이펙티브 자바

Item 3~item 6

2024/01/09

발표자: 박지원

Item 3

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

싱글톤 패턴: 인스턴스를 하나만 생성하는 것

메모리 절약, 데이터 공유 용이

item 1에 제시된 정적 팩터리 메서드로 싱글톤 구현가능

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

방법 1: private 생성자 + public static final

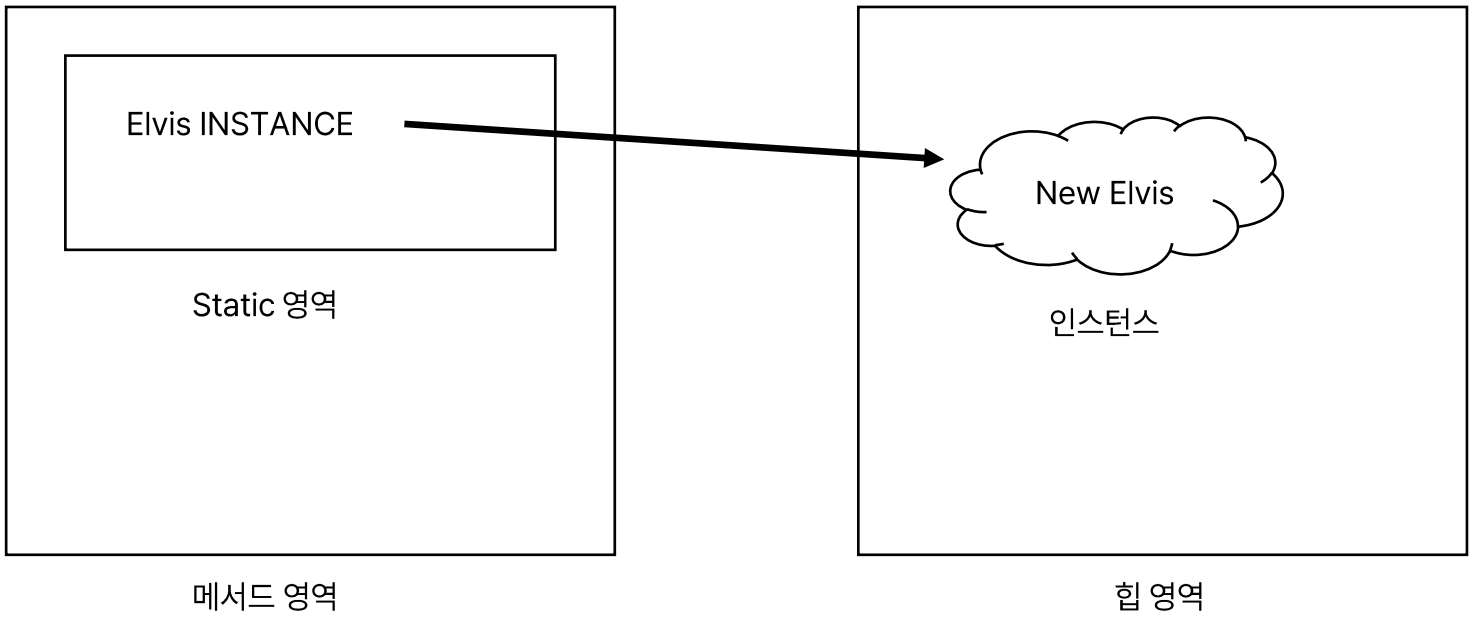
```
1  public class Elvis {
2      public static final Elvis INSTANCE=new Elvis();
3      private Elvis(){
4
5      }
6      public void printSelf(){
7          System.out.println("this = " + this);
8      }
9  }
```

Elvis.INSTANCE 로 접근

명확하고 간단한 방법

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

Static은 클래스 변수로 메서드 영역의 static 영역에서 관리됨



item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

리플렉션을 활용한 private 생성자 접근

```
1  public class ElvisTest {
2
3      @Test
4      public void testSingletonInstance() throws NoSuchMethodException, Ill
5          Elvis instance1 = Elvis.INSTANCE;
6
7          Class<?> classType = Elvis.class;
8          Constructor<?> constructor = classType.getDeclaredConstructor();
9          constructor.setAccessible(true);
10         Elvis instance2 = (Elvis) constructor.newInstance();
11
12
13         assertEquals(instance1, instance2);
14
15     }
16
17 }
```

```
org.opentest4j.AssertionFailedError:
Expected :item1.item3.Elvis@517cd4b
Actual   :item1.item3.Elvis@6cc7b4de
```

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

방법 2: private 생성자 + **private** static final + **정적 팩터리 메소드**

```java
public class Elvis2 {

    private static final Elvis2 INSTANCE=new Elvis2();
    private Elvis2(){

    }
    public static Elvis2 getInstance(){
        return INSTANCE;
    }
    public void printSelf(){
        System.out.println("this = " + this);
    }
}
```

1. 필요시 싱글톤이 아니도록 바꿀 수 있다.

   (return New ...)
2. 제네릭 싱글톤 팩터리로 만들 수 있다.
3. 공급자로 사용할 수 있다.(메서드를 공급자에 넣어 단순 반환)

Elvis2.getInstance로 접근

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

제네릭 싱글톤 팩터리

```
1  public class ElvisTest {
2      public static UnaryOperator<Objects>IDENTITY_FN=(t)->t;
3
4      @SuppressWarnings("unchecked")
5      public static <T>UnaryOperator<T> identityFunction(){
6          return(UnaryOperator<T>) IDENTITY_FN;
7      }
8
9      @Test
10     public void GenericTest()  {
11         UnaryOperator<Elvis2> instance1=identityFunction();
12         UnaryOperator<Elvis2> instance2=identityFunction();
13
14         assertEquals(instance1, instance2);
15     }
16 }
```

상수 IDENTITY_FN 들 사용하여 싱글톤 구현

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

방법 3: **열거형**

```
1  public enum Elvis3 {
2      INSTANCE;
3
4      public void printSelf(){
5          System.out.println("this = " + this);
6      }
7
8  }
```

열거형은 상속이 불가능

item 3 private 생성자나 열거 타입으로 싱글톤임을 보증하라

정리

방법 2의 장점을 사용하지 않는다면 방법 1,3 을 추천

객체 지향 개발에서 클래스간 상속이 있는 경우가 다반사,

유틸리티 클래스와 같은 기타 기능을 제외한 대부분은 방법1을 권장

Item 4

item 4 인스턴스화를 막으려거든 private 생성자를 사용하라

인스턴스 생성이 필요 없는 경우

```
1  import static java.util.Arrays.sort;
2
3  public class Util {
4
5      public static void main(String[] args) {
6          int[] arr=new int[10];
7          for(int i=0;i<10;i++){
8              arr[i]=10-i;
9          }
10         java.util.Arrays.sort(arr);
11         sort(arr);
12     }
13 }
```

```
1  public static void sort(int[] a) {
2          DualPivotQuicksort.sort(a, 0, 0, a.length);
3  }
```

item 4 인스턴스화를 막으려거든 private 생성자를 사용하라

```java
import java.util.ArrayList;
import java.util.Collections;


public class Util2 {

    public static void main(String[] args) {
        ArrayList<Integer> arr=new ArrayList<Integer>();
        for(int i=0;i<10;i++){
            arr.add(10-i);
        }
        System.out.println(Collections.max(arr));
        System.out.println(java.util.Collections.max(arr));
    }
}
```

Item 5

item 5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

```java
public class SpellChecker {
    private static final Lexicon dictionary=new Lexicon();

    private SpellChecker(){

    }

    public static SpellChecker INSTANCE=new SpellChecker();

    public static boolean isValid(String word){
        //....
        return true;
    }
    public static List<String> suggestions(String typ0){
        //....
        return null;
    }
}
```

대부분 클래스는 맴버변수로 다른 클래스를 참조하는 형태(의존적인 상황)

item 5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

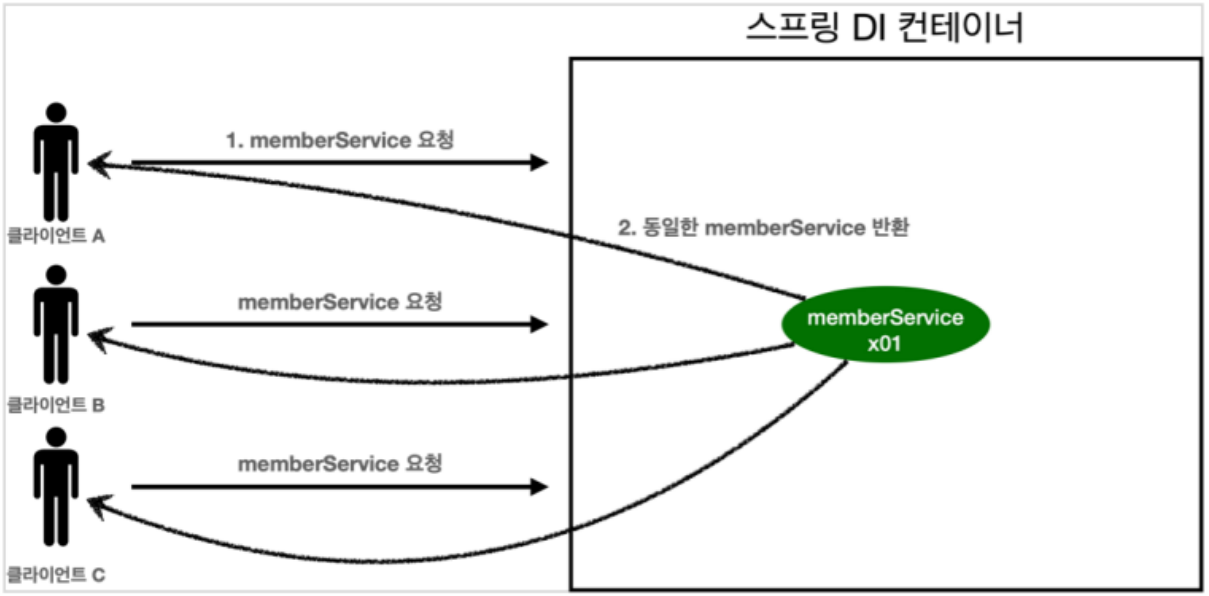
```java
public class SpellChecker {
    private final Lexicon dictionary;

    public SpellChecker(Lexicon dictionary){
        this.dictionary= Objects.requireNonNull(dictionary);
    }

    public boolean isValidWord(String word){
        return dictionary.isValidWord(word);
    }
    public List<String> suggestions(String typ0){
        //....
        return null;
    }
}
```

다형성을 활용할 수 있음

item 5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라

```java
class SpellCheckerTest {

    // Mock Lexicon 클래스를 구현하여 테스트에 사용
    private static class MockLexicon extends Lexicon {
        @Override
        public boolean isValidWord(String word) {
            // 테스트에 필요한 동작 구현
            return Arrays.asList("apple", "banana", "orange").contains(word);
        }
    }

    @Test
    void isValid() {
        // MockLexicon을 사용하여 SpellChecker 객체 생성
        SpellChecker spellChecker = new SpellChecker(new MockLexicon());

        // isValid 메서드 테스트
        assertTrue(spellChecker.isValid("apple"));
        assertTrue(spellChecker.isValid("banana"));
        assertTrue(spellChecker.isValid("orange"));

        assertFalse(spellChecker.isValid("pear"));
        assertFalse(spellChecker.isValid("grape"));
    }
}
```

item 5 자원을 직접 명시하지 말고 의존 객체 주입을 사용하라



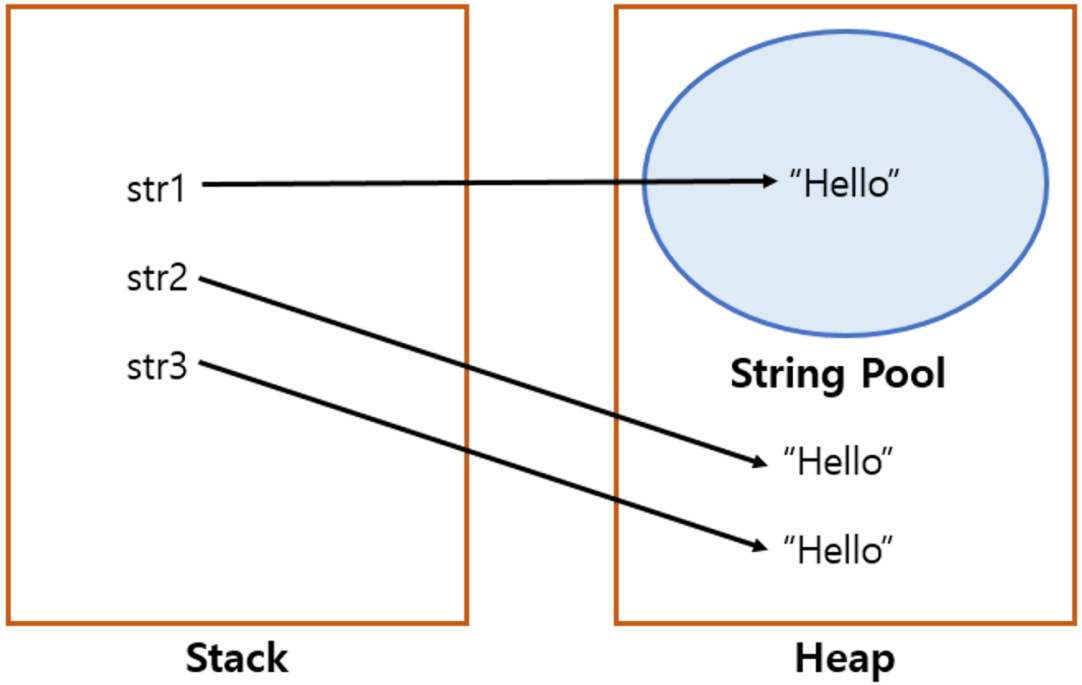
스프링 컨테이너에 빈으로 등록하면 싱글톤 및 의존성 주입을 지원

Item 6

item 6 불필요한 객체 생성을 피하라

```
1  String hello = new String("hello");
2  System.out.println(hello);
```

```
1  String hello = "hello"
2  System.out.println(hello);
```



상수풀을 사용하는 방식

item 6 불필요한 객체 생성을 피하라

```java
public boolean matches(String regex) {
    return Pattern.matches(regex, this);
}
public static boolean matches(String regex, CharSequence input) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    return m.matches();
}
public static Pattern compile(String regex) {
    return new Pattern(regex, 0);
}
```

```java
public class RomanNumerals {
    private static final Pattern ROMAN=Pattern.compile("something");

    static boolean isRomanNumerals(String str){
        return ROMAN.matcher(str).matches();
    }
}
```

Pattern 객체 재사용

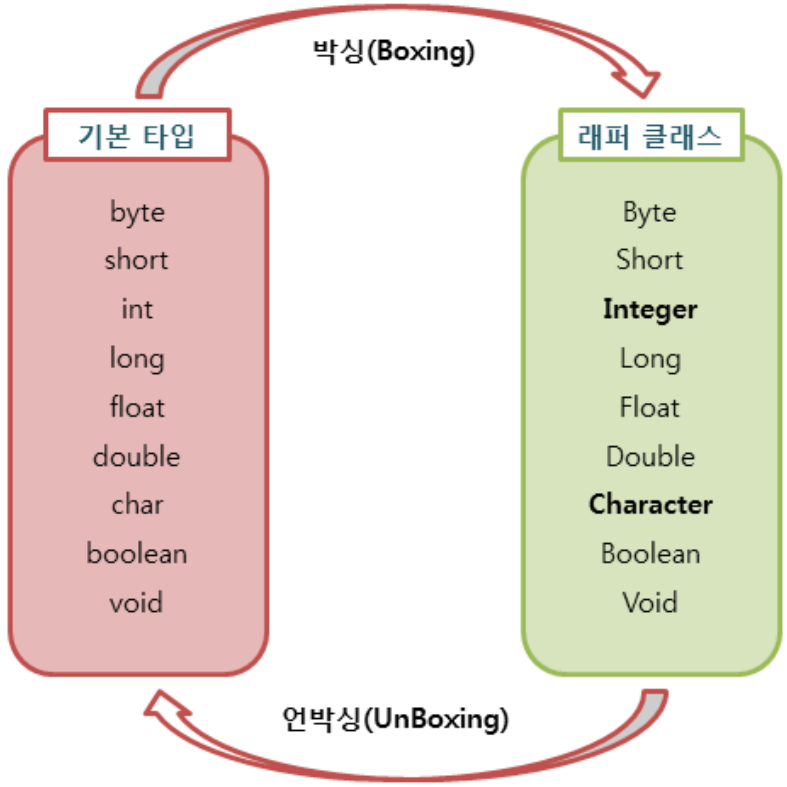
item 6 불필요한 객체 생성을 피하라

```java
public static long sum(){
        Long sum=0L;
        for(long i=0;i<=Integer.MIN_VALUE;i++){
            sum+=i;
        }
        return sum;
}
```

자동 박싱 주의할 것

래퍼클래스를 사용하는 이유가 뭘까

1. Object 반환 가능
2. Util과 같은 패키지 활용가능



item 6 불필요한 객체 생성을 피하라

래퍼클래스를 지양해야 할까?

실제로 래퍼클래스가 성능에 큰 영향을 미칠까?

대부분의 엔티티는 패러클래스로 선언하는 것으로 알고 있다(NULL을 사용 할 수 있기 때문에)