Google 자바 스타일 가이드

목차

1. 소개

이 문서는 Java ™ 프로그래밍 언어의 소스 코드에 대한 Google의 코딩 표준에 대한 **완전한** 정의 역할을합니다 . Java 소스 파일은 여기에있는 규칙을 준수하는 경우에만 *Google* 스타일 로 설명됩니다 .

다른 프로그래밍 스타일 가이드와 마찬가지로 다루는 문제는 형식화의 미적 문제뿐 아니라 다른 유형의 규칙이나 코딩 표준에도 적용됩니다. 그러나이 문서는 주로 우리가 보편적으로 따르는 엄격하고 **빠른 규칙** 에 초점을 맞추고 명확하게 시행 할 수없는 조언 (인간 또는 도구)을 제공 하지 않습니다.

1.1 용어 참고

이 문서에서 달리 명시되지 않는 한 :

1. 용어 클래스 는 "일반"클래스, 열거 형 클래스, 인터페이스 또는 주석 유형 (`@interface`) 을 의미하기 위해 포괄적으로 사용됩니다 .
2. (클래스의) 멤버 라는 용어 는 중첩 된 클래스, 필드, 메서드 또는 생성자 를 의미하기 위해 포괄적으로 사용됩니다 . 즉, 이니셜 라이저와 주석을 제외한 클래스의 모든 최상위 콘텐츠입니다.

3. 주석 이라는 용어 는 항상 구현 주석을 의미합니다 . "문서 주석"이라는 문구를 사용하지 않고 "Javadoc"이라는 일반적인 용어를 사용합니다.

문서 전체에 다른 "용어 참고 사항"이 가끔 나타납니다.

1.2 가이드 노트

이 문서의 예제 코드는 **표준** 이 **아닙니다** . 즉, 예제는 Google 스타일이지만 코드를 표현하는 유일한 세련된 방법을 설명하지 못할 수 있습니다 . 예제에서 선택한 선택적 형식 지정은 규칙으로 적용되지 않아야합니다.

2 소스 파일 기본 사항

2.1 파일 이름

소스 파일 이름은 포함 된 최상위 클래스의 대소 문자 구분 이름 (정확히 하나만 있음)과 `.java` 확장자로 구성됩니다.

2.2 파일 인코딩 : UTF-8

소스 파일은 **UTF-8** 로 인코딩됩니다 .

2.3 특수 문자

2.3.1 공백 문자

줄 종결 자 시퀀스를 제외하고 **ASCII 가로 공백 문자** (**0x20**)는 소스 파일의 아무 곳에 나 나타나는 유일한 공백 문자입니다. 이것은 다음을 의미합니다.

1. 문자열 및 문자 리터럴의 다른 모든 공백 문자는 이스케이프됩니다.
2. 탭 문자는 들여 쓰기에 사용 **되지 않습니다** .

2.3.2 특수 이스케이프 시퀀스

있는 모든 문자를 이스케이프 시퀀스 (`\b` , `\t` , `\n` , `\f` , `\r` , `\"` , `\'` 과 `\\`), 그 순서는 해당 진수 (예를 들면보다는 사용 `\012`) 또는 유니 코드 (예를 들어 `\u000a`) 탈출.

2.3.3 비 ASCII 문자

나머지 비 ASCII 문자의 경우 실제 유니 코드 문자 (예 :) ∞ 또는 동등한 유니 코드 이스케이프 (예 :) 가 사용됩니다. 유니 코드가 문자열 리터럴 및 주석 외부에서 이스케이프하는 것은 권장되지 않지만 코드를 **더 쉽게 읽고 이해할 수** 있는 방법에 따라 선택이 달라집니다 . `\u221e`

> **팁 :** 유니 코드 이스케이프 케이스 및 실제 유니 코드 문자가 사용되는 경우에도 설명 주석이 매우 유용 할 수 있습니다.

예 :

예	토론
`String unitAbbrev = "μs";`	최고 : 댓글이 없어도 완벽하게 선명합니다.
`String unitAbbrev = "\u03bcs"; // "μs"`	허용되지만이 작업을 수행할 이유가 없습니다.
`String unitAbbrev = "\u03bcs"; // Greek letter mu, "s"`	허용되지만 어색하고 실수하기 쉽습니다.
`String unitAbbrev = "\u03bcs";`	나쁨 : 독자는 이것이 무엇인지 전혀 모릅니다.
`return '\ufeff' + content; // byte order mark`	좋음 : 인쇄 할 수없는 문자에는 이스케이프를 사용하고 필요한 경우 주석을 추가합니다.

> **팁 :** 일부 프로그램이 비 ASCII 문자를 제대로 처리하지 못할 수 있다는 두려움 때문에 코드의 가독성을 낮추지 마십시오. 이런 일이 발생하면 해당 프로그램이 **중단** 되고 **수정** 해야합니다 .

3 소스 파일 구조

소스 파일은 다음 **순서** 로 구성됩니다 .

1. 라이센스 또는 저작권 정보 (있는 경우)
2. 패키지 명세서
3. 수입 명세서
4. 정확히 하나의 최상위 클래스

정확히 하나의 빈 줄 이 존재하는 각 섹션을 구분합니다.

3.1 라이선스 또는 저작권 정보 (있는 경우)

라이센스 또는 저작권 정보가 파일에 속하면 여기에 속합니다.

3.2 패키지 설명

패키지 문은 **줄 바꿈되지 않습니다** . 열 제한 (섹션 4.4, 열 제한 : 100)은 패키지 문에 적용되지 않습니다.

3.3 Import 문

3.3.1 와일드 카드 가져 오기 없음

정적 또는 기타 **와일드 카드 가져 오기 는 사용되지 않습니다** .

3.3.2 줄 바꿈 없음

Import 문은 **줄 바꿈되지 않습니다** . 열 제한 (섹션 4.4, 열 제한 : 100)은 import 문에 적용되지 않습니다.

3.3.3 순서 및 간격

수입품은 다음과 같이 주문됩니다.

1. 단일 블록에서 모든 정적 가져 오기.
2. 단일 블록의 모든 비 정적 가져 오기.

정적 가져 오기와 비 정적 가져 오기가 모두있는 경우 하나의 빈 줄이 두 블록을 구분합니다. import 문 사이에는 다른 빈 줄이 없습니다.

각 블록 내에서 가져온 이름은 ASCII 정렬 순서로 나타납니다. (**참고 :** 이는 '.'가 ';'앞에 정렬되므로 ASCII 정렬 순서 인 import 문과 동일하지 않습니다 .)

3.3.4 클래스에 대한 정적 가져 오기 없음

정적 중첩 클래스에는 정적 가져 오기가 사용되지 않습니다. 그들은 일반 수입품으로 수입됩니다.

3.4 클래스 선언

3.4.1 정확히 하나의 최상위 클래스 선언

각 최상위 클래스는 자체 소스 파일에 있습니다.

3.4.2 수업 내용 순서

클래스의 멤버 및 이니셜 라이저에 대해 선택한 순서는 학습 가능성에 큰 영향을 미칠 수 있습니다. 그러나이를 수행하는 방법에 대한 올바른 방법은 없습니다. 다른 클래스는 다른 방식으로 내용을 주문할 수 있습니다.

중요한 것은 각 클래스가 **몇 가지 논리적 순서를** 사용한다는 것입니다.이 **순서** 는 관리자가 요청하면 설명 할 수 있습니다. 예를 들어, 새로운 메서드는 논리적 순서가 아닌 "추가 된 날짜 별 시간순"순서를 생성하므로 클래스 끝에 습관적으로 추가되는 것이 아닙니다.

3.4.2.1 과부하 : 분할되지 않음

클래스에 여러 생성자 또는 동일한 이름을 가진 여러 메서드가있는 경우 이들은 사이에 다른 코드가 없이 (개인 멤버도 포함되지 않음) 순차적으로 나타납니다.

4 서식

용어 참고 : 블록 형 구조 는 클래스, 메서드 또는 생성자의 본문을 나타냅니다. 배열 이니셜 라이저 에 대한 섹션 4.8.3.1에 따라 모든 배열 이니셜 라이저 는 선택적으로 블록과 유사한 구조 인 것처럼 처리 될 수 있습니다.

4.1 교정기

4.1.1 선택 사항 인 경우 중괄호가 사용됩니다.

교정기가 사용되어 `if` , `else` , `for` , `do` 및 `while` 문, 몸이 비어 있거나 단 하나의 문이 포함 된 경우에도.

4.1.2 비어 있지 않은 블록 : K & R 스타일

중괄호 는 비어 있지 않은 블록 및 블록 유사 구조에 대해 Kernighan 및 Ritchie 스타일 (" 이집트 괄호 ")을 따릅니다 .

- 여는 중괄호 앞에 줄 바꿈이 없습니다.
- 여는 중괄호 뒤의 줄 바꿈.
- 닫는 중괄호 앞의 줄 바꿈.
- 닫는 중괄호 뒤의 줄 바꿈 (중괄호가 명령문을 종료하거나 메서드, 생성자 또는 명명 된 클래스 의 본문을 종료하는 경우) 예를 들어 중괄호 뒤에 또는 쉼표 가 오면 줄 바꿈 이 없습니다 . `else`

예 :

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




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

열거 형 클래스에 대한 몇 가지 예외는 섹션 4.8.1, 열거 형 클래스에 있습니다.

4.1.3 빈 블록 : 간결 할 수 있음

빈 블록 또는 블록과 유사한 구조는 K & R 스타일 일 수 있습니다 (섹션 4.1.2에 설명 됨). 또는 다중 블록 명령문 (다중 블록을 직접 포함하는 명령문 : 또는)의 일부가 **아닌 경우** (`{}`) 사이에 문자 나 줄 바꿈없이 열린 직후 닫을 수 있습니다 . `if/else` `try/catch/finally`

예 :

```
// 허용됩니다. void doNothing () {}


// 동일하게 허용됩니다. void doNothingElse () { }
```

```
// 허용되지 않습니다. 다중 블록 문에 간결한 빈 블록이 없습니다. try {
  doSomething (); } catch ( 예외 e ) {}
```

4.2 블록 들여 쓰기 : +2 공백

새 블록 또는 블록과 유사한 구조가 열릴 때마다 들여 쓰기가 두 칸씩 증가합니다. 블록이 끝나면 들여 쓰기는 이전 들여 쓰기 수준으로 돌아갑니다. 들여 쓰기 수준은 블록 전체의 코드와 주석 모두에 적용됩니다. (섹션 4.1.2, 비어 있지 않은 블록 : K & R 스타일 의 예를 참조하십시오 .)

4.3 한 줄에 하나의 문

각 문 뒤에는 줄 바꿈이 있습니다.

4.4 열 제한 : 100

Java 코드의 열 제한은 100 자입니다. "문자"는 모든 유니 코드 코드 포인트를 의미합니다. 아래 명시된 경우를 제외하고이 제한을 초과하는 모든 줄은 섹션 4.5, 줄 바꿈에 설명 된대로 줄 바꿈해야합니다 .

> 각 유니 코드 코드 포인트는 표시 너비가 더 크거나 작아도 하나의 문자로 계산됩니다. 예를 들어 전각 문자를 사용 하는 경우이 규칙이 엄격하게 요구하는 위치보다 먼저 줄을 줄 바꿈하도록 선택할 수 있습니다.

예외 :

1. 열 제한을 따를 수없는 행 (예 : Javadoc의 긴 URL 또는 긴 JSNI 메소드 참조).
2. `package` 및 `import` 문 (섹션 3.2 패키지 문 및 3.3 Import 문 참조).
3. 셸에 잘라서 붙여 넣을 수있는 주석의 명령 줄입니다.

4.5 줄 바꿈

용어 참고 : 합법적으로 한 줄을 차지할 수있는 코드를 여러 줄로 나눈 경우이 작업을 줄 바꿈 이라고 합니다 .

모든 상황에서 줄 바꿈하는 방법을 정확히 보여주는 포괄적이고 결정적인 공식은 없습니다 . 동일한 코드를 줄 바꿈하는 여러 가지 유효한 방법이 매우 자주 있습니다.

> **참고 :** 줄 바꿈의 일반적인 이유는 열 제한을 초과하지 않도록하는 것이지만 실제로 열 제한에 맞는 코드도 작성자의 재량에 따라 줄 바꿈 될 수 있습니다.

> **팁 :** 메서드 또는 지역 변수를 추출하면 줄 바꿈없이 문제를 해결할 수 있습니다.

4.5.1 휴식 장소

줄 바꿈의 주요 지침은 **더 높은 구문 수준** 에서 중단하는 것을 선호하는 **것** 입니다. 또한:

1. 비 할당 연산자 에서 줄이 끊어 지면 기호 앞에 끊어 집니다. (이것은 C ++ 및 JavaScript와 같은 다른 언어의 Google 스타일에서 사용되는 것과 동일한 관행이 아닙니다.)
   - 이는 다음 "연산자 유사"기호에도 적용됩니다.
     - 점 구분 기호 (`.`)
     - 메서드 참조의 두 콜론 (`::`)
     - 유형 바운드 ( ) 의 앰퍼샌드 `<T extends Foo & Bar>`
     - 캐치 블록의 파이프 ( ). `catch (FooException | BarException e)`
2. 할당 연산자 에서 줄이 끊어지면 일반적으로 기호 뒤에 끊어 지지만 어느 쪽이든 허용됩니다.
   - 이는 확장 `for` ("foreach") 문 에서 "할당 연산자와 같은"콜론에도 적용됩니다 .
3. 메서드 또는 생성자 이름은 그 뒤에 오는 여는 괄호 (`(`)에 연결된 상태로 유지 됩니다.
4. 쉼표 (`,`)는 그 앞에있는 토큰에 계속 붙어 있습니다.
5. 람다의 본문이 중괄호가없는 단일 식으로 구성된 경우 화살표 바로 뒤에 줄 바꿈이있을 수 있다는 점을 제외하고는 람다의 화살표 옆에서 줄이 끊어지지 않습니다. 예 :

```
MyLambda < String , Long , Object > lambda = ( String label , Long v




술어 < 문자열 > 술어 = STR ->
    longExpressionInvolving ( STR );
```

> **참고 :** 줄 바꿈의 기본 목표 는 최소한의 줄에 맞는 코드가 아니라 명확한 코드를 만드는 것입니다.

4.5.2 연속 줄을 최소 +4 공백 들여 쓰기

줄 바꿈시 첫 번째 줄 (각 연속 줄) 뒤의 각 줄 은 원래 줄에서 적어도 +4만큼 들여 쓰기됩니다.

연속 줄이 여러 개인 경우 원하는대로 들여 쓰기를 +4 이상으로 변경할 수 있습니다. 일반적으로 두 개의 연속 줄은 구문 상 병렬 요소로 시작하는 경우에만 동일한 들여 쓰기 수준을 사용합니다.

수평 정렬 에 관한 섹션 4.6.3은 특정 토큰을 이전 행과 정렬하기 위해 가변 수의 공백을 사용하는 권장되지 않는 관행을 다룹니다.

4.6 공백

4.6.1 세로 공백

항상 하나의 빈 줄이 나타납니다.

1. 사이 필드, 생성자, 메소드, 중첩 클래스, 정적 초기화 및 인스턴스 초기화 : 연속 구성원 또는 클래스의 초기화.
   - **예외 :** 두 개의 연속 된 필드 사이에 다른 코드가없는 빈 줄은 선택 사항입니다. 이러한 빈 줄은 필요에 따라 필드의 논리적 그룹 을 만드는 데 사용됩니다 .
   - **예외 :** 열거 형 상수 사이의 빈 줄은 섹션 4.8.1 에서 다룹니다 .
2. 이 문서의 다른 섹션에서 요구하는대로 (예 : 섹션 3, 소스 파일 구조 및 섹션 3.3, 가져 오기 문).

예를 들어 코드를 논리적 하위 섹션으로 구성하는 명령문 사이와 같이 가독성을 향상시키는 모든 곳에 빈 줄 하나가 나타날 수 있습니다. 첫 번째 멤버 나 이니셜 라이저 앞 또는 클래스의 마지막 멤버 나 이니셜 라이저 뒤의 빈 줄은 권장되거나 권장되지 않습니다.

여러 개의 연속 된 빈 줄이 허용되지만 필수 (또는 권장)는 아닙니다.

4.6.2 수평 공백

언어 또는 기타 스타일 규칙에서 요구하는 경우 외에 리터럴, 주석 및 Javadoc을 제외하고 단일 ASCII 공간은 다음 위치 **에만** 나타납니다 .

1. 해당 줄에서 뒤에 오는 여는 괄호 ( ) 에서 `if` , `for` 또는 같은 예약어 분리 `catch` (
2. `else` 또는 과 같은 예약어 를 해당 줄에서 앞에 `catch` 오는 닫는 중괄호 (`}`) 에서 분리
3. 여는 중괄호 (`{`) 앞 , 두 가지 예외 :
   - `@SomeAnnotation({a, b})` (공백이 사용되지 않음)
   - `String[][] x = {{"foo"}};` (`{{` 아래 8 번 항목 사이에 공백이 필요하지 않음)
4. 이항 또는 삼항 연산자의 양쪽에 있습니다. 이는 다음 "연산자 유사"기호에도 적용됩니다.
   - 결합 형 바인딩의 앰퍼샌드 : `<T extends Foo & Bar>`
   - 여러 예외를 처리하는 catch 블록에 대한 파이프 :
     `catch (FooException | BarException e)`
   - `:` 향상된 `for` ("foreach") 문의 콜론 ( )
   - 람다 식의 화살표 : `(String str) -> str.length()`

   하지만
   - `::` 메소드 참조 의 두 콜론 ( )은 다음과 같이 작성됩니다. `Object::toString`
   - 다음 `.` 과 같이 쓰여진 점 구분 기호 ( ) `object.toString()`
5. 캐스트 뒤 `,:;` 또는 닫는 괄호 (`)`)
6. `//` 줄 끝 주석을 시작하는 이중 슬래시 ( ) 의 양쪽 . 여기에서는 여러 개의 공백이 허용되지만 필수는 아닙니다.
7. 선언의 유형과 변수 사이 : `List<String> list`
8. 배열 이니셜 라이저의 두 중괄호 바로 안에 옵션
   - `new int[] {5, 6}` 그리고 모두 유효 `new int[] { 5, 6 }`
9. 유형 주석과 `[]` 또는 사이 `...` .

이 규칙은 줄의 시작 또는 끝에 추가 공백을 요구하거나 금지하는 것으로 해석되지 않습니다. 그것은 내부 공간 만을 다룬다 .

4.6.3 수평 정렬 : 필요 없음

용어 참고 : 수평 정렬 은 특정 토큰이 이전 줄의 다른 특정 토큰 바로 아래에 표시되도록 코드에 다양한 수의 추가 공백을 추가하는 방법입니다.

이 관행은 허용되지만 Google 스타일에서 **요구** 하는 것은 **아닙니다** . 이미 사용 된 장소에서 수평 정렬 을 유지할 필요조차 없습니다 .

다음은 정렬없이 정렬을 사용하는 예입니다.

```
private int x ; // 이것은 훌륭한 개인 색상 입니다 . // 이것도


private int    x ; // 허용되지만 향후 편집 개인 색상 색상 ; // 정렬되지 않은 
```

> **팁 :** 정렬은 가독성에 도움이 될 수 있지만 향후 유지 관리에 문제가됩니다. 한 줄만 터치하면되는 미래의 변화를 생각해보십시오. 이 변경으로 이전에 만족 스러웠던 서식이 엉망이 될 **수** 있으며 **허용** 됩니다. 더 자주 코더 (아마도 사용자)에게 근처 줄의 공백을 조정하라는 메시지를 표시하여 연속적인 일련의 재 포맷을 트리거 할 수 있습니다. 그 한 줄 변경은 이제 "폭발 반경"을 갖습니다. 이것은 최악의 경우 무의미한 작업을 초래할 수 있지만 기껏해야 버전 기록 정보를 손상시키고 검토 자의 속도를 늦추고 병합 충돌을 악화시킵니다.

4.7 그룹화 괄호 : 권장

선택적 그룹화 괄호는 작성자와 검토자가 코드가 없으면 코드가 잘못 해석 될 가능성이 없으며 코드를 읽기 쉽게 만들지 않았다는 데 동의하는 경우에만 생략됩니다. 이다 없는 모든 독자가 기억 전체 자바 연산자 우선 순위 테이블이 있다고 가정하는 것이 합리적.

4.8 특정 구조

4.8.1 Enum 클래스

열거 형 상수 뒤에 오는 각 쉼표 뒤에 줄 바꿈은 선택 사항입니다. 추가 빈 줄 (일반적으로 하나만)도 허용됩니다. 이것은 한 가지 가능성입니다.

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




  아니 ,
  MAYBE
}
```

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see Section 4.8.3.1 on array initializers).

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

Since enum classes *are classes*, all other rules for formatting classes apply.

4.8.2 Variable declarations

4.8.2.1 One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

Exception: Multiple variable declarations are acceptable in the header of a `for` loop.

4.8.2.2 Declared when needed

Local variables are **not** habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.

4.8.3 Arrays

4.8.3.1 Array initializers: can be "block-like"

Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
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 No C-style array declarations

The square brackets form a part of the *type*, not the variable: `String[] args`, not `String args[]`.

4.8.4 Switch statements

Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements (or, for the *last* statement group, *zero* or more statements).

4.8.4.1 Indentation

As with any other block, the contents of a switch block are indented +2.

After a switch label, there is a line break, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

4.8.4.2 Fall-through: commented

Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
스위치 ( 입력 ) { 케이스 1 : 케이스 2 :
    prepareOneOrTwo (); // 케이스 3 통과 :
    handleOneTwoOrThree (); 휴식 ; 기본값 :
    handleLargeNumber ( 입력 ); }
```

다음에 주석이 필요하지 않으며 명령문 그룹의 끝에서만 주석이 필요하지 않습니다 . `case 1:`

4.8.4.3 `default` 케이스 존재

각 switch 문에는 `default` 코드가없는 경우에도 문 그룹이 포함됩니다.

예외 : `enum` 유형에 대한 switch 문 은 해당 유형의 가능한 모든 값을 포함하는 명시 적 케이스를 포함하는 경우 명령문 그룹을 생략 할 수 있습니다 . 이를 통해 IDE 또는 기타 정적 분석 도구는 누락 된 사례가있는 경우 경고를 발행 할 수 있습니다. `default`

4.8.5 주석

클래스, 메서드 또는 생성자에 적용되는 주석은 문서 블록 바로 뒤에 나타나며 각 주석은 자체 줄에 나열됩니다 (즉, 한 줄에 하나의 주석). 이러한 줄 바꿈은 줄 바꿈을 구성하지 않으므로 (섹션 4.5, 줄 바꿈) 들여 쓰기 수준이 증가하지 않습니다. 예:

```
@Override @Nullable 공공 문자열 getNameIfPresent () { ... }
```

예외 : 하나의 매개 변수가 주석이 수 대신 예를 들어, 서명의 첫 번째 줄과 함께 나타납니다 :

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

필드에 적용되는 주석도 문서 블록 바로 뒤에 나타나지만이 경우 여러 주석 (매개 변수화 가능)이 같은 줄에 나열 될 수 있습니다. 예를 들면 :

```
@Partial @Mock DataLoader 로더 ;
```

매개 변수, 지역 변수 또는 유형에 대한 주석 형식화에 대한 특정 규칙은 없습니다.

4.8.6 주석

이 섹션에서는 구현 의견을 다룹니다 . Javadoc은 섹션 7, Javadoc 에서 별도로 다루고 있습니다.

줄 바꿈 앞에는 임의의 공백과 구현 주석이 올 수 있습니다. 이러한 주석은 행을 공백이 아닌 것으로 렌더링합니다.

4.8.6.1 블록 주석 스타일

블록 주석은 주변 코드와 동일한 수준에서 들여 쓰기됩니다. `/* ... */` 스타일이나 `// ...` 스타일이 있을 수 있습니다 . 여러 줄 `/* ... */` 주석의 경우 후속 줄은 이전 줄에 `*` 정렬 된 것으로 시작해야합니다 `*` .

```
/ *
 * 이것은 // 그래서 / * 또는 당신은 할 수 있습니다
 * 괜찮아. // 이것은 이것입니다. * 심지어 이것을하십시오. * / * /
```

주석은 별표 또는 기타 문자로 그려진 상자에 포함되지 않습니다.

> **팁 :** 여러 줄 주석을 작성할 때 `/* ... */` 자동 코드 포맷터가 필요할 때 줄을 다시 줄 이도록 하려면 (단락 스타일) 스타일을 사용하십시오. 대부분의 포맷터는 `// ...` 스타일 주석 블록 에서 줄을 다시 감싸지 않습니다 .

4.8.7 수정 자

클래스 및 멤버 수정자는있는 경우 Java 언어 사양에서 권장하는 순서로 나타납니다.

```
public protected private abstract default static final transient volatile 동기화
```

4.8.8 숫자 리터럴

`long` 값을 갖는 정수 리터럴 `L` 은 소문자가 아닌 대문자 접미사를 사용합니다 (digit와의 혼동을 피하기 위해 `1`). 예를 들어, `3000000000L` 가 아니라 `3000000000l` .

5 명명

5.1 모든 식별자에 공통적 인 규칙

식별자는 ASCII 문자와 숫자 만 사용하며 아래에 언급 된 소수의 경우 밑줄로 표시됩니다. 따라서 각 유효한 식별자 이름은 정규식과 일치합니다 `₩w+` .

Google 스타일에서는 특수 접두사 또는 접미사가 사용 **되지 않습니다** . 예를 들어,이 이름은 구글 스타일 없습니다 : `name_` , `mName` , `s_name` 와 `kName` .

5.2 식별자 유형별 규칙

5.2.1 패키지 이름

패키지 이름은 모두 소문자이며 연속 된 단어는 단순히 함께 연결됩니다 (밑줄 없음). 예를 들어, `com.example.deepspace` not `com.example.deepSpace` 또는 `com.example.deep_space` .

5.2.2 클래스 이름

클래스 이름은 UpperCamelCase 로 작성됩니다 .

클래스 이름은 일반적으로 명사 또는 명사구입니다. 예를 들어, `Character` 또는 `ImmutableList` . 인터페이스 이름은 명사 또는 명사 구 (예 `List` :) 일 수도 있지만 때로는 대신 형용사 또는 형용사 구 (예 :) 일 수도 있습니다 `Readable` .

주석 유형 이름 지정에 대한 특정 규칙이나 잘 확립 된 규칙은 없습니다.

테스트 클래스의 이름은 테스트중인 클래스의 이름으로 시작하고 `Test` . 예를 들어, `HashTest` 또는 `HashIntegrationTest` .

5.2.3 메서드 이름

메소드 이름은 lowerCamelCase 로 작성됩니다 .

메서드 이름은 일반적으로 동사 또는 동사 구입니다. 예를 들어, `sendMessage` 또는 `stop` .

JUnit 테스트 메서드 이름 에 밑줄이 표시되어 이름의 논리적 구성 요소를 구분할 수 있으며 각 구성 요소는 lowerCamelCase로 작성됩니다 . 한 가지 전형적인 패턴은 예를 들면 다음과 같습니다 . 테스트 방법의 이름을 정하는 올바른 방법은 없습니
다. `<methodUnderTest>_<state>` pop_emptyStack

5.2.4 상수 이름

상수 이름 사용 `CONSTANT_CASE` : 단일 밑줄로 각 단어를 다음 단어와 구분하는 모든 대문자. 그러나 이며 일정은 정확히?

상수는 내용이 완전히 불변하고 메서드에 감지 가능한 부작용이없는 정적 최종 필드입니다. 여기에는 프리미티브, 문자열, 불변 유형 및 불변 유형의 불변 컬렉션이 포함됩니다. 인스턴스의 관찰 가능 상태가 변경 될 수있는 경우 상수가 아닙니다. 단순히 하려는 개체를 변이 결코 충분하지 않습니다. 예 :

```
// 상수 static final int NUMBER = 5 ; static final ImmutableList < String


  COMMA_JOINER = 결합 자 . on ( ',' ); // Joiner는 불변이기 때문에 static fina
```

```
// 상수가 아님 static String nonFinal = "non-final" ; final String nonStatic


  , SomeMutableType > mutableValues = ImmutableMap . of ( "Ed" , mutableInst
```

이러한 이름은 일반적으로 명사 또는 명사 구입니다.

5.2.5 상수가 아닌 필드 이름

상수가 아닌 필드 이름 (정적 또는 기타)은 lowerCamelCase 로 작성됩니다 .

이러한 이름은 일반적으로 명사 또는 명사 구입니다. 예를 들어, `computedValues` 또는 `index` .

5.2.6 매개 변수 이름

매개 변수 이름은 lowerCamelCase 로 작성됩니다 .

공용 메소드에서 한 문자 매개 변수 이름은 피해야합니다.

5.2.7 지역 변수 이름

지역 변수 이름은 lowerCamelCase 로 작성됩니다 .

최종적이고 변경 불가능한 경우에도 지역 변수는 상수로 간주되지 않으며 상수로 스타일을 지정해서는 안됩니다.

5.2.8 유형 변수 이름

각 유형 변수는 다음 두 가지 스타일 중 하나로 이름이 지정됩니다.

- 임의로 하나의 번호 뒤에 단일 대문자 (예컨대 `E` , `T` , `X` , `T2`)
- 클래스에 사용되는 형식의 이름 (섹션 5.2.2, 클래스 이름 참조)과 대문자 `T` (예 : `RequestT` , `FooBarT`).

5.3 카멜 케이스 : 정의 됨

두문자어 또는 "IPv6"또는 "iOS"와 같은 비정상적인 구조가있는 경우와 같이 영어 구를 낙타 대문자로 변환하는 합리적인 방법이 여러 가지 있습니다. 예측 가능성을 높이기 위해 Google 스타일은 다음과 같은 (거의) 결정 론적 체계를 지정합니다.

이름의 산문 형식으로 시작 :

1. 구를 일반 ASCII로 변환하고 아포스트로피를 제거하십시오. 예를 들어 "Müller의 알고리즘"은 "Muellers 알고리즘"이 될 수 있습니다.
2. 이 결과를 단어로 나누고 공백과 나머지 구두점 (일반적으로 하이픈)으로 분리합니다.
   - 권장 : 일반적으로 사용되는 일반적인 낙타 대문자 모양이 이미있는 단어가 있으면이를 구성 부분으로 분할합니다 (예 : "애드워즈"는 "광고 단어"가 됨). "iOS"와 같은 단어는 실제로 낙타 대소 문자 자체 가 아닙니다 . 어떤 규칙도 위반하므로이 권장 사항은 적용되지 않습니다.
3. 이제 모든 것을 소문자 (약어 포함)하고 다음의 첫 번째 문자 만 대문자로 표시합니다.
   - ... 각 단어, 대문자 낙타 대문자 를 산출 하거나
   - ... 첫 번째 단어를 제외한 각 단어는 낮은 낙타 케이스 를 산출합니다.
4. 마지막으로 모든 단어를 단일 식별자로 결합합니다.

원래 단어의 대소 문자는 거의 완전히 무시됩니다. 예 :

산문 형식	옳은	틀렸다
"XML HTTP 요청"	`XmlHttpRequest`	`XMLHTTPRequest`
"새 고객 ID"	`newCustomerId`	`newCustomerID`
"내부 스톱워치"	`innerStopwatch`	`innerStopWatch`
"iOS에서 IPv6를 지원합니까?"	`supportsIpv60nIos`	`supportsIPv60nIOS`
"YouTube 가져 오기"	`YouTubeImporter` `YoutubeImporter *`	

* 허용되지만 권장되지는 않습니다.

> **참고 :** 일부 단어는 영어에서 모호하게 하이픈으로 연결됩니다. 예를 들어 "nonempty"및 "non-empty"는 모두 정확하므로 메서드 이름 `checkNonempty` 과 `checkNonEmpty` 마찬가지로 모두 정확합니다.

6 프로그래밍 실습

6.1 `@Override` : 항상 사용

메서드는 `@Override` 합법적 일 때마다 주석 으로 표시됩니다 . 여기에는 수퍼 클래스 메소드를 재정의하는 클래스 메소드, 인터페이스 메소드를 구현하는 클래스 메소드, 수퍼 인터페이스 메소드를 재지 정하는 인터페이스 메소드가 포함됩니다.

예외 : `@Override` 부모 메서드가 인 경우 생략 할 수 있습니다 `@Deprecated` .

6.2 포착 된 예외 : 무시되지 않음

아래에 언급 된 경우를 제외하고 포착 된 예외에 대해 아무 조치도 취하지 않는 것은 매우 드뭅니다. (일반적인 응답은 로그에 기록하거나 "불가능"하다고 판단되면으로 다시 던지는 것 `AssertionError` 입니다.)

catch 블록에서 아무 조치도 취하지 않는 것이 진정으로 적절할 때 이것이 정당화되는 이유는 주석에 설명되어 있습니다.

```
try { int i = Integer . parseInt ( 응답 ); return handleNumericResponse ( i )
```

예외 : 테스트에서 포착 된 예외는 이름이이거나로 시작하는 경우 주석없이 무시 될 수 있습니다 `expected` . 다음은 테스트중인 코드가 있음을 보장하기위한 매우 일반적인 관용구이다 않는 주석이 여기에 필요하므로, 예상되는 유형의 예외를 throw합니다.

```
시도 {
  emptyStack을 . 팝 ();
  실패 (); } catch ( NoSuchElementException 예상 됨 ) { }
```

6.3 정적 멤버 : 클래스를 사용하여 정규화

정적 클래스 멤버에 대한 참조가 정규화되어야하는 경우 해당 클래스 유형의 참조 또는식이 아닌 해당 클래스의 이름으로 정규화됩니다.

```
Foo aFoo = ...; Foo . aStaticMethod (); // 좋은 aFoo . aStaticMethod (); // 나쁜
```

6.4 종료 자 : 사용되지 않음

을 재정의하는 것은 **매우 드뭅니다** . `Object.finalize`

> **팁 :** 하지 마십시오. 꼭 필요한 경우 먼저 효과적인 *Java* 항목 7, "종료 자 방지"를 매우주의 깊게 읽고 이해 한 후 수행하지 마십시오.

7 Javadoc

7.1 서식

7.1.1 일반 형식

Javadoc 블록 의 기본 형식은 다음 예와 같습니다.

```
/ **
 * 여러 줄의 Javadoc 텍스트가 여기에 작성됩니다.
 * 일반적으로 포장 ...
 * / public int method ( String p1 ) { ... }
```

... 또는이 한 줄 예제에서 :

```
/ ** 특히 짧은 Javadoc입니다. * /
```

기본 형식은 항상 허용됩니다. Javadoc 블록 (주석 마커 포함) 전체가 한 줄에 들어갈 수있는 경우 한 줄 형식이 대체 될 수 있습니다. 이는와 같은 블록 태그가없는 경우에만 적용됩니다 `@return` .

7.1.2 단락

하나의 빈 줄, 즉 정렬 된 선행 별표 (`*`) 만 포함 된 줄 은 단락 사이와 블록 태그 그룹 (있는 경우) 앞에 나타납니다. 첫 번째를 제외한 각 단락 `<p>` 은 첫 번째 단어 바로 앞에 있으며 뒤에 공백이 없습니다.

7.1.3 블록 태그

사용되는 표준 "블록 태그"중 어떤 순서대로 표시 `@param` , `@return` , `@throws` , `@deprecated` ,이 네 가지 유형의 빈 설명과 함께 표시되지 않습니다. 블록 태그가 한 줄에 맞지 않는 경우 연속 줄은 .NET Framework의 위치에서 4 개 이상의 공백을 들여 씁니다 `@` .

7.2 요약 조각

각 Javadoc 블록은 간단한 **요약 조각으로** 시작됩니다 . 이 조각은 매우 중요합니다. 클래스 및 메서드 인덱스와 같은 특정 컨텍스트에 나타나는 텍스트의 유일한 부분입니다.

이것은 완전한 문장이 아니라 명사구 또는 동사구의 단편입니다. 그것은 않습니다 **하지** 로 시작 `A {@code Foo} is a...` 하거나 `This method returns...` ,도 아니다처럼 완전한 필수 문장을 형성 않습니다 `Save the record.` . 그러나 조각은 완전한 문장 인 것처럼 대문자로 표시되고 구두점으로 표시됩니다.

> **팁 :** 일반적인 실수는 간단한 Javadoc을 형식으로 작성하는 것 `/** @return the customer ID */` 입니다. 이것은 올바르지 않으며로 변경해야합니다 `/** Returns the customer ID. */` .

7.3 Javadoc이 사용되는 곳

상기 최소 javadoc는 모든 대 존재하는 `public` 클래스마다 `public` 또는 `protected` 몇 개의 예외 이러한 클래스의 멤버는, 아래에 언급.

Section 7.3.4, Non-required Javadoc에 설명 된대로 추가 Javadoc 컨텐츠가있을 수도 있습니다 .

7.3.1 예외 : 자명 한 방법

자바 독과 같은 "단순하고 명백한"방법에 대한 선택 사항입니다 `getFoo` 경우에이 곳, 진실로 "반환 foo는"말을하지만,하는 가치가 다른 아무것도 없다.

> **중요 :** 일반적인 독자가 알아야 할 관련 정보를 생략하는 것을 정당화하기 위해이 예외를 인용하는 것은 적절하지 않습니다. 예를 들어,라는 메서드의 경우 일반적인 독자가 "표준 이름"이라는 용어가 무엇을 의미하는지 모를 수 있다면 `getCanonicalName` 문서를 생략하지 마십시오 (라고만 말하는 근거 포함 `/** Returns the canonical name. */`).

7.3.2 예외 : 재정의

Javadoc은 수퍼 타입 메소드를 대체하는 메소드에 항상 존재하는 것은 아닙니다.

7.3.4 필수가 아닌 Javadoc

다른 클래스와 멤버에는 필요하거나 원하는대로 Javadoc이 있습니다 .

구현 주석이 클래스 또는 멤버의 전체적인 목적이나 동작을 정의하는 데 사용될 때마다 해당 주석은 대신 Javadoc로 작성됩니다 (사용 `/**`).

필수가 아닌 Javadoc은 섹션 7.1.2, 7.1.3 및 7.2의 형식화 규칙을 따르도록 엄격하게 요구되지는 않지만 물론 권장됩니다.