네이버 코딩 컨벤션 (Java)

Ver 0.4.1

저작권

목차

1. 가이드의 취지

이 가이드는 프로젝트 구성원 간의 코드 형식에 대한 합의 수준을 높히기 위한 목적으로 작성되었다. 가이드를 적용하는 프로젝트에서 다음과 같은 이득이 있기를 기대한다.

첫째, 기본적인 코드 형식에 대해서 프로젝트마다 초기에 반복 논의하는 시간을 줄인다. 가이드에서 설명한 보편적인 요소를 바탕으로 프로젝트별로 예외나 정교한 규칙에 대한 논의를 짧은 시간 안에 할수 있도록 한다. 단순한 코드 형식보다 깊이 있는 규칙을 정하는 단계로 빨리 진입하도록 돕는다.

둘째, 코드 리뷰 과정에서 핵심로직에 집중할 수 있게 한다. 정교하지 못한 규칙으로 프로젝트를 진행하면 코드 리뷰 때 형식을 맞추기 위한 지적과 반영에 시간을 소모한다. 버그 수정, 기능 추가를 할 때도 줄,공백을 바꾼 부분으로 인한 DIFF가 많이 보인다면 변경의 핵심을 파악하는데 방해가 된다. 컨벤션 가이드는 그런 요소를 최소화하여 핵심적인 변경을 파악하고 리뷰하는 시간을 줄여준다.

네이버의 공통 컨벤션 가이드는 외부의 가이드와 대비해서 아래와 같은 장점을 추구한다.

첫째, 외부의 가이드보다 적극적으로 개발자의 의견이 반영된다. 네이버의 구성원이 가이드 내용, 룰체크 설정을 유지 보수하기 때문이다.

둘째, 오픈소스 컨트리뷰션과 같은 방식으로 가이드의 내용을 수정 제안할 수 있다. 가이드는 프로그램 소스와 유사한 방식으로 가이드를 관리한다. ASCIIDOC 형식인 텍스트 파일로 작성하고 GIT으로 버전을 관리한다.

이 가이드에서 네이버의 개발자 공동체에 의해 유지되고 발전되는 기준을 담아가고자 한다.

2. 운영 규칙

2.1 문의와 요청 경로

가이드 내용에 대한 문의와 의견은 https://oss.navercorp.com/engineering/coding-convention/issues 에 새로운 이슈로 등록한다. 이슈를 등록할 때 제목은 `[언어][규칙 식별자]`로 시작한다. 예시는 다음과 같다.

```
[Java][avoid-1-char-var] 바람직한 사례를 추가
```

규칙 식별자는 각 규칙의 제목 다음에 표기되어 있다.

2.2 프로젝트별로 정의할 파일

라이센스 파일 포함

[file-licence]

프로젝트의 루트 디렉토리에 `LICENSE`라는 파일명으로 라이센스를 명시한다.

프로젝트별 정책 파일 포함

[file-convention]

프로젝트의 루트 디렉토리에 CONVENTION.* 파일명으로 프로젝트별 규칙을 명시한다. 파일의 확장자는 제한을 두지 않으나 `LICENSE.md` , `LICENSE.adoc` 등 해당 파일의 포멧을 나타내는 관례를 따른다. 텍스트 에디터로 열어볼 수 있는 포멧을 권장한다.

공통 가이드의 내용 외에 프로젝트에 특화된 컨벤션을 정의하지 않는다면 생략가능하다.

해당 파일에는 아래와 같은 내용이 포함될 수 있다.

- 공통 가이드에 내용 중 해당 프로젝트에서는 예외로 할 내용
  - 예) 테스트 코드의 함수 이름에서는 "_"를 허용한다.
- 디렉토리/패키지 구성 관례
  - 예) 통합테스트는 /src/main/integration-test에 포함시킨다.
- 해당 프로젝트에 쓰는 라이브러리/프레임워크에 특화된 내용
  - 예) jquery의 bind() 함수대신 on() 함수를 사용한다.

3. 파일 공통 요건

3.1 파일 인코딩은 UTF-8

[encoding-utf8]

모든 소스, 텍스트 문서 파일의 인코딩은 UTF-8로 통일한다.

3.2 새줄 문자는 LF

[newline-lf]

Unix 형식의 새줄 문자(newline)인 LF(Line Feed, 0x0A)을 사용한다. Windows 형식인 CRLF가 섞이지 않도록 편집기와 GIT 설정 등을 확인한다.

4. 이름 (Naming)

4.1 식별자에는 영문/숫자/언더스코어만 허용

[identifier-char-scope]

변수명, 클래스명, 메서드명 등에는 영어와 숫자만을 사용한다. 상수에는 단어 사이의 구분을 위하여 언더스코어(_)를 사용한다. 정규표현식 `[^A-Za-z0-9_]`에 부합해야 한다.

4.2 한국어 발음대로의 표기 금지

[avoid-korean-pronounce]

식별자의 이름을 한글 발음을 영어로 옮겨서 표기하지 않는다. 한국어 고유명사는 예외이다.

- 나쁜 예 : `moohyungJasan` (무형자산)
- 좋은 예 : `intangibleAssets` (무형자산)

4.3 대문자로 표기할 약어 명시

[list-uppercase-abbr]

클래스명, 변수명에 쓰일 단어 중 모든 글자를 대문자로 표기할 약어의 목록을 프로젝트별로 명시적으로 정의한다.

약어의 대소문자 표기는 JDK의 클래스나 라이브러리들 사이에서도 일관된 규칙이 없다. `javax.xml.bind.annotation.XmlElement`처럼 약어를 소문자로 표기하기도 하고, `java.net.HttpURLConnection`처럼 한 클래스 안에서 단어별로 다르게 쓰기도 했다. 그러나 단일 프로젝트에서는 규칙이 명확하지 않으면 같은 단어의 조합을 다른 이름으로 표기할 수 있어서 혼동을 유발한다.

약어가 클래스명에서 대문자로 들어가면 단어 간의 구분을 인지하기에 불리하다. 약어가 연속된 경우 더욱 가독성을 해친다. 예를 들면 XMLRPCHTTPAPIURL과 같은 경우이다. 그래서 기본 정책으로는 약어의 중간단어를 소문자로 표기하고 프로젝트별로 모두 대문자로 표기할 약어의 목록을 명시하는 방식이 가독성을 높이고 규칙을 단순화하는데 유리하다. 즉 프로젝트 내에서 정의한 단어 목록이 없다면 'XmlRpcHttpApiUrl'과 같이 쓴다.

좋은 예. HTTP + API + URL 의 클래스 이름의 경우

- 대문자로 표기할 약어의 목록을 정의하지 않는 경우 : HttpApiUrl
- API만 대문자로 표기할 약어의 목록에 있을 경우 : HttpAPIUrl
- HTTP, API, URL이 대문자로 표기할 약어의 목록에 있을 경우 : HTTPAPIURL

4.4 패키지 이름은 소문자로 구성

[package-lowercase]

패키지 이름은 소문자를 사용하여 작성한다. 단어별 구문을 위해 언더스코어(_)나 대문자를 섞지 않는다.

나쁜 예.

```
package com.navercorp.apiGateway

package com.navercorp.api_gateway
```

좋은 예.

```
package com.navercorp.apigateway
```

4.5 클래스/인터페이스 이름에 대문자 카멜표기법 적용

[class-interface-lower-camelcase]

클래스 이름은 단어의 첫 글자를 대문자로 시작하는 대문자 카멜표기법(Upper camel case)을 사용한다. 파스칼표기법(Pascal case)으로도 불린다.

나쁜 예.

```
public class reservation

public class Accesstoken
```

좋은 예.

```
public class Reservation

public class AccessToken
```

4.6 클래스 이름에 명사 사용

[class-noun]

클래스 이름은 명사나 명사절로 짓는다.

4.7 인터페이스 이름에 명사/형용사 사용

[interface-noun-adj]

인터페이스(interface)의 이름은 클래스 이름은 명사/명사절로 혹은 형용사/형용사절로 짓는다.

좋은 예.

```
public interface RowMapper {

public interface AutoClosable {
```

4.8 테스트 클래스는 'Test'로 끝남

[test-class-suffix]

JUnit 등으로 작성한 테스트 코드를 담은 클래스는 'Test'을 마지막에 붙인다.

좋은 예.

```
public class WatcherTest {
```

4.9 메서드 이름에 소문자 카멜표기법 적용

[method-lower-camelcase]

메서드의 이름에는 첫 번째 단어를 소문자로 작성하고, 이어지는 단어의 첫 글자를 대문자로 작성하는 소문자 카멜표기법(Lower camel case)를 사용한다. 테스트 클래스의 메서드 이름에서는 언더스코어를 허용한다.

4.10 메서드 이름은 동사/전치사로 시작

[method-verb-preposition]

메서드명은 기본적으로는 동사로 시작한다. 다른 타입으로 전환하는 메서드나 빌더 패턴을 구현한 클래스의 메서드에는 전치사를 쓸 수 있다.
좋은 예

- 동사사용 : `renderHtml()`
- 전환메서드의 전치사 : `toString()`
- Builder 패턴 적용한 클래스의 메서드의 전치사 : `withUserId(String id)`

4.11 상수는 대문자와 언더스코어로 구성

[constant_uppercase]

상태를 가지지 않는 자료형이면서 `static final`로 선언되어 있는 필드일 때를 상수로 간주한다. 상수 이름은 대문자로 작성하며, 복합어는 언더스코어(`_`)를 사용하여 단어를 구분한다.

좋은 예.

```
public final int UNLIMITED = -1;
public final String POSTAL_CODE_EXPRESSION = “POST”;
```

4.12 변수에 소문자 카멜표기법 적용

[var-lower-camelcase]

상수가 아닌 클래스의 멤버변수/지역변수/메서드 파라미터에는 소문자 카멜표기법(Lower camel case)을 사용한다.

나쁜 예.

```
private boolean Authorized;
private int AccessToken;
```

좋은 예.

```
private boolean authorized;
private int accessToken;
```

4.13 임시 변수 외에는 1 글자 이름 사용 금지

[avoid-1-char-var]

메서드 블럭 범위 이상의 생명 주기를 가지는 변수에는 1글자로 된 이름을 쓰지 않는다. 반복문의 인덱스나 람다 표현식의 파라미터 등 짧은 범위의 임시 변수에는 관례적으로 1글자 변수명을 사용할 수 있다.

나쁜 예.

```
HtmlParser p = new HtmlParser();
```

좋은 예.

```
HtmlParser parser = new HtmlParser();
```

5. 선언 (Declarations)

클래스, 필드, 메서드, 변수값, import문 등의 소스 구성요소를 선언할 때 고려해야할 규칙이다.

5.1 소스파일당 1개의 탑레벨 클래스를 담기

[1-top-level-class]

탑레벨 클래스(Top level class)는 소스 파일에 1개만 존재해야 한다. (탑레벨 클래스 선언의 컴파일타임 에러 체크에 대해서는 Java Language Specification 7.6 참조)

나쁜 예.

```
public class LogParser {
}

class LogType {
}
```

좋은 예.

```
public class LogParser {
    // 굳이 한 파일안에 선언해야 한다면 내부 클래스로 선언
    class LogType {
    }
}
```

5.2 static import에만 와일드 카드 허용

[avoid-star-import]

클래스를 import할때는 와일드카드(*) 없이 모든 클래스명을 다 쓴다. static import에서는 와일드카드를 허용한다.

나쁜 예.

```
import java.util.*;
```

좋은 예.

```
import java.util.List;
import java.util.ArrayList;
```

5.3 제한자 선언의 순서

[modifier-order]

클래스/메서드/멤버변수의 제한자는 Java Language Specification에서 명시한 아래의 순서로 쓴다.

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

(Java Language Specification - Chapter 18. Syntax 참조)

5.4 애너테이션 선언 후 새줄 사용

[newline-after-annotation]

클래스, 인터페이스, 메서드, 생성자에 붙는 애너테이션은 선언 후 새줄을 사용한다. 이 위치에서도 파라미터가 없는 애너테이션 1개는 같은 줄에 선언할 수 있다.

좋은 예.

```
@RequestMapping("/guests")
public void findGuests() {}
```

좋은 예.

```
@Override public void destroy() {}
```

5.5 한 줄에 한 문장

[1-state-per-line]

문장이 끝나는 ; 뒤에는 새줄을 삽입한다. 한 줄에 여러 문장을 쓰지 않는다.

나쁜 예.

```
int base = 0; int weight = 2;
```

좋은 예.

```
int base = 0;
int weight = 2;
```

5.6 하나의 선언문에는 하나의 변수만

[1-var-per-declaration]

변수 선언문은 한 문장에서 하나의 변수만을 다룬다.

나쁜 예.

```
int base, weight;
```

좋은 예.

```
int base;
int weight;
```

5.7 배열에서 대괄호는 타입 뒤에 선언

[array-square-after-type]

배열 선언에 오는 대괄호([ ])는 타입의 바로 뒤에 붙인다. 변수명 뒤에 붙이지 않는다.

나쁜 예.

```
String names[];
```

좋은 예.

```
String[] names;
```

5.8 `long`형 값의 마지막에 `L`붙이기

[long-value-suffix]

long형의 숫자에는 마지막에 대문자 'L'을 붙인다. 소문자 'l'보다 숫자 '1'과의 차이가 커서 가독성이 높아진다.

나쁜 예.

```
long base = 54423234211l;
```

좋은 예.

```
long base = 54423234211L;
```

5.9 특수 문자의 전용 선언 방식을 활용

[special-escape]

\b, \f, \n,\r,\t, `\", \\ 와 같이 특별히 정의된 선언 방식이 있는 특수 문자가 있다. 이런 문자들은 숫자를 이용한 \008 이나 `₩u0008`와 같은 숫자를 넣은 선언보다 전용 방식을 활용한다.

나쁜 예.

```
System.out.println("---\012---");
```

좋은 예.

```
System.out.println("---\n---");
```

5.10 파일주석으로 저작권/라이센스를 명시

[comment-copyright-licence]

해당 프로젝트의 저작권/라이센스 선언을 파일주석으로 포함한다.

좋은 예.

```
/**
 * Copyright 2015 Naver Corp. All rights Reserved.
 * Naver PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */
```

6. 들여쓰기 (Indentation)

들여쓰기는 코드의 계층을 구분하기 위해 추가하는 문자이다.

6.1 하드탭 사용

[indentation-tab]

탭(tab) 문자를 사용하여 들여쓴다. 탭 대신 스페이스를 사용하지 않는다. 이를 잘 준수하기 위해서 스페이스와 탭을 구별해서 보여주도록 에디터를 설정한다.

6.2 탭의 크기는 4개의 스페이스

[4-spaces-tab]

1개의 탭의 크기는 스페이스 4개와 같도록 에디터에서 설정한다.

6.3 블럭 들여쓰기

[block-indentation]

클래스, 메서드, 제어문 등의 코드 블럭이 생길 때마다 1단계를 더 들여쓴다.

7. 중괄호 (Braces)

중괄호({,}) 는 클래스, 메서드, 제어문의 블럭을 구분한다.

7.1 K&R 스타일로 중괄호 선언

[braces-knr-style]

클래스 선언, 메서드 선언, 조건/반복문 등의 코드 블럭을 감싸는 중괄호에 적용되는 규칙이다. 중괄호 선언은 K&R 스타일(Kernighan and Ritchie style)을 따른다. 줄의 마지막에서 시작 중괄호`{`를 쓰고 열고 새줄을 삽입한다. 블럭을 마친 후에는 새줄 삽입 후 중괄호를 닫는다.

나쁜 예.

```
public class SearchConditionParser
{
    public boolean isValidExpression(String exp)
    {

        if (exp == null)
        {
            return false;
        }

        for (char ch : exp.toCharArray())
        {
             ....
        }

        return true;
    }
}
```

좋은 예.

```
public class SearchConditionParser {
    public boolean isValidExpression(String exp) {

        if (exp == null) {
            return false;
        }

        for (char ch : exp.toCharArray()) {
            ....
        }

        return true;
    }
}
```

7.2 닫는 중괄호와 같은 줄에 `else`, `catch`, `finally`, `while` 선언

[sub-flow-after-brace]

아래의 키워드는 닫는 중괄호(}) 와 같은 줄에 쓴다.

- else
- catch, finaly
- do-while 문에서의 while

나쁜 예.

```
if (line.startWith(WARNING_PREFIX)) {
    return LogPattern.WARN;
}
else if (line.startWith(DANGER_PREFIX)) {
    return LogPattern.DANGER;
}
else {
    return LogPattern.NORMAL;
}
```

좋은 예.

```
if (line.startWith(WARNING_PREFIX)) {
    return LogPattern.WARN;
} else if (line.startWith(DANGER_PREFIX)) {
    return LogPattern.NORMAL;
} else {
    return LogPattern.NORMAL;
}
```

나쁜 예.

```
try {
    writeLog();
}
catch (IOException ioe) {
    reportFailure(ioe);
}
finally {
    writeFooter();
}
```

좋은 예.

```
try {
    writeLog();
} catch (IOException ioe) {
    reportFailure(ioe);
} finally {
    writeFooter();
}
```

나쁜 예.

```
do {
```

```
    write(line);
    line = readLine();
}
while (line != null);
```

좋은 예.

```
do {
    write(line);
    line = readLine();
} while (line != null);
```

7.3 빈 블럭에 새줄 없이 중괄호 닫기 허용

[permit-concise-empty-block]

내용이 없는 블럭을 선언할 때는 같은 줄에서 중괄호를 닫는 것을 허용한다.

좋은 예.

```
public void close() {}
```

7.4 조건/반복문에 중괄호 필수 사용

[need-braces]

조건, 반복문이 한 줄로 끝더라도 중괄호를 활용한다. 이 문서에 언급된 중괄호의 전후의 공백, 제어문 앞 뒤의 새줄 규칙도 함께 고려한다.

나쁜 예.

```
if (exp == null) return false;

for (char ch : exp.toCharArray()) if (ch == 0) return false;
```

좋은 예.

```
if (exp == null) {
    return false;
}

for (char ch : exp.toCharArray()) {

    if (ch == 0) {
        return false;
    }

}
```

8. 줄바꿈 (Line-wrapping)

줄바꿈은 작성한 명령어가 줄 너비를 초과했을 경우 코드 가독성을 위해서 강제로 줄을 바꾸는 것을 말한다.

8.1 최대 줄 너비는 120

[line-length-120]

최대 줄 사용 너비는 120자까지 가능하다.

8.2 package,import 선언문은 한 줄로

[1-line-package-import]

package,import 선언문 중간에서는 줄을 바꾸지 않는다. 최대 줄수를 초과하더라도 한 줄로 쓴다.

8.3 줄바꿈후 추가 들여쓰기

[indentation-after-line-wrapping]

줄바꿈 이후 이어지는 줄에서는 최초 시작한 줄에서보다 적어도 1단계의 들여쓰기를 더 추가한다.

8.4 줄바꿈 허용 위치

[line-wrapping-position]

최대 길이를 초과할 때 줄을 바꾸는 위치는 다음 중의 하나로 한다.

- extends 선언 후
- implements 선언 후
- throws 선언 후
- 시작 소괄호(() 선언 후
- 콤마(,) 후
- . 전
- 연산자 전
  - +, -, *, /, %
  - ==, !=, >=, >,<=, <, &&, ||
  - &, |, ^, >>>, >>, <<, ?
  - instanceof

좋은 예.

```
// 짧은 선언으로 표현했으나 줄이 길어진 상황으로 가정함

public boolen isAbnormalAccess (
    User user, AccessLog log) {
```

```
    String message = user.getId() + "¦" ¦ log.getPrefix()
        + "¦" + SUFFIX;
}
```

9. 빈 줄(Blank lines)

빈 줄은 명령문 그룹의 영역을 표시하기 위하여 사용한다.

9.1 package 선언 후 빈 줄 삽입

[blankline-after-package]

좋은 예.

```
package com.naver.lucy.util;

import java.util.Date;
```

9.2 import 선언의 순서와 빈 줄 삽입

[import-grouping]

import 구절은 아래와 같은 순서로 선언한다.

A. static imports

B. JDK의 클래스

- java.
- javax.

C. 외부 라이브러리의 클래스

- org.
- net.
- com.
- 그 외의 클래스

D. 네이버 내부에서 만든 클래스

- com.nhncorp.
- com.navercorp.
- com.naver.

A,B,C,D 그룹의 사이에는 빈 줄을 삽입한다. 그룹 내부에서 세부적인 구분을 위해 빈줄을 삽입하는 것도 허용한다.

좋은 예.

```
import java.util.Date;
import java.util.List;

import javax.naming.NamingException;

import org.apache.commons.logging.Log;
```

```
import org.apache.commons.logging.LogFactory;
import org.springframework.util.Assert;

import com.google.common.base.Function;

import com.naver.lucy.util.AnnotationUtils;
```

이 규칙은 대부분 IDE에서 자동으로 정리해주는 대로 쓰기 때문에 IDE 설정을 일치시키는데 신경을 써야 한다.

9.3 메소드 사이에 빈 줄 삽입

[blankline-between-methods]

메서드의 선언이 끝난 후 다음 메서드 선언이 시작되기 전에 빈줄을 삽입한다.

좋은 예.

```
public void setId(int id) {
    this.id = id;
}

public void setName(String name) {
    this.name = name;
}
```

10. 공백 (Whitespace)

10.1 공백으로 줄을 끝내지 않음

[no-trailing-spaces]

빈줄을 포함하여 모든 줄은 탭이나 공백으로 끝내지 않는다.

10.2 대괄호 뒤에 공백 삽입

[space-after-bracket]

닫는 대괄호(]) 뒤에 `;`으로 문장이 끝나지 않고 다른 선언이 올 경우 공백을 삽입한다.

나쁜 예.

```
int[]masks = new int[]{0, 1, 1};
```

좋은 예.

```
int[] masks = new int[] {0, 1, 1};
```

10.3 중괄호의 시작 전, 종료 후에 공백 삽입

[space-around-brace]

여는 중괄호({) 앞에는 공백을 삽입한다. 닫는 중괄호(}) 뒤에 `else` ,`catch` 등의 키워드가 있을 경우 중괄호와 키워드 사이에 공백을 삽입한다.

좋은 예.

```
public void printWarnMessage(String line) {
    if (line.startsWith(WARN_PREFIX)) {
        ...
    } else {
        ...
    }
}
```

10.4 제어문 키워드와 여는 소괄호 사이에 공백 삽입

[space-between-keyword-parentheses]

`if`, `for`, `while`, `catch`, `synchronized`, `switch`와 같은 제어문 키워드의 뒤에 소괄호((,)`)를 선언하는 경우, 시작 소괄호 앞에 공백을 삽입한다.

좋은 예.

```
if (maxLine > LIMITED) {
    return false;
}
```

10.5 식별자와 여는 소괄호 사이에 공백 미삽입

[no-space-between-identifier-parentheses]

식별자와 여는 소괄호(() 사이에는 공백을 삽입하지 않는다. 생성자와 메서드의 선언, 호출, 애너테이션 선언 뒤에 쓰이는 소괄호가 그에 해당한다.

나쁜 예.

```
public StringProcessor () {} // 생성자

@Cached ("local")
public String removeEndingDot (String original) {
    assertNotNull (original);
    ...
}
```

좋은 예.

```
public StringProcessor() {} // 생성자

@Cached("local")
public String removeEndingDot(String original) {
    assertNotNull(original);
    ...
}
```

10.6 타입 캐스팅에 쓰이는 소괄호 내부 공백 미삽입

[no-space-typecasting]

타입캐스팅을 위해 선언한 소괄호의 내부에는 공백을 삽입하지 않는다.

나쁜 예.

```
String message = ( String ) rawLine;
```

좋은 예.

```
String message = (String)rawLine;
```

10.7 제네릭스 산괄호의 공백 규칙

[generic-whitespace]

제네릭스(Generics) 선언에 쓰이는 산괄호(<,>) 주위의 공백은 다음과 같이 처리한다.

- 제네릭스 메서드 선언 일 때만 < 앞에 공백을 삽입한다.
- < 뒤에 공백을 삽입하지 않는다.
- > 앞에 공백을 삽입하지 않는다.
- 아래의 경우를 제외하고는 `>`뒤에 공백을 삽입한다.

- 메서드 레퍼런스가 바로 이어질 때
- 여는 소괄호('(')가 바로 이어질 때
- 메서드 이름이 바로 이어질 때

좋은 예.

```
public static <A extends Annotation> A find(AnnotatedElement elem, Class<A> type) { // 제네릭스 메서드 선언
    List<Integer> l1 = new ArrayList<>(); // '(' 가 바로 이어질때
    List<String> l2 = ImmutableList.Builder<String>::new; // 메서드 레퍼런스가 바로 이어질 때
    int diff = Util.<Integer, String>compare(l1, l2); // 메서드 이름이 바로 이어질 때
}
```

10.8 콤마/구분자 세미콜론의 뒤에만 공백 삽입

[space-after-comma-semicolon]

콤마(,)와 반복문(while, for)의 구분자로 쓰이는 세미콜론(;)에는 뒤에만 공백을 삽입한다.

나쁜 예.

```
for (int i = 0;i < length;i++) {
    display(level,message,i)
}
```

좋은 예.

```
for (int i = 0; i < length; i++) {
    display(level, message, i)
}
```

10.9 콜론의 앞 뒤에 공백 삽입

[space-around-colon]

반복문과 삼항연산자에서 콜론(:)의 앞 뒤에는 공백을 삽입한다. 라벨 선언 뒤에는 아무런 문자열이 없으므로 앞에만 공백을 삽입한다.

좋은 예.

```
for (Customer customer : visitedCustomers) {
    AccessPattern pattern = isAbnormal(accessLog) ? AccessPattern.ABUSE : AccessPattern.NORMAL;
    int grade = evaluate(customer, pattern);

    switch (grade) {
        case GOLD :
            sendSms(customer);
        case SILVER :
            sendEmail(customer);
        default :
            inreasePoint(customer)
    }
}
```

10.10 이상/삼항 연산자의 앞 뒤에 공백 삽입

[space-around-binary-ternary-operator]

이항/삼항 연산자의 앞 뒤에는 공백을 삽입한다.

좋은 예.

```
if (pattern == Access.ABNORMAL) {
    return 0;
}

finalScore += weight * rawScore - absentCount;

if (finalScore > MAX_LIMIT) {
    return MAX_LIMIT;
}
```

10.11 단항 연산자와 연산 대상 사이에 공백을 미삽입

[no-space-increament-decrement-operator]

단항 연산자와 연산 대상의 사이에는 공백을 삽입하지 않는다.

- 전위 연산자 : 연산자 뒤에 공백을 삽입하지 않는다.
  - 전위 증감/감소 연산자 : ++,--
  - 부호로 쓰이는 +, -
  - NOT 연산자 : ~, !
- 후위 연산자 : 연산자 앞에 공백을 삽입하지 않는다.
  - 후위 증감/감소 연산자 : ++,--

나쁜 예.

```
int point = score[++ index] * rank -- * - 1;
```

좋은 예.

```
int point = score[++index] * rank-- * -1;
```

10.12 주석문 기호 전후의 공백 삽입

[space-around-comment]

주석의 전후에는 아래와 같이 공백을 삽입한다.

- 명령문과 같은 줄에 주석을 붙일 때 // 앞
- 주석 시작 기호 // 뒤
- 주석 시작 기호 /* 뒤

• 블록 주석을 한 줄로 작성시 종료 기호 */ 앞

좋은 예.

```
/*
 * 공백 후 주석내용 시작
 */

System.out.print(true); // 주석 기호 앞 뒤로 공백

/* 주석내용 앞에 공백, 뒤에도 공백 */
```

Appendix A. Checkstyle 사용법

Checkstyle 은 코딩컨벤션 검사 도구이다. 이 가이드에서 안내하는 규칙을 검사하는 checkstyle 규칙 설정 파일을 제공한다.

A.1 필수 버전

Checkstyle 6.16 버전 이상을 사용해야 한다. 빌드도구와 IDE에서 권장한 버전을 쓰고 있는지 확인을 해야 한다. 6.16 미만의 버전에서는 들여쓰기 규칙을 검사할 때 버그가 있다. 부득이하게 그 6.16 미만의 버전을 써야 할 때는 `<module name="Indentation">`으로 시작하는 선언을 삭제하고 쓴다.

A.2 규칙 설정 파일 다운로드

https://oss.navercorp.com/engineering/coding-convention/blob/master/java/config/naver-checkstyle-rules.xml 에서 다운로드한다.

A.3 검사 실행

실제 프로젝트에서 쓸 때는 IDE나 빌드도구와 연동해서 사용하는것을 권장한다. 그러나 규칙 파일을 커스터마이징 했을 때에는 Checkstyle을 독립적으로 실행을 해서 IDE나 빌드스크립트와 연동하기 전에 먼저 테스트해 보기를 권장한다.

독립적으로 실행할 때는 https://sourceforge.net/projects/checkstyle/files/checkstyle/ 에서 jar 파일을 다운로드하고 아래와 같이 실행한다.

checkstyle 독립실행.

```
java -jar checkstyle-6.17-all.jar -c [규칙파일] [소스폴더]
```

`naver-checkstyle-rules.xml`을 실행할 때는 `-Dconfig_loc` 속성 지정이 필수적으로 필요하다. 예외선언 파일의 위치를 IDE와 Maven 빌드 양쪽에서 참조하기 위해서 필요한 속성이다.

naver-checkstyle-rules.xml 독립 실행 사례.

```
java -Dconfig_loc=config -jar checkstyle-6.17-all.jar -c config/naver-checkstyle-rules.xml src
```

A.4 관련 규칙 확인 방법

규칙 파일에서는 규칙의 ID를 주석으로 달았다. 경고 메시지에서도 규칙의 ID를 표시한다.

```
<!-- [1-line-package-import]-->
<module name="NoLineWrap">
    <property name="tokens" value="PACKAGE_DEF, IMPORT"/>
        <message key="no.line.wrap"
            value="[1-line-package-import] {0} statement should not be line-wrapped."/>
</module>
```

규칙의 ID를 이 가이드에서 검색해서 해당 규칙을 찾는다. HTML문서에서는 #뒤에 규칙의 ID를 붙여서 바로 이동할 수도 있다. 예를 들면 [need-brace]에 규칙에 대한 설명은 http://docs.navercorp.com/coding-convention/java.html#need-braces 링크를 걸 수 있다. 온라인 코드 리뷰를 할 때도 이런 링크를 활용할 수 있다.

A.5 검사할 수 없는 규칙

아래 규칙은 Checkstyle로 검사할 수 없다.

- [avoid-korean-pronounce] : 한국어 발음대로의 표기 금지
- [class-noun] : 클래스 이름에 명사 사용
- [interface-noun-adj] : 인터페이스 이름에 명사/형용사 사용
- [method-verb-preposition] : 메서드 이름은 동사/전치사로 시작
- [test-class-suffix] : 테스트 클래스는 'Test'로 끝남
- [comment-copyright-licence] : 파일 주석으로 저작권/라이센스를 명시
- [space-after-bracket] : 대괄호 뒤에 공백 삽입
- [space-around-comment] : 주석문 기호 전후의 공백 삽입

A.6 검사 대상에서 제외하기

제외 대상을 별도의 파일로 선언

`naver-checkstyle-rules.xml`의 설정에서는 `naver-checkstyle-suppressions.xml`라는 파일이 같은 디렉토리에 있을 경우, 해당 파일을 참조하여 검사 대상에서 일부 파일을 제외한다.

naver-checkstyle-suppressions.xml 선언의 예.

```
<?xml version="1.0"?>
<!DOCTYPE suppressions PUBLIC
"-//Puppy Crawl//DTD Suppressions 1.1//EN"
"http://www.puppycrawl.com/dtds/suppressions_1_1.dtd">

<suppressions>
    <suppress files="UserController.java" checks=".*"/>
    <suppress files="UserService.java" checks=".*"/>
</suppressions>
```

파일명을 정규식으로 설정하는 것도 가능하다. 자세한 선언 방법은 http://checkstyle.sourceforge.net/config_filters.html#SuppressionFilter 을 참조한다.

제외할 영역을 주석으로 표시

`naver-checkstyle-rules.xml`의 설정에서는 아래와 같은 주석문을 인식한다.

- `// @checkstyle:off` : 다음 행부터 검사 대상에서 제외
- `// @checkstyle:on` : 다음 행부터 검사 대상에 포함
- `// @checkstyle:ignore` // 같은 행의 소스를 검사하지 않음

검사 대상에서 제외하는 주석 사용 예.

```
public class MyCAO { // @checkstyle:ignore 외부에 배포된 라이브러리
    public static final String SYSTEM_ID = "MD23D2";
    // @checkstyle:off
    public String CONNECT_URL;
    public String USER_ID;
    // @checkstyle:on
}
```

A.7 커스터마이징

`naver-checkstyle-rules.xml`에 정의된 규칙을 프로젝트에서 더 추가하거나, 수정해서 쓰려고 할 때 참고할만한 정보를 정리한다.

대문자로 표기할 약어를 추가

[list-uppercase-abbr] 규칙에 따라서 대문자로 표기할 약어는 따로 명시해야 한다. `naver-checkstyle-rules.xml` 파일에서 `allowedAbbreviations` 속성에 해당 단어를 추가한다. 레거시 코드에서 많이 쓰이는 `BO`, `DAO`는 이미 포함되어 있다.

대문자로 표기할 약어를 설정 파일에 명시.

```
<!-- [list-uppercase-abbr] -->
<module name="AbbreviationAsWordInName">
    <property name="ignoreFinal" value="false"/>
    <property name="allowedAbbreviationLength" value="1"/>
    <message key="abbreviation.as.word"
    value="[list-uppercase-abbr] Abbreviation in name ''{0}'' must contain no more than {1}"/>
    <property name="allowedAbbreviations" value="DAO,BO"/>
</module>
```

설정 파일을 고치지 않으려면 `@checkstyle:ignore`와 같은 주석을 이용할 수도 있다.

들여쓰기에 탭 대신 스페이스 사용

아래 선언을 `naver-checkstyle-rules.xml` 에서 삭제한다.

탭문자로 들여쓰도록 검사하는 규칙 선언.

```
<!-- [indentation-tab] -->
<module name="RegexpSinglelineJava">
    <property name="format" value="^\t* "/>
    <property name="message" value="[indentation-tab] Indent must use tab characters"/>
    <property name="ignoreComments" value="true"/>
</module>
```

대신 아래와 같이 탭문자가 포함되어 있을때 경고를 보내는 선언을 추가한다.

스페이스로 들여쓰도록 검사하는 규칙 선언.

```
<module name="FileTabCharacter">
    <property name="eachLine" value="true"/>
</module>
```

Appendix B. 빌드 도구 설정

B.1 Maven

`pom.xml`에서 아래와 같은 선언을 추가한다.

인코딩 지정

소스 파일의 인코딩을 maven-compiler-plugin과 maven-resources-plugin에서 UTF-8로 지정한다.

maven-compiler-plugin에서 엔코딩 지정.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.1</version>
    <configuration>
        <source>1.8</source>
        <target>1.8</target>
        <encoding>UTF-8</encoding>
    </configuration>
</plugin>
```

maven-resources-plugin에서 엔코딩 지정.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-resources-plugin</artifactId>
    <version>2.7</version>
    <configuration>
        <encoding>UTF-8</encoding>
    </configuration>
</plugin>
```

Checkstyle 플러그인 설정

<pluginManagement/> 선언에 maven-checkstyle-plugin의 버전과 checkstyle의 버전을 명시한다. checkstyle의 버전은 반드시 6.17 ~ 8.0 사이로 별도로 지정한다.

pluginManagement 선언.

```
<pluginManagement>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-checkstyle-plugin</artifactId>
            <version>2.17</version>
            <configuration>
                <configLocation>naver-style-checks.xml</configLocation>
                <sourceDirectory>${project.build.sourceDirectory}</sourceDirectory>
                <propertyExpansion>config_loc=${basedir}</propertyExpansion>
                <!-- naver-checkstyle-suppressions.xml 파일을 config_loc에서 찾는다 -->
```

```
            </configuration>
            <dependencies>
                <dependency>
                    <groupId>com.puppycrawl.tools</groupId>
                    <artifactId>checkstyle</artifactId>
                    <version>6.17</version>
                </dependency>
            </dependencies>
        </plugin>
    </plugins>
</pluginManagement>
```

<build> → <plugins> 태그 아래에 다음과 같은 선언을 추가한다.

plugin 선언.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-checkstyle-plugin</artifactId>
</plugin>
```

`mvn checkstyle:checkstyle`로 실행하면 검사가 시작된다. `mvn site` 명령으로 검사를 실행하고 싶다면 <reporting> → `<plugins>`태그 아래에 위의 선언을 추가한다.

B.2 Gradle

인코딩 지정

Java plugin의 속성으로 인코딩을 지정한다. 아래와 같이 여러 방법이 가능하다.

java plugin의 속성에서 인코딩을 지정.

```
apply plugin: 'java'

...

// 방법1
compileJava.options.encoding = 'UTF-8'
compileTestJava.options.encoding = 'UTF-8'

// 방법 2
[compileJava, compileTestJava]*.options*.encoding = 'UTF-8'

// 방법 3
tasks.withType(Compile) {
    options.encoding = 'UTF-8'
}
```

Checkstyle 플러그인 설정

아래와 같이 `build.gradle`에 선언한다.

```
apply plugin: 'checkstyle'
```

```
checkstyle {
    configFile = file("${rootDir}/naver-style-checks.xml")
    configProperties = [config_loc: "${rootDir}"]
}
```

config_loc 속성은 `naver-style-supressions.xml`을 찾을 때 참조된다.

Appendix C. 편집기 설정

이 가이드의 규칙을 지키는데 도움이 되는 코드 편집기와 뷰어 설정 방법을 정리한다.

C.1 Eclipse

공백 문자 보이기 설정

탭과 스페이스, 줄바꿈 문자 등을 눈에 보이게 표시한다. `Windows > Preferences > Editors > Text Editors`에서 `Show whitespaces characters`를 선택한다.

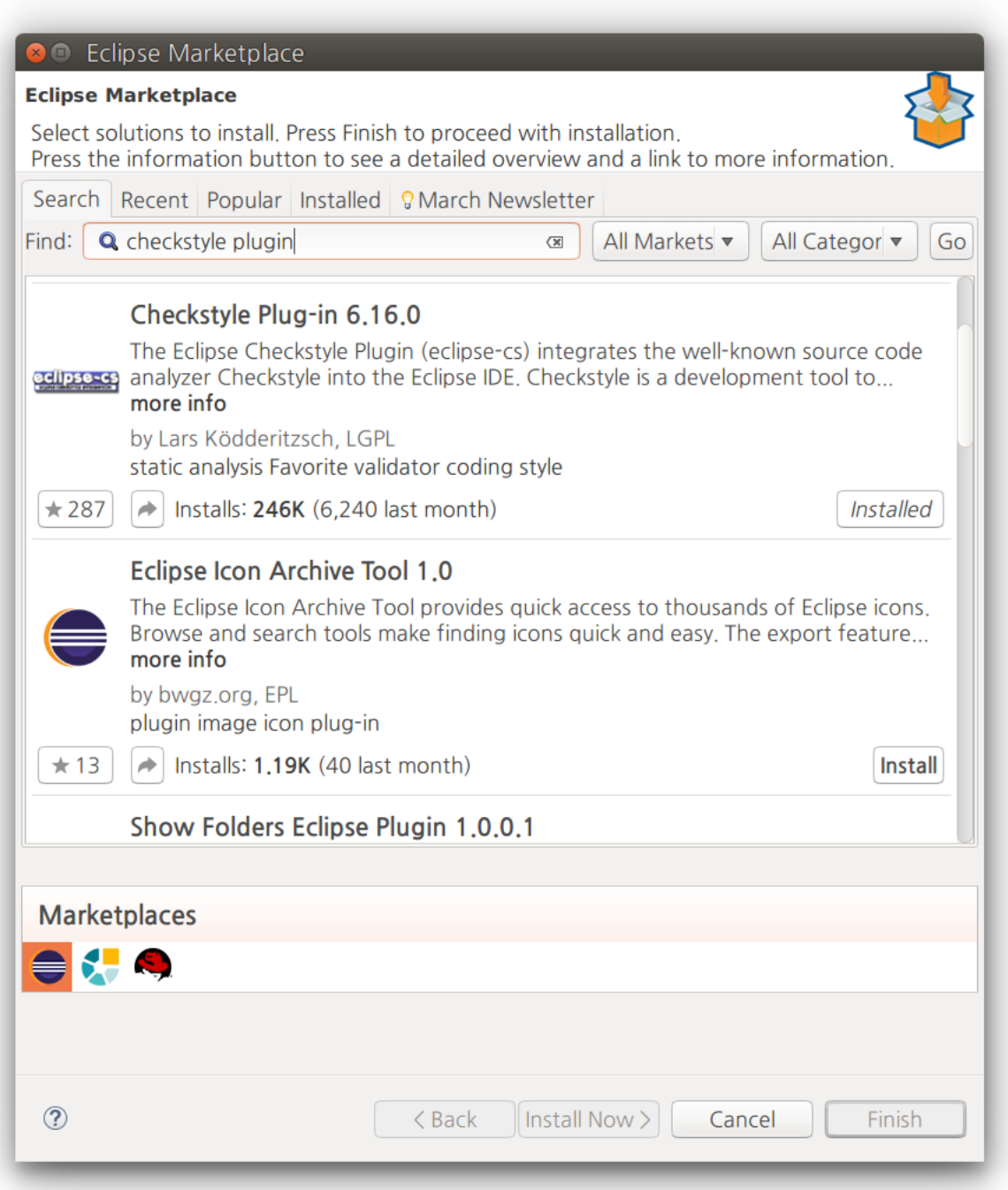
Checkstyle 설정

플러그인 설치

1. 메뉴에서 `Help > Eclise Marketplace`에서 'Checkstyle plugin'으로 검색한다.

2. Eclipse Checkstyle plugin(eclipse-cs)를 찾아서 [`Install`] 버튼을 누른다.

이미 플러그인을 설치했다면 권장하는 버전이 설치되어 있는지 확인을 한다. `Help > Installation Details` 메뉴에서 이를 확인할 수 있다. Eclipse Checkstyle plugin의 버전은 의존하는 checkstyle의 버전과 동일하게 표기되고 있다. `naver-checkstyle-rules.xml`을 수정없이 쓸 때는 Eclipse Checkstyle plugin의 버전도 16.0 이상인지 확인을 한다.



규칙 파일 불러오기

1. `Checkstyle` 메뉴로 이동
   - 여러 프로젝트에서 같은 파일을 활용할 때는 Workspace 전역 규칙으로 불러 올것을 권장한다. `Window` > `Preference` > `Checkstyle` 메뉴로 이동한다.
   - 프로젝트별로 다른 설정을 쓸 때는 프로젝트별 설정을 해야한다. `Project` > `Properties` > `Checkstyle` 메뉴에서 `Local Check Configurations` 탭으로 이동한다.
2. `Checkstyle` 메뉴에서 설정파일 목록의 오른쪽에 있는 `[New]` 버튼을 누른다.
3. `Checkstyle Configuration Properties` 팝업창의 항목 입력
   - `Type` : `External Configuration File` 혹은 `Project Relative Configuration` 선택

- Location : `[Browse]`버튼을 눌러서 `naver-checkstyle-rules.xml`를 찾아서 선택
- Name : 인식할 수 있는 이름. `naver-checkstyle-rules`를 권장
- Protect Checkstyle Configuration File : 체크

프로젝트별 검사 설정

Project > Properties > Checkstyle 메뉴에서 아래 항목을 설정한다.

- Checkstyle active for this project : 체크
- Write formatter/cleanup config (experimental!) : 체크하지 않음
- Simple - use the following check configuration for all files : 앞 단계에서 설정한 naver-checkstyle-rules 선택
- Excluding from checking
  - files outside sourcxe directories : 체크
  - derived (generated) files : 체크

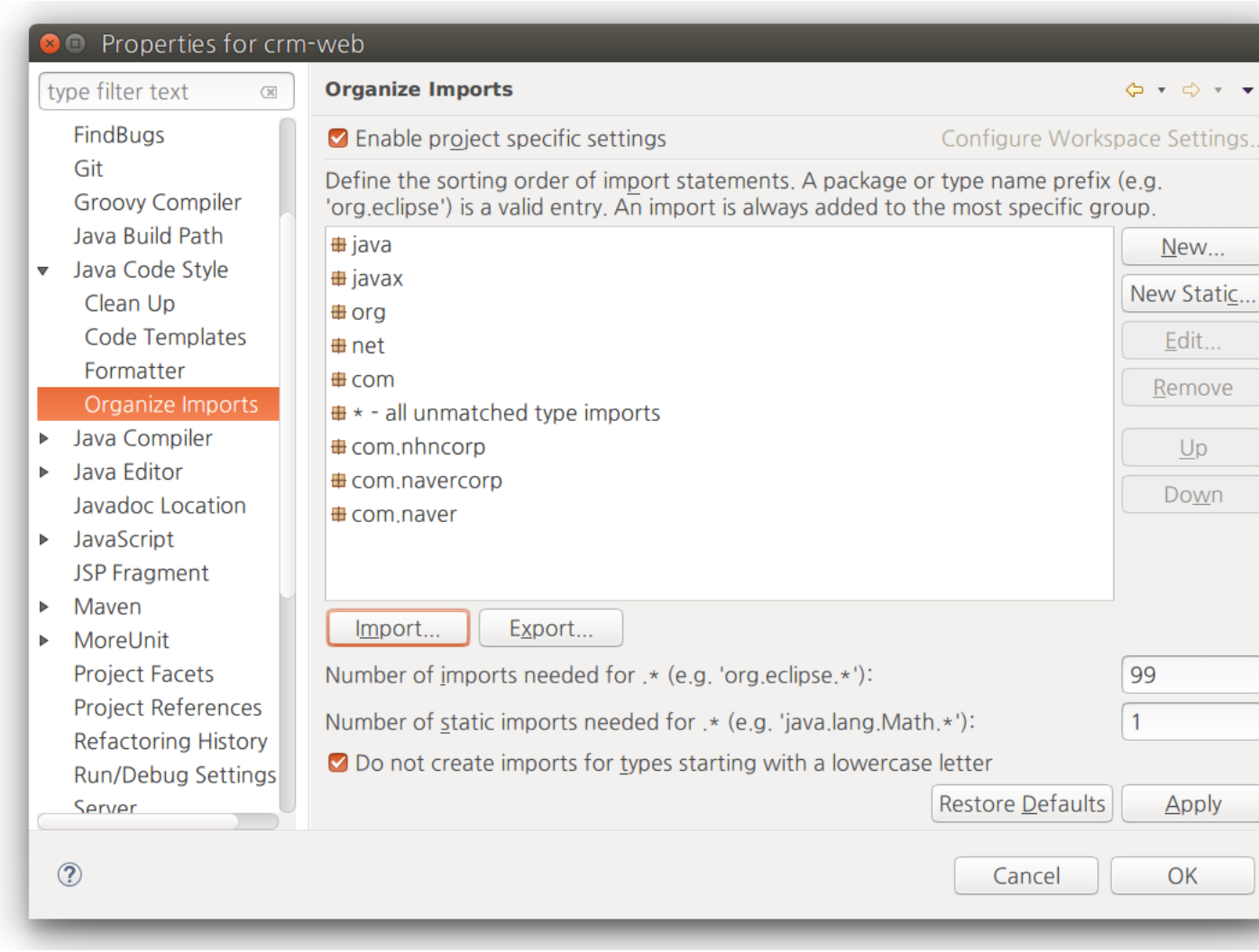
Organize Imports 설정

아래 규칙을 자동으로 지키도록 해준다.

- [avoid-star-import] : static import에만 와일드 카드 허용
- [import-grouping] : import 선언의 순서와 빈 줄 삽입

아래와 같은 순서로 설정한다.

1. importorder 설정 파일 다운로드
   - naver.importorder 파일을 https://oss.navercorp.com/engineering/coding-convention/blob/master/java/config/naver.importorder 에서 다운로드한다.
2. Organize Imports 설정 화면으로 이동
   - Workspace 전역설정은 `Window > Preference > Java > Code style > Organize Imports`로 이동.
   - 프로젝트별 설정은 `Porject > Properties > Java Code style > Organize Imports`로 이동.
3. `Organize Imports`의 항목 입력
   - Enable project specific settings(프로젝트별 설정의 경우) : 체크
   - Define the sorting order of import statement
     - [Import] 버튼을 클릭하여 `naver.importorder`파일 선택
   - Number of imports needed for .* : 99
   - Number of static imports needed for .* : 1



Formatter 설정

들여쓰기, 공백, 줄바꿈 규칙을 지키도록 도와준다.

1. Formatter 설정 파일 다운로드

   - `naver-eclipse-formatter.xml`파일 https://oss.navercorp.com/engineering/coding-convention/blob/master/java/config/naver-eclipse-formatter.xml 에서 다운로드한다.

2. Formatter 메뉴로 이동

   - Workspace 전역설정은 `Window > Preference > Java > Code style > Formatter`에서 이동한다.
   - 프로젝트별 설정은 `Project > Properties > Java Code style > Formatter`로 이동한다.

3. Formatter 항목 설정

   - 프로젝트별 설정의 경우 `Enable project specific settings`를 선택한다.
   - [Import] 버튼을 누른 후, 다운로드 한 naver-eclipse-formatter.xml 파일을 선택한다.
   - `[OK]`버튼을 누른다.

Formatter가 설정되면 코드 편집창에서 `Ctrl + Shift + f` 키로 코드 포멧을 맞출 수 있다. Formatter가 자동으로 맞춰주는 결과가 들지 않을 수도 있기 때문에 선택적으로 사용한다. 예를 들면 줄바꿈 후 들여쓰기를 최소기준인 한 단계보다 깊게 하고 싶을 경우 포멧터를 쓰지 않고 직접 탭문자를 입력할 수도 있다. Eclipse의 포멧터는 Checkstyle에서 검사하는 규칙보다 더 구체적인 규칙으로 코드를 맞춰 주기도 한다.

Save actions 활용

Eclipse에서는 `Ctrl + s`키로 파일을 저장할때 수행하는 동작을 지정할 수 있다. 아래와 같이 메뉴로 이동한다.

- Workspace 전역 설정 : `Window` > `Preference` > `Java` > `Editor Save Actions`
- 프로젝트별 설정 : `Project` > `Properties` > `Java Editor` > `Save Actions`

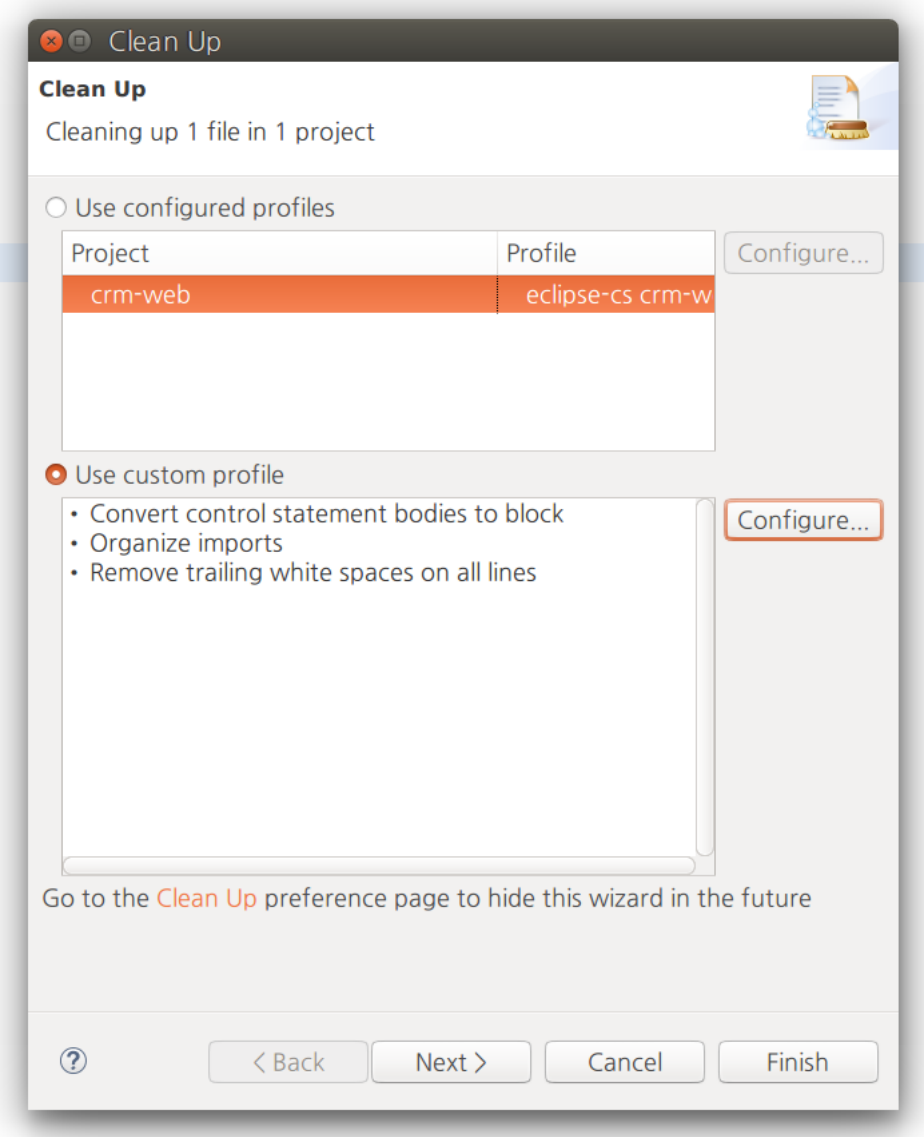
해당 메뉴에서 설정할 수 있는 동작은 아래와 같다.

- `Format source code` : Formatter에 정의된 포메팅을 적용한다. 프로젝트의 상황에 따라서 선택한다.
- `Organize imports` : `Ctrl + Shift + o`를 눌렀을 때와 동일한 동작을 한다.
- `Additional Actions` : 아래 2개의 동작은 설정을 권장한다.
  - `Convert control statement bodies to block` : `{}`가 없는 제어문에 `{}`을 넣어준다. [need-braces] 규칙을 준수하게 해준다.
  - `Remove trailing white spaces on all line` : 줄끝에 붙은 공백을 제거한다. [no-trailing-spaces] 규칙을 준수하게 해준다.

Cleanup 활용

`Save Actions`와 유사하나 프로젝트의 전체 소스에 일괄적으로 적용할 수 있다. 예를 들면 [no-trailing-spaces] 규칙을 준수하기 위해 프로젝트의 모든 파일에서 줄 마지막의 공백을 일괄적으로 없애는 작업을 하는 경우에 사용할 수 있다.

`Source` > `Clean up..` 메뉴로 실행한다. 여러 프로젝트에 반복적으로 수행해야할 작업의 그룹이 있다면, 수행할 작업을 별도의 프로파일로 빼서 정의할 수도 있다. 사전정의된 프로파일을 실행할때는 `Use configured profiles`를 선택한다. `Use custom profile`을 선택하면 수행할 작업을 하나씩 선택할 수 있다.



커스터마이징

이 가이드와 함께 제공되는 Eclipse의 설정파일이 프로젝트에서 재정의한 규칙과 맞지 않을 때 참고할만한 정보를 정리한다.

들여쓰기에 탭 대신 스페이스 사용

Eclipse는 들여쓰기로 4개 크기의 탭을 디폴트로 사용한다. `naver-eclipse-formatter.xml`에서도 동일하게 설정되어 있다.

만약 탭 대신 스페이스로 들여쓰기를 하려는 프로젝트에서는 아래와 같이 설정한다.

- `Windows > Preferences > Editors > Text Editors`에 메뉴로 이동한다.
  - `Insert spaces for Tab`를 선택한다.

`naver-eclipse-formatter.xml`에서는 `org.eclipse.jdt.core.formatter.tabulation.char` 속성을 `space`로 바꾼다.

Formatter에서 탭문자 설정.

```
<setting id="org.eclipse.jdt.core.formatter.tabulation.char" value="space"/>
```

C.2 IntelliJ

공백 문자 보이기 설정

탭과 스페이스를 눈에 보이게 표시한다. `File > Settings > Editor > General > Appearance`에서 `Show whitespaces`를 선택한다. 하위 분류에서 `Leading, Inner,`Trailing`를 모두 선택한다.

Checkstyle 설정

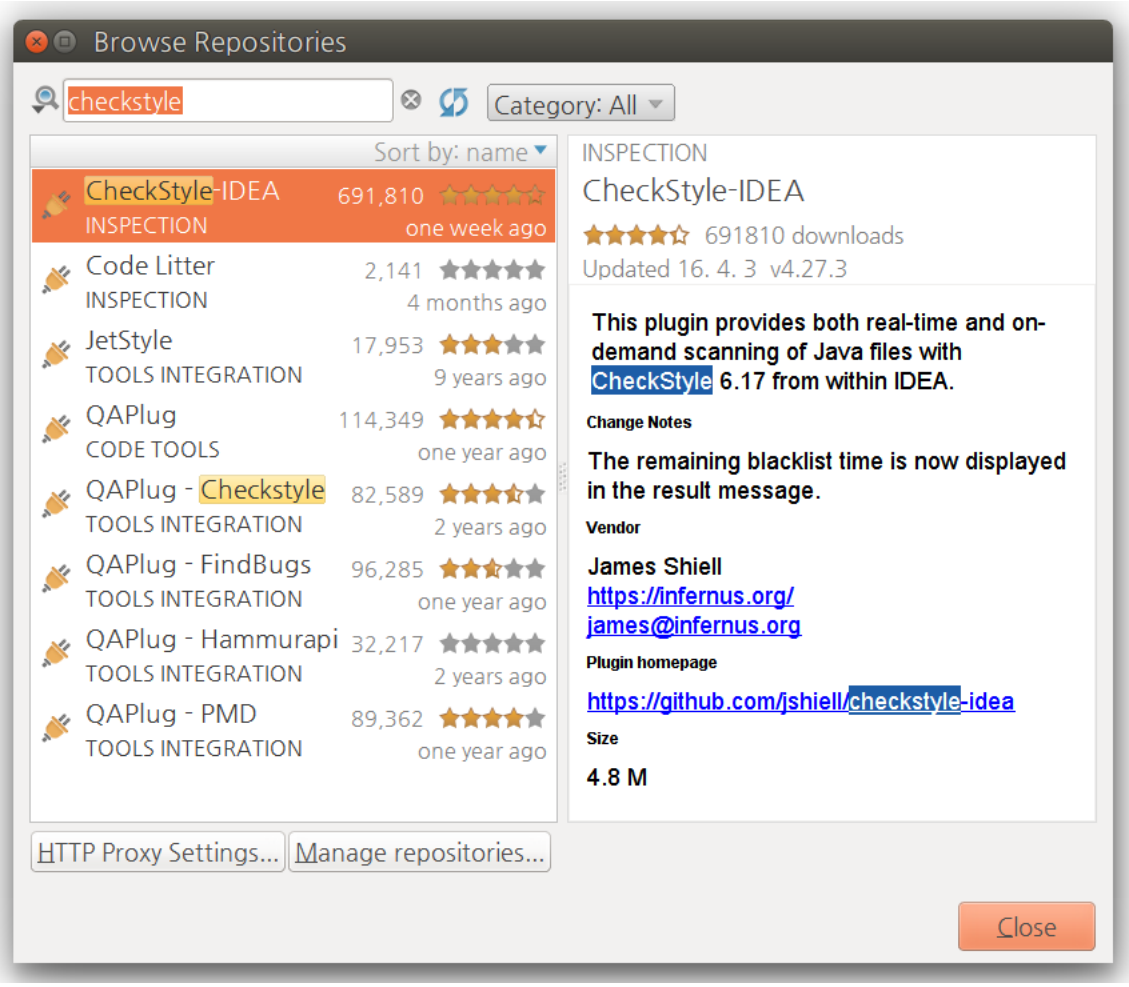
플러그인 설치

1. `File > Settings > Plugins` 메뉴로 이동한다.

2. `[Browse Reposities..]` 버튼 클릭

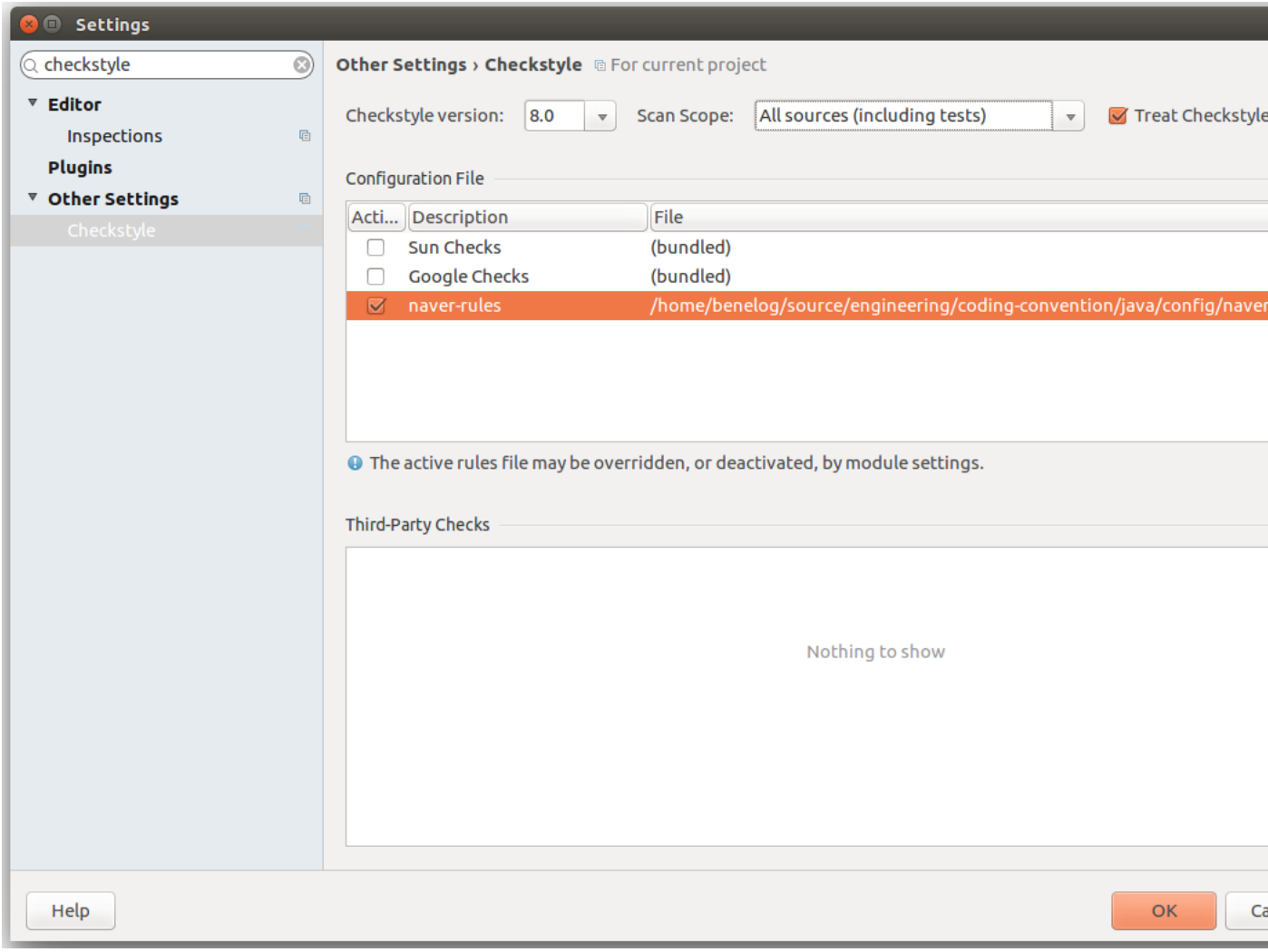
3. 'Checkstyle'의 단어로 검색해서 CheckStyle-IDEA 플러그인을 찾은 후 `[Install]` 버튼을 클릭

이미 플러그인이 설치되어 있다면 권장하는 버전인지 확인을 한다. CheckStyle-IDEA의 사용여부를 선택하는 화면에서 'This plugin provides both real-time and on-demand scanning of Java files with CheckStyle 6.17 from within IDEA.'와 같은 문구로 Checkstyle의 버전을 알려준다. `naver-checkstyle-rules.xml` 을 수정없이 쓰려면 Checkstyle 6.16이상을 쓰는 CheckStyle-IDEA의 버전이 설치되어 있어야한다.



검사 설정

1. `File` > `Settings` > `Other Settings` > `Checkstyle` 메뉴에서 아래 항목들을 설정 한다
   - `Checkstyle versions` : 6.16 ~ 8.0 선택. 디폴트 설정과 다르다면 'Apply' 버튼을 한번 눌러준다.
   - `Scan scope` : `All sources including tests` 선택
   - `Treat Checkstyle errors as warnings` : 체크
2. `File` > `Settings` > `Other Settins` > `Checkstyle` 메뉴로 이동한다.
3. [+] 버튼을 누르면 나오는 입력창에서 아래 항목을 입력한다.
   - `Description` : 식별할 수 있는 이름. `naver-checkstyle-rules`를 권장한다.
   - `Use a Local Checkstyle File` : 선택하고 `[Browse]`버튼을 눌러서 `naver-checkstyle-rules.xml`을 지정한다.
4. `naver-checkstyle-rules` 규칙을 앞의 체크박스를 선택한다.



Warning

2017년 7월 28일 릴리즈된 CheckStyle-IDEA 5.9.0 이상의 버전에서는 checkstyle 8.1 이상을 의존하고 있다. 디폴트 설정으로도 checkstyle 8.1 이상이 선택되어 있다. `naver-checkstyle-rules.xml` 파일을 수정없이 쓰려면 `Checkstyle versions` 항목을 반드시 `8.0` 으로 지정해야 한다.

Tip

checkstyle 8.1 이상 버전을 써야하는 상황이라면 CheckStyle-IDEA 5.10.0 버전을 사용하고 `Checkstyle versions` 항목을 8.2로 지정한다. 규칙 파일은 `naver-checkstyle-rules.xml` 과 같은 디렉토리에 있는 `naver-checkstyle-rules-8.2.xml` 를 사용한다.

Formatter 설정

직접 설정

만약 이 가이드에서 제공하는 Formatter 설정을 활용하기가 어려운 상황이라면 최소한 아래의 항목들은 직접 설정한다.

들여쓰기

- `File > Settings > Editor > Code Style > Java` 메뉴로 이동
  - `User tab charactor` : 선택
  - `Tab Size, Indent` : 4

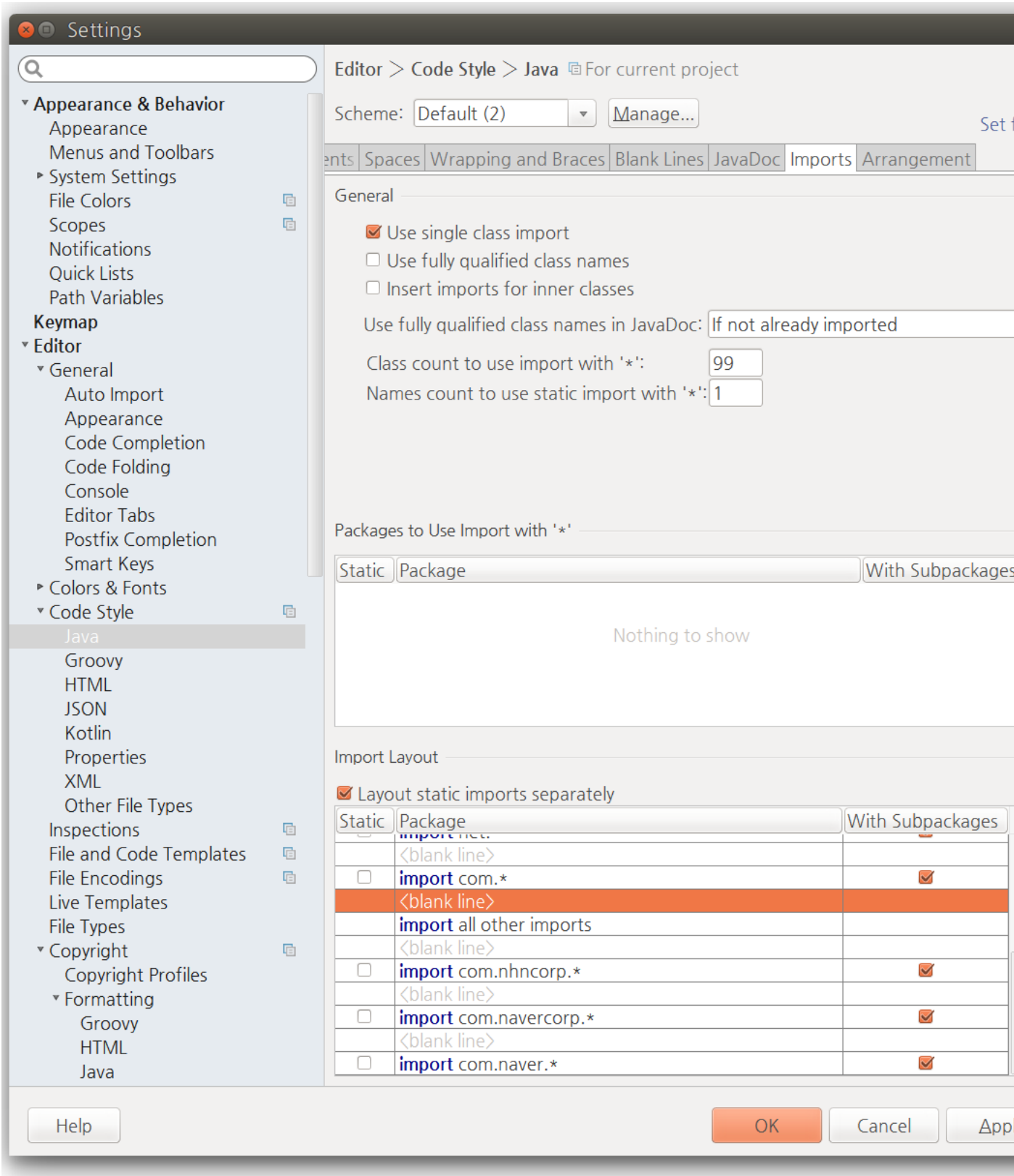
Import 구문 설정

아래 규칙을 자동으로 지키도록 해준다.

- [avoid-star-import] : static import에만 와일드 카드 허용
- [import-grouping] : import 선언의 순서와 빈 줄 삽입

`File > Settings > Editors > Code Style > Java > Imports > Import Layout`로 이동해서 아래 항목을 설정한다.

- General
  - `Use single class import` : 체크
  - `Class count to use import with '*'` : 99
  - `Names count to use static import with '*'` : 1
- Import Layout
  - [import-grouping]의 순서대로 지정
  - 모든 세부 그룹 사이에도 공백을 넣어야 Eclipse의 Formatter와 동일하게 정렬함.



일괄 변환

프로젝트의 홈디렉토리에 커서를 놓은채로 아래의 메뉴를 실행하면 프로젝트의 모든 소스에 해당 설정을 일괄 적용한다.

- File > Line Separators

- Code > Reformat Code
- Code > Auto-Indent Lines
- Code > Optimize Imports

커스터마이징

이 가이드와 함께 제공되는 IntelliJ의 설정파일이 프로젝트에서 재정의한 규칙과 맞지 않을 때 참고할만한 정보를 정리한다.

들여쓰기에 탭 대신 스페이스 사용

이 가이드의 규칙과 다르게 탭대신 스페이스로 들여쓰기를 하고 싶다면 아래와 같이 설정한다.

1. File > Settings > Editor > Code Style > Java 메뉴로 이동한다.
2. Tabs and Indents 탭으로 이동해서 아래 항목을 입력한다.
    - Use tab charactor : 미선택
    - Tab size : 4

C.3 VI

탭의 크기 지정

Unix/Linux는 1탭이 8자리이므로 Java소스를 vi에서 확인하는 경우에는 혼란을 유발할 수 있다. [home]/.vimrc 파일에서 다음을 설정한다.

```
set tabstop=4
set shiftwidth=4
```

tabstop은 \t`문자를 몇개의 크기로 보여줄지를, shiftwidth는 `>>, << 키를 이용할 때 들어가는 간격을 지정한다.

한 줄 최대길이

'textwidth' 옵션을 사용한다.

```
set textwidth=120
```

C.4 Github

Github.com과 Github Enterprise를 사용할 때도 탭의 크기 등을 맞춰야 IDE를 쓸 때와 일관된 레이아웃으로 코드 리뷰를 할 수 있다. `.editconfig`파일을 프로젝트 루트에 커밋을 하면 해당 저장소의 코드 보기/ 편집 기능에서 일괄적으로 탭의 크기 등이 설정된다.

`.editconfig`의 예제.

```
# top-most EditorConfig file
root = true

# 4 space indentation
[*.java]
charset = utf-8
indent_style = tab
indent_size = 4
trim_trailing_whitespace = true
```

만약 저장소 전체 설정을 하기 어려울 경우, 코드를 보는 URL 뒤에 `?ts=4`파라미터를 붙이면 탭의 크기를 지정할 수 있다. 예를 들면 https://github.com/naver/ngrinder/blob/master/ngrinder-controller/src/test/java/org/ngrinder/AbstractNGrinderTransactionalTest.java?ts=4 와 같은 식이다.