9장 일반적인 프로그래밍 원칙

분류	서적 이펙티브 자바
시작일자	@2022년 5월 30일 오후 11:18
종료일자	@2022년 6월 4일 오후 6:36
참고	참고 블로그 [자바봄 기술 블로그] 참고 github.ido
스터디 링크	GitHub Link [이펙티브 자바 스터디]
서적	
Java	

▼ **목차**

아이템 57. 지역변수의 범위를 최소화해라

> item 15의 클래스와 멤버의 접근 권한을 최소화하라와 비슷한 이유이다.

❓ **지역변수 범위를 최소화하는 이유?**

- 지역변수의 유효 범위를 최소한으로 줄이면 코드 가독성과 유지보수성을 높이고, 오류 가능성을 줄여준다.

1. 가장 처음 쓰일 때 선언하기

> 지역변수의 범위를 줄이는 가장 좋은 방법은 처음 쓰일 때 선언하기이다.

- 미리 변수를 선언하고 나중에 할당을 하게 되면 코드가 어수선해서 가독성이 떨어진다.
  또한 변수를 실제로 사용하는 시점에는 타입과 초기값이 기억나지 않을 수 있다.
- 변수의 범위를 제대로 제한하지 않으면 이미 사용된 뒤에도 할당 해제가 되지 않고, 계속 힙 메모리에 유지가 된다.
  즉 GC가 이 변수를 수거하지 못한다는 의미와 같다.

2. 모든 지역변수는 선언과 동시에 초기화 해야 한다.

> 초기화에 필요한 정보가 충분치 않다면 정보가 충분해질 때까지 선언을 미뤄야 한다.

- **try - catch 문에서는 예외이다.**

```
int memberId;
try {
    memberId = memberIdFuture.get();
} catch (InterruptException e) {
  throw new RuntimeException();
}
```

- **반복문은 특이한 방식으로 변수 범위를 초기화 해준다.**

```
for (Iterator<Integer> i = c.iterator(); i.hasNext();) {
  Integer number = i.next();
}

i.hasNext(); // 컴파일 오류 발생
```

> for문에서 정의한 i는 for문 바깥에서는 범위를 벗어나기 때문에 컴파일 에러가 발생한다.

```
Iterator<Integer> i = c.iterator();
while (i.hasNext()) {
  doSomething(i.next());
}

// ...
Iterator<Integer> i2 = c2.iterator();
while (i.hasNext()) {
  // 버그 발생
  // 두 번째 i는 이미 이전 while에서 사용했던 i이기 때문에
  // while은 지역변수의 범위를 제한하지 못하기 때문에 아무런 컴파일 에러가 발생하지 않는다.
  doSomething(i2.next());
}
```

위 코드는 컴파일 에러가 발생하지 않는다.

하지만 아래의 런타임에서 에러가 발생한다.

```
for (Iterator<Integer> i = c.iterator(); i.hasNext();) {
  Integer number = i.next();
}

// ...
for (Iterator<Integer> i2 = c2.iterator(); i.hasNext();) { // compile error
  Integer number = i2.next();
}
```

3. 메서드를 작게 유지하고 한 가지 기능에 집중하도록 만들자

> 한 메서드에서 여러가지 기능을 처리한다면 그 중 한 기능만 관련된 지역변수라도 다른 기능을 수행하는 코드에 접근할 수 있다.
> 이를 방지하기 위해서 메서드를 한 가지 기능만 하도록 쪼개는 것이 좋다.

아이템 58. 전통적인 for문 보다는 for-each 문을 사용하라.

```
List<String> fruits = List.of("Apple", "Orange", "Melon", "Lemon", "Banana");
int[] numbers = {1, 2, 3, 4, 5};

for (Iterator<String> iter = fruits.iterator(); iter.hasNext()) {
    // ...
}

for (int i = 0; i < numbers.length; i++) {
    // ...
}
```

while문 보다는 낫지만 최선의 방법은 아니다.

❓ 최선의 방법?

> 왜 위 방법이 최선이 아닐까?

반복자와 인덱스 변수는 모두 코드를 지저분하게 한다. 즉 필요한 것은 **원소들** 뿐이다.

더군다나 위와 같이 요소의 종류 횟수가 늘어나면 오류가 생길 가능성만 높아진다.

```
List<String> fruits = List.of("Apple", "Orange", "Melon", "Lemon", "Banana");
int[] numbers = {1, 2, 3, 4, 5};

for (String fruit : fruits) {
    // ...
}

for (int number : numbers) {
    // ...
}
```

이렇게 향상된 for문을 사용하면 모든게 해결이 된다.

- 반복자와 인덱스 변수를 사용하지 않으니 코드가 깔끔해지고 오류가 날 일이 없다.
- 하나의 관용구로 컬렉션과 배열을 모두 처리할 수 있어 어떤 컨테이너를 다루는지 신경을 쓸 필요가 사라진다.

주사위를 두 번 굴렸을 때 나올 수 있는 모든 경우의 수 출력

```
public class ForEachTest{
    enum Face {ONE, TWO, THREE, FOUR, FIVE, SIX}

    public static void main(String[] args) {
        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.print(i.next() + " " + j.next() + ", ");
            }
            System.out.println();
        }

        System.out.println();

        for (Face face : faces) {
            for (Face face1 : faces) {
                System.out.print(face + " " + face1 + ", ");
            }
            System.out.println();
        }
```

```
    }
}
```

```
ONE ONE, TWO TWO, THREE THREE, FOUR FOUR, FIVE FIVE, SIX SIX,

ONE ONE, ONE TWO, ONE THREE, ONE FOUR, ONE FIVE, ONE SIX,
TWO ONE, TWO TWO, TWO THREE, TWO FOUR, TWO FIVE, TWO SIX,
THREE ONE, THREE TWO, THREE THREE, THREE FOUR, THREE FIVE, THREE SIX,
FOUR ONE, FOUR TWO, FOUR THREE, FOUR FOUR, FOUR FIVE, FOUR SIX,
FIVE ONE, FIVE TWO, FIVE THREE, FIVE FOUR, FIVE FIVE, FIVE SIX,
SIX ONE, SIX TWO, SIX THREE, SIX FOUR, SIX FIVE, SIX SIX,
```

for-each를 사용할 수 없는 경우

- 파괴적인 필터링
  - 컬렉션을 순회하면서 선택된 원소를 제거해야 할 때
- 변형
  - 리스트나 배열을 순회하면서 그 원소의 값 일부 혹은 전체를 교체해야 할 때
- 병력 반복
  - 여러 컬렉션을 병렬로 순회할 때

```
List<String> list = List.of("A", "B", "C", "D");

// 파괴적인 필터링
for (Iterator<String> iterator = list.iterator(); iterator.hasNext(); ) {
    String element = iterator.next();
    if(element.equals("A")) {
        iterator.remove();
    }
}

list.removeIf(element -> element.equals("A"));

// 변형
String[] arr = new String[]{"A", "B", "C"};

for (int i = 0; i < arr.length; i++) {
    String s = arr[i];
    if ("a".equals(s)) {
        arr[i] = "d";
    }
}

// 병렬 반복
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, ... }

static List<Suit> suits = Arrays.asList(Suit.values());
static List<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(); ) {
        // i.next() 메서드가 '숫자(suit) 하나당' 한 번씩만 불려야 하는데
        // 카드(Rank) 하나당 불리고 있다.
        deck.add(new Card(i.next(), j.next()));
    }
}
```

📋 간단 정리

- for-each 문은 명료하고 유연하며 버그를 예방해주는 역할을 한다.
- 성능 저하도 없다.

→ 가능한 for문이 아닌 for-each를 애용하자!

아이템 59. 라이브러리를 익히고 사용하라

1. 라이브러리에 미숙할 경우

```
public class RandTest {
    static Random rnd = new Random();
    static int random(int n){
        return Math.abs(rnd.nextInt()) % n;
    }
```

```
    public static void main(String[] args) {
        int n = 2 * (Integer.MAX_VALUE / 3);
        int low = 0;
        for (int i = 0; i < 1_000_000; ++i)
            if (random(n) < n/2)
                low++;
        System.out.println(low);
    }
}
```

> 다양한 문제가 있다.

- n이 크지 않은 2잊 ㅐ곱수라면 얼마 지나지 않아 같은 수열이 반복된다.
- n이 2의 제곱수가 아니라면 몇몇 숫자가 평균적으로 더 자주 반복된다.
  - 난수라면 평균적으로 low의 값이 50만으로 나와야하지만, 66만에 가깝게 나온다.
- 지정한 범위 바깥의 수가 튀어날 수 있는 조건이 있다.
  - rnd.nextInt() 가 Integer.MIN_VALUE 를 반환하면 결과적으로 음수가 반환되기도 한다.

❓ **Math.abs는 절댓값인데 왜 음수가 반환된다는거지??**

스택오버플로우 설명

```
// 우리의 예상
Intger.MIN_VALUE (-2_147_483_648) -> Math.abs() -> +2_147_483_648
// 현실
Intger.MIN_VALUE = 2_147_483_647
// 결과
Math.abs() 를 통해 변환한 값이 Integer.MAX_VALUE 보다 크다!
-> +2_147_483_648 = -2_147_483_648
```

간단히 말하면 범위값을 넘어가게 되면서 부호비트가 변하기 때문이다.

❓ **그럼 어떻게 해결하는건데?**

> 간단하다. 다른 함수를 사용하면 되는 것이다.

- Random.nextInt(intger)를 사용하면 된다.
- 하지만 이후 내용을 보면 알 수 있겠지만 Random을 쓰지 말라고 한다.

🌟 **이처럼 라이브러리에 대해 제대로 이해하고 익히지 않으면 예상치 못한 오류를 만나게 된다.**
예제의 오류가 실제 서비스에서 난다고 생각하면 정말 끔찍할 것 같다.
하지만 이걸 일반 프로그래머가 신경쓰며 개발하기란 쉽지가 않다. 그렇더라도 라이브러리에 대해 이해하고 익히는 습관을 들여야한다.

2. 표준 라이브러리를 사용해야 하는 이유

> 코드를 작성한 전문가의 지식과 앞서 사용한 다른 프로그래머의 경험을 활용할 수 있다.

자바 7 부터는 Random → ThreadLocalRandom으로 대체됐다.

포크-조인 풀, 병렬 스트림인 경우라면 SplittableRandom을 사용하자.

- 핵심적인 일과 크게 관련 없는 문제를 해결하느라 시간을 허비하지 않아도 된다.
  - 애플리케이션 기능 개발에 집중할 수 있다.
- 따로 노력하지 않아도 성능이 지속해서 개선된다.
  - 사용자가 많고 업계 표준 벤치마크를 사용하기 때문
- 기능이 점점 많아진다.
  - 부족한 부분에 대해서 개발자 커뮤니티에서 논의가 계속되면서 이를 보완해 릴리즈에 기능이 추가된다.
- 표준 라이브러리를 활용한 코드는 많은 사람에게 낯익은 코드가 된다.
  - 가독성이 좋고, 유지보수 하기 편하며, 재활용 하기 좋다.

🌟 하지만 프로그래머들이 직접 구현해서 사용하는 경우도 많은데 대부분이 라이브러리에 그러한 기능이 있는지 모르는 경우가 많다. 매번 릴리즈 마다 새로운 기능을 설명하는 웹페이지가 있으니 한 번씩 읽어보는게 좋을 것 같다.

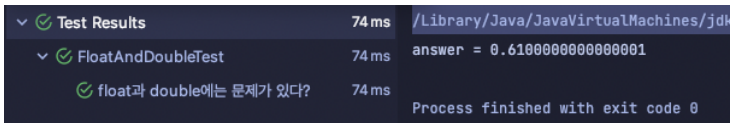
3. 주로 쓰는 라이브러리

- java.lang, java.util, java.io, Collection framework, Stream, java.util.concurrent

아이템 60. 정확한 답이 필요하다면 float와 double은 피하라.

❓ float과 더블은 문제가 있다?

```
@Test
@DisplayName("float과 double에는 문제가 있다?")
void 플롯더블테스트(){
    double answer = 1.03 - 0.42; // 0.61 예상
    System.out.println("answer = " + answer);
    Assertions.assertNotEquals(answer, 0.61);
}
```



🌟 **float과 double은 부동소수점 방식을 사용하고 있기 때문!**

고정 소수점과 부동 소수점

> 컴퓨터는 숫자를 2진수로 표현을 한다.

❓ 그럼 2진수로 표현하는게 문제인가?

- 정답은 아니다. 표현 자체의 문제는 없다.
- 하지만 소수점을 표함한 실수를 표현할 때는 상황이 달라진다.
- 소수점의 위치를 표현하고, 무엇이 정수 부분이고 무엇이 실수 부분인지 구분해야 하기 때문이다.

→ 이를 위해 고정 소수점과 부동 소수점 방식을 사용한다.

고정 소수점 방식

> 소수점의 위치를 고정시키는 것이다.

실수를 표현하는 5비트의 고정 소수점 방식 자료형이 있다고 가정한다.

2지트를 정수부 3비트를 실수부로 사용한다고 하면

`12.345` 이런 식으로 소수점의 위치가 고정되어 있기 ㄸ문에 정수를 표현하는 것과 똑같이 실수를 표현할 수 있다.

🌟 **오차 없는 정확한 여산이 가능하다는 장점!**
하지만 표현 범위가 제한적이다
즉 123.45, 1234.5 같이 표현하는게 불가능하다.

부동 소수점 방식

> 지수부와 가수부로 구분한다.

소수점이 고정적이지 않으므로 더 폭넓은 범위를 표현할 수 있다.

필연적으로 오차가 발생하여 근사값을 구하게 된다.

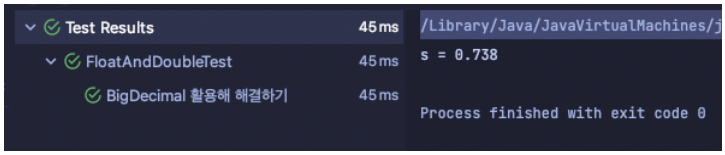
자바의 float과 double은 부동소수점 방식으로 표현돼 있다.

❓ 그럼 해결을 못하는것인가?

- `BigDecimal` 를 활용해 해결할 수 있다.

```
@Test
@DisplayName("BigDecimal 활용해 해결하기")
void 플롯더블대신사용() {
    BigDecimal bigDecimal1 = new BigDecimal("1.25");
```

```
    BigDecimal bigDecimal2 = new BigDecimal("0.512");
    String s = bigDecimal1.subtract(bigDecimal2).toString(); // 0.738
    System.out.println("s = " + s);
    assertEquals(s, "0.738");
}
```



- 정수 타입 이용

  실수 타입을 정수 타입으로 사용하는 방식이다.

  현실의 예로 0.1 달러 = 10센트 인 것처럼 표현한다는 것이다.

  정수 기본형을 사용하기 때문에 연산도 정확하고 사용하기도 편리하며 빠르다.

  하지만 모든 상황에서 사용할 수가 없기에 무조건 좋은 것은 아니다.

아이템 61. 박싱된 기본 타입보다는 기본 타입을 사용하라.

자바의 데이터 타입

- 기본 타입인 int, double, boolean 등이 있다.
- 참조 타입인 Integer, String, Double 등이 있다.

기본타입은 모두 대응되는 참조 타입이 있으며 이를 **박싱된 기본타입**이라고 한다.

> 사실 별로 신경안쓰고 개발했던 것 같다.
> 그냥 쓰다가 오류 뜨면 참조형으로 바꾸고 그랬었따 ㅎㅎ;

기본타입과 박싱된 기본 타입

- 기본 타입은 값만 가지고 있으며, 박싱된 기본 타입은 값뿐만 아니라 식별성이란 속성도 가지고 있다.

```
final int number1 = 1;
final int number2 = 1;

System.out.println(number1 == number2); // true

final Integer number3 = 1;
final Integer number4 = 1;
System.out.println(number3 == number4); // true
System.out.println(number3.equals(number4)); // true

final Integer number5 = new Integer(1);
final Integer number6 = new Integer(1);

System.out.println(number5 == number6); // false
System.out.println(number5.equals(number6)); // true
```

- 기본타입의 값은 null을 가질수 없으나 박싱된 기본 타입의 값은 null을 가질 수 있다.

```
int number = null // 에러
Integer number = null; // 가능
```

- 기본 타입이 박싱된 기본 타입보다 시간과 메모리 사용면에서 더 효율적이다.

예시

- Integer로 오름차순으로 정렬하는 비교자

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

- 제대로 동작은 하지만 문제가 있다.

- Integer를 new 예약어로 같은 값을 넣어 사용하게 되면 `i==j` 는 false가 나오게 된다.
- 이 예시는 String도 new 예약어가 아닌 리터럴로 생성하라는 이전의 아이템과 같은 이유이다.
- `==` 대신 `equals` 를 사용하자.

❓ **그럼 오토박싱을 사용하면 되잖아?**

> 오토박싱은 항상 말하지만 지양해야한다.
> 성능저하가 심하며 아이템6의 내용을 다시 생각해보면 하지 말아야하는 이유를 알 수 있다.

NullPointerException

기본타입과 박싱된 기본 타입을 혼용한 연산에서는 기본적으로 박싱된 기본타입의 박싱이 풀리게 된다.(오토 언박싱)

→ Intger가 null이라면?? → **NPE가 발생!!**

박싱 타입을 사용해야 할 때

1. 컬렉션의 원소, 키 값으로 사용할 때

   → 애초에 기본 타입 쓰고 싶어도 못쓴다.

2. 제네릭을 사용할 때

   → 제네릭에서도 기본타입을 지원해주지 않는다.

3. 리플렉션을 통해 메서드를 호출 할 때 (아이템 65)

아이템 62. 다른 타입이 적절하다면 문자열 사용을 피하라

문자열은 다른 타입을 대신하기에 적절하지 않다.

> 말 그대로 입력받을 데이터가 문자열인 경우에만 사용하자.

숫자 → int double float

참 거짓 → boolean Enum 자료형

문자열은 열거 타입을 나타내기에 적절하지 않다.

```
public static final int APPLE_FUJI         = 0;
public static final int APPLE_PIPPIN       = 1;
public static final int APPLE_GRANNY_SMITH  = 2;

public static final int ORANGE_NAVEL  = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOOD  = 2;
```

⛔ **정수 열거 타입의 문제점**

> 정수 열거 패턴의 문제는 아이템 34에서 다뤘다.

- 타입의 안전성을 보장받을 수 없다.
- 접두어를 사용하여 이름 충돌을 방지해야 한다. → 별도의 이름 공간이 없다.
- 프로그램이 깨지기 쉽다.

평범한 상수를 나열한 것 뿐이라 컴파일하면 그 값이 클라이언트 파일에 그대로 새겨진다.

즉 API의 상수 값이 바뀌면 클라이언트도 재컴파일해야한다.

⛔ **문자열 패턴의 문제점**

> 상수의 의미를 출력하는 것도 좋지만 이를 하드 코딩해야 한다.
> → 오타가 있어도 컴파일러가 확인할 방법이 없다.

열거 패턴 → 열거 타입

```
public enum Apple { FUJI, PIPPIN, GRANNY_SMITH }
public enum Orange { NAVEL, TEMPLE, BLOOD }
```

문자열은 혼합 타입을 대신하기 어렵다.

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

각 요소를 분리해서 사용할려고 #으로 구분을 했다.

하지만 이렇게 하면 String에서 제공되는 것 외에 메서드들을 사용하기 너무 까다롭다.

toString, equals ComapreTo 등등

```
public class studyInfo {
    private String team;
    private int itemNum;

    private static class compoundKey{
        private String team;
        private int itemNum;

        public compoundKey(String team, int itemNum) {
            this.team = team;
            this.itemNum = itemNum;
        }
        public studyInfo compound(){
            return new studyInfo(this);
        }
    }

    private studyInfo(compoundKey compoundKey){
        team = compoundKey.team;
        itemNum = compoundKey.itemNum;
    }

    public static void main(String[] args) {
        ArrayList<Integer> itemList = new ArrayList<>(List.of(60,61,62));
        Iterator<Integer> i = itemList.iterator();

        studyInfo student1 = new studyInfo.compoundKey("B팀",i.next()).compound();
        studyInfo student2 = new studyInfo.compoundKey("A팀",i.next()).compound();
        studyInfo student3 = new studyInfo.compoundKey("B팀",i.next()).compound();

        System.out.println("student1: " + student1.team + " 아이템 " + student1.itemNum);
        System.out.println("student2: " + student1.team + " 아이템 " + student2.itemNum);
        System.out.println("student3: " + student1.team + " 아이템 " + student3.itemNum);
    }
}
/*
student1: B팀 아이템 60
student2: A팀 아이템 61
student3: B팀 아이템 62
*/
```

문자열은 권한을 표기하기에 적합하지 않다.

> 권한을 문자열로 표현하는 경우 보안이 취약해지며 의도적으로 같은 키를 사용하여 값을 탈취하는 문제점이 생길 수 있다.

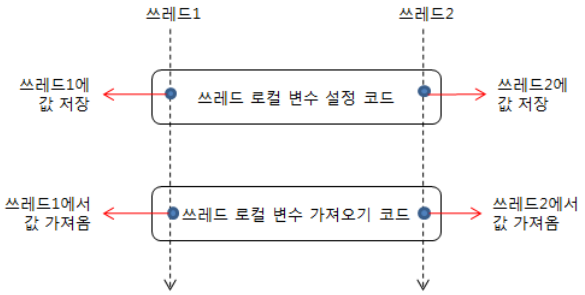
권한을 문자열로 관리하는 ThreadLocal 예시

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

  // 현 스레드의 값을 키로 구분해 저장한다.
  public static void set(String key, Object value);

    // (키가 가리키는) 현 스레드의 값을 반환한다.
    public static Object get(String key);
}
```

- 스레드 구분용 문자열 키가 전역 이름 공간에서 공유된다는 문제점
- 두 클라이언트가 서로 소통하지 못해 같은 키를 쓰기로 결정했다면, 의도하지 않게 같은 변수를 공유하게 된다.



해결하기 위해서 API는 문자열 대신 위조할 수 없는 키를 사용한다.

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

    public static class Key { // (권한)
        Key() { }
    }
    public staatic getKey() {
        return Key;
    }

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

- 위 방법은 문자열 기반 API의 문제 두 가지를 모두 해결해주지만, 개선의 여지가 존재한다.
- set, get메서드는 정적 메서드일 이유가 없으니 Key 클래스의 인스턴스 메서드로 옮긴다.
- Key는 더 이상 스레드 지역변수를 구분하기 위한 키가 아니라, 그 자체가 **스레드 지역변수**가 된다.
- 결과적으로 지금 톱레벨 클래스인 ThreadLocal은 별달리 하는 일이 없어지므로 치워버리고, 중첩 클래스 Key의 이름을 ThreadLocal로 바꾼다.

```
// Key -> ThreadLocal
public final class ThreadLocal {
    public ThreadLocal() { }
    public void set(Object value);
    public Object get();
}
```

- get으로 얻은 Object를 실제 타입으로 형변환을 해서 사용해야 하기에 타입이 안전하지 않다.
- ThreadLocal을 매개변수화 타입으로 선언하여 문제를 해결하면 된다.

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

아이템 63. 문자열 연결은 느리니 주의하라.

`+` 를 이용해서 문자열을 연결하면 편리하다.

하지만 문제가 있다. 엄청나게 느리다는 것이다.

❓ **왜 느리지?**

> 문자열 연결 연산자로 문자열 N개를 잇는 시간은 N^2에 비례한다.
> 그리고 문자열은 불변이기에 불변인 문자열을 연결하기 위해서는 양쪽의 내용을 복사해 새로운 문자열을 만들기 때문이다.

해결 방법 StringBuilder 와 StringBuffer

StringBuilder

- 빌더 패턴을 이용해서 String을 처리한다고 생각하면 편하다.
- Thread-Safe 하지는 않지만 싱글쓰레드 환경에서는 연산처리가 굉장히 빠르빈다.

StringBuffer

- StringBuilder와 기능은 똑같지만, 멀티쓰레드 환경을 고려해 synchronized 키워드를 사용해 동기화를 할 수 있다.

  → Thread-Safe한 객체이다.

String.join

```
public static String join(CharSequence delimiter, CharSequence... elements) {
    Objects.requireNonNull(delimiter);
    Objects.requireNonNull(elements);
    // Number of elements not likely worth Arrays.stream overhead.
    StringJoiner joiner = new StringJoiner(delimiter);
    for (CharSequence cs: elements) {
        joiner.add(cs);
    }
    return joiner.toString();
}
```

- StringBuilder와 비슷하게 동작하지만, 구분자 넣는 기능이 추가됐다.

아이템 64. 객체는 인터페이스 타입으로 참조하라

1. 객체는 클래스가 아닌 인터페이스 타입으로 선언하라.

```
//좋은 예시
Set<String> set = new LinkedHashSet<>();

//나쁜 예시
LinkedHashSet<String> set = new LinkedHashSet<>();
```

즉 상세한 클래스보단 그 상위 인터페이스로 선언을 하라는 의미이다.

이렇게 선언할 경우 나중에 새로운 타입으로 선언해도 추가적인 변경이 필요가 없다.

프로그램이 유연해진다.

```
//LinkedHashSet -> HashSet으로 구현체를 변경
Set<String> set = new HashSet<>();

set.add(...);
set.remove(...);
set.size(...);
set.contains(...);

public void ...(Set<String> set, ...)
```

```
//LinkedHashSet -> HashSet으로 구현체를 변경
HashSet<String> set = new HashSet<>();

set.add(...);
set.remove(...);
set.size(...);
set.contains(...);

//컴파일 에러
public void ...(LinkedHashSet<String> set, ...)
```

2. 적합한 인터페이스가 없다면 당연히 클래스로 참조해야 한다.

String, BingInteger 같은 값 클래스

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    /**
     * The value is used for character storage.
    **/
}
```

클래스 기반으로 작성된 프레임워크가 제공하는 객체

java.io 패키지

OutputStream 을 구현하는 클래스는

FileOutputStream, ByteArrayOutputStream 가 있다.

확장한 기능을 사용하기 위해선 클래스를 참조해야 한다.

```
public abstract class OutputStream implements Closeable, Flushable {

...

}

OutputStream outputStream = new FileOutputStream("");
```

특정 구현 클래스 보다는 기반 클래스인 OutputStream을 사용하는 것이 더 좋다.

3. 좀 더 자세한 예시

김영한 강사님 강의에서 들었던 예제

```
public interface Repository<T> {
    T save(T t);
    Optional<T> get(Long id);
}
```

```
public class MemoryMemberRepo implements Repository<Member> {

    private static final HashMap<Long ,Member> memoryDB = new HashMap<>();
    private static Long key = 1L;

    @Override
    public Member save(Member member) {
        memoryDB.put(key,member);
        key++;
        return member;
    }

    @Override
    public Optional<Member> get(Long id) {
        if(memoryDB.containsKey(key)){
            return Optional.of(memoryDB.get(key));
        }
        return Optional.empty();
    }
}
```

```
public class RDBMemberRepo implements Repository<Member> {
    //TODO DB 커넥션 연결함.
    @Override
    public Member save(Member member) {
        //TODO : DB에 저장 했음
    }

    @Override
    public Optional<Member> get(Long id) {
        //TODO : DB 검색해서 가져옴
    }
}
```

❓ 상위 인터페이스가 있음에도 불구하고 Service단에서 구현 클래스의 타입을 사용한다면?

> Service는 MemoryDB, RDB 둘 중 하나만의 기능에만 의존하게 된다.

```
public class MemberService {

    // 상세 클래스인 MemoryMemberRepo를 참조하고 있다.
    private MemoryMemberRepo repo = new MemoryMemberRepo();
    // 상세 클래스인 RDBMemberRepo를 참조하고 있다.
    private RDBMemberRepo repo = new RDBMemberRepo();

    public Member save(Member member){
        return repo.save(member);
    }
```

```
    public Member getMember(Long id){
        Optional<Member> member = repo.get(id);
        return member.orElseThrow(IllegalArgumentException::new);
    }

}
```

하지만 내가 MemoryMemberRepo를 쓰다가 중간에 RDBMemberRepo를 사용하고 싶다고 해서 코드를 수정하게 될 경우 객체지향적으로 코드를 짜지 못한 것이다.

해결 방법

```
public class MemberService {

    private final Repository<Member> repo;

    public MemberService(Repository<Member> repo) {
        this.repo = repo;
    }

   // 기능 생략
}
```

사용할 때 어떠한 구현체를 주입하냐에 따라서 MemberService는 RDB를 사용할지 MemoryDB를 사용할지 결정하게 된다.

아이템 65. 리플렉션보다는 인터페이스를 사용하라.

1. 리플렉션이란?

> Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine. This is a relatively advanced feature and should be used only by developers who have a strong grasp of the fundamentals of the language. With that caveat in mind, reflection is a powerful technique and can enable applications to perform operations which would otherwise be impossible.
> 오라클 API 문서

java.lang.reflection를 이용하면 프로그램에서 임의의 클래스에 접근할 수 있다.

- 클래스의 생성자, 메서드, 필드에 해당하는 constructor, method, field 인스턴스를 가져올 수 있다.
- 클래스의 멤버 이름, 필드 타입, 메서드 시그니처 등을 가져올 수 있다.
- 위의 것들을 활용해서 실제로 인스턴스를 생성하거나, 메서드를 호출, 필드에 접근할 수 있다.
- **컴파일 당시에 존재하지 않던 클래스도 이용**할 수 있다.

2. 리플렉션의 단점

- 컴파일타임 검사가 주는 이점을 하나도 누릴 수 없다.

  → 존재하지 않던 클래스도 이용하기 때문
- 리플렉션을 이용하면 코드가 지저분해지고 장황해진다.
  - 각종 오류 처리로 인해 로직을 보기 힘들며, 코드를 읽기가 힘들어진다.
- 성능이 떨어진다.
  - 훨씬 느리고, 책에 의하면 int 변환 메서드의 경우 일반 메서드에 비해 11배나 느리다고 한다.

3. 근데 왜 사용할까?

> 리플렉션은 아주 제한된 형태로만 사용해야 그 단점을 피하고 이점만 취할 수 있다.
> 하지만 이럼에도 해결하지 못하는 단점들은 존재한다.

- 인터페이스 및 상위클래스로 참조해 사용하자!

  즉 리플렉션은 인스턴스 생성에만 쓰고 ,이렇게 만든 인스턴스는 인터페이스나 상위클래스로 참조해 사용하자.

  `SomeInterface instance = InstanceMakerUsingReflection()`

4. 사용법

- 생성자 : Constructor

- 메서드 : Method
- 필드 : Field

```
1. 클래스 얻기
Class<?> class = Class.forName();

2. 생성자 얻기
Constructor<?> constructor = class.getDeclaredConstructor();

3. 클래스 인스턴스 만들기
constructor.newInstance();

4. 메서드 실행
Method.invoke()
```

5. 예제

- args를 통해첫 번째 인수로 지정한 클래스가 무엇이냐에 따라 저장되는 순서가 달라진다.

  → 컴파일타임에는 어떤 클래스인지 모른다.

- HashSet을 지정하면 무작위 순서가 될 것이고, TreeSet을 지정하면 알파벳 순서가 될 것이다.

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

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

    // 위에서 만든 클래스를 통해 집합 인스턴스 생성
    Set<String> s = null;
    try{
        s = cons.newInstance();
    } catch (IllegalAccessException e) {
        fatalError("생성자에 접근할 수 없습니다.");
    } catch (InstantiationException e) {
        fatalError("클래스를 인스터화 할 수 없습니다.");
    } catch (InvocationTargetException e) {
        fatalError("생성자가 예외를 던졌습니다." + e.getCause());
    } catch (ClassCastException e){
        fatalError("Set을 구현하지 않은 클래스 입니다.");
    }

    // 생성한 인스턴스 사용
    s.addAll(Arrays.asList(args).subList(1, args.length));
    System.out.println(s);
}

private static void fatalError(String msg){
    System.out.println(msg);
    System.exit(1);
}
```

```
args[0] 가 HashSet 일 때 : [a, b, c, d, e, f, z, j, k, l]
args[0] 가 TreeSet 일 때 : [a, b, c, d, e, f, j, k, l, z]
```

당장 이 코드만 봐도 단점을 2가지나 찾을 수 있다.

- 컴파일타임에 잡을 수 있는 다수의 예외를 런타임에서 잡아야한다.
- 일반적으로 생성했으면 생성자 호출 한 줄로 끝날 인스턴스 생성을 몇십줄에 걸쳐서야 해냈다.

📋 개인적인 결론

리플렉션은 인스턴스 생성에서만 사용하고, 생성된 인스턴스는 인터페이스로 형변환해서 사용하는게 깔끔할 것 같다.

아이템 66. 네이티브 메서드는 신중히 사용하라.

1. 자바 네이티브 인터페이스(JNI) 란?

- 자바 프로그램이 네이티브 메서드를 호출하는 기술
- 네이티브 메서드란, C나 C++같은 네이티브 프로그래밍 언어로 작성한 메서드

2. 네이티브 메서드의 주요 쓰임 3가지

- **레지스트리 같은 플랫폼 특화 기능 사용**
  - 자바가 성숙해지면서 하부 플랫폼의 기능들을 점차 흡수하고 있다.

    → 네이티브 메서드를 사용할 필요가 계속 줄어들고 있다
  - 자바 9는 새로 processAPI를 추가해 OS 프로세스에 접근할 수 있는 길을 만들어주었다.

```
public class JavaProcess {
    public static void printProcessInfo(){
        ProcessHandle processHandle = ProcessHandle.current();
        ProcessHandle.Info processInfo = processHandle.info();

        System.out.println("processHandle.pid(): " + processHandle.pid());
        System.out.println("processInfo.arguments(): " + processInfo.arguments());
        System.out.println("processInfo.command(): " + processInfo.command());
        System.out.println("processInfo.startInstant(): " + processInfo.startInstant());
        System.out.println("processInfo.user(): " + processInfo.user());
    }
    public static void main(String[] args) {
        printProcessInfo();
    }
}
```

```
/Library/Java/JavaVirtualMachines/jdk-11.0.13.jdk/Contents/Home/bin/java ...
processHandle.pid(): 68908
processInfo.arguments(): Optional[[Ljava.lang.String;@5abca1e0]
processInfo.command(): Optional[/Library/Java/JavaVirtualMachines/jdk-11.0.13.jdk/Contents/Home/bin/java]
processInfo.startInstant(): Optional[2022-06-04T09:21:53.391Z]
processInfo.user(): Optional[jihoon]
```

- **네이티브 코드로 작성된 기존 라이브러리 사용**
  - 대체할 만한 자바 라이브러리가 없는 네이티브 라이브러리를 사용해야 할 때는 네이티브 메서드를 써야 한다.
- **성능 개선을 목적으로 성능에 결정적인 영향을 주는 영역만 따로 네이티브 언어로 작성**
  - 성능을 개선할 목적으로 네이티브 메서드를 사용하는 것은 권장하지 않는다.
  - JVM이 계속 발전해왔으며, 현재는 대부분의 작업에서 다른 플롯팸에 견줄만한 성능을 보여주기 때문이다.

3. 네이티브 메서드의 단점

1. 메모리 훼손 오류

- 네이티브 언어가 안전하지 않으므로 네이티브 메서드를 사용하는 어플리케이션도 메모리 훼손 오류로부터 더 이상 안전하지 않다.
- 가비지 컬렉터가 네이티브 메모리는 자동 회수하지 못하고, 심지어 추적조차 할 수 없다.

2. 이식성

- 네이티브 언어는 자바보다 플랫폼을 많이 타서 이식성도 낮고 디버깅도 더 어렵다.

3. 성능과 비용, 가독성

- 주의하지 않으면 속도가 오히려 느려질 수 있다.
- 자바 코드와 네티이브 코드의 경계를 넘나들 때마다 비용도 추가된다.
- 네이티브 메서드와 자바 코드 사이의 '접착 코드(glue code)'를 작성해야 하는데,
  이는 귀찮은 작업이거니와 가독성도 떨어진다.

정리

- 네이티브 메서드가 성능을 개선해주는 일은 그리 많지 않으니 사용할 때 한 번 더 생각해보고 구현 후 꼭 비교해야 한다.
- 네이티브 코드는 최소한만 사용하고 철저히 테스트해야한다.

아이템 67. 최적화는 신중히 하라.

1. 빠른 프로그램보다는 좋은 프로그램를 작성하라

- 성능보다는 **견고한 구조**에 집중하세요
- 견고한 구조란 캡슐화가 잘 된, 다른 모듈과 결합성이 낮은 아키텍처를 이야기한다.
- **성능같은 구현 상의 문제는 나중에 최적화해서 해결할 수 있지만,**
  아키텍처의 결함은 시스템 전체를 다시 작성하지 않고는 해결하기 불가능할 수 있다.

2. 변경하기 어려운 설계에서는 성능을 생각하라

- 변경하기 가장 어려운 설계 요소는 API, 네트워크 프로토콜, 영구 저장용 데이터 포맷 등이 있다.
- 이런 설계 요소들은 완성 후에 변경하기 어렵거나 불가능할 수 있으며, 동시에 시스템 성능을 심각하게 제한할 수도 있다.
- 만약 API, 네트워크 프로토콜, 영구 저장용 데이터 포맷을 설계할 일이 있다면 성능을 염두해야 한다.

  → 변경하기 어려운 설계는 항상 시작전에 다양한 고려사항들을 체크해야한다.

3. API를 설계할 때 성능에 주는 영향을 고려하라

- 위에 말한 것처럼 API는 완성 후 변경은 매우 어려운 일이다.
- 아래를 참조해 설계할 때 성능에 주는 영향을 고려하세요
  - 내부 데이터를 변경할 수 없도록 만드세요 (아이템 50)
  - 상속이 아닌 컴포지션 관계로 만드세요 (아이템 18)
  - 구현 타입이 아닌 인터페이스 타입을 만드세요 (아이템 64)

4. 문제의 원인이 되는 곳에 최적화를 하라

- 최적화해봤자 성능이 생각보다 좋아지지 않는 경우가 있으며 심지어 성능을 더 나빠지게 할 수도 있다.

  → 최적화 이전 성능과 이후 성능을 꼭 측정해두자!
- 느릴 거라고 짐작한 부분을 최적화했는데 성능에 별다른 영향을 주지 않는다면 시간만 허비한 꼴이 된다.
- 프로파일링 도구를 사용하면 최적화를 어디에 하면 좋을지 알려준다.
  - 사용하면 메서드의 소비 시간과 호출 횟수 같은 런타임 정보를 알 수 있다.
  - 참고로 대부분의 프로파일링 도구는 **유료**이다.
- 자바 코드의 성능을 알기 쉽게 보여주는 벤치마킹 프레임워크인 JMH(Java Microbenchmark Harness)도 있습니다.
  - OpenJDK에서 개발한 무료 성능 측정 툴
  - JVM에서 돌아가는 언어를 전부 측정이 가능하다.
  - 해당 메서드의 가동시간을 알 수 있다.

> 🌟 **성능 개선, 최적화도 좋지만 견고한 구조의 프로그램을 만드는 것도 중요하다.**
> 둘 다 잡고 싶어서 코드를 작성하는 경우가 많지만 사실상 견고한 구조의 프로그램을 만들면 성능은 자연스럽게 따라오기 때문이다.
> 시스템을 구현했다면 성능을 측정해보고 충분히 빠르다면 최적화를 하지 않아도 괜찮다.

아이템 68. 일반적으로 통용되는 명명 규칙을 따르라

> 🌟 **자바의 명명 규칙은 크게 철자와 문법, 두 범주로 나눠진다.**
> 철자 규칙은 패키지, 클래스, 인터페이스, 메서드, 필드, 타입 변수의 이름을 다룬다.
> 이러한 규칙들은 특별한 이유가 없는 한 반드시 따라야 한다.

패키지와 모듈

- 이름은 각 **요소를 점(.)으로 구분하여 계층적**으로 짓는다.
- 요소들은 모두 소문자 알파벳 혹은 (드물게) 숫자로 이뤄진다.
- 조직의 인터넷 도메인 이름을 역순으로 사용한다.
  - com.google.common
    - com: company
- 예외적으로 표준 라이브러리와 선택적 패키지들은 각각 java와 javax로 시작한다.
  - java.util.Objects;
  - javax.persistence.*;

- 많은 기능을 제공하는 경우 계층을 나눠 더 많은 요소로 구성해도 좋다.
  - java.util.concurrent.atomic

클래스와 인터페이스

- 이름은 하나 이상의 단어로 이뤄지며, **각 단어는 대문자로 시작**한다.
  - List, FutherTask
- 약자의 경우 **첫 글자만 대문자**로 하는 쪽이 훨씬 많다.
  - HttpUrl vs HTTPURL

메서드와 필드

- **첫 글자를 소문자로 쓴다**는 점만 빼면 클래스 명명 규칙과 같다.
  - remove, ensureCapacity
- 단, '상수 필드'는 예외다.
- **상수 필드를 구성하는 단어는 모두 대문자**로 쓰며, **단어 사이는 밑줄로 구분**한다.
- 이름에 밑줄을 사용하는 요소로는 상수 필드가 유일하다.

변수

- 타입 매개변수 이름은 보통 한 문자로 표현한다.
- T: 임의의 타입
- E: 컬렉션 원소의 타입
- K: 맵의 키, V: 맵의 값
- X: 예외
- R: 메서드의 반환

네이밍(문법 규칙)

- 객체를 생성할 수 있는 클래스의 이름은 **보통 단수 명사나 명사구**를 사용한다.
  - Thread, PriorityQueue
- 객체를 생성할 수 없는 클래스의 이름은 **보통 복수형 명사**로 짓는다.
  - Collectors, Collections
- **어떤 동작**을 수행하는 **메서드의 이름은 동사나 동사구**로 짓는다
  - append, drawImage
- **boolean 값**을 반환하는 메서드는 보통 **is**나 (드물게) has로 시작하고 명사, 명사구, 형용사구 등으로 끝나도록 짓는다
  - isDigit, isEmpty, isEnabled

  → 개인적으로 is를 사용한다.
- 반환 타입이 boolean이 아니거나 해당 인스턴스의 속성을 반환하는 메서드의 이름은 보통 명사, 명사구, get으로 시작하는 동사구로 짓는다.
  - size, hashCode, getTime

특별한 메서드 이름

- 객체의 타입을 바꿔서, 다른 타입의 또 **다른 객체를 반환**하는 메서드는 보통 **to 형태**로 짓는다.
  - toString, toArray
- 객체의 내용을 다른 뷰로 보여주는 메서드는 asType 형태로 짓는다.
  - asList

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<>(a);
```

```
}
List<Integer> list = Arrays.asList(1, 2, 3, 4);
```

- 객체의 값을 기본 타입 값으로 변환하는 메서드의 이름은 보통 typeValue 형태로 짓는다.
  - intValue
- 정적 팩터리의 이름은 다음과 같이 짓는다.(아이템 1)
  - from, of, valueOf, getInstance, newInstance ...

정리

표준 명명 규칙은 모든 개발자가 공용으로 사용하는 규칙이기에 자연스럽게 나와야한다.

"오랫동안 따라온 규칙과 충돌한다면 그 규칙을 맹종해서는 안 된다." 상식이 이끄는대로 따르자.

📕 8장 메서드