


# 이펙티브 자바 CP.8

🕒 작성 일시	@2023년 3월 4일 오후 2:23
🕒 최종 편집 일시	@2023년 3월 8일 오후 11:18
📄 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

▼ 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