9장 일반적인 프로그래밍 원칙

아이템 57. 지역변수의 범위를 최소화 하라

자바는 문장을 선언할 수 있는 곳이면 어디든 변수를 선언할 수 있기 때문에 가장 처음 쓰일 때 선언하는 것이 좋으며, 스코프가 오염되는 것을 방지할 수 있다.

지역변수 초기화 시점

- 거의 모든 지역변수는 선언과 동시에 초기화해야 한다.
- 예외적으로 try-catch를 사용할 때는 try 블록 안에서 초기화해야 한다.

반복문은 while 문보다는 for 문을 사용하자

- while 문을 사용할 때는 반복문 블록 밖에 불필요한 변수가 존재하기 때문에 잠재적인 오류 가능성이 있다.

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

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

아이템 58. 전통적인 for문 보다는 for-each문을 사용하라.

- 향상된 for문이라고 불리는 for-each문은 for문보다 깔끔하고 예상치 못한 오류를 줄여준다.

```
for 문으로 컬렉션 순회하기
for (Iterator<Element> i = c.iterator(); i.hasNext(); ) {
    Element e = i.next();
```

```
    //Something job..
}
```

```
for 문으로 배열 순회하기
for (int i = 0; i < a.length; i++) {
    //Something job...
}
```

- 컬렉션과 배열을 순회하는 for-each문

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

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,

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()));
```

- for문으로 나타냈을 때 굉장히 지저분하다. 그리고 마지막줄 i.next()를 호출하는 부분처럼 오류를 발생시킬 수 있다.

컬렉션이나 배열의 중첩 반복을 위한 for-each 문

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

for-each문을 사용할 수 없는 상황

파괴적인 필터링

컬렉션을 순회하면서 선택된 원소를 제거해야 한다면 반복자의 remove 메서드를 호출해야 한다. 자바 8부터는 Collection의 removeIf 메서드를 사용해 컬렉션을 명시적으로 순회하는 일을 피할 수 있다.

변형

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

병렬 반복

여러 컬렉션을 병렬로 순회해야 한다면 각각의 반복자와 인덱스 변수를 사용해 엄격하고 명시적으로 제어해야 한다.

아이템 59. 라이브러리를 익히고 사용하라.

아주 특별하거나 해당 프로젝트에서만 쓰이는 기능이 아니라면 누군가가 라이브러리 형태로 구현해놓았을 것이다.→바퀴를 재발명하지 말자.

- 표준라이브러리의 이점
  - 표준 라이브러리를 사용하면 그 코드를 작성한 전문가의 지식과 이것을 사용한 다른 개발자의 경험, 노하우 문서까지 활용할 수 있다
  - 핵심 비즈니스 로직 외에 들이는 시간이 줄어든다.
  - 따로 노력하지 않아도 성능이 지속적으로 개선되며 기능이 추가된다. 라이브러리 개발자들이 계속 노력하기 때문이다.
- 코드가 많은 사람들에게 낯익은 코드가 되며 유지보수성과 재활용성이 좋아진다.
  - java.lang
  - java.io
  - java.util : 컬렉션, 스트림, 동시성
- google guava : 구글에서 만든 자바 라이브러리

아이템60. 정확한 답이 필요하다면 float와 double은 피하라.

float와 double타입은 과학과 공학 계산용으로 설계되었다. 따라서 정확한 결과가 필요할 때는 사용하면 안된다.

- **그럼 대안이 있을까?**

    - BigDecimal
        - 범위가 크고 소수점을 직접 관리하지 않아도 되지만 느리다.
    - int or long
        - 사용하기가 간편하지만 범위가 좁고 소수점을 직접 관리해야한다.
- 소수점 추적을 시스템에 맡기거나 숫자가 너무 크다면 BigDecimal
- 소수점을 직접 추적할 수 있고 성능이 중요하며, 숫자가 너무 크지 않다면 기본형을 사용하자.

아이템61. 박싱된 기본 타입보다는 기본 타입을 사용하라.

기본 타입과 박싱된 기본 타입의 차이

- 기본 타입은 값만 가지고 있으나, 박싱된 기본 타입은 값에 더해 식별성이라는 속성을 갖는다
- 기본 타입의 값은 유효하나, 박싱된 기본 타입은 유효하지 않은 값(null)을 가질 수 있다
- 기본 타입이 박싱된 기본 타입보다 시간과 메모리 사용면에서 더 효율적이다

박싱된 원시타입의 문제

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

- i와 j의 값의 동등 대신 인스턴스 동치를 판단하여 false가 출력된다.
- 박싱된 기본 타입에 == 연산자를 사용하면 오류가 발생한다.
- 기본 타입을 다루는 비교자가 필요하다면 **Comparator.naturalOrder()**를 사용하라
    - 비교자를 직접 만들면 비교자 생성 메서드나 기본 타입을 받는 정적 compare 메서드를 사용해야 한 이때, Interger 매개변수의 값을 기본 타입 정수로 저장하거나 파라미터로 넘긴 후, 모든 비교를 이 기본 타입 변수로 수행한다.
- NullPointerException이 발생할 수 있다.
    - 박싱된 기본 타입 변수의 초기값은 null이다

    - 이와 같이 초기화를 시키지 않은 상태로 기본 타입 변수와 비교(==)를 시도하면 오토언박싱이 일어나고 *NullPointerException*이 발생한다
    - 기본 타입과 박싱된 기본 타입이 혼용된 연산에서는 오토언박싱과 *NullPointerException*이 발생할 수 있음을 주의해야 한다
- 성능이 저하될 수 있다.

아이템 62. 다른 타입이 적절하다면 문자열 사용을 피하라.

문자열(String) 사용에 대한 주의점: 문자열을 다른 데이터 타입의 대용으로 사용하지 마라.

- 문자열은 다양한 데이터를 표현하는 데 편리하지만, 데이터가 원래 다른 타입일 경우에는 해당 타입을 직접 사용하는 것이 좋다.
- 예를 들어, 파일, 네트워크, 사용자 입력으로부터 데이터를 받을 때, 이 데이터가 숫자나 다른 타입(수치, 객체 등)일 경우에는 해당 타입을 직접 사용하는 것이 좋다.
    - 즉, 숫자를 입력 받는 경우에는 문자열 대신 int, float, BigInteger와 같은 적절한 수치형 타입으로 변환해야 하며, 사용자 정의 객체를 입력 받는 경우에는 해당 객체의 타입으로 변환해야 한다.

문자열은 열거 타입을 대신하기에 적합하지 않다.

- **타입 안전**: 문자열이 틀렸거나 잘못 입력되는 경우를 컴파일 시에 잡아낼 수 있다. 반면 문자열은 컴파일 시 오타나 오류를 발견하기 어렵다.
- **코드 가독성**: 열거 타입을 사용하면 해당 값이 어떤 목적으로 사용되는지 명확하게 알 수 있다.
- **기능의 확장**: 열거 타입에는 메서드를 추가하거나 필드를 정의할 수 있다.

문자열은 혼합 타입을 대신하기에 적합하지 않다.

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

- 각 요소를 개별로 접근하려면 문자열을 파싱해야하기 때문에 처리시간이 증가할 수 있으며, 오류를 발생시킬 수 있다.
- 각 요소를 별도의 필드로 갖는 전용 클래스로 만들자.

문자열은 권한을 표현하기에 적합하지 않다.

```
public class ThreadLocal {
    private ThreadLocal() {} // 객체 생성 불가
    //스레드의 값을 키로 구분해 저장
    public static void set(String key, Object value);
    //키가 가리키는 현 스레드의 값을 반환한다.
    public static Object get(String key);
}
```

- 만약 다른 스레드에서 동일한 키를 사용해 set을 호출하면, 현재 스레드의 값이 덮어씌워진다.
- 이는 보안에 취약해질 수 있다.
- get,set 메서드는 정적 메서드일 필요가 없으니 key 클래스의 인스턴스 메서드로 바꾸자.

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

- get 메서드가 Object 타입을 반환하기 때문에 형변환 해서 사용해야 하므로 안전하지 않다.
- 제네릭을 도입하여 형변환으로 인한 오류를 줄여보자.

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

아이템 63.문자열 연결은 느리니 주의하라

문자열 연결 연산자(+)의 성능 문제: 문자열 연결 연산자(+)로 문자열 n개를 잇는 시간은 n^2에 비례한다.

- 문자열은 불변 클래스이므로 두 문자열을 복사해야 하므로 성능 저하가 발생한다.

```
// 문자열 연결을 잘못 사용한 예 - 느리다!
public String statement() {
    String result = "";
    for (int i = 0; i < numItems(); i++) {
        result += lineForItem(i); // 문자열 연결
    }
    return result;
}
```

변화가 잦을 경우 String 대신 StringBuilder를 사용하자.

```
// StringBuilder를 사용하면 문자열 연결 성능이 크게 개선된다.
public String statement2() {
    StringBuilder sb = new StringBuilder(numItems() * LINE_WI
    for (int i = 0; i < numItems(); i++) {
       sb.append(lineForItem(i));
    }
    return sb.toString();
}
```

아이템64. 객체는 인터페이스를 사용해 참조하라.

매개변수, 반환값, 변수, 필드를 전부 인터페이스 타입으로 선언하라.

```
// 좋은 예. 인터페이스를 타입으로 사용했다.
Set<Son> sonSet = new LinkedHashSet<>();

// 쉽게 구현 클래스를 교체할 수 있다.
Set<Son> sonSet = new HashSet<>();
```

```
// 나쁜 예. 클래스를 타입으로 사용했다.
LinkedHashSet<Son> sonSet = new LinkedHashSet<>();
```

주의해야 할 점: 원래의 클래스가 특정 메소드를 제공하고, 그 메소드를 사용하는 코드가 있다면, 새로운 클래스도 반드시 그 메소드를 제공해야 한다.

- 예시:
    - LinkedHashSet은 순서를 가지는 반면, HashSet는 순서를 가지지 않는다. 원래 LinkedHashSet을 사용할 때, 순서에 의존적인 코드를 구현했다면, HashSet을 사용하더라도 같은 기능을 제공해야 한다

적합한 인터페이스가 없다면 적합한 인터페이스가 없다면 당연히 클래스로 참조해야 한다.

1. 값 클래스

- 실제 값을 표현하는 데 중점을 두기 때문에, 주로 불변(final)하며, method보다는 데이터를 나타낸다. 또한, 적합한 인터페이스가 거의 없다.
- 따라서 값 클래스를 매개변수, 변수, 필드, 반환 타입으로 사용해도 괜찮다.

2. 클래스 기반 작성된 프레임워크

- 클래스가 프레임워크의 기본적인 구성요소로 사용된다. 또한, 적합한 인터페이스가 거의 없다.
- 따라서 클래스 타입을 직접 사용해야 할 수 있다.

3. 인터페이스에 없는 특별한 메소드를 제공하는 클래스

- 예: PriorityQueue 클래스는 Queue 인터페이스에는 없는 comparator 메소드를 제공한다.
- 이러한 메소드를 사용해야 하는 경우 클래스 타입을 직접 사용해야 한다.

적합한 인터페이스가 없다면 클래스의 계층구조 중 필요한 기능을 만족하는 가장 상위의 클래스를 타입으로 사용하자.

아이템65.리플렉션보다는 인터페이스를 사용하라.

리플렉션

- 자바에서 동적인 프로그래밍을 가능하게 해주는 강력한 기능이다.
- 프로그램이 실행 중에 자신의 구조를 동적으로 탐색하거나 조작하는 능력이다.
- 프로그램이 자신을 조사하고 수정할 수 있게 하는 기능이며, 이는 컴파일 시점에는 알 수 없는 클래스나 메소드에 대해 런타임에 동적으로 접근하는 것을 가능하게 한다.

1. 클래스의 정보를 가져올 수 있다.
    - 자바의 Class 객체를 통해 임의의 클래스에 접근할 수 있다.
    - Class 객체는 해당 클래스의 메타데이터를 나타내며, 이를 통해 클래스의 이름, 슈퍼 클래스, 인터페이스 등의 정보를 얻을 수 있다.
    - Class 객체는 Class.forName() 메소드나 .class 리터럴 등을 이용해 얻을 수 있다.
    - 프로그램 실행 중인 동안 어떤 클래스의 정보를 가져올 수 있다.
        - Class 객체에서 getConstructors(), getMethods(), getFields() 등의 메소드를 호출하면, 해당 클래스의 생성자, 메소드, 필드에 대한 정보를 Constructor[], Method[], Field[] 등의 형태로 가져올 수 있다.
        - 이 정보는 생성자, 메소드 필드 등에 대한 정보를 포함하며, 이들의 이름, 필드의 타입, 메소드의 시그니처(파라미터 타입과 반환 타입) 등을 포함한다.
2. 각각에 연결된 실제 생성자, 필드, 메소드에 접근해 조작할 수 있다.
    - 예
        - Constructor.newInstance() 메소드를 호출하면 해당 생성자를 통해 새 인스턴스를 생성할 수 있다.
        - Method 객체의 invoke() 메소드를 사용하면, 해당 Method 객체가 나타나는 메소드를 실제로 호출할 수 있다.
        - Field.set 또는 Field.get()을 통해 필드 값을 설정하거나 가져올 수도 있다.

컴파일 당시에는 존재하지 않는, 즉 런타임에 동적으로 로드되는 클래스를 이용할 수 있다

리플렉션의 단점은?

- 컴파일타임 검사가 주는 이점을 사라진다.
    - 일반적인 자바 코드는 컴파일러에 의해 타입 검사가 이루어진다.
    - 이는 프로그래머가 잘못된 타입의 객체를 메소드에 전달하거나, 존재하지 않는 메소드를 호출하려고 시도하는 경우 같은 실수를 사전에 잡아준다.
    - 그러나 리플렉션을 사용하면 이러한 타입 검사는 런타임에 이루어진다.
    - 따라서 리플렉션을 사용한 코드는 존재하지 않는 메소드를 호출하거나, 잘못된 타입의 인수를 전달하는 등의 오류를 컴파일 시점에는 감지할 수 없으며, 이는 런타임 오류를 초래할 수 있다.

- 리플렉션을 이용하면 코드가 지저분하고 장황해진다.
  - 일반적인 메소드 호출은 간결하지만, 리플렉션을 통한 메소드 호출은 해당 메소드의 이름, 파라미터 타입, 실제 인수 등을 모두 처리해야 하므로 코드가 복잡해진다.
- 성능이 떨어진다.
  - 일반적인 메소드 호출에 비해 상당히 느리다.
  - 이는 메소드를 찾고, 접근 제한자를 확인하고, 인수를 언박싱하고 다시 박싱하는 등의 작업 때문이다

리플렉션 사용시 주의할 점

리플렉션은 인스턴스 생성에만 사용해야 한다.

- 컴파일 타임에 사용할 수 없는 클래스를 사용해야만 하는 경우라도, 일반적으로 해당 클래스는 어떤 인터페이스나 상위 클래스를 구현하거나 상속받은 경우가 많다.
- 이런 경우, 리플렉션은 해당 클래스의 인스턴스를 생성하는 데에만 사용하고, 그 생성된 인스턴스는 그 클래스가 구현하거나 상속받은 인터페이스나 상위 클래스를 통해 참조하고 사용하는 것이 좋다.(item 64)

```
// 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>>) Class.forName(arg
    } catch (ClassNotFoundException e) {
        fatalError("클래스를 찾을 수 없습니다.");
    }

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

```
    // 집합의 인스턴스를 만든다.
    Set<String> s = null;
    try {
        s = cons.newInstance();
    } catch (IllegalAccessException e) {
        fatalError("생성자에 접근할 수 없습니다.");
    } catch (InstantiationException e) {
        fatalError("클래스를 인스턴스화할 수 없습니다.");
    } catch (InvocationTargetException e) {
        fatalError("생성자가 예외를 던졌습니다: " + e.getCause());
    } catch (ClassCastException e) {
        fatalError("Set을 구현하지 않은 클래스입니다.");
    }

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

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

리플렉션과 의존성 관리

리플렉션은 런타임에 다른 클래스, 메소드, 필드에 접근하게 해준다.

이 기능은 특히 런타임에 존재하지 않을 수 있는 코드와의 의존성을 관리해야 할 때 유용하다.

예를 들어, 프로그램이 특정 라이브러리의 최신 기능을 이용하려 할 때, 이 최신 기능이 오래된 버전에서는 존재하지 않을 수 있다. 이런 경우 리플렉션을 이용하여 최신 기능이 존재하는지 런타임에 확인하고, 존재한다면 이를 이용하게 된다.

주의해야 할 점은 런타임에 존재하지 않는 코드에 접근하는 것이 실패할 때 대체 방법을 준비해야 한다.

아이템66. 네이티브 메서드는 신중히 사용하라.

네이티브 메서드는 어디에 사용될까?

1. 레지스트리 같은 플랫폼 특화 기능
    - 특정 플랫폼에만 특화된 기능을 사용하기 위해 네이티브 메소드를 사용할 수 있다.
        - ex. 윈도우에만 제공하는 레지스트리 기능을 자바에서 사용하고자 하는 경우 네이티브 메소드를 사용할 수 있다.
    - 하지만 자바가 발전하면서 플랫폼 특화 기능들을 자체적으로 제공하는 경우가 많아졌다.
        - ex. 자바 9 이상부터는 OS 프로세스에 접근하는 API를 제공한다.
2. 네이티브 코드로 작성된 기존 라이브러리 사용
    - 기존 라이브러리를 재사용하기 위해서는 네이티브 메소드를 사용해야 한다.
    - ex. 레거시 데이터를 사용하는 레거시 라이브러리
3. 성능 개선의 목적으로 성능에 영향을 주는 부분만 사용
    - 특정 연산이 자바로 작성한 것보다 네이티브 언어로 작성한 경우 훨씬 좋다면, 그 부분을 네이티브 메소드로 작성하여 성능을 개선할 수 있다.
    - **현재는 성능을 개선할 목적으로 네이티브 메소드를 사용하는 것은 거의 권장하지 않는다.**

네이티브 메서드의 단점

- 네이티브 언어는 메모리 훼손 오류로부터 안전하지 않다.
    - 자바는 메모리 관리를 자동으로 처리해주는 언어이므로, 개발자가 직접 메모리를 관리할 필요가 없다.
    - 그러나 네이티브 언어를 사용하면 개발자가 직접 메모리를 할당하고 해제해야 한다.
    - 이 과정에서 메모리 할당이나 해제를 잘못 처리하면 메모리 훼손 오류가 발생할 수 있다.
- 자바보다 플랫폼에 의존적이므로 이식성이 낮다.

    - 자바 코드는 다양한 플랫폼엣 동작시킬 수 있으나, 네이티브 언어로 작성된 코드는 특정 운영체제나 하드웨어에 의존적이므로, 자바 코드보다 이식성이 낮다.
- GC는 네이티브 메모리를 자동 회수하지 못하기 때문에 추적할 수 없다.
- 성능, 비용, 디버깅 측면에서 모두 좋지 않다.
    - 자바와 네이티브 코드의 전환은 비용이 많이 들며, 이로 인해 성능이 저하될 수 있다.

아이템 67. 최적화는 신중히 하라.

빠른 프로그램보다는 좋은 프로그램을 작성하라.: 아키텍처 수준에서의 성능 최적화를 위해 가독성이 좋고, 재사용성이 높고 수정이 용이한 코드를 작성하라.

- 프로그램의 아키텍처가 잘 설계되어 있으면, 개별 구성 요소의 성능 문제는 대체로 구성 요소 내부의 개별적인 최적화를 통해 해결할 수 있다.
- 즉, 정보 은닉 또는 캡슐화 원칙을 따르므로 각 요소를 독립적으로 설계하고 개선할 수 있게 만든다.
- 반면, 성능 문제가 아키텍처 수준에서 발생하는 경우, 일반적으로 전체 시스템을 변경해야 하며 이는 대규모의 재설계와 많은 리소스가 필요하다.
- 결론적으로, 좋은 프로그램을 작성하는 것이 중요하며, 이를 통해 **개별 구성 요소를 독립적으로 최적화할 수 있는 유연성을 얻을 수 있다.**
- 또한, **아키텍처 수준에서의 성능 최적화가 중요**하며, 이는 프로그램 설계 초기 단계부터 고려되어야 한다.

성능을 제한하는 설계를 피하라.

- 성능에 큰 영향을 미치는 설계 요소들은 신중하게 결정해야 한다.
- 특히 컴포넌트 간의 소통 방식이나 외부 시스템과의 연결 방식 같은 핵심적인 설계 요소들은 한 번 결정되면 추후에 변경하기 어렵기 때문에 초기 설계 단계에서 성능을 고려하여 결정해야 한다.
- API, 네트워크 프로토콜, 영구 저장용 데이터 포맷 등의 설계 요소들은 시스템의 성능에 큰 영향을 미치며, 변경하기 어렵기 때문에 초기 설계 단계에서 신중히 결정해야 한다.

API를 설계할 때 성능에 주는 영향을 고려하라.

- public 타입을 가변으로 만들면 불필요한 방어적 복사를 유발할 수 있다.(item 50)

    - 메모리 사용이 증가하고, 프로그램 성능이 저하될 수 있다.
    - 따라서 가능하면 불변 객체를 사용하여 이러한 문제를 방지하고, 필요한 경우에만 가변 객체를 사용하자.
- 컴포지션으로 해결할 수 있는 경우에도 상속을 이용하면 상위 클래스에 영원히 종속되고, 성능에 제약까지 물려받는다.(item 18)
    - 상속은 코드의 재사용성을 높이지만, 상위 클래스의 특성을 물려받아 변경이 어렵다.
    - 반면, 컴포지션을 이용하면 필요한 부분만을 선택하여 클래스를 구성할 수 있어, 유연성이 높아지고 성능 최적화에 유리하다.
- 굳이 인터페이스가 아닌 구현 타입을 사용하면, 차후 개선된 구현체를 사용하기 어려워진다.
    - 클래스보다 인터페이스를 이용하면 코드의 유연성을 높이고, 성능 최적화에 좋다.

```
public abstract class Component implements ImageObserver, Men
                                           Serializable
{
    public Dimension getSize() {
        return size();
    }

    @Deprecated
    public Dimension size() {
        return new Dimension(width, height); // 방어적 복사 수행
    }
}
```

getSize 메소드는 Dimension 클래스의 새 인스턴스를 반환한다. Dimension 클래스는 가변으로 설계되어, getSize를 호출할 때마다 새로운 Dimension 인스턴스가 생성된다. 이는 성능 저하를 유발한다.

```
public abstract class Component implements ImageObserver, Men
    public int getWidth() {
        return width;
    }

    public int getHeight() {
```

```
        return height;
    }
}
```

- 따라서, 자바 2에서는 getWidth와 getHeigth 메소드를 추가하여 성능 문제를 개선하려 하였다.
- 이 두 메소드는 각각 너비와 높이의 기본 타입을 반환하므로, getSize를 호출할 때마다 새로운 Dimension 객체를 생성하는 것보다 훨씬 효율적이다.

성능을 위해 API를 왜곡하지 마라.

- API 왜곡은 장기적으로 API의 사용성을 저하시키고 유지보수를 어렵게 만든다.
- 기술의 발전으로 성능 문제는 자연스럽게 해결될 수 있지만, 일단 왜곡된 API는 수정이 어렵고 관리의 어려움이 지속될 수 있다.

각각의 최적화 시도 전후로 성능을 측정하라.: 프로파일링 도구를 사용하자.

- 프로파일링 도구는 성능 최적화에서 굉장히 중요한 역할을 한다.
- 코드에서 가장 많은 시간을 소비하거나 가장 빈번하게 호출되는 부분을 식별하는 데 도움을 줘, 개발자는 성능 최적화 노력을 어디에 집중해야 할지 결정할 수 있다.
- 또한, 개별 메소드의 소비 시간과 호출 횟수 같은 런타임 정보를 제공한다.
    - 이정보를 통해
        - 개발자는 성능 문제가 있는 지점을 쉽게 찾을 수 있다.
        - 알고리즘이 적절하게 선택되었는지 확인할 수 있다.
- JMH(Java Microbenchmark Harness)는 프로파일링 도구가 아니다.
    - 그러나 자바 코드의 성능을 상세하게 측정하고 분석하기 위한 도구로, 마이크로 벤치마킹 프레임워크다.
    - 이를 사용하면 코드의 특정 부분이나 작은 단위의 성능을 측정하고, 여러 구현 간의 성능을 비교할 수 있다.
- 자바의 성능 모델은 C/C++보다 덜 정교하다.
    - 이는 자바의 고수준 추상화 때문으로, 프로그래머가 작성하는 코드와 CPU에서 수행하는 명령 사이에 추상화 격차가 있다.
    - 이로 인해 최적화의 효과를 일정하게 예측하기 어렵다.
    - 이런 문제를 해결하기 위해서는 프로파일링 도구와 같은 도구를 사용하여 실제 실행 시간을 측정하고 분석하는 것이 필요하다.

아이템 68. 일반적으로 통용되는 명명규칙을 따르라.

자바 언어의 명명 규칙

- 크게 철자와 문법 두 가지로 나뉜다.

철자 규칙

- 특별한 이유가 없는 한 반드시 따라야 한다.
- 지키지 않을 경우 API는 사용하기 어렵고, 유지보수하기 어렵다.
- 가독성이 떨어진다.

패키지와 모듈 이름

- 각 요소를 점(.)으로 구분하여 계층적으로 짓는다.
- 조직 외부에서도 사용될 수 있다면, 조직의 웹 도메인 이름을 역순으로 사용한다.
    - 웹 도메인 이름은 전세계적으로 유일하므로 패키지 이름의 충돌을 피할 수 있다.
    - ex. com.google
    - 예외적으로 표준 라이브러리와 선택적 패키지들은 java와 javax로 시작한다.
- 각 요소는 8자 이하의 짧은 단어 혹은 약어를 추천한다.
    - ex. utilities보다는 util처럼 의미가 통하는 약어를 사용하는 것도 좋다.
    - ex. 여러 단어로 구성된 이름이라면 awt처럼 첫글자를 사용하는 것도 좋다.

클래스와 인터페이스 이름

- 하나 이상의 단어로 이루어지며, 각 단어는 대문자로 시작한다.
    - ex. List, FutherTask
- 단어의 첫글자만 딴 약자나 max, min처럼 널리 통용되는 줄임말을 제외하고는 단어를 줄여 쓰면 안 된다.

메소드와 필드 이름

- 첫 글자를 소문자로 쓴다는 점만 빼면 클래스 명명 규칙과 같다.
    - ex. remove, ensureCapacity 등
- 단 상수 필드는 예외다.

    - 모두 대문자로 쓰며 단어 사이는 밑줄로 구분한다.
    - ex. VALUES, NEGATIVE_INFINITY 등

더보기

- 지역 변수의 경우, 비교적 유추하기 쉬우므로 약어를 사용해도 좋다.
    - ex. i, denom, houseNum 등

타입 매개변수

- 입력 매개변수도 지역변수이지만, 문서에 등장하는 만큼 지역변수보다는 신경써야 한다.
- T: 임의의 타입
- E: 원소의 타입
- K와 V: 맵의 키와 값
- X: 예외
- R: 메소드 반환 타입
- 임의 타입의 시퀀스: T, U, V 혹은 T1, T2, T3

문법 규칙

- 클래스
    - 객체를 생성할 수 있는 클래스(열거 타입 포함)
        - 보통 단수 명사나 명사구를 사용한다.
        - ex. Thread, PriorityQueue, ChessPiece 등
- 객체를 생성할 수 없는 클래스
    - 보통 복수형 명사를 사용한다.
    - Collectors, Collections 등
- 인터페이스
    - 클래스와 동일하게 사용
    - able 혹은 ible로 끝나는 형용사 사용
        - ex. Runnable, Iterable, Accessible

메소드

- 어떤 동작을 수행하는 메소드

    - 동사나 동사구로 짓는다.
    - ex. append, drawImage
- boolean 값을 반환하는 메소드
    - is, has로 시작
    - 명사나 명사구 혹은 형용사로 기능하는 아무 단어나 구로 끝나도록 짓는다.
    - ex. isDigit, isProbablePrime, isEmpty, isEnabled, hasSiblings 등
- 반환 타입이 boolean이 아니거나 해당 인스턴스의 속성을 반환하는 메소드
    - 명사나 명사구 혹은 get으로 시작하는 동사구로 짓는다.
    - ex. size, hashCode, getTime 등
- 객체의 타입을 바꿔서, 다른 타입의 또 다른 객체를 반환하는 인스턴스 메소드
    - 보통 toType 형태로 짓는다.
    - ex. toString, toArray 등
- 객체의 내용을 다른 뷰로 보여주는 메소드
    - asType 형태로 짓는다.
    - ex. asList 등
- 객체의 값을 기본 타입 값으로 반환하는 메소드
    - 보통 typeValue 형태로 짓는다.
    - ex. intValue 등
- 정적 팩터리
    - 다양하지만 from, of, valueOf, instance, getInstance, newInstance, getType, newType을 흔히 사용한다.

필드 이름

- 클래스, 인터페이스, 메소드 이름에 비해 덜 명확하고 덜 중요하다.
- API 설계를 잘 했다면 필드가 직접 노출될 일이 거의 없기 때문이다.
- boolean 타입의 필드
    - 보통 boolean 접근자 메소드에서 앞 단어를 뺀 형태다.
    - ex. initialized, composite 등
- 다른 타입의 필드

    - 명사나 명사구를 사용한다.
    - ex. height, digits, bodyStyle 등
- 지역변수
    - 필드와 비슷하나 더 느슨하다.