 여러 차례 호출되더라도 **실제로 생성되는 객체는 하나**이고 최초 생성 이후에 호출된 생성자는 **최초의 생성자가 생성한 객체를 리턴**한다. 이와 같은 디자인 유형을 싱글턴 패턴이라고 한다. 주로 공통된 객체를 여러개 생성해서 사용하는 DBCP(DataBase Connection Pool)와 같은 상황에서 많이 사용된다.

→ 객체의 인스턴스를 '한 개'만 생성되게 하는 패턴



싱글톤 패턴의 장점

- 한 개의 인스턴스만을 고정 메모리 영역에 생성해 추후 해당 객체에 접근할 때 메모리 낭비를 방지
- 이미 생성된 인스턴스를 활용하여 속도가 빠름.
- 전역으로 사용하는 인스턴스이기 때문에 다른 여러 개의 클래스에서 데이터 공유가 가능함.

Spring의 각각의 의존성 주입 방법

Spring에서 의존성을 주입하는 방법으로는 주로 아래 세 가지를 사용한다.

1. **생성자 주입(private final)**
2. @Autowired
3. 필드 주입

1. **생성자 주입(Constructor Injection)**
   - 생성자 주입은 생성자를 통해서 의존 관계를 주입받는 방법이다.
   - ServiceBean이 만들어지는 시점에서 모든 의존관계를 BeanFactory를 통해 가져온다.

```
@Controller
public class MemberController {
    private final MemberService memberService;

    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }
}
```

2. **수정자 주입(Setter Injection)**
   - Setter 메소드에 @Autowired 어노테이션을 붙이는 방법이다.

```
@Controller
public class MemberController {
    private MemberService memberService;

    @Autowired
    public void setMemberService(MemberService memberService) {
      this.memberService = memberService;
    }
}
```

   - 수정자 주입을 사용하면 Setter 메소드를 public으로 열어두어야 하기 때문에 언제 어디서든 변경이 가능하다는 문제가 있음.

3. **필드 주입(Field Injection)**
   - 필드에 @Autowired 어노테이션만 붙여주면 자동으로 의존성 주입이 된다.

```
@Controller
public class MemberController {

    @Autowired
    private MemberService memberService;
}
```

- 프레임워크에 의존적이고 객체지향적으로 좋지 않음.

→ 어떤 주입 방식을 사용하는 것이 좋은가?

Spring Framwork reference는 `생성자를 통한 주입`을 권장.

생성자 주입 방식의 장점

1. **순환 참조를 방지할 수 있다.**

   개발을 하다가 보면 여러 컴포넌트 간에 의존성이 생겨, A가 B를 참조하고 B가 A를 참조하는 상황

   → 수정자/필드 주입은 빈이 생성된 후에 참조하기 때문에 경고 없이 구동.

   실제 코드가 호출될 때까지 문제를 알 수 없음

```
@Service
public class AService {

    // 순환 참조
    @Autowired
    private BService bService;

    public void HelloA() {
        bService.HelloB();
    }
}
@Service
public class BService {

    // 순환 참조
    @Autowired
    private AService aService;

    public void HelloB() {
        aService.HelloA();
    }
}
```

   반면, 생성자를 통해 주입하고 실행하면 **BeanCurrentlyInCreationException**이 발생!

의존 관계의 내용을 외부로 노출시켜 실행 시점에서 오류를 체크할 수 있다.

2. **객체의 불변성 확보가 가능하다.**
   - 생성자로 의존성을 주입할 때 **final로 선언**할 수 있고, 이로인해 런타임에서 의존성을 주입받는 객체가 변할 일이 없어지게 된다.
   - 수정자/일반 메소드 주입을 이용하면 수정의 가능성이 열려있어 OOP의 다섯가지 원칙 중 OCP(개방-폐쇄의 원칙)를 위반하게 된다.
   - 생성자 주입을 통해 변경의 가능성을 배제하고 불변성을 보장하는 것이 좋다.

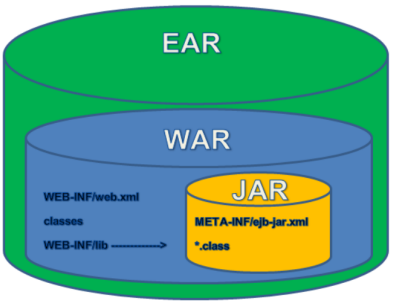
```
@Controller
public class MemberController {
    private final MemberService memberService;

    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }
}
```

3. **테스트에 용이하다.**
   - 생성자 주입을 사용하면 테스트 코드를 좀 더 편리하게 작성할 수 있다.
   - 독립적으로 인스턴스화가 가능한 POJO(Plain Old Java Object)를 사용하면, DI 컨테이너 없이도 의존성을 주입하여 사용할 수 있다.

```
public class MemberService {
    private final MemberRepository memberRepository;

    public MemberService(MemberRepository memberRepository) {
      this.memberRepository = memberRepository;
    }

    public void doSomething() {
        // do Something with memberRepository
    }
}

public class MemberServiceTest {
    MemberRepository memberRepository = new MemberRepository();
    MemberService memberService = new MemberService(memberRepository);

    @Test
    public void testDoSomething() {
        memberService.doSomething();
    }
```

JAR, WAR의 차이점

- spring 프로젝트를 서버에 배포할 때 JAR, WAR 2가지 방법으로 배포 할 수 있다.
- 기본적으로 JAR, WAR 모두 Java의 jar 옵션 (java -jar)을 이용해 생성된 압축 파일로, 애플리케이션을 쉽게 배포하고 동작시킬 수 있도록 관련 파일(리소스, 속성 파일 등)을 패키징 한 것이다.



JAR (Java Archive)

- JAVA 어플리케이션이 동작할 수 있도록 자바 프로젝트를 압축한 파일
- Class (JAVA 리소스, 속성 파일), 라이브러리 파일을 포함
- JRE(JAVA Runtime Environment)만 있어도 실행 가능

WAR (Web Application Archive)

- Servlet / Jsp 컨테이너에 배치할 수 있는 웹 애플리케이션(Web Application) 압축파일 포맷
- 웹 관련 자원(JSP, Servlet, JAR, Class, XML, HTML, Javascript)을 포함
- 사전 정의된 구조(WEB-INF, META-INF)를 사용
- 별도의 웹서버(WEB) 또는 웹 컨테이너(WAS) 필요
- 즉, JAR파일의 일종으로 웹 애플리케이션 전체를 패키징 하기 위한 JAR 파일.