Effective Java

Item 57 : 지역변수의 범위를 최소화하라

Item 58 : for-each 문을 사용하라

Item 59 : 라이브러리를 익히고 사용하라

Item 57

지역변수의 범위를 최소화하라

for 문 원소 삭제

Quiz 1

```java
List<Integer> list = new ArrayList<>();
list.add(2);
list.add(2);
list.add(3);

int loopCount = 0;
for (int i = 0; i < list.size(); i++) {
    loopCount++;
    if (list.get(i) == 2) {
        list.remove(i);
    }
}
```

loopCount = 2

list = [2, 3]

```
for (i = 0; list.size = 3) {
    list = [2, 2, 3]
}

for (i = 1; list.size = 2) {
    list = [2, 3]
}
```

for 문 원소 삭제

Quiz 2

```
List<Integer> list = new ArrayList<>();
list.add(2);
list.add(2);
list.add(3);

int loopCount = 0;
for (Integer i : list) {
    loopCount++;
    if (i == 2) {
        list.remove(i);
    }
}
```

ConcurrentModificationException!

for 문 원소 삭제

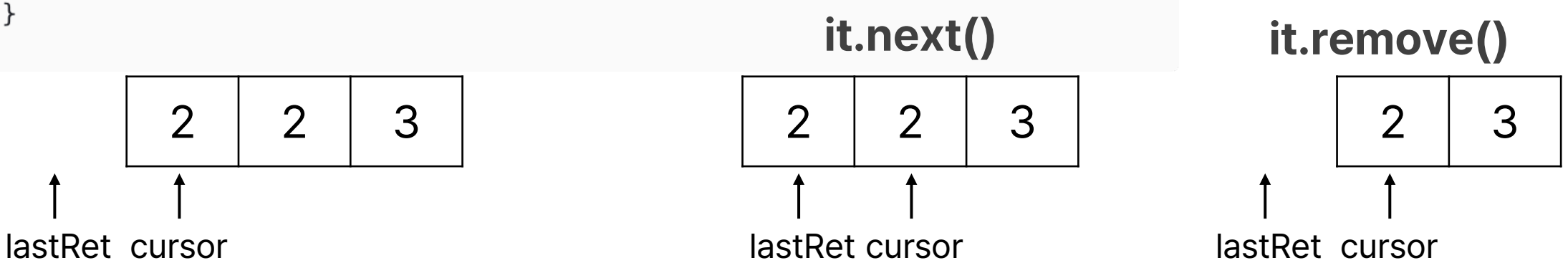
Quiz 3

```
List<Integer> list = new ArrayList<>();
list.add(2);
list.add(2);
list.add(3);

int loopCount = 0;
for (Iterator<Integer> iterator = list.iterator(); iterator.hasNext(); ) {
    loopCount++;
    if (iterator.next() == 2) {
        iterator.remove();
    }
}
```

loopCount = 3

list = [3]



for 문 원소 삭제

Quiz 4

```
List<Integer> list = new ArrayList<>();
list.add(2);
list.add(2);
list.add(3);

list.removeIf(i -> i == 2);
```

list = [3]

```
default boolean removeIf(Predicate<? super E> filter) {
    Objects.requireNonNull(filter);
    boolean removed = false;
    final Iterator<E> each = iterator();
    while (each.hasNext()) {
        if (filter.test(each.next())) {
            each.remove();
            removed = true;
        }
    }
    return removed;
}
```

그 외

지역변수는 가장 처음 쓰일 때 선언
(+ 선언과 동시에 초기화)

단, try-catch 문은 어쩔 수 없다.

```
String str;
try {
    str = method1();
} catch (Exception e) {
    e.printStackTrace();
    str = method2();
}
```

while 문 보다는 for 문

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
    Integer e = it.next();
    ...
}
```

```
for (Iterator<Integer> it = list.iterator(); it.hasNext(); ) {
    Integer e = it.next();
    ...
}
```

복잡한 연산은 미리 저장

```
for (int i = 0, n = expensive(); i < n; i++) {
    ...
}
```

Item 58

for-each 문을 사용하라

for-each 문을 사용할 수 없는 상황

순회하면서 원소 제거

```
int loopCount = 0;
for (Integer i : list) {
    loopCount++;
    if (i == 2) {
        list.remove(i);
    }
}
```

원소 변경

```
for (int i = 0; i < list.size(); i++) {
    list[i] += 3;
}
```

Item 59

라이브러리를 익히고 사용하라

Random

```
Random random1 = new Random(10000000);
for (int i = 0; i < 3; i++) {
    System.out.print(random1.nextInt() + " ");
}
System.out.println("\n-----------------------------");

Random random2 = new Random(10000000);
for (int i = 0; i < 3; i++) {
    System.out.print(random2.nextInt() + " ");
}
```

```
-1975323259 -989177031 -2050808487
-----------------------------
-1975323259 -989177031 -2050808487
```

seed를 넣지 않으면 시각 이용

```
public Random() {
    this( seed: seedUniquifier() ^ System.nanoTime());
}

private static long seedUniquifier() {
    // L'Ecuyer, "Tables of Linear Congruential Generators of
    // Different Sizes and Good Lattice Structure", 1999
    for (; ; ) {
        long current = seedUniquifier.get();
        long next = current * 1181783497276652981L;
        if (seedUniquifier.compareAndSet(current, next))
            return next;
    }
}

private static final AtomicLong seedUniquifier
        = new AtomicLong( initialValue: 8682522807148012L);
```

```
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    } while (!seed.compareAndSet(oldseed, nextseed));
    return (int) (nextseed >>> (48 - bits));
}
```

Random

Many applications will find the method `Math.random` simpler to use.

Instances of `java.util.Random` are threadsafe. However, the concurrent use of the same `java.util.Random` instance across threads may encounter contention and consequent poor performance. Consider instead using `java.util.concurrent.ThreadLocalRandom` in multithreaded designs.

Instances of `java.util.Random` are not cryptographically secure. Consider instead using `java.security.SecureRandom` to get a cryptographically secure pseudo-random number generator for use by security-sensitive applications.

간단하게 사용하려면 **Math.random** 을 사용해라 (static method)

멀티 스레드 환경에서 Random 객체는 threadsafe 함.
하지만 성능이 떨어진다. **ThreadLocalRandom**을 사용해라

```
protected int next(int bits) {
    long oldseed, nextseed;
    AtomicLong seed = this.seed;
    do {
        oldseed = seed.get();
        nextseed = (oldseed * multiplier + addend) & mask;
    } while (!seed.compareAndSet(oldseed, nextseed));
    return (int) (nextseed >>> (48 - bits));
}
```

Random은 암호학적으로 안전하지 않으니, **SecureRandom**을 사용해라

바퀴를 다시 발명하지 마라