프로그램 표준
코딩규칙

2010.12

개발자들 간에 일관된 코딩규칙을 적용하게 되면, 코드의 가독성과 이해도가 높아져, 코드 검토시간 감소와 관리 용이성 등 효율적인 협업이 가능하다. 또한, 운영시 발생할 수 있는 오동작을 감소시켜 유지보수를 용이하게 한다.

행정안전부

[제목 차례]

제1장 개요

제1절 표준 코딩규칙의 필요성

컴파일러나 인터프리터는 개발자가 작성한 프로그램을 컴퓨터가 이해할 수 있는 바이너리 포맷으로 변환 처리하기 때문에 프로그램의 문법적인 구조만 일치하면, 소스에 대해서 문제 삼지 않는다. 반면 개발자가 작성한 코드는 여러 사람이 공유하고, 검토 대상이 되기 때문에 코딩 스타일 및 컨벤션의 기준이 필요하다.

프로젝트 개발 과정에서 어떤 문제를 해결하기 위해서 여러 개발자가 협업을 하기도 하고, 때로는 타인이 작성한 개발 소스를 유지 보수하기도 한다. 최초 개발자가 개발된 프로그램을 계속 유지보수하는 경우는 거의 없다. 따라서 코딩규칙에 대한 지침 및 기준이 제시되어져야 하고, 개발자는 그 기준에 맞게 코딩이 이루어져야 한다.

표준 코딩은 코드 작성에 관한 지침, 규칙 및 규정의 집합으로 변수 명명, 코드 들여쓰기, 괄호나 키워드의 위치 정하기 등 주위를 해서 작성해야 하는 문장과 함수, 각 언어의 특성 및 개발 환경으로 인해서 발생하는 지침 등을 포함한다. 표준 코딩규칙은 많은 개발자들이 동일한 코드를 작업할 경우 일관되게 지켜야 하며, 특히 초보 개발자는 코드를 작성할 때 표준을 지키는 것이 매우 중요하다. 코딩 표준을 준수하여 코드를 작성하면 코드 검토에 소요되는 시간을 상당히 절약할 수 있기 때문이다.

표준 코딩규칙은 일반적으로 코드의 가독성, 확장성 및 신뢰성을 향상시키기 위한 목적으로 사용한다.

- **가독성을 목적으로 한 규칙** : 주석, 줄 바꿈, 들여 쓰기 및 공백과 같은 빈 공간의 처리와 조건문의 형식화 등의 규칙을 포함한다. 프로그램 소스가 일관되게 정열된 형식으로 코딩되면 가독성이 높아진다.
- **확장성을 목적으로 한 규칙** : 상속, 캡슐화 및 상수에 관한 규칙 등이 포함된다. 명시적인 숫자 대신 상수를 사용하면, 변경 시 한번 상수의 값을 변경하는 것만으로 코드 전체에서 상수의 모든 인스턴스를 한 번에 변경할 수 있어 확장성이 높아진다.
- **신뢰성을 목적으로 한 규칙** : 제한된 자원, 동기화 및 예외 처리 규칙 등이 포함된다. 제한된 자원을 적절하게 해제하면 메모리 부족이나 데드락 같은 문제가 줄어든다.

개발자가 표준 코딩규칙을 준수하면 개발된 소프트웨어의 유지보수가 쉬워지고, 결함을 감소시키고, 비용을 절감하며 코드 품질이 향상된다.

제2절 표준 코딩규칙 내용 및 기술 방법

본 책자의 표준 코딩규칙은 SUN Java Code Conventions, MISRA-C, The CERT Oracle Secure Coding Standard for Java 및 The Cert C Secure Coding Standard를 참조하고, 국내 이 분야의 권위를 가진 전문가의 의견을 반영하여 작성되었다.

본 책자의 구성은 공통 코딩규칙과 Java 의존적인 코딩규칙 및 C 의존적인 코딩규칙으로 구분하고 있다.

1. **공통 코딩규칙** : Java/C에서 공통적으로 적용할 수 있는 규칙으로, 명칭, 소스 형식, 주석, 변수 선언, 상수, 수식 및 문장같이 주로 프로그램의 외형을 구성하는 문장 및 표현에 기반을 둔 규칙과 보안기능과 관련된 규칙을 다루고 있으며, 7개의 규칙으로 구성 되어 있다. 준수해야 하는 규칙과 사용해서는 안 될 문법 등을 Good 코드와 Bad 코드 예제를 두어 설명하고 있다.

2. **Java 의존적인 코딩규칙** : Java의 언어적인 특성에서 발생되는 코딩규칙으로 14개의 하위 규칙으로 구성이 되어 있으며, Good 코드 예제와 Bad 코드 예제를 제시하고, 예제에 대한 주석 및 설명으로 이해도를 제고한다.

3. **C 의존적인 코딩규칙** : C의 언어적인 특성에서 발생되는 코딩규칙으로 9개의 하위 규칙으로 구성이 되어 있으며, Good 코드 예제와 Bad 코드 예제를 제시하고, 예제에 대한 주석 및 설명으로 이해도를 제고한다.

제2장 프로그램 표준 코딩규칙

제1절 공통 코딩규칙

1. 명칭에 관한 규칙

- **명칭의 길이는 31자 이내로 한다.**
  명칭의 길이는 컴파일러에 따라서 제한하는 경우가 있으며, 명칭의 길이가 필요 이상으로 길어질 경우 오타발생 확률 증가 및 가독성이 저하된다.

■ Bad 코드의 예-Java

```
public class InformationManagementFromDatabaseServer { // 클래스명이 31 자 초과.
    String collectInformationFromDatabaseServer() { // 메서드명이 31 자 초과.
        int initial_database_management_startdate_status = 0; // 변수명이 31 자 초과.
        ...
    }
}
```

■ Good 코드의 예-Java

```
public class InfoMgtFromDBServer { // 클래스명이 31 자 초과하지 않음.
    String collectInfoFromDBServer() { // 메서드명이 31 자 초과하지 않음.
        int init_dbmgt_sd_status = 0; // 변수명이 31 자 초과하지 않음.
        ...
    }
}
```

■ Bad 코드의 예-C

```
int collectInformationFromDatabaseServer() { // 함수명이 31 자 초과.
    int initial_database_management_startdate_status = 0; // 변수명이 31 자 초과.
    ...
}
```

■ Good 코드의 예-C

```
int collectInfoFromDBServer() { // 함수명이 31 자 초과하지 않음.
    int init_dbmgt_sd_status = 0; // 변수명이 31 자 초과하지 않음.
    ...
}
```

- **동일한 변수명과 메서드(함수)명을 사용하지 않는다.**
  동일한 이름을 가진 변수명과 메서드(함수)명은 가독성이 저하된다. 개발자가 변수에 접근 시에 메서드(함수)를 호출할 수 있기 때문에 혼란을 줄 수 있다.

■ Bad 코드의 예-Java

```
public class Foo {
// 멤버변수명과 메서드명이 동일.
    private String name;
    public String name() {
        return name;
    }
}
```

■ Good 코드의 예-Java

```
public class Foo {
// 멤버변수명과 메서드명이 상이.
    private String name;
    public String getName() {
        return name;
    }
}
```

■ Bad 코드의 예-C

```
// 변수명과 함수명이 동일.
char*   name;
...
char* name() {
    name;
}
```

■ Good 코드의 예-C

```
// 변수명과 함수명이 상이.
char*   name;
...
char* getName() {
    name;
}
```

- **명칭은 "_" 이외의 특수 문자를 사용하지 않는다.**
  다른 프로그래밍 언어와 명칭에 대한 호환을 위해서 특수문자는 "_"만 허용한다.

■ Bad 코드의 예-Java/C

```
int count_email@others = 0; // 변수명에 "_" 이외의 특수문자(@) 사용.
```

■ Good 코드의 예-Java/C

```
int count_email_others = 0; // 변수명에 "_" 이외의 특수문자 사용하지 않음.
```

2. 소스 형식에 관한 규칙

- **하나의 소스 파일은 2000줄 이내로 작성한다.**
  하나의 파일에 너무 많은 코드를 작성할 경우, 프로그램의 문맥 파악이 어렵고, 파일 관리 및 유지 보수 측면에서도 효율성이 저하된다.
- **한 줄의 길이는 80자 이내의 문자로 한다.**
  한 줄에 많은 코드를 작성할 경우 가독성이 저하된다.
- **메서드나 함수의 내용은 70줄 이내로 작성한다.**
  한 화면에 전체 내용이 보이지 않아, 메서드(혹은 함수)의 앞부분과 뒷부분의 문맥을 파악이 어려워 가독성이 저하된다.
- **중괄호는 시작 문장의 마지막 열에 삽입, 닫는 중괄호는 새로운 시작 열에 삽입한다. (if문, else-if문, else문, for문, while문, do-while문, switch문, try-catch문)**
  중괄호를 별도의 줄에 삽입하는 것은 코드길이가 더 길어져서 아래위로 스크롤해가며 읽어야 함으로 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
// if, else 문의 여는 중괄호는 다음 줄에 삽입.
if (condition)
{
    statements;
}
else (condition)
{
    statements;
}
```

■ Good 코드의 예-Java/C

```
// if, else 문의 여는 중괄호는 문장의 마지막 열에 삽입.
if (condition) {
    statements;
} else {
    statements;
}
```

- **하나의 문장이 여러 줄로 작성될 경우 아래와 같은 규칙에 의해 행을 나눈다.**

  a. 80자 초과 시, 쉼표 다음 문자부터 새로운 행을 시작한다.

  b. 이전 행과 동일한 수준의 표현식과 열을 맞춘다.

  하나의 문장을 여러 줄로 작성하는 경우 명확하게 구분하지 않으면 소스코드의 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
int result = getEmployees(records_from_employee_table, read_count, records_to_employee
                    _table, write_count);
// 한 문장이 여러 줄에 작성될 경우 쉼표 다음 문자부터 새로운 행을 시작하지 않음.
// 2 번째 줄에서 들여 쓰기가 이전 행과 다름.
```

■ Good 코드의 예-Java/C

```
int result = getEmployees(records_from_employee_table, read_count,
                records_to_employee_table, write_count);
// 한 문장이 여러 줄에 작성될 경우 인자의 “,” 이후 내용을 다음 줄에 위치.
// 2 번째 줄에서 들여 쓰기가 이전 행과 동일 수준의 열을 맞춤.
```

- **동일 수준의 문장은 같은 위치에서 시작하고 끝난다.**

  문장의 시작/끝을 명확하게 구분하기 위해서는 시작/끝의 위치를 맞추어야 한다. 같은 수준의 문장에 동일한 들여쓰기를 적용하지 않을 경우 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
if(score==MAX) {
    score--;
return score; // 동일 수준의 들여쓰기를 적용 하지 않음.
}
```

■ Good 코드의 예-Java/C

```
if(score==MAX) {
    score--;
    return score; // 동일 수준의 들여쓰기를 적용.
}
```

3. 주석에 관한 규칙

- **프로그램은 최초작성자, 최초작성일, 최종변경일, 목적, 개정이력 및 저작권을 기술하는 주석으로 시작해야 한다.**

  각 파일의 목적을 정확히 파악할 수 있도록 주석처리하여 소스코드의 가독성과 유지보수성을 높인다.

  - 최초작성자 : 프로그램을 최초 작성한 개발자명
  - 최초작성일 : 프로그램을 최초 작성한 일자
  - 최종변경일 : 프로그램을 최종 변경한 일자
  - 목적 : 프로그램을 작성한 목적
  - 개정이력 : 프로그램을 변경한 변경자, 변경일자 및 변경내용
  - 저작권 : 프로그램을 최초 작성한 개발회사

■ Bad 코드의 예 -Java

```
// 프로그램의 주석 정보가 없음
public class StringHelper {
    ...
}
```

■ Good 코드의 예-Java

```
/*
* 최초작성자 : 김길동
* 최초작성일 : 2009.10.10.
* 최종변경일 : 2009.10.20.
* 목적 : 문자열을 조작하고 처리하는 유틸리티
* 개정이력 : 홍길동, 2009.10.15, utf-8 지원
*            홍길동, 2009.10.20, checksum 기능 추가
* 저작권 : 대한민국(주)
*/
public class StringHelper {
    ...
}
```

※ C코드도 동일하게 적용한다.

- **메서드나 함수 주석은 목적, 매개변수, 반환 값 및 변경이력을 기술하는 주석으로 시작한다.**

  메서드나 함수 정의 앞부분에 다음의 정보를 기술하면, 유지보수성과 가독성이 향상된다.

  - 목적 : 메서드나 함수를 작성한 목적
  - 매개변수 : 메서드나 함수의 인자로 사용되는 변수 설명
  - 반환값 : 메서드나 함수의 결과값 설명
  - 변경이력 : 메서드나 함수를 변경한 변경자, 변경일자 및 변경내용

■ Bad 코드의 예-Java

```
// 메서드의 주석 정보가 없음.
private Connection getConnection() {
    ....
    return conn;
}
```

■ Good 코드의 예-Java

```
/*
* 목적 : DB Connector 정보를 가지고 옴.
* 매개변수 : dbConnInfo : Connection 객체를 얻기 위한 정보.
* 반환값 : Java.sql.connection
* 개정이력 : 홍길동, 2009.10.20, null 체크 기능 추가
*/
private Connection getConnection(Properties dbConnInfo) {
    ....
    return conn;
}
```

※ C코드도 동일하게 적용한다.

4. 변수 선언에 관한 규칙

- **같은 용도의 변수는 같은 선언에 둔다.**
  하나의 문장에 구분자를 통해 서로 다른 용도의 변수를 선언하면 각 변수에 대한 의미를 정확하게 전달하기 어렵다.

■ Bad 코드의 예-Java/C

```
// 동일용도(인덱스)로 사용되는 변수를 여러 줄에 선언.
int i = 0;
int j = 0;
...
for ( i = 0; i < 10; i++) {
    for ( j = 0; j < 10; j++) {
        ...
    }
}
```

■ Good 코드의 예-Java/C

```
int i = 0, j = 0;    // 동일 용도(인덱스)로 사용되는 변수를 동일 줄에 선언.
...
for ( i = 0; i < 10; i++) {
    for ( j = 0; j < 10; j++) {
        ...
    }
}
```

- **불필요한 변수를 선언하지 않는다.**
  불필요한 변수 선언으로 개발자로 하여금 코드에 대한 이해를 저하시켜 유지보수가 어렵다.

■ Bad 코드의 예-Java/C

```
int unusedVar;  // 사용되지 않은 변수.
```

■ Good 코드의 예-Java/C

```
int usedVar = 0;
....
usedVar = getData(usedVar); // 사용된 변수.
```

- **배열을 선언하는 경우 반드시 요소 수를 명시적으로 선언하거나 초기화에 의해 묵시적으로 결정되도록 한다.**
  요소 수가 결정되지 않은 배열은 배열 범위에 대해 예측할 수 없으므로 배열 범위를 벗어나는 참조를 통한 보안 문제를 유발할 수 있다.

■ Bad 코드의 예-Java/C

```
int arry[]; // 배열을 초기화하지 않음.
```

■ Good 코드의 예-Java/C

```
int arry[] = {1, 2}; // 배열을 초기화.
```

- **지역 변수는 선언과 동시에 초기화한다.**
  지역 변수는 선언과 동시에 초기화 하여 초기화되지 않은 변수의 사용을 사전에 방지해야 한다.
  ※ C인 경우 지역 변수 초기화가 필수이지만, Java인 경우는 컴파일러가 이를 탐지하기 때문에 해당 사항이 없다.

■ Bad 코드의 예-C

```
int poorVar; // 초기화 안 됨.
```

■ Good 코드의 예-C

```
int goodVar = 0; // 초기화 됨.
```

5. 상수에 관한 규칙

- **8진수로 표현된 상수를 사용하지 않는다.**
  8진수는 가독성이 매우 떨어지므로, 10진수나 16진수 표기법을 사용한다.

■ Bad 코드의 예-Java/C

```
int octal = 0377;  // 8진수 표현.
```

■ Good 코드의 예-Java/C

```
int hexa = 0xFF; // HEX 표현
int decimal = 255; // 10진수 표현
```

- **숫자 리터럴을 직접적으로 소스코드 안에 삽입하지 않는다.**
  반복문의 카운터로 사용될 수 있는 (-1, 0, 1)을 제외한 숫자는 가독성을 저하시킨다. 소스코드에 직접 기술한 숫자 리터럴은 그 의미를 파악하기 어렵다.

■ Bad 코드의 예-Java/C

```
int circleArea = 10 * 10 * 3; // 숫자를 하드코딩.
```

■ Good 코드의 예-Java

```
// 계산될 숫자를 미리 선언.
final int RADIUS = 10;
final int APPROXIMATED_PI = 3;
...
int circleArea = RADIUS * RADIUS * APPROXIMATED_PI;
```

■ Good 코드의 예-C

```
// 계산될 숫자를 미리 선언.
const int RADIUS = 10;
const int APPROXIMATED_PI = 3;
...
int circleArea = RADIUS * RADIUS * APPROXIMATED_PI;
```

6. 수식에 관한 규칙

- **단항 연산자는 피연산자와 붙여 쓴다.**
  단항 연산자는 피연산자가 무엇인지 명확하게 알 수 있도록 붙여쓰지 않을 경우 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
var ++; // 피연산자와 붙여 쓰지 않음.
```

■ Good 코드의 예-Java/C

```
var++; // 피연산자와 붙여 사용.
```

- **.(dot) 연산자를 제외한 모든 이항 연산자 전후는 공백으로 구분한다.**
  이항 연산자 전후에 공백을 삽입하는 것은 연산자와 피연산자의 구분을 명확하게 할 수 있으며, 소스코드의 가독성을 높일 수 있다.

■ Bad 코드의 예-Java/C

```
total = val1+val2*val3-val4; // 연산자 전후에 공백 없음.
```

■ Good 코드의 예-Java/C

```
total = val1 + (val2 * val3) - val4; // 연산자 전후에 공백을 둠.
```

- **조건부 연산자에서 "?" 연산자 앞에 이항 연산식이 나타날 경우 괄호로 구분한다.**
  조건부 연산자는 3개의 피연산자를 갖기 때문에 복잡한 구조를 갖는다. 또한 첫 번째 피연산자가 이항 연산식이 될 경우 혼란을 가중시킬 수 있다.

■ Bad 코드의 예-Java/C

```
x >= 0 ? x : -x; // 이항 연산식에 괄호 없음.
```

■ Good 코드의 예-Java/C

```
(x >= 0) ? x : -x; // 이항 연산식에 괄호 붙임.
```

- **증감 연산자는 수식에서 다른 연산자와 결합하여 사용하지 않는다.**
  증감 연산자가 다른 연산자와 결합하여 사용되면 직관적으로 이해할 수 없게 되어 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
int val1, val2, sum=0;
...
sum = val1 + (--val2);  // 증감 연산자가 다른 연산자와 결합.
```

■ Good 코드의 예-Java/C

```
int val1, val2, sum=0;
...
--val2;
sum = val1 + val2;  // 증감 연산자가 다른 연산자와 결합하지 않음.
```

- **.(dot), ->(포인터) 연산자를 제외한 모든 이항 연산자 전후는 공백으로 구분한다.**
  이항 연산자와 피연산자 전후에 공백을 두는 것은 구분을 명확하게 하지 않을 경우 가독성이 저하된다.
  ※ 단, Java는 포인터 연산자가 없음.

■ Bad 코드의 예-Java/C

```
int total,valueA,valueB = 0;  // 연산자 전후에 공백 없음.
...
total=valueA+valueB;  // 연산자 전후에 공백 없음.
```

■ Good 코드의 예-Java/C

```
int total, valueA, valueB = 0; // 연산자 전후에 공백을 둠
...
total = valueA + valueB; // 연산자 전후에 공백을 둠.
```

- **3 개 이상의 연산자를 사용하는 경우 괄호로 연산의 우선순위를 표현한다.**
  연산자의 우선순위와 결합법칙에 적합한 수식도 3 개 이상의 연산자를 사용하는 경우 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
if(valueA==0&&valueB==0)  // 연산자 우선순위를 위한 구분 없음.
```

■ Good 코드의 예-Java/C

```
 if( (valueA == 0) && (valueB == 0) ) // 연산자 우선순위를 위해 괄호를 둠.
```

- **비트 연산자는 부호있는 자료에 사용하지 않는다.**
  비트 연산자는 부호의 의미를 포함하지 않으므로, 부호있는 연산자에 시프트 연산을 하게 되면 부호 비트와 숫자 값을 가지는 비트가 그 의미를 잃게 된다.

■ Bad 코드의 예-Java/C

```
int j = -10;
int k = j << 2; // 음수에 비트연산자 적용.
```

■ Good 코드의 예-Java/C

```
unsigned int j = 10;
unsinged int k = j << 2; // 양수에 비트연산자 적용.
```

7. 문장에 관한 규칙

- **switch-case 문장에서 case 문에 break 문이 없는 경우 주석을 작성한다.**
  switch-case 문장에서 case 문에 break 문이 없는 경우, 다음 case 문으로 제어의 흐름이 넘어간다. 주석이 없는 경우 프로그램 이해도가 떨어져 가독성이 저하된다.

■ Bad 코드의 예-Java/C

```
switch (condition) {
    case 1:
        statements // break 문이 없는 경우 주석문이 없음.
    case 2:
        statements
        break;
    default:
        statements
        break;
}
```

■ Good 코드의 예-Java/C

```
switch (condition) {
    case 1:
        statements    // case 1 이후 case 2가 동일하게 적용이 되어야 함.
    case 2:
        statements
        break;
    default:
        statements
        break;
}
```

- **switch-case 문에서는 반드시 default 문을 작성하고 마지막 항목에 위치한다.**
  default 문을 작성하지 않으면, 모든 case문을 만족하지 않을 경우에 대한 처리가 누락이 되어 프로그램 오류 발생 가능성이 높다

■ Bad 코드의 예-Java/C

```
// default문 없음.
switch (condition) {
    case 1:
        statements
        break;
    case 2:
        statements
        break;
}
```

■ Good 코드의 예-Java/C

```
switch (condition) {
    case 1:
        statements
        break;
    case 2:
        statements
        break;
    default: // default문 작성.
        statements
        break;
}
```

- **goto 문을 사용하지 않는다.**

  goto 문은 프로그램 제어의 흐름을 복잡하게 하여, 가독성이 저하되며, 제어의 흐름이 예측할 수 없는 방향으로 진행되어 오류발생 가능성이 높다.

  ※ Java에서는 goto문 대용으로 break문 사용

■ Bad 코드의 예-Java

```
L1: while (true) {
      if (condition) {
      break L1;  // break문 사용으로 흐름 파악이 복잡.
      }
}
```

■ Good 코드의 예-Java

```
while (true) {
    ...
    if ( fun1(condition) ) {
        break;
    }
}
boolean fun1(int condition) {
     return (condition == 1) : true ? false;
}
```

■ Bad 코드의 예-C

```
L1: while (true) {
    if (condition) {
        goto L1;  // goto문 사용으로 흐름 파악이 복잡.
    }
}
```

■ Good 코드의 예-C

```
while (true) {
    ...
    if ( fun1(condition) ) {
        break;
    }
}
boolean fun1(int condition) {
     return (condition == 1) : true ? false;
}
```

- **for 문을 제어하는 수식에 실수 값을 사용하지 않는다.**
  실수값은 표현 한계로 인하여 부정확한 값을 가지는 경우가 대부분이므로 의도하지 않은 동작을 유발할 수 있다.

■ **Bad 코드의 예-Java/C**

```
float i = 0.0; // for문의 제어 인자로 실수를 사용
for(i=0.0; i<1.0; i=i+0.1)
```

■ **Good 코드의 예-Java/C**

```
int i = 0; // for문의 제어 인자로 정수를 사용
for(i=0; i<10; i++)
```

- **for 문을 제어하는 수치 변수는 루프 내에서 변화되지 않아야 한다.**
  루프 내에서 루프 제어 변수가 변화할 경우 반복 횟수를 예측할 수 없어 프로그램 이해도가 떨어지고, 잘못 계산된 루프제한 변수는 무한루프를 발생시킨다.

■ **Bad 코드의 예-Java/C**

```
int i =0, j = 10;
for( i = 0; i < 10; i++ ) {
    ...
    i = i + j; // for문의 제어 변수가 loop내에서 사용.
}
```

■ **Good 코드의 예-Java/C**

```
int i =0, j = 10;
for( i = 0, j = 0 ; i < j; i++, i++) { // for문의 제어 변수를 loop내에서 사용하지 않음.
    ...
}
```

- **반복문 내부에서 반복중단을 위한 break 문은 가능한 한번만 사용하도록 한다.**
  break 문이 산재되어 있는 경우 프로그램의 동작을 예측하기 어렵다.

■ Bad 코드의 예-Java/C

```
// 반복문을 중단하기 위해 여러 번 break 문 사용.
int value = 0;
while(1) {
    if(value == 0) {
        break;
    }
    else if(value == 100) {
        break;
    }
}
```

■ Good 코드의 예-Java/C

```
// 반복문을 중단하기 위해 한번 break 문 사용.
int value=0;
while(1) {
    ...
    if( (value == 0) || (value == 100) ) {
        break;
    }
    ...
}
```

- **if ~ else if 문은 반드시 else 문으로 끝나도록 한다.**
  else 문으로 종료하지 않는 경우, 처리가 누락되는 경우가 발생하여 오류를 발생시킬 수 있다.

■ Bad 코드의 예-Java/C

```
// if ~ else if문에 else 문이 없음.
if(value == 0) {
    ...
}else if (value < 0) {
    ...
}
```

■ Good 코드의 예-Java/C

```
// if ~ else if문에 else 문이 존재.
if(value == 0) {
    ...
} else if (value < 0) {
    ...
} else {
    ...
}
```

제2절 Java 의존적인 코딩규칙

1. 명칭에 관한 규칙

- **클래스, 인터페이스, 상수, 변수 및 메서드의 명칭은 구분하여 작성한다.**
  (a) 클래스의 이름은 대문자로 시작한다.
  (b) 인터페이스의 이름은 대문자로 시작한다.
  (c) 상수의 이름은 "_" 및 대문자로 한다.
  (d) 변수의 이름은 소문자로 시작한다.
  (e) 메서드의 이름은 소문자로 시작하고 첫 번째 단어는 동사로 작성한다.

■ Bad 코드의 예-Java

```
(a) public class person { }  // 클래스명이 소문자로 시작.
(b) public interface person { } // 인터페이스명이 소문자로 시작.
(c) final int Max_Length = 255; // 상수명에 소문자 사용.
(d) int Variables;              // 변수명이 대문자로 시작.
(e) public int Info() { }     // 메서드명이 대문자로 시작하고 동사로 시작하지 않음.
```

■ Good 코드의 예-Java

```
(a) public class Person { }  // 클래스명이 대문자로 시작.
(b) public interface Person { } // 인터페이스명이 대문자로 시작.
(c) final int MAX_LENGTH = 255; // 상수명에 "_" 및 대문자 사용.
(d) int variables;              // 변수명이 소문자로 시작.
(e) public int getInfo() { }     // 메서드명이 소문자로 시작하고 동사로 시작.
```

- **동일한 클래스명과 메서드명을 가져서는 안 된다.**
  생성자가 아닌 메서드는 자기 자신의 클래스와 동일한 명을 사용하면, 가독성이 떨어지고. 개발자가 구성자로 오해할 가능성이 있어 혼란을 준다.

■ Bad 코드의 예-Java

```
public class MyClass {
    public MyClass() {}
    public void MyClass() { // 구성자로 오해 가능성이 있음.
        ...
    }
}
```

■ Good 코드의 예-Java

```
public class MyClass {
    public MyClass() {}
    public void getMyClassName() { // 클래스명과 다름.
        ...
    }
}
```

2. 프로그램 작성 순서에 관한 규칙

- **Java소스 파일의 첫 번째 문장을 package로 시작한다**

  package 문장이 없는 프로그램은 클래스명간 충돌이 발생할 수 있으며, 클래스의 접근 제어가 용이하지 않음으로 향후 확장 시에 비용이 많이 든다.

  ※ package 문장 다음에 import 문을 기술한다.

■ Bad 코드의 예-Java

```
// package 문장이 없음.
import Java.io.File;
```

■ Good 코드의 예-Java

```
// package 문장으로 시작.
package com.srs.admin.service;
import Java.io.File;.
```

3. 클래스의 정의에 관한 규칙

- **계층 구조의 상위에 위치한 클래스는 추상 클래스로 작성한다.**

  계층 구조의 상위에 위치한 클래스는 범용성을 가져야 하기 때문에 추상적으로 작성하는 것이 좋다.

■ Bad 코드의 예-Java

```
// 클래스 계층 구조상 최상위 클래스를 추상 클래스로 정의하지 않음.
class Directory {
...
}
class FileDirectory extends Directory {
...
}
class UnixFileDirectory extends FileDirectory  {
...
}
```

■ Good 코드의 예-Java

```
// 클래스 계층 구조상 최상위 클래스를 추상 클래스로 정의.
abstract class Directory {
...
}
class FileDirectory extends Directory {
...
}
class UnixFileDirectory extends FileDirectory {
...
}
```

4. 필드와 메서드 정의에 관한 규칙

- **추상클래스에는 하나 이상의 추상메서드를 작성한다.**
  클래스의 확장과 기능의 유연성을 확보하기 위해 추상클래스에 추상메서드를 정의하는 것이 좋다.

■ Bad 코드의 예-Java

```
public abstract class Foo {
// 정의된 모든 메서드가 추상메서드가 아님.
     void int method1() {
          ...
     }
     void int method2() {
          ...
     }
}
```

■ Good 코드의 예-Java

```
public abstract class Foo {
// 정의된 메서드가 추상메서드.
     abstract void int method1() {
          ...
     }
     void int method2() {
          ...
     }
}
```

- **추상클래스의 빈 메서드는 추상메서드로 정의해야 한다.**
  추상클래스에 빈 메서드는 상속을 받은 하위클래스에 대한 해당 메서드의 구현을 강제화 할 수 없기 때문에 추상메서드로 선언한다.

■ Bad 코드의 예-Java

```
public abstract class Foo {
     void int method1() {} // 추상클래스에 구현이 없는 메서드
     abstract void int method2() {
          ...
     }
}
```

■ Good 코드의 예-Java

```
public abstract class Foo {
     abstract  void int method1(); // 추상메서드 정의.
     abstract void int method2() {
          ...
     }
}
```

- **오버라이드 메서드를 호출하는 생성자 사용을 금지한다.**

  하위클래스의 생성자에서 상위클래스의 생성자를 호출 시, 상위클래스의 생성자에서 오버라이드 메서드를 호출하면, 상위클래스의 인스턴스 생성이 완료되지 않은채, 객체의 메서드를 호출하기 때문에 NullPointerException 예외가 발생한다.

■ Bad 코드의 예-Java

```
public class SeniorClass {
    public SeniorClass(){
        toString(); //구성자에 오버라이드 메서드를 호출.
    }
    public String toString() {
        return "SeniorClass";
    }
}

public class JuniorClass extends SeniorClass {
    private String name;
    public JuniorClass() {
        super(); // NullPointerException 에러가 발생.
        name = "JuniorClass";
    }
    public String toString() {
        return name.toUpperCase();
    }
}
```

■ Good 코드의 예-Java

```
public class SeniorClass {
    public SeniorClass(){}
    public String toString() {
      return "SeniorClass";
    }
}

public class JuniorClass extends SeniorClass {
    private String name;
    public JuniorClass() {
        name = super.toString(); // 상위클래스의 인스턴스 생성 후 메서드 호출
    }
    public String toString() {
        return name.toUpperCase();
    }
}
```

- **클래스 정적 변수 또는 메서드는 클래스 이름을 통해 참조해야 한다.**
  객체를 생성한 클래스 정보가 객체 이름에 반영되지 않기 때문에, 클래스 이름을 통해 참조하여야 클래스 변수/메서드 소속을 명확하게 할 수 있고 의미파악이 용이하다.

■ Bad 코드의 예-Java

```
public class SeniorClass {
    public static String name;
    public SeniorClass(){}
    public String getName() {
        return name;
    }
}
public class JuniorClass {
    public JuniorClass() {}
    public String getName() {
        SeniorClass sc = new SeniorClass();
        return sc.getName(); // static 변수 참조를 객체명을 사용하여 참조.
    }
}
```

■ Good 코드의 예-Java

```
public class SeniorClass {
    public static String name;
    public SeniorClass(){}
    public String getName() {
        return name;
    }
}
public class JuniorClass {
    public JuniorClass() {}
    public String getName() {
        return SeniorClass .getName(); // static 변수 참조를 클래스명을 사용하여
                                       // 참조.
    }
}
```

5. 상속에 관한 규칙

- **서브클래스는 슈퍼클래스에서 사용한 변수이름과 동일한 이름의 변수를 사용하지 않는다.**
  서브클래스에서 슈퍼클래스와 동일한 이름의 변수는 잘못 참조로 인한 데이터 오류가 발생할 수 있다.

■ Bad 코드의 예-Java

```
public class SeniorClass {
    public String name;
    ...
}
public class JuniorClass extends SeniorClass {
    public String name;  // 상위클래스와 동일한 멤버변수 사용
    public JuniorClass() {}
    ...
}
```

■ Good 코드의 예-Java

```
public class SeniorClass {
    public String name;
    ...
}
public class JuniorClass extends SeniorClass {
    public String firstName;  // 상위클래스와 동일하지 않은 멤버변수 사용
    public JuniorClass() {}
    ...
}
```

6. 자료형에 관한 규칙

- **volatile을 사용하지 않는다.**
  Java의 메모리 모델에 대한 전문적인 지식이 필요하며, 유지보수 또는 이식성을 위해서 사용해서는 안 된다.

■ Bad 코드의 예-Java

```
public class Delux {
    private volatile String name;  // volatile 사용.
    ...
}
```

■ Good 코드의 예-Java

```
public class Delux {
    public String name; // volatile 사용 안 함.
    ...
}
```

- **short 타입을 사용하지 않는다.**
  short 타입은 JVM 내부적으로 int로 변환하여 계산한 이후 다시 short 타입으로 변환되므로, short 타입은 계산 시에 많은 부정적인 영향을 미친다.

■ Bad 코드의 예-Java

```
public class Delux {
    private short value;  // short 타입 사용.
    ...
    private void Cal(short data) {
        data = data + 10;
        this.value = data;
    }
}
```

■ Good 코드의 예-Java

```
public class Delux {
    private int value;  // short 대신 int 타입 사용.
    ...
    private void Cal(int data) {
        data = data + 10;
        this.value = data;
    }
}
```

- **가능한 안전한 자료형을 사용한다.**

  실수 표현으로 float보다 double을 사용하고, 논리값을 갖는 변수는 int 보다 boolean과 같은 자료형을 사용하는 것이 코딩이 간단하고 에러를 감소시킨다.

■ Bad 코드의 예-Java

```
public class Delux {
    private float value; // 실수 표현.
    private int value;  // 1:true, -1:false 용도로 사용.
    ...
}
```

■ Good 코드의 예-Java

```
public class Delux {
    private float value; // 실수 표현.
    private boolean value;  // true, false 용도로 사용.
    ...
}
```

7. 문자열에 관한 규칙

- **모든 문자열 연산에 대해 null 값을 검사한다.**
  NullPointerException으로부터 프로그램을 보호할 수 있다.

- **문자열 변수는 null로 초기화하지 않는다.**
  null로 초기화된 문자열 변수는 NullPointerException을 발생할 수 있다.

- **문자열 비교시 ==를 사용 하지 말고 equals()를 사용한다.**
  문자열의 비교시 "=="를 사용하면 항상 거짓이므로 프로그래머가 의도한 방식으로 수행 되지 않기 때문에 결과를 예측할 수 없다.

■ Bad 코드의 예-Java

```
String fname = null ; // null로 초기화
String lname = null ; // null로 초기화

lname = getLastName(); // null 값 체크 없음
if ( lname == fname ) {  // String을 == 비교
      ...
}
```

■ Good 코드의 예-Java

```
String fname = "" ; // null blank로 초기화
String lname = ""1 ; // null blank로 초기화

lname = getLastName();
if ( lname^=null && lname.equals(fname) ) // lname에 대한 null체크 및 equals() 비교
```

8. 캡슐화에 관한 규칙

- **클래스 멤버변수를 public으로 선언하지 않는다.**
  public으로 선언할 경우, 객체지향언어의 특성인 캡슐화에 위배될 수 있으며, 외부에서의 접근이 가능해져, 예상하지 못한 접근으로 인해 프로그램이 오작동 할 수 있다.

■ Bad 코드의 예-Java

```
public class Delux {
    public String name; // 멤버변수를 public으로 선언
    ...
}
```

■ Good 코드의 예-Java

```
public class Delux {
    private String name; // 멤버변수를 private으로 선언
    ...
}
```

9. 패키지에 관한 규칙

- **패키지 선언 시 불필요한 패키지와 폐기된 API를 사용하지 않는다.**
  사용하지 않는 패키지를 import하면 메모리가 비효율적으로 사용되며, 폐기된 API를 사용하면 런타임 시 시스템에서 에러가 발생할 가능성이 높다.

■ Bad 코드의 예-Java

```
import Java.util.HashMap;
import Java.util.ArrayList;  // 프로그램에서 사용하지 않는 패키지 사용.
import Java.util.Date;       // deprecated API.
...
HashMap hm = new HashMap();
...
```

■ Good 코드의 예-Java

```
import Java.util.HashMap;
// 프로그램에서 사용하지 않는 패키지(Java.util.ArrayList)는 삭제.
import Java.util.Calendar;  // Java.util.Date를 Java.util.Calendar로 대체.
...
HashMap hm = new HashMap();
...
```

- **패키지를 import 할 때 "*"를 사용하지 않는다.**
  "*"는 모든 해당 클래스를 import 하기 때문에 시스템 메모리 효율을 저하시키고, 정확한 import 클래스를 알 수 없고 클래스 간 의존성을 이해하기 어렵다.

■ Bad 코드의 예-Java

```
import Java.io.*;   // 패키지를 import시 "*"사용.
```

■ Good 코드의 예-Java

```
import Java.io.File;   // 프로그램에서 사용하는 클래스만 import함.
```

10. 동기화에 관한 규칙

- **메서드가 public이고 멤버 변수 값을 사용할 경우 동기화 메서드로 정의한다.**
  쓰레드 사이의 멤버 변수를 사용할 때 동기화를 고려하지 않으면, 멤버 변수의 상태 변화로 인해 예측하지 못한 결과가 발생될 수 있다.

■ Bad 코드의 예-Java

```
Class MyClass {
    private int numOfUsers; /* 현재의 동시 사용자 수를 표시 */
    public void addCurrentUsers() {
        ...
        numOfUsers++;
    }
    public void removeCurrentUsers() { // 동기화 메서드 사용 안함.
        ...
        numOfUsers++;
    }
}
```

■ Good 코드의 예-Java

```
Class MyClass {
    private int numOfUsers; /* 현재의 동시 사용자 수를 표시 */
    public void synchronized addCurrentUsers() {
        ...
        numOfUsers++;
    }
    public void synchronized removeCurrentUsers() { // 동기화 메서드 사용.
        ...
        numOfUsers++;
    }
}
```

- **실행 속도에 큰 영향이 없으면, 동기화문 보다 동기화 메서드를 사용한다.**
  문장내에 동기화 문을 사용하는 경우, 프로그램의 확장 시 동기화 처리가 복잡하고, 캡슐화가 용이하지 않으며, 보안 취약성이 증가한다.

■ Bad 코드의 예-Java

```
public void func1 {
    synchronized {  // 문장 내에서 동기화문 사용.
    ...
    }
}
```

■ Good 코드의 예-Java

```
public void synchronized func1 {  // 동기화 메서드 사용.
    ...
}
```

11. 가비지 콜렉터에 관한 규칙

- **명시적으로 가비지 콜렉터를 호출하는 코드를 작성하지 않는다.**
  가비지 콜렉터의 잦은 호출은 시스템 성능에 악영향을 준다. 시스템의 성능 및 메모리 관리의 효율성을 위하여 가비지 콜렉터는 WAS가 담당하도록 한다.

■ Bad 코드의 예-Java

```
System.gc(); // GC(가비지 콜렉터) 사용.
```

■ Good 코드의 예-Java

```
// GC(가비지 콜렉터) 사용을 삭제함.
```

12. 제한된 자원에 관한 규칙

- **제한된 시스템 자원은 사용 후 반드시 해제한다.**
  입출력 스트림, 공유 자원 등은 사용 후 반드시 해제한다. 특히, 해제하지 않은 공유 자원은 메모리 부족 및 교착 상태를 유발하여 서비스 거부를 발생할 수 있다.

■ Bad 코드의 예-Java

```
// 입력 스트림을 사용 후 해제하지 않음
FileInputStream fis = new FileInputStream(new File(a.txt"));
..
```

■ Good 코드의 예-Java

```
FileInputStream fis = new FileInputStream(new File(a.txt"));
..
finally {
    if ( fls != null ) {
        fis.colse(); // 입력 스트림을 사용 후 해제
    }
}
```

13. 예외 처리에 관한 규칙

- **광범위한 예외 클래스인 Exception을 사용하여 예외처리를 하지 않는다.**
  광범위한 예외처리에 의존할 경우, 호출하는 프로그램에서 모든 예외처리를 해야 하므로, 호출 프로그램의 예외처리 로직이 복잡하여 오류 가능성이 높다.

- **예외를 처리할 때, 민감한 정보를 외부에 노출하지 않는다.**
  민감한 정보를 외부에 노출시키면, 해커가 많은 정보를 획득할 수 있다.

■ Bad 코드의 예-Java

```
try {
    ...
} catch ( Exception e) { // 광범위한 예외처리
    System.out.println(cardNumber+"확인"); // 카드번호가 콘솔에 노출
}
```

■ Good 코드의 예-Java

```
try {
    ...
} catch ( MyException e) {  // 개발자가 정의한 예외 처리
    throws new MyException("에러 확인"); // 간단한 메시지로 에러 처리
}
```

14. J2EE 애플리케이션 개발에 관한 규칙

- **J2EE 상에서 프로그래밍 할 때, 쓰레드를 직접 사용하지 않는다.**
  개발자가 쓰레드를 직접 사용하면 WAS와 충돌이 발생할 수 있으며, 이로 인해 시스템 교착상태를 유발하여 서비스 거부를 발생할 수 있다.

■ Bad 코드의 예-Java

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
// Thread를 직접 생성.
    Runnable r = new Runnable() {
        public void run() {
            ...
        }
    };
    new Thread(r).start();
}
```

■ Good 코드의 예-Java

```
public void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
// 쓰레드 대신에 Java 클래스의 인스턴스 생성.
    New MyClass().main();
}
Public class MyClass {
    public void main() {
        ...
    }
};
```

- **getClassLoader()를 사용하지 않는다.**
  J2EE에서 getClassLoader()의 사용은 개발자가 의도한 대로 프로그램이 동작되지 않을 수 있어 예기치 않은 예외 상황이 발생할 수 있다.

■ Bad 코드의 예-Java

```
ClassLoader cl = Bar.class.getClassLoader();  // getClassLoader() 메서드 사용.
```

■ Good 코드의 예-Java

```
ClassLoader cl = Thread.currentThread().getContextClassLoader(); // 반드시 현재의 Thread 객체를 통해서 클래스 로더를 얻음.
```

- **Java.io.Serializable를 구현한 클래스는 serialVersionUID 멤버변수를 제공한다.**
  J2EE에서 Java.io.Serializable를 구현한 클래스가 serialVersionUID의 멤버변수를 사용하지 않으면, JVM이 계산한 값에 의해서 default값으로 처리되어, deserialization하는 과정에서 예기치 않은 예외를 발생할 수 있다.

■ Bad 코드의 예-Java

```
// serialVersionUID를 사용 안함.
public class Foo implements Java.io.Serializable {
      private String name;
}
```

■ Good 코드의 예-Java

```
public class Foo implements Java.io.Serializable {
    private static final long serialVersionUID = 101L; // serialVersionUID 사용.
    private String name;
}
```

- **System.exit()을 사용하지 않는다.**
  공격자가 System.exit()의 루틴을 알고 있으면, 악성코드를 삽입하여 해당 서비스의 호출을 통한 시스템의 서비스 거부 공격이 가능하다.

■ Bad 코드의 예-Java

```
try {
     ...
} catch (ApplicationSpecificException ase) {
     System.exit(1); // System.exit() 사용.
}
```

■ Good 코드의 예-Java

```
try {
     ...
} catch (ApplicationSpecificException ase) {
     throw MyException("에러발생"); // System.exit()을 삭제하고 예외를 처리.
}
```

제3절 C 의존적인 코딩규칙

1. 명칭에 관한 규칙

- **매크로, 상수, 변수 및 함수의 명칭은 구분이 가능하게 작성한다.**
  클래스, 인터페이스, 상수, 변수 및 메서드는 명칭에 대해 명확하게 구분할 수 있도록 작성하여 프로그램의 가독성을 높인다.
  (a) 매크로의 이름은 "_" 및 대문자로 한다.
  (b) 상수의 이름은 "_" 및 대문자로 한다.
  (c) 변수의 이름은 소문자로 시작한다.
  (d) 함수의 이름은 소문자로 시작하고 첫 번째 단어는 동사로 작성한다.

■ Bad 코드의 예-C

```
(a) #define Max_Length 255  // 매크로명에 소문자 사용.
(b) const int Max_Length = 255; // 상수명에 소문자 사용.
(c) int Variables = 0;              // 변수명이 대문자로 시작.
(d) int Info() { }    // 함수명이 대문자로 시작하고 동사로 시작하지 않음.
```

■ Good 코드의 예-C

```
(a) #define MAX_LENGTH 255  // 매크로명에 "_"및 대문자 사용.
(b) const int MAX_LENGTH = 255; // 상수명에 "_"및 대문자 사용.
(c) int variables = 0;              // 변수명이 소문자로 시작.
(d) int getInfo() { }    // 함수명이 소문자로 시작하고 동사로 시작.
```

- **특정 변수의 주소를 저장하기 위한 포인터 변수의 이름은 참조하는 변수 이름의 첫 글자를 대문자로 바꾸고 앞에 "p"를 붙인다.**
  특정 변수에 대한 포인터 변수는 그 변수와 연관관계에 있다는 것을 명칭으로 나타내어 가독성을 높인다.

■ Bad 코드의 예-C

```
int count = 0;
int *sum += count;     // 포인터 변수 sum 앞에 "p"없음.
```

■ Good 코드의 예-C

```
int count, sum = 0;
int *pSum += count; // 참조하는 변수 이름의 첫 글자를 대문자로 바꾸고 앞에 "p"를 붙임.
```

- **특정 변수의 주소를 저장하기위해 사용하는 포인터 변수에 동일한 자료형의 다른 변수의 주소를 저장하지 않는다.**
  특정변수에 대한 포인터 변수에 다른 변수의 주소를 저장하게 되면 의미를 혼동하게 될 수 있는 가능성이 존재한다.

■ Bad 코드의 예-C

```
char* name;
...
name = getAddress(); // name에 getAddress() 결과 저장.
```

■ Good 코드의 예-C

```
char* name;
...
name = getName(); // name에 getName()의 결과 저장.
```

2. 상수에 관한 규칙

- **상수는 부호있는 자료형을 기본으로 사용하지만, 만약 부호없는 자료형을 사용시에는 "u"를 붙여 사용한다.**
  부호없는 자료형을 사용시에 "u'를 붙여 가독성을 높인다.

■ Bad 코드의 예-C

```
#define MAX 10; // 부호없는 자료형을 표현하지 않음.
```

■ Good 코드의 예-C

```
#define MAX 10u; // 부호없는 자료형을 "u"를 사용.
```

3. 수식에 관한 규칙

- **sizeof의 인자에 연산을 포함하지 않는다.**
  sizeof의 인자로 사용되는 연산자는 피연산자의 수식 값을 평가하지 않고, 자료형에 대한 크기만을 계산하므로, 개발자로 하여금 혼동을 줄 수 있다.

■ Bad 코드의 예-C

```
int* value = (int *)malloc(sizeof(a=b+c)); // sizeof의 인자에 수식 포함
```

■ Good 코드의 예-C

```
int a = b + c;
int* value = (int *)malloc(sizeof(a)); // sizeof의 인자에 수식 포함하지 않음.
```

4. 자료형에 관한 규칙

- **포인터 변수에 자료형이 일치하지 않는 주소나 정수 값을 저장하지 않는다.**
  포인터 변수에 자료형이 일치하지 않는 주소 값이나 정수 값을 저장하게 되면, 동작을 예측할 수 없게 되어 보안상의 문제점을 발생할 수 있다.

■ Bad 코드의 예-C

```
int *pValueInt;
char valueChar = 'A';
pValueInt = &valueChar; // 자료형이 일치하지 않음
```

■ Good 코드의 예-C

```
int *pValueInt;
int valueInt = 0;
pValueInt = &valueInt; // 자료형이 일치
```

- **비트 필드는 unsigned int나 signed int 형으로 선언한다.**
  비트 필드는 unsigned/signed int 형 이외의 자료형으로 선언된 경우, 동작이 정의되어 있지 않아 예측할 수 없게 되어 보안상의 문제점을 발생할 수 있다.

■ Bad 코드의 예-C

```
struct st {
    char a : 2;    // 비트필드에  char 사용
    unsigned int b : 3;
};
```

■ Good 코드의 예-C

```
struct st {
    unsigned int a : 2;  // 비트필드에  unsigned int사용
    unsigned int b : 3;  // 비트필드에  unsigned int사용
};
```

5. 초기화에 대한 규칙

- **배열 초기화는 반드시 차원에 따라 중괄호를 사용한다.**
  배열을 초기화할 때 차원에 따라 중괄호를 중복하여 사용하지 않으면 초기화되는 값을 혼동할 여지가 있어 소스코드의 가독성을 저하시키고 에러가 발생할 수 있다.

■ Bad 코드의 예-C

```
int arry[2][2]={2, 3, 4, 5}; // 데이터 초기화 시 2차원배열 구분 없음.
```

■ Good 코드의 예-C

```
int arry[2][2]={ {2, 3} , {4, 5} }; // 데이터 초기화 시 2차원배열 구분.
```

6. 동적 메모리 관리에 대한 규칙

- **메모리 할당과 해제는 동일한 모듈의 동일한 수준에서 수행한다.**
  메모리 할당/해제가 같은 모듈이나 동일 블록 내에서 수행되지 않으면, 할당된 메모리가 해제되지 않아 메모리 누수나 중복해제로 인한 문제가 발생할 수 있다.

■ Bad 코드의 예-C

```
while(1) {
    value = (int *) malloc(sizeof(int));
    …
}
free(value);  // while loop의 외부에서 free 함
…
```

■ Good 코드의 예-C

```
while(1) {
    value = (int *) malloc(sizeof(int));
    …
    free(value); // while loop 내부에서 free 함
}
…
```

- **메모리 할당 함수의 초기화 여부를 인지하여 사용한다.**
  메모리 할당 함수사용 시 할당된 메모리 영역에 가비지 데이터가 존재하여 계산 시 예측할 수 없는 동작을 발생할 수 있다.

- **동적으로 할당된 메모리 블록을 참조하기 위해서 사용되는 포인터 변수는 반드시 널 포인터인지 여부를 검사한 후 사용한다.**
  동적으로 할당된 메모리 블록을 참조하는 포인터 변수가 널 포인터 값을 가지는 경우, 프로그램을 비정상 종료시키거나 부적절한 메모리 참조 가능성이 있다.

■ Bad 코드의 예-C

```
int *value;
value = (int *)malloc(sizeof(int)); // 함수 호출 이후 결과값 체크하지 않음
while(1) {
    …
    *value = *value + 2; // value를 초기화하지 않고 사용
    …
}
```

■ Good 코드의 예-C

```
int *value;
value = (int *)malloc(sizeof(int));
if (value == NULL) {  // 함수 호출 이후 결과값 체크
   //에러 처리 루틴
}
*value = 0; // value 초기화
while(1) {
    …
    *value = *value + 2;
    …
}
```

7. 포인터와 배열에 관한 규칙

- **포인터 변수의 산술 연산은 배열 요소에 대한 참조 목적인 경우에만 사용한다.**
  배열 요소에 대한 참조가 아닌 포인터 변수에 산술 연산을 하게 되면 예측할 수 없는 메모리 영역을 참조하게 되어 보안상의 문제가 발생할 수 있다.

■ Bad 코드의 예-C

```
unsigned char value[3] = { 'a', 'b', 'c' };
unsigned char *pValue=value;
int i = 0;
for ( i = 0; i < sizeof(value) ; i++ ) {
    printf("%c\n",pValue++;)  // 포인터가 가르키는 주소
}
```

■ Good 코드의 예-C

```
unsigned char value[3] = { 'a', 'b', 'c' };
unsigned char *pValue=value;
int i = 0;
for ( i = 0; i < sizeof(value) ; i++ ) {
    printf("%c\n",*pValue++;)  // 포인터가 가르키는 주소의 데이터
}
```

- **포인터 변수 사이의 비교 연산은 같은 배열을 참조하는 경우에만 사용한다.**
  서로 다른 배열요소를 참조하면, 각 배열요소의 초기 메모리의 위치가 상이하기 때문에 포인터간 연산은 이상동작을 수행하여 보안상의 문제를 발생할 수 있다.

■ Bad 코드의 예-C

```
unsigned char value1[3] = { 'a', 'b', 'c' };
unsigned char value2[3] = { 'A', 'B', 'C' };
unsigned char *pValue1 = value1;
unsigned char *pValue2 = value2;
if ( ( pValue2 - pValue1 ) > 0 ) { // 서로 다른  배열 요소 참조, 예측 불가
     ...
else {
     doSomething();
}
```

■ Good 코드의 예-C

```
unsigned char value[3] = { 'a', 'b', 'c' };
unsigned char *pValue1 = &value[0];
unsigned char *pValue2 = &value[1];
if ( (pValue2 - pValue1) > 0 )  { // 동일한 배열 요소 참조, 예측 가능
    ...
} else {
    doSomething();
}
```

8. 라이브러리 사용에 관한 규칙

- **setjmp 매크로와 longjmp 함수는 사용하지 않는다.**
  setjmp와 longjmp는 일반적인 함수 호출 메커니즘을 따르지 않고 OS 내부적인 context를 사용하므로, 메커니즘 이해가 부족하면, 디버깅 및 유지보수가 어렵다.

■ Bad 코드의 예-C

```
// setjump, longjump 사용.
jmp_buf jb;
if ( setjump(jb) == 0 ) {
      ...
      func1();
}
void func1(void) {
      ...
      longjump(jb,1);
}
```

■ Good 코드의 예-C

```
// setjump, longjump 사용을 제거.
// jmp_buf jb;
boolean flag=true;
while ( try(flag) ) {
      func1();
}
boolean try(boolean flag) {
       return flag;
}
void catch(char* errMsg) {
      error(errMsg);
}
void func1(void) {
      if ( conditon ) {
          ..
          flag=false;
      }.
      else {
          catch("오류");
      }
}
```

9. 버퍼 오버플로우에 관한 규칙

- **gets(), strcpy(), strcat() 함수는 사용하지 않는다.**
  문자열을 읽거나 조작 시, 위 함수는 버퍼 오버플로우를 유발하여, 프로그램 중지, 프로그램의 비정상적인 종료, 유해한 코드를 실행하는 원인이 된다.
  ※ fgets(), strlcpy(), strlcat()과 같은 안전한 API를 사용한다.

■ Bad 코드의 예-C

```
// gets() 사용.
#defne BUFSIZE 256
char buf[BUFSIZE];
gets(buf)
```

■ Good 코드의 예-C

```
// fgets() 사용.
#defne BUFSIZE 256
char buf[BUFSIZE];
fgets(buf,BUFSIZE,stdin);
```

- **for문의 루프 인자는 타입에 unsigned short int를 사용하지 않는다.**
  short int는 주의하여 사용하지 않으면 버퍼 오버플로우를 유발할 수 있다. short int 대신 32 bit의 long int나 64 bit의 long long 타입을 사용한다.

■ Bad 코드의 예-C

```
// unsigned short int 사용.
unsigned short int i=0;
...
for ( int i = 0; i < MAXSIZE; i++ ) {
    ...
}
```

■ Good 코드의 예-C

```
// unsigned long long 사용.
unsigned long long i=0L;
...
for ( i = 0L; i < MAXSIZE; i++ ) {
    ...
}
```

제3장 용어정리 및 약어표

제1절 용어정리

- **가비지 콜렉터(Garbage Collector)** : 어떤 프로그램이 수행되는 도중에 해당 프로그램에서 생성된 객체(인스턴스)를 프로그램이 더 이상 사용하지 않을 때, 해당 객체(인스턴스)를 메모리에서 해체하는 역할을 한다.
- **데드락(Deadlock)** : 무한 교착상태로서 A쓰레드가 B쓰레드에 의해, B쓰레드는 A쓰레드에 의해 조건이 만족되어야 작동 하는 경우 무한 대기상태에 빠지게 되는데 이런 경우를 데드락이라 한다.
- **동기화 메서드(Synchronized Method)** : 여러 개의 쓰레드가 동시적으로 특정 메서드를 수행 할 때 쓰레드 간의 충돌을 방지하기 위한 메커니즘
- **디버깅(Debugging)** : 코드가 실행되면서 변수 값이나 메모리 사용 등의 문제의 원인을 찾아내서 수정하는 것을 의미한다.
- **라이브러리(Library)** : 라이브러리는 소프트웨어를 만들 때 쓰이는 클래스나 서브루틴들의 모임을 가리키는 말이다.
- **매개변수(Parameter)** : 몇 개의 변수 사이에 함수관계를 정하기 위해서 사용되는 또 다른 하나의 변수. 파라미터 또는 보조변수(補助變數)라고도 한다.
- **멀티 환경(Multi Environment)** : 여러 개의 응용 프로그램이 쓰레드로 불리는 처리 단위를 복수 생성하여 복수의 처리를 병행하는 것. 즉, 응용 프로그램 내에서의 다중 작업(multitasking)을 처리 할 수 있는 환경을 말한다.
- **메모리 할당 함수** : 메모리를 할당하기 위한 API(malloc(), calloc() 등)
- **메모리 해제 함수** : 메모리 할당 함수를 사용하고 난후 메모리를 반환하기 위한 함수(free() 등)
- **반환 값(Return Value)** : 어떤 동작을 시켰을 때 그 결과로서 돌려받게 되는 값. 예를 들어, 소프트웨어 프로그램에서 함수를 호출하였을 때 그 결과로서 수행 결과나 '0', '−1' 등 미리 지정된 값을 받을 수도 있으며, 그 반환값을 출력하는 대신 하나의 변수에 할당 할 수도 있다.
- **비트 연산자** : 비트끼리의 연산을 의미한다.(예, &, |, ^, ~, <<, >>)
- **서블릿(Servlet)** : Java 서블릿(Java Servlet)은 Java를 사용하여 웹페이지를 동적으로 생성하는 서버 측 프로그램 혹은 그 사양을 말한다.
- **암호화 알고리즘** : 데이터의 기밀성을 유지하기 위해서 암호화 및 복호화를 위한 알고리즘

○ **역참조 연산자** : 간접연산자라고도 하며 포인터 변수를 통해 포인터 변수가 가리킨 변수의 값을 역으로 취하거나 변경한다.

○ **전역 변수(Global Variable)** : 함수 외부에서 선언되어 모든 함수에서 사용할 수 있고 프로그램이 실행 될 때 메모리가 할당되며 프로그램이 종료되어야 메모리에서 소멸된다.

○ **증감 연산자** : 증감 연산자는 변수의 값을 1 증가 또는 감소시키는 연산자이다.

○ **지역 변수** : 함수나 일정영역 내에서 선언되어 함수가 종료되면 메모리 내에서 소멸된다.

○ **캡슐화(Encapsulation)** : 클래스 내부의 데이터를 외부에 직접 노출시키지 않고 내부에 은닉시켜 보호 하는 것이다.

○ **컴파일러(Compiler)** : 컴퓨터의 프로그램 작성을 보다 간단하게 하기 위한 소프트웨어로 일상 언어에 가까운 문장으로 작성한 프로그램을 기계어로 번역하는 것이다.

○ **콤마 연산자** : 나열 연산자라고도 하며 성격이 동일한 자료형 또는 문장을 나열할 때 사용된다.

○ **포인터 변수** : 해당 변수의 주소를 저장 하는 변수로서 저장된 주소를 통하여 그 주소가 가리키는 변수에 접근하게 해주는 것이 포인터 변수의 역할이다

○ **해쉬 함수(Hash)** : 주어진 원문에서 고정된 길이의 의사난수를 생성하는 연산기법이며 생성된 값은 '해쉬 값'이라고 한다. MD5, SHA, SHA-1, SHA-256 등의 알고리즘이 있다.

○ **Pool** : 제한된 자원을 효율적으로 사용하기 위한 기술로써, 한정된 개수의 자원을 생성 하고, 사용자가 요구할 때 풀에 있는 자원을 제공함으로써 자원의 생성 및 해제를 위한 시간을 줄이고 자원을 효율적으로 관리할 수 있다.

○ **synchronized** : Java에서 임계코드를 동기화하기 위해서 제공하는 구문이다.

제2절 약어표

○ **API** : Application Programming Interface

○ **J2EE** : Java 2 Platform, Enterprise Edition

○ **JVM** : Java Virtual Machine

[부록1] 금지 API 목록

▪ JAVA API 목록

구분	Banned API	Recommended API
API 악용	System.exit()	N/A
	Socket()	EJBHome.create();
	String(word.getBytes("euc-kr"), 0xFF);	String(word.getBytes("euc-kr"),"euc-kr");
	System.out.println()	Logger.println()
	Throwable.printStackTrace()	Logger.printStackTrace()
	Runtime.exec()	Executor.safeExec()
	Session.getId()	Randomizer.getRandomString(betternottouseatall)
	ServletRequest.getUserPrincipal()	Authenticator.getCurrentUser()
	ServletRequest.isUserInRole()	AccessController.isAuthorized*()
	Session.invalidate()	Authenticator.logout()
	Math.Random.*	Randomizer.*
	File.createTempFile()	Randomizer.getRandomFilename()
	ServletResponse.setContentType()	HTTPUtilities.setContentType()
	ServletResponse.sendRedirect()	HTTPUtilities.safeSendRedirect()
	RequestDispatcher.forward()	HTTPUtilities.safeSendForward()
	ServletResponse.addHeader()	HTTPUtilities.safeSetHeader()/safeSetHeader()
	ServletResponse.addCookie()	HTTPUtilities.safeAddCookie()
	ServletRequest.isSecure()	HTTPUtilties.isSecureChannel()
	ServletContext.log()	Logger.*
	java.security and javax.crypto	Encryptor.*
	java.net.URLEncoder()/Decoder()	Encoder.encodeForURL()/decodeForURL()
	java.sql.Statement.execute()	PreparedStatement.execute()
	ServletResponse.encodeURL	HTTPUtilities.safeEncodeURL()
	ServletResponse.encodeRedirectURL()	HTTPUtilities.safeEncodeRedirectURL()
가비지 콜렉터	System.gc()	N/A
J2EE 애플리케이션	Thread.run()	Thread.start()
보안 기능	Cipher.getInstance("DES")	Cipher.getInstance("AES") Cipher.getInstance("TripleDES")
	MessageDigest.getInstance("MD5")	MessageDigest.getInstance("SHA-256")
	Cipher.getInstance("*RSA/NONE/NoPadding*");	Cipher.getInstance("RSA/ECB/OAEPWithMD5AndMGF1Padding");
불충분한 캡슐화	Class.forName();	new MyClass();
	e.printStackTrace()	new MyException()
	main()	N/A
세션	session.setMaxInactiveInterval(*-1*)	session.setMaxInactiveInterval(10*60)

▪ C API 목록

구분	Banned API	Recommended API
API 악용	ldap_simple_bind_s(ld, NULL, NULL)	ldap_simple_bind_s(ld, id, passwd)
	getlogin()	getlogin_r()
	gets()	fgets()
	strcat()	strlcat()
	strcpy()	strlcpy()
	_mbscpy()	strlcpy()
	_mbscat()	strlcat()
라이브러리	setjump, longjump	N/A
	atoi(), atol()	strtol()
	atof()	strtod()
보안 기능	Banned API	Recomended API
	EVP_EncryptInit(&ctx, EVP_des_ecb(), NULL, NULL)	EVP_EncryptInit(&ctx, EVP_aes_ecb(), NULL, NULL)
	MD5(text, MAX_TEXT_LENGTH, out)	SHA2(text, MAX_TEXT_LENGTH, out)
	RSA_public_encrypt(size, text, out, rsa_p, SA_NO_PADDING)	RSA_public_encrypt(size, text, out, rsa_p,RSA_PKCS1_OAEP_PADDING)
	RSA_generate_key(512, 35, NULL, NULL)	RSA_generate_key(1024, 35, NULL, NULL)

프로그램 표준 코딩규칙

발행처 행정안전부(www.mopas.go.kr)
발행처 2010년 12월
문의처 행정안전부 정보보호정책과
Tel. 02-2100-3628

인쇄처 OO
Tel: (02) OOO-OOOO

서울시 종로구 세종로 55 정부중앙청사
행정안전부 정보보호정책과
Tel. 02-2100-3628 / Fax. 02-2100-4221