01

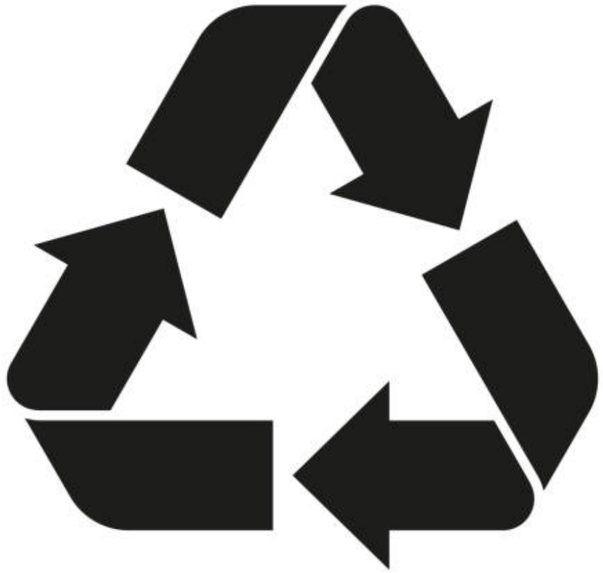
불필요한 객체 생성을 피하라

Effective Java Item 6

02

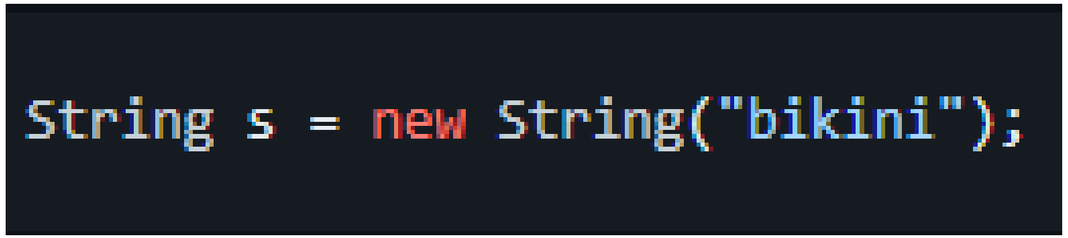
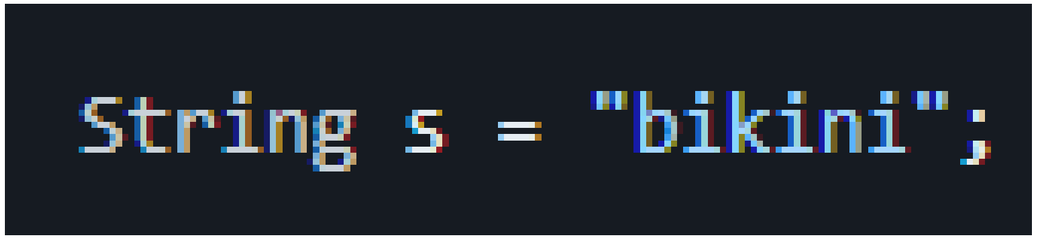
불필요한 객체 생성

- 똑같은 기능 객체 매번 생성 < 객체 하나 재사용



03

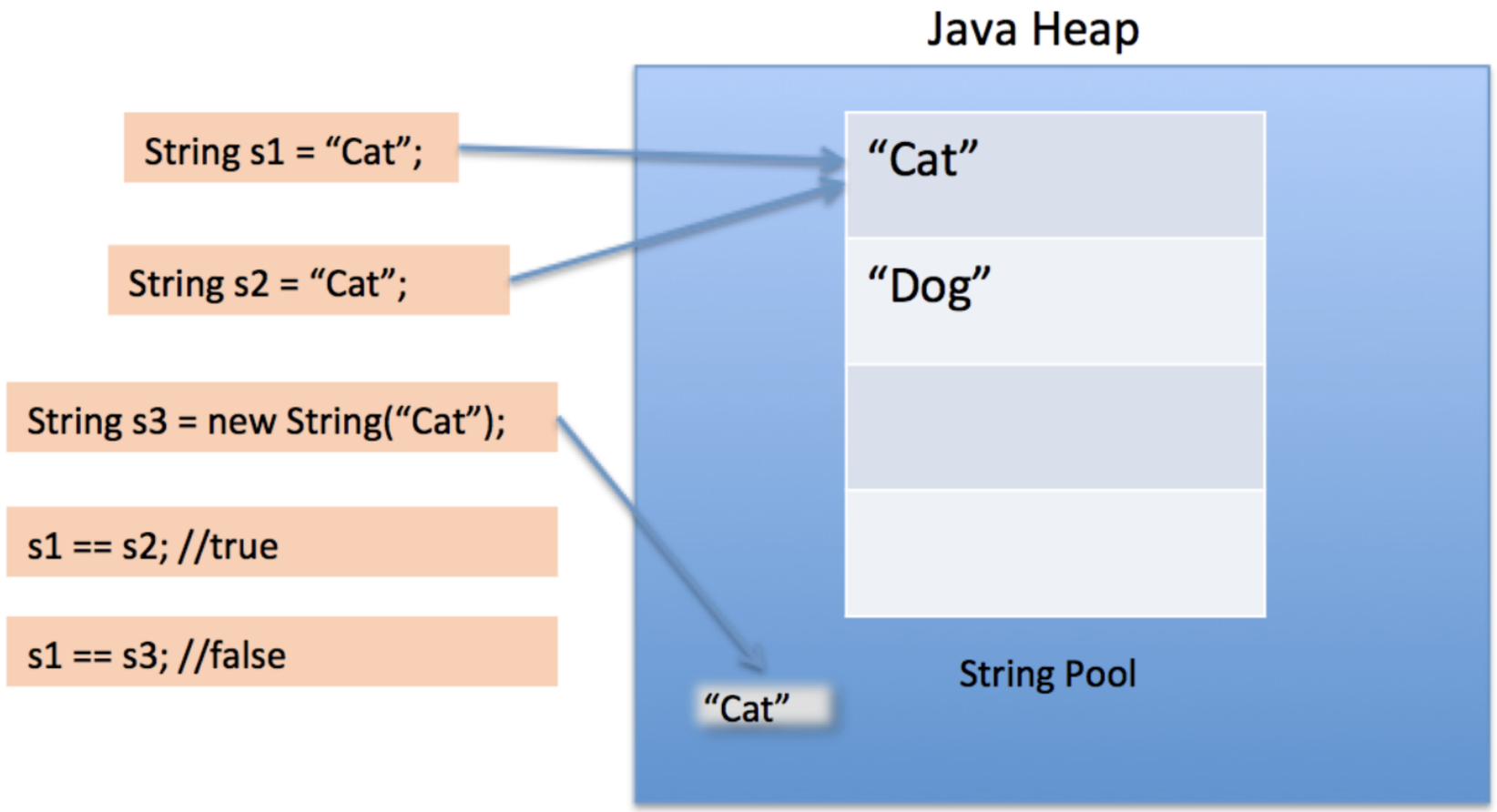
예제 1

불필요한 객체 생성	불필요한 객체 생성 피하는 법
	

String 선언 시 문자열 리터럴로 선언해 불필요한 객체 생성 피하기

04

String 동작 방식



Spring Constant Pool

문자열 리터럴은 pool에 저장되어 재사용된다.

05

예제 2

불필요한 객체 생성	불필요한 객체 생성 피하는 법
`Boolean tf = new Boolean("true")`	`Boolean tf = Boolean.valueOf("true")`

생성자 대신 정적 팩토리 메소드 사용해 불필요한 객체 생성 피하기

06

예제 3

불필요한 객체 생성

```
// 코드 6-1 성능을 훨씬 더 끌어올릴 수 있다!
static boolean isRomanNumeralSlow(String s) {
        return s.matches("^(?=.)M*(C[MD]|D?C{0,3})" + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");
}
```

⁉

07

예제 3

불필요한 객체 생성

```
public boolean matches( @NonNls @NotNull String regex) {
    return Pattern.matches(regex,  input: this);
}
```

```
public static boolean matches( @NotNull @NonNls String regex, CharSequence input) {
    Pattern p = Pattern.compile(regex);
    Matcher m = p.matcher(input);
    return m.matches();
}
```

```
public static Pattern compile( @NonNls @NotNull String regex) {
    return new Pattern(regex,  f: 0);
}
```

08

예제 3

불필요한 객체 생성 피하는 법

```
// 코드 6-2 값비싼 객체를 재사용해 성능을 개선한다.
private static final Pattern ROMAN = Pattern.compile(
        "^(?=.)M*(C[MD]|D?C{0,3})" + "(X[CL]|L?X{0,3})(I[XV]|V?I{0,3})$");

static boolean isRomanNumeralFast(String s) {
        return ROMAN.matcher(s).matches();
}
```

객체 생성해 캐싱해 놔서 불필요한 객체 생성 피하기

09

예제 4

불필요한 객체 생성

```
// 코드 6-3 꼼찍이 느리다! 객체가 만들어지는 위치를 찾았는가?
    private static long sum() {
        Long sum = 0L;
        for (long i = 0; i <= Integer.MAX_VALUE; i++)
            sum += i;
        return sum;
    }
```

예제 4

불필요한 객체 생성

```
private static long sum(){
    long sum = 0L;
    for(long i = 0; i<=Integer.MAX_VALUE; i++)
        sum += i;
    return sum;
}
```

박싱된 기본 타입 말고 기본 타입 사용해 불필요한 객체 생성 피하기

11

객체 재사용

1. String 생성자 대신 문자열 리터럴로 선언해 객체 재사용

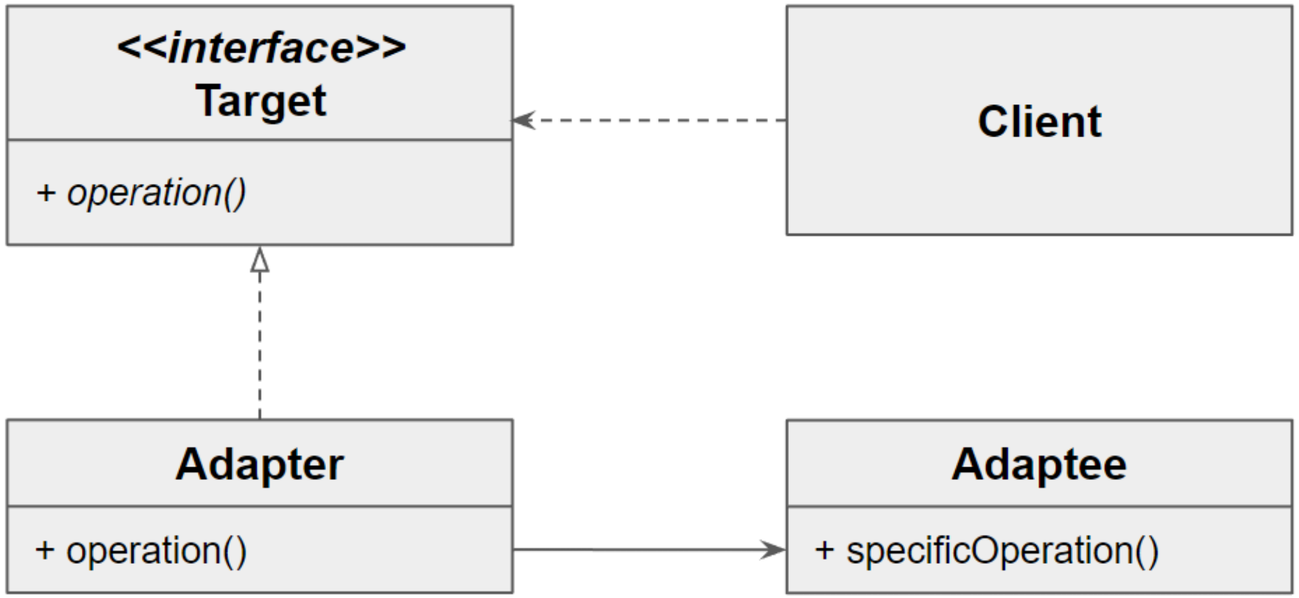
2. 생성자 대신 정적 팩토리 메소드 사용해 객체 재사용

3. 인스턴스 클래스 초기화 시 직접 생성해 캐싱해 객체 재사용

4. 박싱된 기본 타입보단 기본 타입 사용해 불필요한 객체 생성 피함

객체 재사용...?

Adapter 패턴



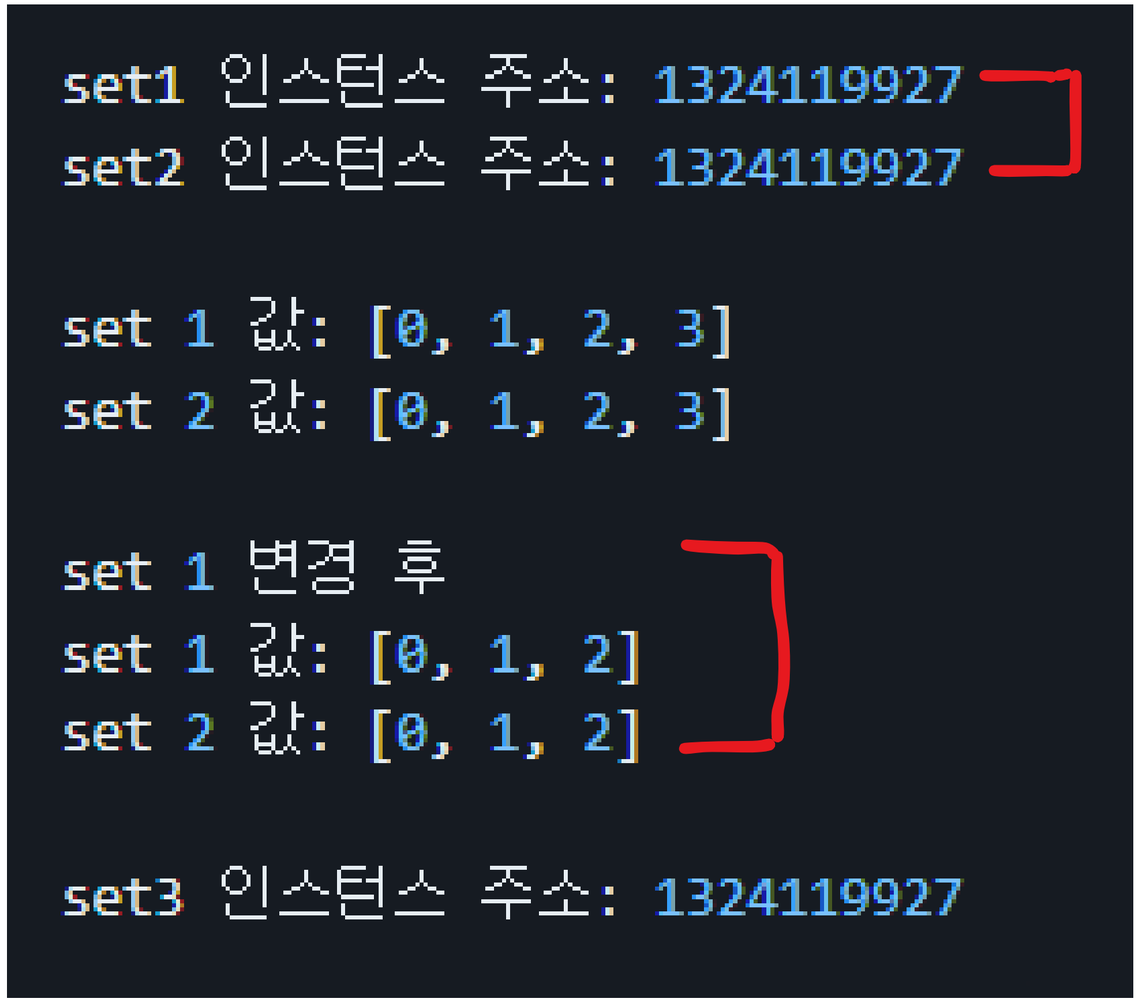
```
c static void main(String[] args) {
    Map map = new HashMap<Integer, String>();

    map.put(0, "00");
    map.put(1, "11");
    map.put(2, "22");
    map.put(3, "33");

    Set<Integer> set1 = map.keySet();
    System.out.println("set1 인스턴스 주소: "+ System.identityHashCode(set1));

    Set<Integer> set2 = map.keySet();
    System.out.println("set2 인스턴스 주소: "+ System.identityHashCode(set2));

    System.out.println();
    System.out.println("set 1 값: " + set1);
    System.out.println("set 2 값: " + set2);

    set1.remove(3);

    System.out.println();
    System.out.println("set 1 변경 후");
    System.out.println("set 1 값: " + set1);
    System.out.println("set 2 값: " + set2);

    System.out.println();
    Set<Integer> set3 = map.keySet();
    System.out.println("set3 인스턴스 주소: "+ System.identityHashCode(set3));
```


예시 - keySet

keySet 메소드는 새로운 Set 인스턴스 생성?

- 2개의 주소 확인
- 2개중 1개에만 값 제거

결과



keySet은 같은 Set 인스턴스 반환

- Set 인스턴스 주소 동일
- set1에 변화가 set2에도 적용

결론

무조건 객체 생성 피하는 것이 아닌
불필요한 객체 생성 줄이자!