


이펙티브 자바 CP.8

| | |
|------------|--|
| 🕒 작성 일시 | @2023년 3월 4일 오후 2:23 |
| 🕒 최종 편집 일시 | @2023년 3월 12일 오후 4:32 |
| 📄 유형 | 이펙티브 자바 |
| 👤 작성자 |  종현 박 |
| 👥 참석자 | |
| 🗨 언어 | |

8 일반적인 프로그래밍 원칙

- 57. 지역변수의 범위를 최소화하라
- 58. 전통적인 for 문보다는 for-each문을 사용하라
- 59. 라이브러리를 익히고 사용하라
- 60. 정확한 답이 필요하다면 float 과 double은 피하라
- 61. 박싱된 기본 타입보다는 기본 타입을 사용하라
- 62. 다른 타입이 적절하다면 문자열 사용을 피하라
- 63. 문자열 연결은 느리니 주의하라
- 64. 객체는 인터페이스를 사용해 참조하라
- 65. 리플렉션보다는 인터페이스를 사용하라
- 66. 네이티브 메서드는 신중히 사용하라
- 67. 최적화는 신중히하라
- 68. 일반적으로 통용되는 명명 규칙을 따르라

8 일반적인 프로그래밍 원칙

▼ 57. 지역변수의 범위를 최소화하라

- 개요
 - 클래스와 멤버의 접근 권한을 최소화하라 (아이템 15)와 취지가 비슷하다.
 - 지역변수의 유효 범위를 최소로 줄이면 코드 가독성과 유지보수성이 높아지고 오류 가능성은 낮아진다.
- 지역변수의 범위를 줄이는 가장 좋은 방법은 역시 가장 처음에 쓰일 때 선언하기 이다.
 - 지역변수를 생각 없이 선언하다 보면 변수가 쓰이는 범위보다 너무 앞서 선언하거나, 다 쓴 뒤에도 여전히 살아 있게 되기 쉽다.

```
// C++, 개인적으로 학부생때 이런 코드를 짜는데, 이해가 안 됐다.
int i, j;
for (i = 0; i < 10; i++) {
    ...
}
... // i 관련된 코드 없음
```

- 지역변수의 범위는 선언된 지점부터 그 지점을 포함한 블록이 끝날 때까지이므로, 실제 사용하는 블록 바로 바깥에 선언된 변수는 그 블록이 끝난 뒤까지 살아 있게 된다.

- 거의 모든 지역변수는 선언과 동시에 초기화해야 한다.

- 초기화에 필요한 정보가 충분하지 않다면 충분해질 때까지 선언을 미뤄야 한다.
- `try-catch` 문에서는 예외다.
변수를 초기화하는 표현식에서 검사 예외를 던질 가능성이 있다면 `try` 블록 안에서 초기화해야 한다.
변수 값을 `try` 블록 바깥에서도 사용해야 한다면 `try` 블록 앞에서 선언해야 한다.

- 반복문

- 반복문의 변수의 값을 반복문 종료된 뒤에도 써야 하는 상황이 아니라면 `while` 문 보다는 `for` 문을 쓰는 편이 낫다.
- ex) 컬렉션이나 배열을 순회하는 권장 관용구

```
for (Element e : c) {
    ... // e 로 무언가 한다.
}
```

- ex) 컬렉션이나 배열의 index 를 사용해야 하는 경우의 관용구

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // e와 i 로 무언가 한다.
}
```

- ex) 문제가 될 수도 있는 상황

```
Iterator<Element> i = c.iterator();
while(i.hasNext()) {
    doSomething(i.next());
}
...
Iterator<Element> i2 = c2.iterator();
while(i.hasNext()) { // 버그
    doSomethingElse(i2.next());
}
```

- 문제 `i2.hasNext()` 가 아닌 `i.hasNext()` 로 항상 비어 있다고 생각 할 수 있다.
- `for` 문을 사용하면 반복문 안에서 지역변수가 초기화, 종료 됨으로, 이런 문제가 나올 수 없다.

- ex) 문제 코드를 `for` 문 으로 변경한 코드

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ...
}
// i 를 찾을 수 없다며 컴파일 에러를 낸다.
for (Iterator<Element> i2 = c.iterator(); i2.hasNext(); ) {
    Element e2 = i2.next();
}
```

```
...
}
```

- `for` 문의 경우 복붙하는 코드에서 `c` 를 `c2` 만 바꿔줘도 된다.
 - `while` 문 보다 짧아서 가독성이 좋다.
- 메서드를 작게 유지하고 한 가지 기능에 집중하도록 만드는게 좋다.
 - 한 메서드에서 여러 가지 기능을 처리한다면 그 중 한 기능과만 관련된 지역변수라도 다른 기능을 수행하는 코드에서 접근할 수 있을 것이다.

▼ 58. 전통적인 `for` 문보다는 `for-each`문을 사용하라

- 전통적인 `for` 문

```
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
    ... // e 로 뭔가를 한다.
}
for (int i = 0; i < a.length; i++) {
    ... // a[i]로 무언가 한다.
}
```

- `while` 문 보다는 낫지만 (아이템 57) 가장 좋은 방법은 아니다.
 - 반복자와 인덱스 변수는 모두 코드를 지저분하게 할 뿐 필요한 건 원소들이다.
 - `i.next()`, `i` 는 사용하지는 않지만, 등장횟수가 있어 오류가 발생할 가능성이 높아진다.
- `for-each` 문

```
for (Element e : elements) {
    ... // e로 무언가를 한다.
}
```

- 반복자와 인덱스 변수를 사용하지 않으니 코드가 깔끔해지고 오류 날 일도 없다.
 - 컬렉션을 중첩해 순회해야 한다면 `for-each` 문의 이점은 더욱 커진다.
- 버그가 있는 `for` 문

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING }
...

static Collection<Suit> suits = Arrays.asList(Suit.values());
static Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<>();
for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
    for (Iterator<Rank> j = ranks.iterator(); j.hasNext(); )
        deck.add(new Card(i.next(), j.next()));
```

- `i.next()` 가 `Suit` 하나당 한 번씩만 불러야 하는데, 안쪽 반복문에서 호출되는 바람에 카드 하나 당 한 번씩 불러서, 숫자가 바닥 나면 `NoSuchElementException` 을 던질 것이다.

- 또 다른 버그

```
enum Face { ONE, TWO, THREE, FOUR, FIVE, SIX }
...
Collection<Face> faces = EnumSet.allOf(Face.class);

for (Iterator<Face> i = faces.iterator(); i.hasNext(); )
    for (Iterator<Face> j = faces.iterator(); j.hasNext(); )
        System.out.println(i.next() + " " + j.next());
```

- 예외를 던지지는 않지만, 의도대로 동작하지 않는다.
- 1, 1 ~ 6, 6 총 6개만 출력하고 끝낸다. 의도대로면 36개가 나와야함.

- 해결 코드

```
for (Suit suit : suits)
    for (Rank rank : ranks)
        deck.add(new Card(suit, rank));
```

- **for-each** 문을 사용할 수 없는 상황

1. 파괴적인 필터링

- 컬렉션을 순회하면서 선택된 원소를 제거해야 한다면 반복자의 `remove` 메서드를 호출해야 한다.
- `for-each` 원소에서 `remove(Object e)` 를 사용하면 `ConcurrentModificationException` 에러가 발생한다.
- 자바 8부터는 `Collection` 에는 `removeIf` 라는 메서드를 제공하면서, 명시적으로 순회하는 일을 피할 수 있다.

2. 변형

- 리스트나 배열을 순회하면서 그 원소의 값 일부 혹은 전체를 교체해야 한다면 리스트의 반복자나 배열의 인덱스를 사용해야 한다.

3. 병렬 반복

- 여러 컬렉션을 병렬로 순회해야 한다면 각각의 반복자와 인덱스 변수를 사용해 엄격하고 명시적으로 제어해야 한다.
- 위 3가지 경우에는 위에 나왔던 `for` 문을 사용하되, `for` 문 사용시 문제들을 주의하자.

- `for-each` 를 사용하려면 `Iterable` 인터페이스를 구현하자.

```
public interface Iterable<E> {
    // 이 객체의 원소들을 순회하는 반복자를 반환한다.
    Iterator<E> iterator();
}
```

- 원소들의 묶음을 표현하는 타입을 작성해야 한다면 `Iterable` 을 구현하도록 하자.

- 정리

- 전통적인 `for` 문과 비교해서 `for-each` 문이 더 명료하고, 유연하고, 버그를 예방해준다.

- 성능저하도 없다.
- `for-each` 문을 사용하지 못하는 경우 3가지를 제외하고는 사용하도록 하자.

▼ 59. 라이브러리를 익히고 사용하라

- 흔히 마주치는 문제
 - ex) 무작위 정수를 생성하는 코드

```
static Random random = new Random();

static int random(int n) {
    return Math.abs(random.nextInt()) % n;
}
```

- `n` 이 크지 않은 2의 제곱수라면 얼마 지나지 않아 같은 수열이 반복된다.
- `n` 이 2의 제곱수가 아니라면 몇몇 숫자가 평균적으로 더 자주 반환된다.
 - `n` 값이 크면 이 현상은 더 두드러진다.

```
int n = 2 * (Integer.MAX_VALUE / 3);
int low = 0;
for (int i = 0; i < 1000000; i++)
    if (random(n) < n / 2)
        low++;
System.out.println(low);
```

- 메서드를 1,000,000 번 돌려서 `n/2 <` 이 `true` 인 경우가 약 50만 개가 나와야 하지만, 666,666 에 가까운 숫자를 얻는다.
- 지정한 범위 '바깥'의 수가 종종 튀어나 올 수 있다.
 - `random.nextInt()` 가 반환 값을 `Math.abs` 를 이용해 음수가 아닌 정수로 매핑하기 때문이다.
 - `nextInt()` 가 `Integer.MIN_VALUE` 를 반환하면 `Math.abs` 도 `Integer.MIN_VALUE` 를 반환하고, 나머지 연산자는 음수를 반환한다.
- 이 문제들은 의사난수 생성기, 정수론, 2의 보수 계산 등에 깊이가 있어야 한다.
 - 다행히 이 문제들은 `Random.nextInt(Int)` 가 이미 해결해냈다.
 - 이 메서드의 자세한 동작 방식은 몰라도 된다.
- **표준 라이브러리 장점**
 - 표준 라이브러리를 사용하면 그 코드를 작성한 전문가의 지식과 여러분보다 앞서 사용한 다른 프로그래머들의 경험을 활용할 수 있다.
 - 핵심적인 일과 크게 관련 없는 문제를 해결하느라 시간을 허비하지 않아도 된다.
 - 따로 노력하지 않아도 성능이 지속해서 개선됨. (버전업 하면서 표준 라이브러리가 업뎃)
 - 기능이 점점 많아진다.

- 작성한 코드가 많은 사람에게 낯익은 코드가 된다. 자연스럽게 더 읽기 좋고, 유지보수하기 좋고, 재활용 쉬운 코드가 된다.
- **Java 라이브러리가 방대하여 모든 API 문서를 보는건 어렵겠지만, 적어도 `java.lang`, `java.util`, `java.io` 와 그 하위 패키지들은 익숙해져야 한다.**
 - 추가적으로 컬렉션 프레임워크, 스트림 라이브러리(아이템 45 ~ 48), `java.util.concurrent` 동시성 기능 (아이템 80, 81) (내가 `synchronized` 동시성 문제를 해결하지 않아도 됨)
- 정리
 - 구현하기 전 먼저 구글링(찾아보자)을 해보자. 대부분 표준 라이브러리에서 구현되어 있을 가능성이 크다.
 - 누구나 쓰는 라이브러리를 사용하는 것이 더 좋은 코드를 작성할 수 있다. (**github star**)

▼ 60. 정확한 답이 필요하다면 float 과 double은 피하라

- float, double 타입
 - 부동소수점 연산에 쓰이며, 넓은 범위의 수를 빠르게 정밀한 ‘근사치’로 계산하도록 세심하게 설계되었다.
 - 따라서 정확한 결과가 필요할 때는 사용해서 안 된다.
 - 특히, 금융 관련 계산에는 맞지 않는다.
 - 학부생때 배웠던 내용으로는, 0.1 을 표현하려면, 이진수로 표현하기 위해서는, 2^0 , 2^{-1} , 2^{-2} , 2^{-3} , 2^{-4} , 2^{-5} ... 에서 2^{-4} , 2^{-5} 를 사용하고 $0.06125 + 0.030625 + a$ 로 해서 정확한 소수점의 수를 구할 수 없다.
 - 예를 들어 $1.03 - 0.57$ 을 하면 0.46 이 아닌 0.46000000000000001 을 출력한다.
 - 반올림을 한다해도 문제가 발생할 수 있다.
- ex) 금융 계산 부동소수점 타입 사용

```
public static void man(String [] args){
    double funds = 1.00;
    int itemsBought = 0;
    for (double price = 0.10; funds >= price; price += 0.10) {
        funds -= price;
        itemsBought++;
    }
    System.out.print(itemsBought);
    System.out.print(funds);
}
```

- 결과로 4, 0 출력할 것으로 예상하지만, 결과는 3, 0.3999999999999999 를 출력한다.
- 이를 위해 소수점 계산에는 `BigDecimal`, `int` 혹은 `long` 을 사용해야한다.
- ex) BigDecimal 사용해서 문제 해결

```
public static void man(String [] args){
    final BigDecimal TEN_CENTS = new BigDecimal("0.10");

    int itemsBought = 0;
    BigDecimal funds = new BigDecimal("1.00");
```

```

    for (BigDecimal price = TEN_CENTS; funds.compareTo(price) >= 0; price = price.add(TEN_CENTS)) {
        funds.subtract(price);
        itemsBought++;
    }
    System.out.print(itemsBought);
    System.out.print(funds);
}

```

- 기본 타입보다 쓰기가 훨씬 불편하고, 훨씬 느리다.
- ex) 단위를 최소 단위에 맞춰서 기본 타입으로 해결

```

public static void man(String [] args){
    int funds = 100;
    int itemsBought = 0;
    for (int price = 10; funds >= price; price += 10) {
        funds -= price;
        itemsBought++;
    }
    System.out.print(itemsBought);
    System.out.print(funds);
}

```

- 정리
 - 정확한 답이 필요한 경우 `float, double` 를 피해라
 - 소수점 추적은 시스템에 맡기고, 불편함과 성능 저하에 신경 쓰지 않겠다면 `BigDecimal` 을 사용하라
 - `BigDecimal` 은 여덟 가지 반올림 모드를 이용해 반올림을 완벽히 제어할 수 있다.
 - 반면, 성능이 중요하고 소수점을 직접 추적할 수 있고, 숫자가 크지 않다면, `int` 나 `long` 을 써라
 - 숫자 표현 범위가 10^9 인 경우 `int` , 10^{18} 인 경우 `long` , 그 이상인 경우 `BigDecimal`

▼ 61. 박싱된 기본 타입보다는 기본 타입을 사용하라

- 기본 타입에 대응하는 참조 타입
 - `int` → `Integer`
 - `double` → `Double`
 - `boolean` → `Boolean`
- 오토 박싱, 오토 언박싱
 - 오토 박싱 - 기본 타입이 참조 타입으로 변경되는 것
 - 오토 언박싱 - 참조 타입이 기본 타입으로 변경되는 것
- 기본 타입과, 박싱된 기본 타입의 주된 차이
 1. 기본 타입은 값만 갖고 있으나, 박싱된 기본 타입은 값에 더해 식별성이란 속성도 갖는다.
 2. 기본 타입의 값은 언제나 유효하나, 박싱된 기본 타입은 유효하지 않는 값 `null` 을 갖을 수 있다.
 3. 기본 타입이 박싱된 기본 타입보다 시간과 메모리 면에서 더 효율적이다
- ex) `Integer` 값을 오름차순으로 정렬하는 비교자 - 잘못 구현됨

```
Comparator<Integer> naturalOrder =
    (i, j) -> (i < j) ? -1 : (i == j ? 0 : 1);
```

- `i == j` 는 참조형 식별성을 검사하게 된다.
- 즉, 이 결과는 같은 값 `Integer` 이 올 경우 `false` 가 됨으로, 같은 값에 1을 반환한다.
- 같은 객체를 비교하는게 아니라면, 박싱된 기본 타입에 `==` 연산자를 사용하면 에러 난다
- 기본 타입을 다루는 비교자가 필요하면 `Comparator.naturalOrder` 을 사용하자
 - 박싱된 기본 타입에서 `compareTo` 메서드를 잘 구현해놨다.
 - 비교자를 직접 만들려면 비교자 생성 메서드나 기본 타입을 받는 정적 `compare` 메서드를 사용해야 한다. (아이템 14)
 - 위 문제를 고치려면 기본타입의 지역 변수를 2개 두어 언박싱 후 비교 로직을 수행한다.

```
Comparator<Integer> naturalOrder = (iBoxed, jBoxed) -> {
    int i = iBoxed, j = jBoxed;
    (i < j) ? -1 : (i == j ? 0 : 1);
};
```

- ex) 기본 타입과 박싱된 기본 타입의 연산

```
Integer i;
if (i == 42)
    System.out.println("출력 될까?");
```

- 해당 소스는 `NullPointerException` 을 던진다.
- `Integer` 기본 값은 `null` 값이고, 기본 타입과 박싱된 기본 타입의 혼용한 연산에서는 박싱된 기본 타입의 박싱이 자동으로 풀린다.
- ex) 박싱과 언박싱의 반복으로 성능이 느려지는 코드

```
public static void main(String[] args) {
    Long sum = 0L;
    for (long i = 0; i <= Integer.MAX_VALUE; i++) {
        sum += i;
    }
    System.out.println(sum);
}
```

- 오류나 경고 없이 컴파일 되지만, `sum += i` 시 박싱, 언 박싱이 반복해서 일어나 성능이 굉장히 느려진다.
- 박싱된 기본 타입을 써야 할 때
 1. 컬렉션의 원소, 키, 값으로 쓰인다.
컬렉션은 기본 타입을 담을 수 없음으로 어쩔 수 없이 박싱된 타입을 써야만 한다.
매개변수화 타입이나 매개변수화 메서드의 타입 매개변수로든 박싱된 기본타입을 써야하기 때문이다.

2. `ThreadLocal<Integer>`

3. 리플렉션 (아이템 65) 를 통해 메서드 호출시 박싱된 기본 타입 사용

• 정리

- 기본 타입과 박싱된 기본 타입 중 하나를 선택해야 한다면, 가능하다면 기본 타입
 - 간단하고 빠름, 박싱된 타입을 써야 한다면 주의하자.
- 오토 박싱이 박싱된 기본 타입을 사용할 때의 번거러움을 줄여주지만, 그 위험은 유효하다.
- 박싱된 기본 타입을 `==` 연산자로 비교하면 식별성 비교가 이뤄지는데, 원하는 결과가 아닌 가능성이 높다.
 - `a = new Integer(42), b = new Integer(42)`
 - `a == b` 는 `false` 를 반환
- 기본 타입과 박싱된 기본 타입을 혼용하면, 언 박싱 과정에서 `nullPointerException` 을 던질 수 있다.
- 기본 타입을 박싱하는 작업은 필요 없는 객체를 생성하는 부작용이 나올 수 있다.

▼ 62. 다른 타입이 적절하다면 문자열 사용을 피하라

- 문자열은 다른 값 타입을 대신하기에 적합하지 않다.
 - 파일, 네트워크, 키보드 입력으로 데이터를 받을 때 문자열을 받지만, 입력받은 데이터가 진짜 문자열일 때만 그렇게 하는게 좋다.
 - 수치형이라면, `int, float, BigInteger` 등 적당한 수치 타입으로 변환해야 한다.
 - 기본 타입이든 참조 타입이든, 적절한 값 타입이 있다면 그것을 사용하고, 없다면 새로 하나 작성하라.
- 문자열은 열거 타입을 대신하기에 적합하지 않다.
 - 상수 열거할 때는 문자열보다는 열거 타입이 훨씬 낫다. (아이템 34)
- 문자열은 혼합 타입을 대신하기에 적합하지 않다.
 - 여러 요소가 혼합된 데이터를 하나의 문자열로 표현하는 것은 대체로 좋지 않은 생각임.
 - ex) 혼합 타입을 문자열로 처리한 예

```
String compoundKey = className = "#" + i.next();
```

- 혹여라도 구분자 #이 두 요소 중에 하나에서 쓰였다면, 문제가 된다.
 - 각 요소를 개별로 접근하려면 문자열을 파싱해야 해서 느리고, 귀찮고, 오류 가능성도 커진다.
 - 적절한 `equals, toString, compareTo` 메서드를 제공할 수 없으며, `String` 이 제공하는 기능에만 의존해야한다.
 - 차라리 `private` 정적 멤버 클래스로 전용 클래스를 만드는게 낫다. (아이템 24)
- 문자열은 권한을 표현하기에 적합하지 않다.
 - 권한을 문자열로 표현하는 경우가 종종 있다.
 - ex) 문자열을 사용해 권한을 구분함 - 문자열 키로 스레드별 지역변수 식별

```
public class ThreadLocal {
    private ThreadLocal() { } // 객체 생성 불가

    private static void set(String key, Object value);

    public static Object get(String key);
}
```

- 이 방식의 문제는 스레드 구분용 문자열 키가 전역 이름공간에서 공유된다는 점이다.
- 의도대로 동작은, 각 클라이언트가 고유한 키를 제공해야 한다.
그런데 만약 두 클라이언트가 서로 소통하지 못해 같은 키를 쓰기로 결정한다면, 의도지 않게 같은 변수를 공유하게 된다.

▼ String 도 객체 생성이 아니냐? 왜 문제가 되느냐?

```
String s = "ss";
String s1 = "ss";
String s2 = new String("ss");

System.out.println(s == s1); // true
System.out.println(s == s2); // false
System.out.println(s1 == s2); // false
```

- 자세한 내용은 String Constant Pool 를 찾아보자.

- ex) 해결책 문자열 대신 위조할 수 없는 키를 사용하면 해결된다.

```
public class ThreadLocal {
    private ThreadLocal() { }

    public static class Key {
        key() { }
    }

    public static Key getKey() {
        return new Key();
    }

    public static void set(Key key, Object value);
    public static Object get(Key key);
}
```

- 고유한 키를 제공할 수 있게 된다.
- `set` 과 `get` 이제 정적 메서드일 이유가 없으니 `Key` 클래스의 인스턴스 메서드로 바꾼다.
이렇게 되면 `Key` 는 더 이상 스레드 지역변수를 구분하기 위한 키가 아닌, 그 자체가 스레드 지역 변수가 된다. `ThreadLocal` 은 별달리 하는 일이 없어짐으로 치우고 `Key` 이름을 `ThreadLocal` 로 바꿔 버리자.

```
public final class ThreadLocal {
    public ThreadLocal();
    public void set(Object value);
    public Object get();
}
```

- 여기서 `Object` 를 실제 타입으로 형변환해야 됨으로 타입 안전하지 않다.
- `ThreadLocal` 을 매개변수화 타입(아이템 29)로 선언하자

```
public final class ThreadLocal<T> {
    public ThreadLocal();
    public void set(T value);
    public T get();
}
```

- 정리
 - 더 적합한 데이터 타입이 있거나 새로 작성할 수 있다면, 문자열을 쓰지말자
 - 문자열은 잘못 사용하면 번거롭고, 덜 유연하고, 느리고, 오류 가능성이 있다.
 - 문자열을 잘못 사용하는 흔한 예로는 기본 타입, 열거 타입, 혼합 타입이 있다.
- 최근 했던 문제
 - OTP 기능을 개발 중
 - `Map` 구현체를 사용하면서, `String` Key 값으로 `user_Id::OTP_number` 로 설정 했었다. value 값으로는 `System.currentTimeMillis()` 값이다.
 - 재발급 로직에서 `map.keySet().iterator()` 로 `value.contains(user_Id + "::")` 로 있는지 판단 하여 해당 데이터를 하나 날렸는데, 이 방식이 아닌, 내부 정적 클래스를 만들어서 처리하면 더 좋을 것 같다. (물론, `user_Id`, `OTP_number` 에 `::` 가 들어가진 않지만, 문자열 처리 방식 로직이 늘어져서 가독성이 떨어지고, `String` 객체를 생성함으로 느려질 것 같다.

▼ 63. 문자열 연결은 느리니 주의하라

- 문자열 연결 연산자 +
 - 여러 문자열을 하나로 합쳐주는 수단이다.
 - 하지만, 엄청나게 성능 저하를 일으킨다.
 - 문자열 n 개를 잇는 시간은 n^2 이 걸린다.
 - 문자열은 불변이라서, 두 문자열을 연결할 경우 양쪽의 내용을 모두 복사해야 함으로
- ex) `String` 연결 연산자 사용

```
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++)
        result += lineForItem(i);
    return result;
}
```

- ex) `StringBuilder` 를 사용함

```
public String statement2() {
    StringBuilder b = new StringBuilder(numItems() * LINE_WIDTH);
    for (int i = 0; i < numItems(); i++)
```

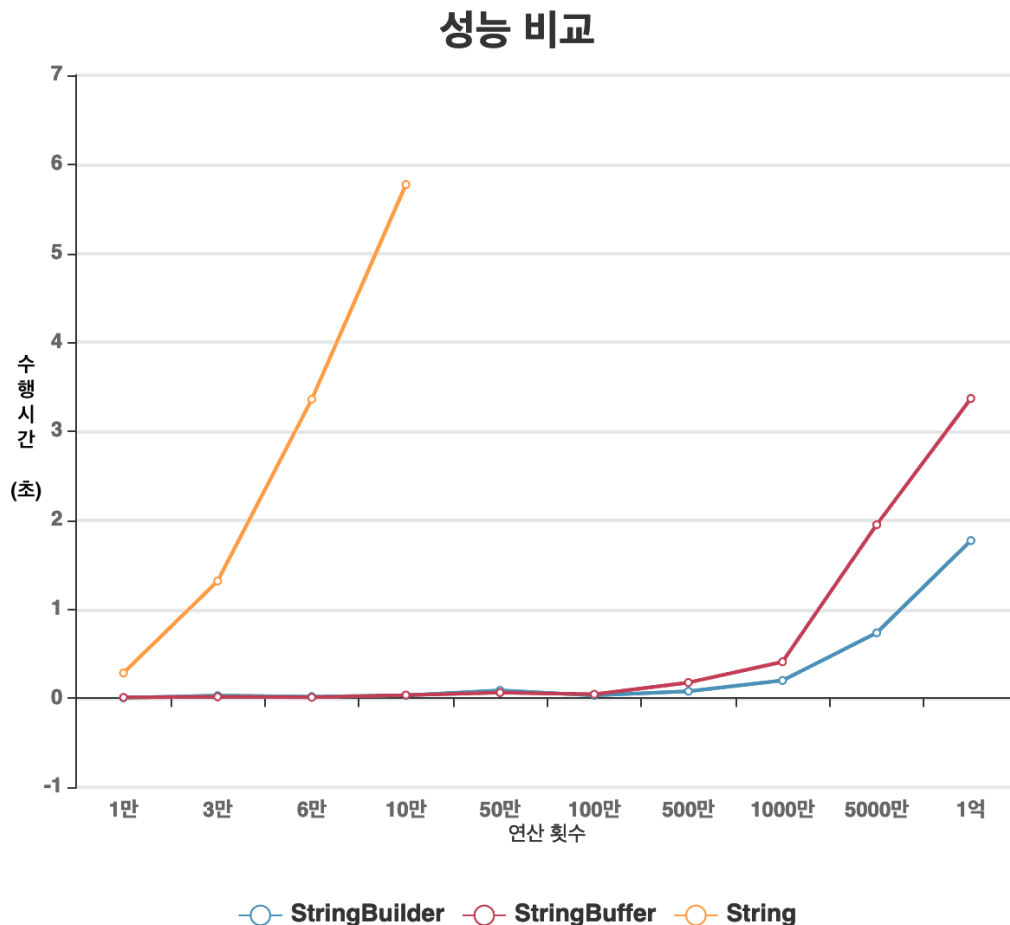
```

        b.append(lineForItem(i));
        return b.toString();
    }

```

- 성능이 훨씬 좋다
- 멀티 쓰레드 환경에서는 `StringBuffer` 를 쓰자
 - 사용 방법은 `StringBuilder` 와 같다
- 정리
 - 문자열 연결이 많은 경우는 `StringBuilder` 를 쓰고
 - 멀티 쓰레드 환경이면 `StringBuffer` 를 쓰자.

▼ 성능비교 (구글링)



▼ 64. 객체는 인터페이스를 사용해 참조하라

- 적합한 인터페이스가 있다면 클래스 말고 인터페이스를 사용해라
 - 매개변수 뿐만 아니라, 반환 값, 필드, 변수를 전부 인터페이스 타입으로 써라
 - 실제 클래스를 사용하는 상황은 생성자로 생성 할 때만이다.
 - 인터페이스를 타입으로 사용하는 습관을 두면 프로그램이 유연해질 것이다.

- 나중에 구현 클래스를 교체하고자 한다면, 새(인터페이스를 확장(구현)한 클래스의 생성자, 다른 정적 팩터리)를 사용하면 된다.
- 주의점
 - 원래 클래스가 인터페이스의 일반 규약 이외의 특별한 기능을 제공하며, 주변 코드가 이 기능에 기대어 동작한다면 새로운 클래스도 반드시 같은 기능을 제공해야 한다.
 - 예를 들어, `LinkedHashSet` 은 순서를 보장한다. (적재 순서) 그런데, `HashSet` 으로 변경하려면, 문제가 발생할 수 있다. (순서 보장 X)
 - **적합한 인터페이스가 없다면 당연히 클래스를 참조해야 한다.**
 1. `String, Integer` 같은 값 클래스
 - 값 클래스를 여러 가지로 구현 될 수 있다고 생각하고 설계하는 일은 없다.
 - 따라서 `final` 인 경우가 많고 상응하는 인터페이스가 별도로 존재하는 경우가 드물다.
 - 이런 값 클래스의 경우 매개변수, 변수, 필드, 반환 타입으로 사용해도 된다.
 2. 클래스 기반으로 작성된 프레임 워크가 제공하는 객체
 - 이런 경우라도 특정 구현 클래스 보다는 기반 클래스(추상 클래스) 를 사용해 참조하는게 좋다.
 - `java.io` 패키지의 `OutputStream` 등등
 3. 인터페이스에 없는 특별한 메서드를 제공하는 클래스
 - ex) `PriorityQueue` 클래스(우선 순위 큐)는 `Queue` 인터페이스에 없는 `comparator` 메서드를 제공한다.
 - 클래스 타입을 직접 사용하는 경우는 이런 추가 메서드를 꼭 사용해야 하는 경우로 최소화 해야 하며, 절대 남발하지 말아야 한다.
 - **적합한 인터페이스가 없다면 클래스의 계층구조 중 필요한 기능을 만족하는 가장 덜 구체적인 (상위 클래스 타입을 사용하자.)**
- 구현 타입을 변경하려는 의도
 - 원래 것보다 성능이 좋거나, 신기능을 제공할 수 있기 때문이다.
 - 선언 타입과 구현 타입을 동시에 바꾸면 되지 않느냐?
 - 선언 타입이 변경되면서, 이전에 사용한 선언 타입의 메서드가 변경된 선언 타입의 메서드가 지원하지 않으면 컴파일 되지 않을 것이다.
- 최근 했던 문제
 - 최근 OTP 기능을 구현하면서, `Map<String, Long> otpMap = new HashMap<>()` 으로 설정 했다.
 - 이걸 구현하면서, 웬만하면 멀티 스레드 환경에서도 잘 동작할 것으로 보이는데, 혹시 모르니, 주석으로 아래 처럼 남겨놨다.

```
/*
 * 동시성 문제가 나지 않을 것으로 판단된다.
 * 만약, 동시성 문제가 발생되면 OtpMap 생성을 hashMap 대신 ConcurrentHashMap 를 사용하자
 */
private final Map<String, Long> otpMap = new HashMap<>();
```

- 이 처럼 선언 타입은 인터페이스로 하는 것이 유연하다.

▼ 65. 리플렉션보다는 인터페이스를 사용하라

• 리플렉션 설명

- 리플렉션을 이용하면 프로그램에서 임의의 클래스에 접근할 수 있다.
- `Class` 객체가 주어지면 그 클래스의 생성자, 메서드, 필드에 해당하는 인스턴스를 가져올 수 있고, 이어서 인스턴스들로는 그 클래스의 멤버 이름, 필드 타입, 메서드 시그니처 등을 가져올 수 있다.
- 인스턴스를 이용해 각각에 연결된 실제 생성자, 메서드, 필드를 조작할 수도 있다.
 - 인스턴스들을 통해 해당 클래스의 인스턴스를 생성하거나, 메서드를 호출하거나, 필드에 접근할 수 있다는 것이다.
- 리플렉션을 사용하면 컴파일 당시에 존재하지 않던 클래스도 이용할 수 있다.
 - Spring DI (의존관계 주입)은 어떻게 `@Component` 어노테이션으로 싱글톤을 유지하며, 필요시 인스턴스를 줄 수 있는 걸까? (리플렉션 사용)

• 리플렉션 단점

- 컴파일타임 타입 검사가 주는 이점을 하나도 누릴 수 없다.
 - 예외 검사도 마찬가지다. 프로그램이 리플렉션 기능을 써서 존재하지 않는 혹은 접근할 수 없는 메서드를 호출하려 시도하면, 런타임 오류가 발생한다.
- 리플렉션을 이용하면 코드가 지저분하고 장황해진다.
- 성능이 떨어진다.
 - 리플렉션을 통한 메서드 호출은 일반 메서드 호출보다 훨씬 느리다.

• 리플렉션은 아주 제한된 형태로만 사용해야 그 단점을 피하고 이점만 취할 수 있다.

- 리플렉션은 인스턴스 생성에만 쓰고, 이렇게 만든 인스턴스는 인터페이스나 상위 클래스로 참조해 사용해야한다.

• ex) 리플렉션으로 생성하고 인터페이스로 참조해 활용한다.

```
public static void main(String [] args) {
    // 클래스 이름을 Class 객체로 변환
    Class<? extends Set<String>> cl = null;
    try {
        cl = (Class<? extends Set<String>>) // 비검사 형변환
            Class.forName(args[0]);
    } catch (ClassNotFoundException e) {
        logger.error("클래스를 찾을 수 없습니다.");
    }

    // 생성자를 얻는다.
    Constructor<? extends Set<String>> cons = null;
    try {
        cons = cl.getDeclaredConstructor();
    } catch (NoSuchMethodException e) {
        logger.error("매개변수 없는 생성자를 찾을 수 없습니다.");
    }

    // 집합의 인스턴스를 만든다.
    Set<String> s = null;
    try {
        s = cons.newInstance();
    }
```

```

    } catch (IllegalAccessException e) {
        logger.error("생성자에 접근 할 수 없습니다.");
    } catch (InstantiationException e) {
        logger.error("클래스를 인스턴스화 할 수 없습니다.");
    } catch (InvocationTargetException e) {
        logger.error("생성자가 예외를 던졌습니다." + e.getCause());
    } catch (ClassCastException e) {
        logger.error("Set을 구현하지 않은 클래스입니다.");
    }
}

// 생성한 집합을 사용한다.
s.add(Array.asList(args).subList(1, args.length));
logger.info(s);
}

```

이 프로그램은 손쉽게 제네릭 집합 테스터로 변환할 수 있다. 즉, 명시한 `Set` 구현체를 공격적으로 조작해보며, `Set` 규약을 잘 지키는지 검사해볼 수 있다. 비슷하게, 제네릭 집합 성능 분석 도구로 활용할 수도 있다.

대부분의 리플렉션 기능은 위 처럼 인스턴스를 생성하고 인터페이스나 상위 클래스를 참조해 사용하는 방식으로만 사용하는게 좋다

- 리플렉션의 단점 두가지를 보여준다.
 - 런타임에 총 6가지가 되는 예외를 던질 수 있다.
 - 인스턴스를 리플렉션 없이 생성했다면 컴파일시 잡을 예외들이다.
 - 클래스 이름만으로 인스턴스를 생성해내기 위해 무려 25줄이나 작성했다.
 - 생성자 호출 한 줄이면 끝날 일이었다.
 - 참고로, 리플렉션 예외 각각을 잡는 대신 모든 리플렉션 예외의 상위 클래스인, `ReflectiveOperationException` 을 잡도록 하여 예외를 줄일 수 있다.
- 이 프로그램을 컴파일하면 비검사 형변환 경고가 뜬다.
 하지만 `Class<? extends Set<String>>` 으로 형변환은 심지어 명시한 `Set` 클래스를 구현하지 않았더라도 성공할 것이라, 실제 문제로 이어지진 않는다.
 단, 그 클래스의 인스턴스를 생성하려 할 때 `ClassCastException` 을 던진다. (아이템 27)
- 정리
 - 리플렉션은 복잡한 특수 시스템을 개발시 강력한 기능이지만, 단점도 많다.
 - 컴파일 타임에는 알 수 없는 클래스를 사용하는 프로그램을 작성한다면 리플렉션을 사용해야 할 것이다.
 - 단, 되도록 객체 생성에만 사용하고, 생성한 객체를 이용할 때는 적절한 인터페이스나 컴파일타임에 알 수 있는 상위 클래스로 형변환해 사용해야 한다.

▼ 66. 네이티브 메서드는 신중히 사용하라

▼ 67. 최적화는 신중히하라

▼ 68. 일반적으로 통용되는 명명 규칙을 따르라