# Google Java Style Guide

## 1. 소개

이 문서는 Java 소스코드를 위한 Google의 코딩 표준을 **완전히** 기술한다. Java 소스코드가 이 문서의 규칙을 준수한 경우에만 *Google Style*으로 작성되었다고 말할 수 있다.

다른 프로그래밍 스타일 가이드처럼, 이 문서는 포맷의 미적 문제뿐 아니라 다른 유형의 관습(convention)이나 코딩 표준에 대해서도 다루고 있다. 그러나 이 문서는 주로 당신이 보편적으로 따르는 **엄격한 규칙**(hard-and-fast rules)에 초점이 맞춰져 있으며, (사람에 의해서든 툴에 의해서든)명확히 시행할 수 없는 조언을 주는 것을 지양한다.

### 1.1 용어 정리

이 문서 내에서 따로 언급하지 않는 한,

1. 클래스(class)라는 용어는 일반적인 클래스, enum 클래스, 인터페이스, 또는 어노테이션 타입(`@interface`)을 의미하는데 포괄적으로 사용된다.
2. (클래스의 )멤버(member)는 중첩 클래스, 필드, 메소드, 생성자를 의미하는데 포괄적으로 사용된다. 즉, 이니셜라이저와 주석을 제외한 모든 최상위 컨텐츠를 의미한다.
3. 주석(comment)이라는 용어는 항상 코드내의 주석을 의미한다. 우리는 "Javadoc"이라는 일반적인 용어 대신 "문서 주석(documentation comments)"이라는 용어를 사용하지 않는다.

이외의 "용어 정리"는 문서 중간에 종종 기술되어있다.

### 1.2 가이드 노트

이 문서 내 모든 예제 코드는 **비 규범적**(non-normative)이다. 즉, Google 스타일을 보여주는 예제가 그 코드를 작성하는 최적의 방법이라고 할 수 없다. 예시에서 쓰인 선택적인 포맷을 규칙으로 적용해서는 안된다.

## 2. 소스 파일 기본

### 2.1 파일 이름

소스 파일 이름은 그 파일의 최상위 클래스와 같은 이름에 `.java`확장자를 가지며 대소문자를 구분한다.

### 2.2 파일 인코딩 : UTF-8

소스 파일은 **UTF-8**로 인코딩한다.

### 2.3 특수 문자

### 2.3.1 공백 문자

줄바꿈을 제외하고, 소스파일의 공백문자는 오직 **ASCII horizontal space character** (**0x20**) 뿐이다. 이것은 다음을 의미한다.

1. 문자열(string)과 문자 리터럴(character literals)의 모든 다른 공백문자는 이스케이프 처리한다.
2. 들여쓰기에 **Tab**을 사용하지 않는다.

### 2.3.2 특수한 이스케이프 시퀀스

특수한 이스케이프 시퀀스(`\b`, `\t`, `\n`, `\f`, `\r`, `\"`, `\'` and `\\`)를 가진 문자는 8진수(e.g. `\012`) 또는 유니코드(e.g. `\u000a`) 이스케이프 태신 해당 이스케이프를 사용한다.

### 2.3.3 Non-ASCII 문자

ASCII문자가 아닐 경우 실제 유니코드(e.g. `∞`) 또는 대응하는 유니코드 이스케이프(e.g. `\u221e`)를 사용한다. 문자열 리터럴과 주석에서 유니코드를 이스케이프 처라하는것은 권장하지 않으나, 선택은 전적으로 코드를 더 읽기 쉽고 이해하기 쉽게 만드는 데에 달려있다.

> 팁: 유니코드 이스케이프의 경우 실제 유니코드 문자를 사용할 때 주석으로 설명하는 것이 도움이 된다.

| 예시 | 설명 |
| --- | --- |
| `String unitAbbrev = "μs";` | 가장 좋음: 주석 없이도 명확함. |
| `String unitAbbrev = "\u03bcs"; // "μs"` | 허용되나 사용할 이유가 없음. |
| `String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"` | 허용되나 이상하고 실수하기 쉬움. |
| `String unitAbbrev = "\u03bcs";` | 나쁨: 읽는 사람이 무슨 의미인지 알 수 없음. |
| `return '\ufeff' + content; // byte order mark` | 좋음: 출력되지 않을 수 있는 문자를 위해 이스케이프 처리하고, 필요에따라 주석을 작성함. |

> 팁: 몇몇 프로그램이 non-ASCII 문자를 처리하지 못 할까 두려워 읽기 어려운 코드를 작성하지 말것. 만약 그런 일이 발생한다면 프로그램은 멈출(broken)것이며 반드시 수정되어야 한다.

## 3. 소스 파일 구조

소스 파일은 **다음 순서**로 구성된다.

1. 라이센스 또는 저작권 정보(존재한다면)
2. Package 구문
3. import 구문
4. 단 하나의 최상위 클래스

존재하는 각각의 영역은 **딱 한 줄의 공백**으로 구분된다.

## 3.1 라이센스, 저작권 정보(존재한다면)

만약 파일에 라이센스 및 저작권 정보가 들어간다면 이곳에 적는다.

## 3.2 Package 구문

패키지 구문은 줄바꿈하지 않는다. 컬럼수 제한이 적용되지 않는다.(4.4 컬럼수 제한: 100)

## 3.3 Import 구문

### 3.3.1 Wildcard imports 금지

와일드카드 import, static 어떤 것이든 사용하지 않는다.

### 3.3.2 줄 바꿈 금지

import구문은 줄바꿈 하지 않는다. 컬럼수 제한이 적용되지 않는다.(4.4 컬럼수 제한: 100)

### 3.3.3 순서 및 간격

Import는 다름 순서를 따른다.

1. 하나의 블록에 모든 static import
2. 하나의 블록에 모든 non-static import

만약 static과 non-static import가 둘 다 있다면, 두개의 블럭 사이에 빈 한줄로 구분한다. import문 사이에 다른 빈줄은 만들지 않는다. 블록 내 각 import문은 ASCII문자 순서로 정렬한다.(**Note**: 이는 ';'앞에 '.'가 위치하기 때문에, import구분이 ASCII 정렬 순서와 같지 않다.)

### 3.3.4 클래스 static import 금지

static 중첩 클래스를 위해 static import를 사용하지 않는다. 일반적은 import를 사용하여 가져온다.

## 3.4 .클래스 선언

### 3.4.1 최상위 클래스는 하나만 선언

각 최상위 클래스는 그 파일에 하나만 존재한다.

### 3.4.2 클래스 컨텐츠 순서

멤버와 클래스 이니셜라이저의 순서는 학습용이성(learnability)에 큰 영향을 줄 수 있다. 하지만, 이것에 정해진 정답은 없다. 각 클래스는 서로 다른 방식으로 순서가 배열될 수 있다.

각 클래스에서 중요한 것은 코드 관리자가 설명할 수 있는 논리적인 순서를 사용하는 것이다. 에를들어, 새로운 메소드를 클래스의 맨 뒤에 추가하는 것은 "추가된 순서를 연대순으로" 나타내는 것이기 때문에 논리적인 순서가 아니다.

#### 3.4.2.1 오버로딩(Overload)은 분리하지 않음

클래스가 같은 이름의 생성자나 메소드를 여러개 가진 경우 사이에 다른 코드(private 멤버 포함) 없이 순차적으로 작성한다.

# 4 Formatting

**용어 정리**: *블록같은 구조*는 클래스, 메소드, 생성자의 본문을 나타낸다. 배열 이니셜라이저는 블록과 같은 구조로 간주할 수 있다.

## 4.1 괄호

### 4.1.1 선택사항으로 사용되는 괄호

괄호는 `if`, `else`, `for`, `do`, `while` 문의 본문이 없거나 한 줄이더라도 사용된다.

### 4.1.2 비어있지 않는 블록: K & R 스타일

괄호는 비어있지 않은 블록이나 블록 같은 구조일 경우 the Kernighan and Ritchie style ("Egyptian brackets")를 따른다.

- 여는 괄호 전에 줄바꿈 하지 않는다.
- 여는 괄호 뒤에 줄바꿈 한다.
- 닫느 괄호 전에 줄바꿈 한다.
- 메소드, 생성자, 클래스의 닫는 괄호일 경우에 뒤에 줄바꿈한다. 예를 들어, `else`나 ';'뒤에는 줄바꿈 하지 않는다.

예시:

```
return () -> {
  while (condition()) {
    method();
  }
};

return new MyClass() {
  @Override public void method() {
    if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

enum클래스의 몇 가지 예외는 4.8.1. Enum classes에서 소개한다.

### 4.1.3 비어있는 블록: 줄일 수 있음

빈 블록이나 비어있는 구조는 (4.1.2에서 서술한 대로) K & R 스타일을 적용할 수 있다. 다중 블록 구문 (`if/else` or `try/catch/finally`)이 아닌 경우, 사이에 어떤 문자나 줄바꿈이 없다면 `{}` 처럼 바로 닫아도 된다.

예시:

```
// This is acceptable
void doNothing() {}

// This is equally acceptable
void doNothingElse() {
}
```

```
// This is not acceptable: No concise empty blocks in a multi-block statement
try {
  doSomething();
} catch (Exception e) {}
```

## 4.2 블록 들여쓰기: 스페이스 2칸

새 블록이나 블록 같은 구분이 열릴 때 마다, 들여쓰기가 2칸 증가한다. 블록이 끝날 때, 들여쓰기가 이전 들여쓰기 레벨로 돌아간다. 들여쓰기 레벨은 코드와 주석 둘 다 적용된다.(4.1.2 비어있지 않는 블록: K & R 스타일의 예시를 참고하라)

## 4.3 한 줄에 한 구문

각 구문 마지막에 줄바꿈한다.

## 4.4 컬럼수 제한:100

Java 코드는 100자 컬럼수 제한을 갖는다. 이때, 문자는 모든 유니코드 포인트를 의미한다. 아래의 예외를 제외하고 4.5 줄바꿈에 나온대로 줄바꿈한다.

> 각 유니코드 포인트는 화면에서 크던 작던 하나의 문자로 취급한다. 에를 들어, 전각 문자 (fullwidth characters)를 사용한다면 이 규칙에서 요구하는 것 보다 일찍 줄바꿈 할 수 있다.

**예외:**

1. 컬럼 글자수 제한을 준수할 수 없는 경우(예를 들어, Javadoc의 긴 URL이나 긴 JSNI 메소드 레퍼런스)
2. `package` 또는 `import` 구문(3.2 Package 구문과 3.3 Import 구문을 참고하라)
3. shell에서 붙여넣기 할 수 있도록 주석화 한 경우

## 4.5 줄바꿈

**용어 정리:** 한 줄로 작성할 수 있는 코드를 여러 줄로 분할하는 행위를 줄 바꿈(*line-wrapping*)이라 한다.

모든 상황에 정확히 맞는 포괄적이고 결정론 적인 규칙은 존재하지 않는다. 몇 가지 상황에 적절한 방법이 있을 뿐이다.

> 참고: 컬럼 글자 수 제한 때문에 줄바꿈을 하는 것이 일반적이지만, 제한 수를 넘지 않더라도 작성자의 판단에 의해 줄바꿈될 수 있다.

> 팁: 메소드나 지역변수를 추출하여 줄바꿈 하지 않도록 할 수 있다.

### 4.5.1 언제 바꿀까

줄바꿈의 주요 지시어는 다음과 같다(높은 구문 레벨에서 줄바꿈하길 선호됨)

1. 비 대입 연산자에서 컬럼이 넘어서면 심볼 앞에서 줄바꿈한다.(이는 구글의 C++이나 Javascript같은 다른 언어의 스타일과 같지 않다.)
    - 이는 연산자 형 심볼에도 적용된다.
        - the dot separator(`.`)
        - the two ':(colons)' of a method reference (`::`)
        - an '&(ampersand)' in a type bound (`<T extends Foo & Bar>`)
        - a pipe in a catch block (`catch (FooException | BarException e)`).
2. 대입 연산자에서 컬럼이 넘어가면 일반적으로 심볼 뒤에 줄바꿈 한다. 하지만, 다른 방식도 허용한다.
    - 이는 또한 할당 연산자형 심볼인 `for`("foreach")의 콜론에도 적용된다.
3. 메소드나 생성자의 이름 뒤에 열린 괄호(`(`)는 줄바꿈하지 않는다.
4. 콤마(`,`)는 앞의 항목에 붙여 쓴다.
5. 람다에 인접한 화살표는 람다의 내용이 괄호없는 한줄로 구성된 경우를 제외하고 줄바꿈하지 않는다.

```
MyLambda<String, Long, Object> lambda =
    (String label, Long value, Object obj) -> {
        ...
    };

Predicate<String> predicate = str ->
    longExpressionInvolving(str);
```

> 참고: 줄바꿈의 기본 목표는 코드를 명확하게 하는 것다. 라인 수를 줄이는 것은 필요치 않다.

### 4.5.2 연속된 들여쓰기는 적어도 스페이스 4칸 이상

줄바꿈 할 때, 연속된 라인은 원래 줄에서 4칸 이상 들여쓰기 한다.

연속된 라인이 여러 줄일 경우 4칸 이상 원하는 만큼 들여쓰기 한다. 일반적으로 두 라인 이상 분법적으로 수평적인 항목으로 시작한다면 동일한 들여쓰기 레벨을 적용한다.

4.6.3 수평 정렬 : 필요하지 않음에서 이전 행과 어떤 토큰을 정렬하기 위해 스페이스를 다양하게 쓰지 않는 것이 좋음을 보여준다.

# 4.6 공백(Whitespace)

## 4.6.1 수직 공백

빈 줄은 다음과 같이 사용된다. (즉, 다음의 경우 항상 한 줄씩 띄운다.)

1. 클래스의 연속 멤버(또는 이니셜라이저) 사이 : 필드, 생성자, 메소드, 중첩된 클래스, 정적 이니셜라이저, 인스턴스 이니셜라이저.
    - 예외 : 두 개의 연속 필드 사이의 빈 줄(그 사이에 다른 코드는 없음)은 선택사항이다. 이러한 빈 줄은 필드의 논리적 그룹을 만들 때 필요한 만큼 사용된다.
    - 예외 : Enum 상수 사이의 빈 줄은 Section 4.8.1에서 다룬다.
2. 다른 Section에서 요구하는 대로 (Section 3, Source file structure 와 Section 3.3, Import statements)

코드의 논리적인 하위 섹션을 구성하기 위해서, 코드의 가독성을 높이기 위해 어디서나 한 줄씩 띄울 수 있다. 첫 번째 멤버 또는 이니셜라이저 이전 또는 마지막 멤버 또는 클래스의 이니셜라이저 이후의 빈 행은 필수가 아니다.

연속된 여러개의 빈 줄은 허용하지만 절대 권장하지는 않는다.

## 4.6.2 수평 공백

언어 또는 다른 스타일 규칙에 필요한 것을 넘어서 문자, 주석 및 Java doc을 제외하고, 하나의 ASCII 스페이스는 다음 위치에만 나타난다.

1. `if`, `for`, `catch` 등 과 같은 예약어를 구분하기 위해 그 다음에 오는 여는 괄호 ( `(`)사이에 공백 사용
2. `else`, `catch` 등 예약어를 구분하기 위해 그 이전에 오는 앞에 닫는 중괄호( `}` ) 사이에 공백 사용
3. 여는 중괄호( `{` ) 앞에는 두 가지 예외가 있음, 이를 제외하고 여는 중괄호앞은 공백 사용
    - `@someAnnotation({a, b})` (공백 없음)
    - `String[][] x = {{"foo"}};` (아래 항목 8에 따라 `{{` 사이에 공백이 필요 없음)
4. 이항 또는 삼항 연산자의 양쪽에 위치한다. 이는 다음과 같은 "연산자 형" 기호에도 적용된다:
    - 논리곱 타입의 앤퍼센트 : `<T extends Foo & Bar>`
    - 여러 예외를 처리하는 catch 블록의 파이프 : `catch (FooException | BarException e)`
    - `for`("foreach")문 안에서의 콜론( `:` )
    - 람다 표현 : `(String str) -> str.length()`

      다음 경우에는 공백을 사용하지 않는다.
    - `Object::toString` 과 같이 쓰여진 메소드 참조의 두 가지 콜론( `::` )
    - `Object.toString()` 과 같이 쓰여진 점 구분 문자( `.` )
5. `,:;` 의 뒤, 형변환을 위한 닫는 괄호( `)` ) 의 뒤는 공백 사용
6. 코드 끝에 오는 주석의 시작인 더블 슬래시( `//` ) 좌우로 한 칸씩 띄운다. 여러칸도 허용하지만 필수는 아님.
7. 변수 선언시 타입과 변수명 사이에 공백 사용 `List<String> list`
8. 배열 이니셜라이저에서 양쪽 괄호 안쪽
    - `new int[] {5, 6}` 과 `new int[] { 5, 6 }` 모두 허용
9. 타입 어노테이션 `[]` 와 `...` 사이는 공백 사용

위의 규칙은 라인의 시작 또는 끝의 추가 공백에는 적용하지 않고 코드 *내부* 공간에만 적용된다.

### 4.6.3 수평 정렬 : 필요하지 않음

**용어 노트:** *수평정렬*은 특정 토큰을 이전 행의 다른 토큰과 줄 맞추기 위해 임의의 추가 스페이스르 입력하는 것을 말한다.

이것은 허용되나 구글 스타일에서는 **필요하지 않으며** 이미 사용하고 있더라도 *유지*할 필요가 없다.

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

> **팁:** 정렬이 가독성을 높이긴 하지만 향후 유지보수에 문제를 일으킨다. 한 줄만 건드리게 되면 잘 맞춰놓은 포맷은 엉망이 되며 정렬을 맞추기 위해 근처 라인의 연속적인 수정이 발생할 수 있다. 한 줄 변경이 "blast radius"가 되어 최악의 경우에는 버전 기록 정보가 손상되고 리뷰에 시간이 걸리게 되고 병합 충돌이 발생할 수있다.

## 4.7 그룹화를 위한 괄호 : 권장

선택적 그룹화 괄호는 작성자와 리뷰자가 코드가 잘못 해석될 가능성이 없고 더이상 코드를 읽기 쉽게 만들 가능성이 없다는데 동의할 때에만 생략가능하다. 모든 독자들이 자바 연산자 우선 순위표를 암기한다고 가정하는 것은 합리적이지 않다.

## 4.8 특수 구조

### 4.8.1 Enum classes

각 enum 상수 뒤의 콤마 뒤의 줄 바꿈은 선택사항이다. 추가적인 빈 줄(일반적으로 한 줄)도 허용한다.

```
private enum Answer {
  YES {
    @Override public String toString() {
      return "yes";
    }
  },

  NO,
  MAYBE
}
```

메소드가 없고 도큐먼트도 없는 enum 클래스는 배열 이니셜라이저처럼 작성할 수 있다. (Section 4.8.3.1)

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

enum 클래스도 *클래스* 이므로 모든 다른 규칙은 클래스의 형식을 적용한다.

## 4.8.2 변수 선언

#### 4.8.2.1 한 번에 하나의 변수 선언

모든 변수 선언(필드, 로컬)은 오직 하나의 변수만을 선언한다 : `int a, b;` 와 같이 사용하지 않는다.

**예외**: `for` 루프의 헤더에서는 여러 변수 선언을 허용한다.

#### 4.8.2.2 필요할 때 선언

지역 변수는 습관적으로 블록 또는 블록과 유사한 구조를 포함하는 시작에 선언되지 않는다. 대신에 지역 변수는 변수의 스코프를 최소화하기 위해 처음 사용되는 지점(이유 내에서)에 가깝게 선언된다. 지역 변수는 일반적으로 선언후에 즉시 이니셜라이저를 가지거나 초기화된다.

## 4.8.3 배열(Arrays)

#### 4.8.3.1 배열 이니셜라이저: 블록 처럼 가능

배열 이니셜라이저는 선택적으로 "블록 같은 구성"인 것처럼 작성할 수 있다. 예를 들어, 다음 사항이 모두 허용된다. (전체 목록이 아님) :

```
new int[] {           new int[] {
  0, 1, 2, 3            0,
}                       1,
                        2,
new int[] {             3,
  0, 1,                 }
  2, 3
}                       new int[]
                            {0, 1, 2, 3}
```

#### 4.8.3.2 C-Style 배열 선언 안됨

대괄호는 변수가 아닌 *타입*의 일부분이다 : `String[] args` 이지 `String args[]` 은 아니다.

## 4.8.3 Switch 문

**용어 노트:** *switch 블록*의 중괄호 안에는 하나 이상의 *statement group*이 있다. 각 statement group은 하나 이상의 *switch label*( `case FOO:` 이나 `default:` )로 구성되어 있고 그에 따르는 구문을 가진다.

#### 4.8.4.1 들여쓰기 (Indentation)

다른 블록과 마찬가지로 switch 블록의 내용은 +2로 들여쓰기한다.

switch label이후, line break를 한다. 정확히 블록 시작처럼 들여쓰기를 +2한다.  다음 switch label은 블록이 닫힌것 처럼 이전 들여쓰기 레벨로 돌아온다.

#### 4.8.4.2 Fall-through : 주석

switch  블록 내에서, 각 statement group은 갑자기 종료되거나(`break`, `continue`, `return` 또는 thrown exception와 함께), 실행이 다음 statement group으로 계속되는 것을 나타내는 설명으로 표시한다. fall through 아이디어를 전달하는 모든 주석은 충분하다. (일반적으로 `// fall through`) 마지막 statement group은 이 주석을 달 필요가 없다.

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

`case 1:` 의 경우처럼 구문이 없는 경우에는 주석이 필요없다.

#### 4.8.4.3 `default` case는 존재한다.

코드가 없더라도 `default` statement group은 존재해야 한다.

**예외:** Enum 타입의 switch 문인 경우 생략 할 수 있다. 만약 모든 가능한 타입의 값을 커버하는 명시적인 케이스를 포함하지 않을 경우, IDEs나 다른 정적 분석 툴에서 경고를 나타낼 수 있다.

### 4.8.5 어노테이션

 클래스, 메소드, 생성자에 적용한 어노테이션은 documentation 블록 다음에 즉시 나타낸다. 각 어노테이션은 한줄에 하나씩 작성된다. 어노테이션은 잘 바꿈에 포함되지 않는다. (Section 4.5, Line-wrapping), 즉 들여쓰기 레벨은 증가되지 않는다.

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

**예외**: 파라미터가 없는 어노테이션은 한 줄에 같이 나타낼 수 있다.

```
@Override public int hashCode() { ... }
```

필드에 적용된 어노테이션은  documentation 블록 다음에 즉시 나타내는데, 이 경우에는 여러 어노테이션들을 같은 라인에 작성했다.

```
@Partial @Mock DataLoader loader;
```

### 4.8.6 주석

이 섹션은 *implementation comment*를 다룬다. Javadoc은 Section 7에 있다.

임의의 줄 바꿈은 임의의 공백 뒤에 주석이 뒤따를 수 있다. 그런 주석은 그 선을 명백하게 만든다.

#### 4.8.6.1 블록 주석 스타일

블록 주석은 주변 코드와 동일한 들여쓰기 레벨을 가진다. `/* ... */` 스타일, `// ...` 스타일이다. 멀티라인을 위한 `/* ... */` 주석의 경우, 이후 라인은 `*`로 시작해야 하며 이전행의 `*`과 줄맞춤 한다.

```
/*
 * This is          // And so           /* Or you can
 * okay.            // is this.          * even do this. */
 */
```

주석은 asterisk(*)나 그외 다른 문자로 만든 박스로 감싸지 않는다.

> **팁:** 멀티라인 주석을 작성할때 자동 코드 포매터가 필요하다면 `/* ... */` 스타일을 사용한다. 대부분의 포매터는 `// ...` 스타일의 주석 블록을 자동으로 정렬해주지 않는다.

### 4.8.7 제어자(Modifiers)

클래스나 맴버 제어자를 나타낼 때 Java Language Specification에서 추천하는 순서대로 표기한다.

```
public protected private abstract default static final transient volatile
synchronized native strictfp
```

### 4.8.8 숫자 리터럴(Numeric Literals)

`long` 값 숫자 리터럴은 대문자 `L`접미사를 사용한다. 절대 소문자를 사용하지 않는다. (숫자 1과의 혼동을 피하기 위해서)

예시, `3000000000L` rather than `3000000000l`.

# 5. Naming

## 5.1 모든 식별자에 대한 공통 규칙

식별자는 오직 ASCII 문자와 숫자만 사용하고 아래에서 언급하는 소수의 경우에서는 밑줄(_)을 사용한다. 따라서 각 유효한 식별자 이름은 정규식 `\w+`와 일치한다.

구글 스타일에서 특별한 접두어나 접미사는 사용되지 않는다. 예를 들어 `name_` , `mName`, `s_name` , `kName`과 같은 이름은 구글 스타일이 아니다.

## 5.2 식별자 유형별 규칙

### 5.2.1. 패키지 이름

패키지 이름들은 모두 소문자이며 연속된 단어가 간단히 연결된다 (밑줄 없음).

올바른 예 : `com.example.deepspace`

올바르지 않은 예: `com.example.deepSpace`, `com.example.deep_space`

### 5.2.2 클래스 이름

클래스 이름은 UpperCamelCase 형식으로 작성한다.

클래스 이름은 보통 명사 혹은 명사구이다. 예 : `Character` 또는 `ImmutableList` . 인터페이스 이름도 또한 명사 혹은 명사구 일수 있지만 ( 예 : `List` ) , 때때로 형용사 혹은 형용사구일 수 도 있다 (예 : `Readable` ).

주석 유형을 명명하기 위한 구체적인 규칙이나 심지어 잘 확립된 규약이 없다.

*테스트(Test)* 클래스는 테스팅할 클래스의 이름으로 시작되고 `Test` 로 끝난다. 예 : `HashTest` 또는 `HashIntegerationTest`

### 5.2.3 메소드 이름

메소드 이름은 lowerCamelCase 형식으로 작성한다.

메소드 이름은 보통 동사 혹은 동사구이다. 예 : `sendMessage` 또는 `stop` .

밑줄은 JUnit *테스트* 메소드 이름에 밑줄이 표시되어 논리 컴포넌트를 구분하고 각 컴포넌트는 lowerCamelCase로 작성된다. 일반적인 패턴 중 하나는 *<methodUnderTest>_<state>* 이다 (예: `pop_emptyStack` ). 테스트 메소드 이름을 정하기 위한 하나의 올바른 방법은 없다.

### 5.2.4 상수 이름

상수 이름은 `CONSTANT_CASE` 를 사용한다. : 모두 대문자이며 각 단어 사이는 하나의 밑줄로 구분한다. 상수가 정확히 무엇일까?

상수는 내용이 매우 불변적이고 메소드가 검출 가능한 부작용이 없는 static final 필드이다. 여기에는 primitives, Strings, 불변 types, 불변한 유형의 불변 collections을 포함한다. 만약 어떤 인스턴스의 관찰 가능한 상태가 바뀐다면 상수가 아니다. 절대로 객체를 변경하지 않으려고 하는것만으로는 충분 하지 않다.

예

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35,
"Ann", 32);
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is
immutable
```

```
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }

// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements =
ImmutableSet.of(mutable);
static final ImmutableMap<String, SomeMutableType> mutableValues =
    ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

상수 이름은 보통 명사 혹은 명사구이다.

### 5.2.5 상수가 아닌 필드 이름

상수가 아닌 필드 이름(static 혹은 그 외)은 lowerCamelCase 형식으로 작성한다.

이름들은 보통 명사 혹은 명사구이다. 예 : `computedValues` 또는 `index`

### 5.2.6 매개변수 이름

매개변수 이름은 lowerCamelCase 형식으로 작성한다.

public 메소드의 하나의 문자로 이루어진 매개변수 이름은 피해야한다.

### 5.2.7 지역 변수 이름

지역 변수 이름은 lowerCamelCase 형식으로 작성한다.

심지어 final이고 불변할때 지역 변수는 상수로 간주되지 않고 상수 스타일로 지정해서도 안된다.

### 5.2.8 타입 변수 이름

각 타입 변수 이름은 아래 두 개중 하나로 지정된다.

- 하나의 대문자로 선택적으로 하나의 숫자가 따라올 수 있다. (예 : `E`, `T`, `X`, `T2`)
- 클래스에 사용되는 형식(5.2.2 클래스 이름)의 이름과 그  뒤에 대문자 T가 따라온다.

## 5.3 Camel case : defined

영어 어구를 camel case로 변환하는 두 가지 이상의 합리적인 방법이 있을 수 있다.  (예:  "IPv6" 혹은 "iOS"와 같이 두문자어 혹은 특이한 구문이 보여질 때) 예측 가능성을 향상시키기 위해 구글 스타일은 (거의) 결정론적인 체계를 지정한다.

이름의 산문 형식으로 시작하면 다음과 같다.

1. 어구를 일반 ASCII 로 바꾸고  아포스트로피(')를 제거한다. (예 :  "Müller's algorithm" -> "Muellers algorithm". )

2. 1번의 결과를 단어들로 나누고 공백과 남은 구두점 (일반적으로 하이픈(-))으로 나눈다.
    - *권장*: 일반적으로 camel-case 모양의 단어가 이미 있는 경우 해당 단어를 구성 부분으로 나눈다. (예 : "AdWords" -> "ad words") "iOS"와 같은 단어는 camel-case가 아니며 *어떤* 관습에도 위배되므로 이 권고 사항이 적용 되지 않는다.
3. 이제 모든 것은 (두문자 포함) 소문자로 바꾸고 첫번째 문자만 대문자를 표시한다.
    - ***upper camel case*** : 각 단어 모두 첫번째 문자만 대문자
    - ***lower camel case*** : 첫번째 단어를 제외한 각 단어 첫번째 문자만 대문자
4. 마지막으로 모든 단어들을  단일 식별자로 결합한다.

# 6. 프로그래밍 실습

## 6.1 `@Override` : 항상 사용

메소드는 합법적일 때마다 `@Override` 주석으로 표시된다. 클래스 메소드 오버라이딩하는 수퍼클래스 메소드, 인터페이스 메소드를 구현한 클래스 메소드, 수퍼인터페이스를 다시 지정하는 인터페이스 메소드를 포함한다.

**예외** : 부모 메서드가 `@Deprecated`이면 `@Override`는 생략될 수 있다.

## 6.2 Caught exception(포착된 예외) : 무시되지 않음

아래에 언급 된걸 제외하고 caught exception에 대한 응답으로 아무것도 안하는 것이 드물다. (일반적으로 응답은 로그에 기록하거나 불가능이라고 간주되면 `AssertionError`로 다시 던져진다. )

캐치 블록에서 아무 조치도 취하지 않는 것이 적절 할 경우, 정당화되는 이유는 주석에 설명되어있다.

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

**예외** : 테스트에서는 만약 이름이 `expected`로 시작한다면 caught exception은 주석 없이 무시될 수 있다. 다음은 테스트 중인 코드가 예상되는 유형의 예외를 처리하도록 하는 일반적인 관용구이므로 주석은 필요하지 않다.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

## 6.3 Static 멤버 : 클래스를 사용하여 규정

static 클래스 멤버를 참조(reference)가 규정될 때 클래스의 타입의 참조 혹은 표현이 아니라 클래스의 이름으로규정된다.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

## 6.4 Finalizer : 사용되지 않음

`Object.finalize`를 오버라이드 하는것은 극히 드물다.

**Tip** : 하지 말 것. 만약 절대적으로 필요한 경우, 먼저 *Effective Java* Item 7의 "Avoid Finalizers"를 읽고 이해한 다음 사용하지 말아야한다.

# 7. Javadoc

## 7.1 서식 (Formatting)

### 7.1.1 일반적인 서식

Javadoc 블록의 기본 서식은 밑에 예시와 같다.

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

혹은 단일 행 서식은 다음과 같다.

```
/** An especially short bit of Javadoc. */
```

기본 서식은 항상 허용된다. 단일 행 서식은 Javadoc 블록의 전체( 주석 마커 포함)가 단일 행에 적합할 때 대체 될 수 있다. `@return` 태그가 없는 블록일 때에만 적용 된다.

### 7.1.2 문단

하나의 빈 행 ( 즉, 별표(*)만 포함하는 줄) 은 문단 사이에서 나타나고 블록 태그의 그룹이 존재한다면 블록 태그 그룹 앞에 나타난다. 첫번째 단락을 제외한 문단은 첫번째 단어 바로 앞에 `<p>`가 있으며 뒤에 공백이 없다.

### 7.1.3 블록 태그

사용되는 표준 "블록 태그"는 `@param`, `@return`, `@throws`, `@deprecated` 순서로 나타나며 빈 설명에 함께 나타나지 않는다. 만약 블록 태그가 단일 행에 맞지 않는다면 연속행은 `@` 위치에서 4개 이상의 공백으로 들여 쓰기 된다.

## 7.2 The summary fragment

각 Javadoc 블록은 간단한 **summary fragment**로 시작한다. 이 fragment는 매우 중요하다. 클래스 및 메소드 인덱스와 같이 특정 컨텍스트에 나타나는 텍스트의 유일한 부분이다.

Fragment는 완전한 문장이 아닌 명사구, 동사구이다. `A {@code Foo} is a ...` 혹은 `This method returns ...` 로 시작되지 않으며 `Save the record.`와 같이 완전한 문장이 아니다. 하지만 fragment는 완전한 문장인 것처럼 대문자로 표시되고 문장 부호로 표시된다.

**Tip** : 간단한 Javadoc을 쓸 때 `/** @return the customer ID */` 와 같이 실수한다. 올바른 표현은 `/** Returns the customer ID.*/` 이다.

## 7.3 Where Javadoc is used

아래에 몇가지 예외를 제외하고  최소로 Javadoc은 모든 `public` 클래스와 클래스의 모든 `public` 혹은 `protected` 멤버에서 나타난다.

추가 Javadoc 내용은 존재 할 수 있다. (7.3.4 Non-required Javadoc에 설명 되어있음)

### 7.3.1 예외 : self-explanatory 메소드

Javadoc은 `getFoo` ("Returns the foo" 밖에 말할 가치가 없는 경우) 와 같이 간단하지만 명백한 메소드의 경우 선택 사항이다.

**Important** : 이 예외를 일반적인 독자가 알아야하는 관련 정보를 생략하기위해 정당화 하는 것은 옳지 않다. 예를들어 메소드 이름이 `getCanonicalName` ( `/** Returns the canonical name. */` 만 쓴다는 이유만으로)이면 만약 일반적인 독자가 "canonical name"이 무엇인지 모르는 경우 해당 문서를 생략하지 말아야한다.

### 7.3.2 예외 : overrides

Javadoc은 수퍼타입 메소드를 오버라이드 하는 메소드에 항상 나타나지 않는다.

## 7.3.4 Non-required Javadoc

다른 클래스와 멤버들은 Javadoc이 필요하다.

구현 주석이 클래스 또는 멤버의 전반적인 목적 혹은 행위를 정의하는데 사용될 때마다 주석은 Javadoc으로 작성된다.( `/**`를 사용하여)

필수가 아닌 Javadoc은 7.1.2, 7.1.3 , 7.2 의 서식 규칙을 엄격하게 준수 할 필요는 없지만 권장된다.