< wegra’s Java Naming & Coding Conventions>

<1.0.2>

Revision History

Date	Version	Description	Author
2002.01.09		Requested	wegra (이복연)
2002.04.12	Public Draft		wegra
2002.05.16	1.0.0	Final release	wegra
2002.08.29	1.0.1	Minor modifications	wegra
2003.04.10	1.0.2	Minor modifications	wegra

Changes in 1.0.2

- assertion 관련 규약 추가.
- 예제 추가.
- More minor modifications.

Changes in 1.0.1

- // PENDING, // REMIND 주석에 들여쓰기 적용.
- 축약어를 모두 대문자 표기에서 일반 단어와 같은 규칙을 적용하는 것으로 변경 (축약어가 연속해서 올 경우 구분 용이).
- More minor modifications.

Table of Contents

wegra's Java Naming & Coding Conventions

1. Introduction

1.1 Purpose

명명법 표준화의 가장 큰 목적은 가독성 향상과 체계적이고 일관적인 코딩을 통한 개발자들 간의 원활한 의사소통 지원, 이름 충돌로 인한 오류 방지, 제품의 유지보수 비용 절감 등이다. 또한 개인적 차원에서도 자신이 작성한 여러 프로그램들 간의 코딩 방식 통일로 유용한 부분을 추려 라이브러리화 하기 쉽고 예전에 작성한 코드의 낯섦이 사라지게 되는 등의 이점이 생긴다.
이미 표준 규약이 나와 있는 상태에서 다시 나름대로의 규약을 정하는 이유는 표준 규약에서 명시되지 않은 세세한 부분들의 통일과 부족한 부분의 보충하기 위함이다.
본 규약은 자바 표준 규약에 익숙한 개발자라면 누구나 무리 없이 본 규약에 따라 작성된 코드를 이해할 수 있도록 작성되었다.

1.2 Scope

자바 명명법과 주석 표기 방식, 기타 코딩 스타일을 표준화한다.
Java Language 이외의 언어들과는 무관하다.
J2SE 1.4.x 이하 버전에 해당한다.

1.3 References

1. The Java Language Specification - Second Edition

2. Naming Conventions

명명 규약을 따르게 되면 일반적으로 더 낳은 가독성이 제공되고 이름 충돌로 인한 오류 발생을 최소화 시킬 수 있게 된다. 본 명명 규약은 자바 표준 규약을 최대한 유지하도록 하였지만 약간의 변경 사항과 축소된 부분을 포함한다. 축소된 부분은 본 명명 규약이 개인적 차원의 규약이므로 자바 표준 규약에서 필요 없는 부분을 제거한 것이며 이 경우에도 최대한 표준 규약에 거스르는 일은 없도록 하였다.

2.1 Primary Rules

① 모든 이름은 명확한 의미를 전달할 수 있도록 짓는다.
② 모든 이름은 영문 알파벳, 숫자, 밑줄(_) 만을 사용하여 짓는다.
③ 모든 이름의 길이는 두 자 이상이어야 한다. 단, 반복문의 카운트 변수 제외.
④ 'temp', 'tmp'와 같이 정확한 용도를 파악하기 힘든 이름은 사용하지 않는다.

2.2 Package Names

① 모든 패키지 명은 영문 소문자만을 사용한다.
② 라이브러리 패키지는 항시 'org.wegra.'로 시작한다.
③ 패키지는 기능 단위로 구분하며 가능한 짧은 이름을 사용하되 해당 패키지의 용도를 명확히 구분할 수 있도록 한다.
④ 표준 패키지의 확장/보충 기능을 담당할 경우 표준 패키지의 이름을 따른다.

예:
```
org.wegra.awt
org.wegra.lang
org.wegra.net
```

2.3 Class and Interface Type Names

① 인터페이스 명은 클래스 명과 동일한 규칙을 적용한다.
② 추상 클래스(Abstract Class)의 구분은 따로 하지 않는다.
③ 클래스 명은 명사 또는 명사구를 사용한다.
④ 클래스의 역할을 정확히 표현할 수 있는 단어를 택하되 너무 길어지지 않도록 주의한다. (표현력과 길이 사이의 적절한 타협점을 찾도록 할 것)
⑤ 일반적으로 축약어를 사용하지 않으며 광범위하게 사용되는 축약어만을 허용한다.
⑥ 각 단어의 첫 문자는 대문자로 하며 그 외에는 소문자 사용을 원칙으로 한다.

예:
```
ClassLoader
SecurityManager
Thread
BufferedInputStream
HttpConnection
```

2.4 Exception and Error Class Type Names

① 'Class and Interface Type Names' 과 동일한 규칙을 적용한다.
② Exception 클래스의 이름은 반드시 'Exception'으로 끝맺는다.
③ Error 클래스의 이름은 반드시 'Error'로 끝맺는다.

예:
```
TimeoutException
VerifyError
```

2.5 Method Names

① 메소드 명은 동사 또는 동사구를 사용한다.
② 메소드의 기능을 정확히 표현할 수 있는 단어를 택하되 너무 길어지지 않도록 주의한다. (표현력과 길이 사이의 적절한 타협점을 찾도록 할 것)
③ 일반적으로 축약어를 사용하지 않으며 광범위하게 사용되는 축약어만을 허용한다.
④ 첫 단어는 소문자로 시작하며 뒤이은 각 단어들의 첫 문자는 대문자, 그 외에는 소문자 사용을 원칙으로 한다.
⑤ 속성에 관한 메소드들은 'get' 이나 'set' 으로 시작한다. 속성 명을 V 라 한다면 값을 얻는 메소드는 'getV', 값을 설정하는 메소드는 'setV' 라는 이름을 갖게 된다.
⑥ boolean 조건 검사 메소드는 'isV' 또는 'hasV' 형태의 이름을 갖는다.
⑦ 객체의 내용을 특정 포맷 F 로 변경시키는 메소드는 'toF' 의 형태를 갖는다.
⑧ 길이를 반환하는 메소드 명은 'getLength' 로 한다.
(자바 표준 명명법에서는 length 로 하고 있으나 여기서는 ①, ⑤번 규약에 따르도록 한다. 이로써 지역 변수 명과 충돌되는 경우가 없어진다)

```
예:    draw()
       getId()
       setLocale()                // 속성 설정
       isVisible()                // boolean 조건 검사
       hasMoreElements()          // boolean 조건 검사
       getLength()                // 자바 표준 규약과 다르다
       toByteArray()              // 포맷 변경
```

⑨ 이벤트 리스너의 메소드는 '명사' + '동사(과거/진행형)' 의 형태를 갖는다.

```
예:    windowClosing()
       actionPerformed()
```

2.6 Field Names

필드 명명 규약은 자바 표준 규약과 비교적 많은 차이를 보이는 부분이다.
주의해서 숙지하도록 하자.

① 필드 명은 명사 또는 명사구를 사용한다.
② 필드의 용도를 정확히 표현할 수 있는 단어를 택하되 너무 길어지지 않도록 주의한다. (표현력과 길이 사이의 적절한 타협점을 찾도록 할 것)
③ 일반적으로 축약어를 사용하지 않으며 광범위하게 사용되는 축약어만을 허용한다.
④ 클래스 변수는 static 을 의미하는 's' 로, 인스턴스 변수는 field 를 의미하는 'f' 로 시작한다.
(클래스/인스턴스 변수 사이의 구분뿐 아니라 지역 변수나 메소드, 파라미터 명과도 명확히 구분되기 때문에 프로그래머의 실수로 인한 오류를 줄일 수 있다)
⑤ 각 단어들의 첫 문자는 대문자, 그 외에는 소문자 사용을 원칙으로 한다.

```
예:    sInstanceCount        // 클래스 변수
       fElements             // 인스턴스 변수
       fUniqueId             // 축약어도 같은 규칙 적용
```

2.7 Constant Names

자바에서의 상수란 'static final' 수식자를 갖는 변수, 또는 인터페이스에 정의된 변수를 말한다.

① 상수 명은 하나 이상의 단어가 '_' 로 결합된 형태를 취하며 대문자만을 사용한다.

② 같은 속성에 대한 선택값을 나타내는 상수들은 공통된 접두어 또는 접미어를 사용한다.

```
예:    MIN_VALUE
       MAX_VALUE            // 동일 접미어 사용
```

2.8 Local Variable and Parameter Names

지역 변수와 파라미터는 하나의 메소드 내부에서만 사용되는 변수를 말한다.

① 파라미터 명은 지역 변수 명과 동일한 규칙을 적용한다.
② 지역 변수 명은 명사 또는 명사구를 사용한다.
③ 변수의 용도를 정확히 표현할 수 잇는 단어를 택하되 너무 길어지지 않도록 주의한다.
(표현력과 길이 사이의 적절한 타협점을 찾도록 할 것)
④ 첫 단어의 시작은 소문자, 뒤이은 단어들의 첫 문자는 대문자, 그 외에는 소문자 사용을 원칙으로 한다.

```
예:    offset
       elementsCount
```

⑤ 원 의미가 훼손되지 않는 범위 내에서, 또는 관용적으로 널리 사용되고 있는 단어의 경우 원형을 그대로 사용하지 않고 줄여 쓰는 것을 허용한다.

```
예:    buffer à buf
```

⑥ 다음의 경우를 제외하고 한 문자로 된 변수의 선언을 금한다.

- 반복문 내에서 사용하는 카운트 변수

```
예:    for ( int i = 0; i < MAX_LENGTH; i++ ) {
              …
       }
```

- try-catch 문의 Exception 파라미터 명

```
예:    try {
              …
       } catch ( Exception e ) {
              …
       }
```

- Graphics 객체는 일반적으로 'g'로 사용

```
예:    public void paint(Graphics g) {
              …
       }
```

⑦ 클래스 변수나 인스턴스 변수 명과 중복되지 않는 이름을 사용한다.

3. Comment Style & Usages

주석 처리 형태는 자바의 기본 형태와 많은 차이를 보인다. /* */ 형태 주석은 주요 용도를 // 형태 주석이 대체하여 거의 사용되지 않게 되었고 // PENDING, // REMIND 라는 새로운 형태의 주석이 추가되었다.
모든 주석을 그 내용을 가능한 자세히 서술하도록 한다.

3.1 Usage of '/* … */ Style' Comment

① 라이선스 등 코드 구현과 상관 없는 설명을 첨부.
② 코드의 일부분을 실행되지 않도록 불활성화.
③ 부득이하게 코드 라인 처음, 또는 중간에 주석 추가.
④ 이상의 경우 외에는 사용을 금한다.

3.2 Usage of '// Style' Comment

① 한 라인의 주석을 처리한다.
② 여러 라인의 설명을 나타낼 경우 /* */ 형태를 사용하는 대신 각 라인을 // 로 시작한다.
③ // 이후 한 칸의 공백을 유지한다.

3.3 Usage of '// PENDING … // END PENDING' Style Comment

① 아직 구현이 완료되지 않은 부분을 표시한다. 프로토타입만 정의해 두었거나 임시로 간략히 구현해 둔 곳을 표시하여 실수로 완성치 못한 부분이 남겨지지 것을 방지할 목적으로 사용된다.
② 타인이 보아도 의도를 명확히 이해할 수 있도록 가능한 쉽고 자세하게 기술한다.
③ 제품 완성 후에는 가능한 모든 // PENDING 주석이 없어져야 한다.
④ 적용 범위의 들여쓰기 위치와 상관없이 항상 라인의 첫 칸에서 시작한다.
- 항시 첫 칸부터 시작되므로 쉽게 눈에 띄어 확실히 각인시킨다.
- 들여쓰기가 깊은 곳에서 주석 처리의 어려움 방지
⑤ 내포된 // PENDING 주석은 내포 깊이에 맡게 들여쓰기를 적용한다.
⑥ 완전한 형태는 아래와 같다.

```
// PENDING(작성자 이름) yyyy.mm.dd: 내용
// [여러 라인이 필요할 경우]
… (적용 범위)
// END PENDING
```

예:

```
    public void transmit(Packet packet) {
  // PENDING(이복연) 2002.01.10: 패킷의 타입에 따라 적절한 수신자에
  // 전달하도록 한다.
  // 현재는 모든 패킷이 버려지도록 되어 있다.
  // 패킷의 형태는 packet.getType() 을 통해 알 수 있다.
        ....          // 적용 내용
  // END PENDING
     }
```

3.4 Usage of '// REMIND … // END REMIND' Style Comment

① 헷갈리기 쉬운 부분이나, 얼핏 이상하게 보일 수 코드에 작성 시의 의도를 명확히 밝혀 추후 문제가 발생하는 일을 방지할 목적으로 사용된다.
② 타인이 보아도 의도를 명확히 이해할 수 있도록 가능한 쉽고 자세하게 기술한다.

③ 추후 코드가 명확한 형태로 완전히 새로 작성되지 않는 이상 가능한 // REMIND 주석은 없애지 않도록 한다.
④ 적용 범위의 들여쓰기 위치와 상관없이 항상 라인의 첫 칸에서 시작한다.
- 항시 첫 칸부터 시작되므로 쉽게 눈에 띄어 확실히 각인시킨다.
- 들여쓰기가 깊은 곳에서 주석 처리의 어려움 방지
⑤ 내포된 // REMIND 주석은 내포 깊이에 맡게 들여쓰기를 적용한다.
⑥ 완전한 형태는 아래와 같다.

```
// REMIND(작성자 이름) yyyy.mm.dd: <내용>
// [여러 라인이 필요한 경우]
… (적용 범위)
// END REMIND
```

예:

```
        for ( /* 생략 */ ) {
  // REMIND(이복연) 2002.01.10: 이 부분은 프로그램 퍼포먼스에 커다란
  // 영향을 미치므로 가독성이나 객체지향적 코딩 보다는 퍼포먼스 극대화에
  // 주안점을 두어 작성되었다.
  // 아래 코드를 수정할 시는 반드시 수정 전과의 퍼포먼스 비교를 수행해야
  // 함을 잊지 말자.
            ....          // 적용 내용
  // END REMIND
        }
```

3.5 Usage of '/** … */' Style Comment

① javadoc 을 통한 자바 표준 API 문서를 만들어 내기 위한 주석 형태이다.
② 모든 패키지, 클래스, 메소드, 필드, 상수는 이 형태의 설명문을 포함할 것을 권장한다.
③ javadoc 표준 태그에 @fix 라는 비표준 태그가 추가된다.
④ 기본적으로 다음과 같은 형태를 권장한다. javadoc 표준에 입각한 정보 추가는 허용된다.

- Package Comment

```
/**
 * <설명>
 */
```

- Class & Interface Comment

```
/**
 * <설명>
 *
 * @author [작성자]
 * @version [버전], [yyyy.mm.dd]            // 최초 작성일
 * @fix([수정자명]) [yyyy.mm.dd]: [수정 내용]
   // 메소드/필드/상수의 추가/삭제/이름변경/Deprecated 등의 변화가 있을 시.
   // 메이저 버전 변화 시 각 메소드의 @fix 절과 함께 초기화한다.
 */
```

- Method Comment

```
/**
 * <설명>
 *
 * @param  [파라미터명] [파라미터 설명]      // 각 파라미터 당 하나씩
 * @return [반환값설명]                     // 반환값이 존재할 경우
 * @throws [예외명]  [설명]                 // 각 예외 당 하나씩
                                            // (런타임 예외 생략 가능)
```

```
     * @author [작성자]                          // 클래스 작성자와 다를 경우만
     * @fix(<수정자명>) [yyyy.mm.dd]: [수정 내용]
       // 메소드 버그 수정 내용
       // 클래스의 메이저 버전 변화 시 초기화한다.
     */
- Field Comment
    /**
     * <설명>
     */
  or
    /** …… */                                  // 단축형 허용
- Constant Comment
    /**
     * <설명>
     * {@value}                                // 생략 가능, javadoc 1.4 이상
     */
  or
    /** <설명> {@value} */                     // 단축형 허용
```

4. Coding Conventions

4.1 General Rules

① 한 라인의 길이는 80 칸을 넘지 않도록 유지한다.
② 들여쓰기 간격은 4 칸으로 한다.
③ 한 라인에 둘 이상의 문장이 올 수 없다.
④ 콤마(,) 앞에는 공백이 없도록, 뒤로는 한 칸의 공백을 둔다.
⑤ unary 연산자를 제외한 모든 연산자는 앞뒤로 한 칸씩의 공백을 둔다.
⑥ 모든 키워드 뒤에 오는 소괄호는 키워드와 한 칸의 공백을 둔다.
적용되는 키워드는 if, switch, for, while, catch 가 있다.
⑦ 한 메소드 내에서 적용 범위(scope)가 겹치지 않는다고 해서 같은 이름의 지역 변수가 중복 선언되지 않도록 한다.
단, 반복문의 카운트 변수 제외
⑧ 하나의 변수를 여러 용도로 사용하지 않는다.
⑨ 변수 초기화는 가능한 선언 시 명시적으로 수행한다.
⑩ 선언문이 아닌 문장의 길이가 80 칸이 넘어 라인을 나눌 경우 다음과 같은 규칙을 적용한다.
- 콤마(,) 다음, 연산자 앞에서 나눈다.
- 괄호로 묶인 부분은 가능한 나누지 않는다.
- 나뉘어진 부분은 두 번의 들여쓰기를 적용한다.
- 나뉘어진 부분을 포함하는 소괄호가 존재한다면 소괄호의 위치 다음 칸까지 들여 쓴다.
⑪ 가능한 Label 로의 분기를 사용하지 않고 해결할 수 있도록 설계한다.
⑫ 문장의 끝을 나타내는 세미콜론(;)은 항시 앞의 문장에 공백 없이 붙인다.

4.2 Source File Format

① 하나의 소스 파일에는 하나의 클래스, 또는 인터페이스 만이 포함되도록 한다.
(내부 클래스/인터페이스 제외)
② 클래스를 선언하는 파일의 구성 순서
- 라이선스, 정책 등의 구현 코드와 직접적 관계가 없는 설명문 (/* */ 주석 사용)
- 패키지 선언
- import 선언
- 클래스 설명 (/** */ 주석 사용)
- 클래스 선언
  . 상수 선언
  . 클래스 변수 선언
  . 정적 초기화 구문
  . 인스턴스 변수 선언
  . 생성자 선언
  . 클래스 메소드 선언
  . 인스턴스 메소드 선언
  . 이벤트 처리 메소드 선언
  . 내부 클래스 선언 (클래스 구조와 동일)

③ 인터페이스를 선언하는 파일의 구성 순서
- 라이선스, 정책 등의 구현 코드와 직접적 관계가 없는 설명문 (/* */ 주석 사용)
- 패키지 선언

- import 선언
- 인터페이스 설명 (/** */ 주석 사용)
- 인터페이스 선언
  . 상수 선언
  . 추상 메소드 선언

④ import 선언 시의 세부 규칙
- 광범위하게 사용되는 표준 패키지를 먼저 선언한다.
- 서브 패키지가 없는 것을 가장 먼저 선언한다.
- 서브 패키지들끼리는 우선 순위가 없다.
- 같은 상위 패키지를 공유하는 것들끼리 묶어 가독성을 높인다.
- 프로젝트에 종속적인 패키지를 가장 나중에 선언한다.
- wegra's library 패키지는 프로젝트 종속적 패키지 바로 앞에 선언한다.

예:
```
import java.net.*;          // 자바 표준 패키지
import java.nio.channels.*; // 같은 상위 패키지를 공유
import java.nio.charset.*;  //          "
import javax.swing.*;       // 자바 표준 확장 패키지
import org.omg.*;           // 기타 패키지
import org.wegra.awt.*;     // wegra's library 패키지
import ca.vm.event.*;       // 프로젝트 종속적 패키지
```

4.3 Declarations

4.3.1 Common Rules

① Body 를 여는 중괄호는 선언부와 같은 라인의 끝에 한 칸 띄워 위치 시킨다.
② 단, 선언부가 두 라인 이상 되는 경우 Body 를 여는 중괄호는 선언부 다음 라인에서 선언부와 같은 들여쓰기를 적용하여 열어준다.
③ Body 를 닫는 중괄호는 Body 마지막 다음 라인에 선언부의 들여쓰기 위치에 맞춰 닫아준다.
④ Body 는 여는 중괄호 다음 라인부터 선언문보다 한 단계 깊은 들여쓰기를 적용하여 기술한다.
⑤ Body 가 너무 길어지거나 다른 선언문과의 명확한 구분이 필요한 경우 닫는 중괄호 뒤에 한 칸을 띄우고 간략한 주석을 첨부한다.

예:
```
public void handle() {      // ①
    ...                     // ③
    ...
    if ( ... ) {
        ...
    } // end if              // ⑤
}
```

⑥ 각각의 선언부들의 명확한 구분을 위해 사이에 한 라인 이상을 띄우도록 한다.
⑦ 필드/메소드의 경우 외부 접근 허용/확장에 대한 분명한 정책이 없다면 반드시 private 접근 수식자를 사용한다.

4.3.2 Class Declarations

① 둘 이상의 클래스 수식자가 올 경우 다음과 같은 순서를 지켜 기술한다.
[public | protected | private] [abstract] [static] [final] [strictfp]
② 선언부가 80 칸을 넘어설 경우 implements 구문을, 또는 extends 와 implements 구문을

각각 다음 라인에 들여쓰기를 두 번 적용하여 기술한다.

예:
```
public class HashSet extends AbstractSet
        implements Cloneable, Collection, Serializable, Set
{
    ...
}
or
public class HashSet
       extends AbstractSet
       implements Cloneable, Collection, Serializable, Set
{
    ...
}
```

4.3.2.1 Constant Declarations

① public, protected, package(default/없음), private 순으로 선언한다.
즉, public 상수들이 코드의 가장 윗 라인에, protected 등이 그 뒤를 잇는다.

4.3.2.2 Field Declarations

① 둘 이상의 필드 수식자가 올 경우 다음과 같은 순서를 지켜 기술한다.
[public | protected | private] [static] [final] [transient] [volatile]
② 클래스 변수들을 먼저 선언한 후 인스턴스 변수들을 선언한다.
③ 변수들은 public, protected, package(default/없음), private 순으로 선언한다.
④ 클래스 변수 선언부와 인스턴스 변수 선언부 사이에는 눈에 띌 정도의 공백 라인을 둔다.
⑤ 클래스 변수 초기화는 변수 선언 시, 또는 정적 초기화 구문을 통해 초기화 한다.
⑥ 모든 인스턴스에 동일하게 적용되는 인스턴스 변수 초기화는 선언 시 같이 초기화한다.
⑦ 인스턴스 별로 다르게 적용될 수 있는 인스턴스 변수의 초기화는 생성자에서 초기화한다.

4.3.2.3 Method Declarations

① 둘 이상의 메소드 수식자가 올 경우 다음과 같은 순서를 지켜 기술한다.
[public | protected | private] [abstract] [static] [final] [synchronized]
[native] [strictfp]
② 예외를 던질 경우 throws 부터 다음 라인에 기술한다.
이 때 메소드 선언부를 기준으로 들여쓰기를 두 번 더 적용한다.
③ throws 부분이 한 라인을 초과하지 않도록 예외를 남용하지 않는다.
④ 입력 인자가 많아 한 라인에 표시가 불가능할 경우 콤마(,)를 기준으로 라인을 나누어 기술한다. 이 때 첫 입력 인자가 시작하는 위치까지 들여쓰기를 적용한다.
⑤ 메소드 명과 여는 소괄호 사이는 공백을 두지 않는다.
⑥ 소괄호는 파라미터와 한 칸의 공백을 유지하며, 만약 파라미터가 없을 경우 여는 소괄호와 닫는 소괄호 사이는 공백을 두지 않는다.

예:
```
private void drawTriangle(Triangle triangle, Point origin) {
    ...
}
```

⑦ 메소드 몸체의 내용이 없거나 한 라인으로 표현될 경우라도 반드시 중괄호를 통해 몸체를 명시한다.

예:
```
public final int getId() {
```

```
        return fId;
    }
```

4.3.2.4 Constructor Declarations

① 4.3.2.3 Method Declarations 와 동일한 규칙이 적용된다.
단, 사용할 수 있는 수직자가 [public | protected | private]로 제한된다.

4.3.3 Interface Declarations

① Class 선언 규약 중 적용 가능한 모든 것을 그대로 수용한다.
② 둘 이상의 수식자가 올 경우 다음과 같은 순서를 지켜 기술한다.
[public | protected | private] [static] [strictfp]
③ 자바 명세에서 허용하고 있는 abstract 수식자는 사용하지 않는다.
④ 필드(상수)의 수식자는 반드시 public static final 이어야 한다.
자바 명세에서는 생략 가능하지만 상수임을 확실히 나타내기 위해 본 규약에서는 불허한다.
⑤ 메소드의 수식자는 반드시 public abstract 이어야 한다.
자바 명세에서는 생략 가능하지만 의미를 분명히 하기 위해 본 규약에서는 불허한다.

4.3.4 Array Type Declarations

① 모든 배열은 '[]'를 타입 명 뒤에 오도록 선언한다.
자바 명세에서 허용하고 있는 변수 명 뒤에 '[]' 이 오는 형태는 허용하지 않는다.
배열의 의미를 하나의 타입으로 통일하기 위함이다.

```
예:    int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };
       byte[] buffer;
       byte matrix[][];            // 허용하지 않음
```

4.4 Other Statements

본 절에서 언급되는 Statement 들은 다음에 기술된 형태로만 사용하도록 한다.

4.4.1 The if Statement

① 기본 형태

```
if ( BooleanExpression ) {
    Statements
}
```

② 확장 형태 1

```
if ( BooleanExpression ) {
    Statements
} else {
    Statements
}
```

③ 확장 형태 2

```
if ( BooleanExpression ) {
    Statements
} else if ( BooleanExpression ) {
    Statements
```

```
} else {
    Statements
}
```

④ 이상 세 형태의 혼합

4.4.2 The switch Statement

① 기본 형태

```
switch ( NumericExpression ) {
    case ConstantExpression:
        [ Statements ]
        [ break|continue[ Label] |return; ]
    ...    // case 문 반복
    default:
        Statement
        break|continue[ Label]|return;
} // end switch
```

② case 문에서 의도적으로 break 를 사용하지 않을 경우 주석을 사용하여 반드시 명시한다.

③ default 와 둘 이상의 case 를 반드시 포함하도록 한다.
그렇지 않을 경우 if 문을 사용하도록 한다.
단, 추후 확장이 예상되는 경우, 또는 다른 부분과의 일관성 측면에서 필요하다면 허용.

④ 코드 일관성 유지를 위해 default 문 마지막에 반드시 break;를 삽입하도록 한다.

4.4.3 The while and do-while Statement

① 기본 형태 1 (while)

```
while ( BooleanExpression ) {
    Statements    // 반드시 하나 이상의 Statement 가 오도록 한다.
    [ break|continue[ Label]; ]
}
```

② 기본 형태 2 (do-while)

```
do {
    Statements
    [ break|continue[ Label]; ]
} while ( BooleanExpression );
```

4.4.4 The for Statement

① 기본 형태

```
for ( LoopInit; BooleanExpression; LoopAdjust ) {
    Statements
    [ break|continue[ Label]; ]
}
```

4.4.5 The synchronized Statement

① 기본 형태

```
synchronized ( Lock ) {
    Statements
```

```
}
```

4.4.6 The try-catch-finally Statement

① 기본 형태

```
try {
    Statements
} catch ( FormalParameter ) {    // catch 문은 한 번 이상 반복
    Statements
} finally {
    Statements
}
```

4.4.7 The assert Statement (J2SE 1.4 이상)

① 기본 형태

```
assert BooleanExpression
```

② 기본 형태 2

```
assert BooleanExpression : ErrorStatement
```