[**자바의 개념, 기본 원리**](http://aesthetica7.tistory.com/entry/개발자가-놓치기-쉬운-자바의-개념-기본-원리)

**목차**

**[1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-1" \t "_blank) 객체지향의 구멍 static**

[1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-1.1" \t "_blank) Java는 객체지향 언어이다?

[1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-1.2" \t "_blank) 전역변수

**[2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2" \t "_blank) Java는 Pointer언어이다? (Java에는 Pointer밖에 없다?)**

[2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.1" \t "_blank) Java는 primitive형을 제외하곤 모두 Pointer이다

[2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.2" \t "_blank) null은 객체인가?

[2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.3" \t "_blank) String에 대하여

[2.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.4" \t "_blank) 객체지향의 캡슐화 파괴 주의

[2.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.5" \t "_blank) 배열에 대하여

[2.5.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.5.1" \t "_blank) 배열은 object 인가?

[2.5.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.5.2" \t "_blank) 배열의 length는 왜 field(member variable)인가?

[2.5.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.5.3" \t "_blank) final과 배열에 대하여...

[2.5.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.5.4" \t "_blank) "Java에서의 다차원 배열은 존재하지 않는다."

[2.6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.6" \t "_blank) 인수(parameter/argument)전달의 개념

[2.6.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.6.1" \t "_blank) "Java에서 parameter(argument) 전달은 무조건 'call by value' 이다"

[2.6.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.6.2" \t "_blank) "C와 같은 언어는 static linking이지만, Java는 dynamic linking이다."

[2.7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.7" \t "_blank) GC 에 대하여 잠깐!

[2.7.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.7.1" \t "_blank) "Garbage Collection은 만능이 아니다."

[2.8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.8" \t "_blank) Java Pointer 결론

[2.8.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-2.8.1" \t "_blank) "결국 Java에는 pointer가 있는 것인가, 없는 것인가?"

**[3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3" \t "_blank) 상속과 interface의 문제점**

[3.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.1" \t "_blank) 상속

[3.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.1.1" \t "_blank) 상속에 있어서의 생성자(constructor)

[3.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.1.2" \t "_blank) "down cast는 본질적으로 매우 위험하다"

[3.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.1.3" \t "_blank) "추상클래스에 final이 있으면 compile error이다"

[3.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.2" \t "_blank) interface

[3.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.2.1" \t "_blank) "interface는 interface일뿐 다중 상속의 대용품이 아니다."

[3.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-3.3" \t "_blank) 상속 제대로 사용하기

**[4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4" \t "_blank) package와 access 제어에 관한 이해**

[4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.1" \t "_blank) package

[4.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.1.1" \t "_blank) "package는 '계층구조' 인가?"

[4.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.1.2" \t "_blank) "compiler 가 인식하는 class검색 순서(소스코드내 클래스가 발견될 경우 그 클래스의 위치를 찾는 순서)"

[4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.2" \t "_blank) access 제어

[4.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.2.1" \t "_blank) "interfacde member의 access 제어"

[4.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-4.2.2" \t "_blank) 그렇다면 interface를 다른 package에 대하여 숨기고 싶으면 어떻게 하는가?

**[5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5" \t "_blank) 기타 Java 기능**

[5.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.1" \t "_blank) Thread

[5.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.1.1" \t "_blank) "Multi Thread에서는 흐름은 복수이지만 data는 공유될 수 있다."

[5.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.1.2" \t "_blank) "Thread는 객체와 직교하는 개념이다."

[5.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.1.3" \t "_blank) "Synchronized 의 이해"

[5.1.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.1.4" \t "_blank) "Thread 사용법의 정석은?"

[5.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2" \t "_blank) Exception

[5.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.1" \t "_blank) "finally 절은 반드시 어떠한 경우에도 실행되는가?"

[5.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.2" \t "_blank) "예외의 종류 3가지 (Error, RuntimeException, 그밖의 Exception)"

[5.2.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.2.1" \t "_blank) Error

[5.2.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.2.2" \t "_blank) RuntimeException

[5.2.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.2.3" \t "_blank) 그밖의 Exception

[5.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.2.3" \t "_blank) "OutOfMemoryError는 어떻게 처리해야 하는가?"

[5.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.3" \t "_blank) Object Serialize

[5.3.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.3.1" \t "_blank) "Serialize를 위해서는 marker interface인 java.io.Serializable interface를 implements해야한다."

[5.3.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.3.2" \t "_blank) "super class는 Serializable이 아닌데 sub class만 Serializable인 경우의 문제점"

[5.3.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.3.3" \t "_blank) "transient field의 복원(?)관련"

[5.3.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.3.4" \t "_blank) "Stack Overflow에 주의하라!"

[5.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.4" \t "_blank) "nested class / inner class / 중첩클래스"

[5.4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.4.1" \t "_blank) "중첩클래스의 개념"

[5.4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.4.2" \t "_blank) "내부클래스는 부모의 참조를 몰래 보유하고 있다."

[5.4.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.4.3" \t "_blank) "local inner class에 대하여"

[5.4.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-5.4.4" \t "_blank) "anonymous class(무명클래스)에 대하여"

**[6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6" \t "_blank) 이래도 Java가 간단한가**?

[6.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.1" \t "_blank) method overload 에서의 혼란?

[6.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.1.1" \t "_blank) "overload란 이름이 가고 인수가 다른 method에 compiler가 다른 이름을 붙이는 기능"

[6.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.1.2" \t "_blank) "그렇다면 overload에서 실제로 혼동되는 부분은 무엇인가?"

[6.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.1.3" \t "_blank) (참고) 또다른 혼동, overload한 method를 override 하면?

[6.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2" \t "_blank) 상속/override/은폐 에서의 복잡함

[6.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2.1" \t "_blank) "Java class의 member 4 종류"

[6.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2.2" \t "_blank) "override시 method 이름에 대한 함정"

[6.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2.3" \t "_blank) "또다른 나의(?) 실수 - 말도 안되는 오타"

[6.2.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2.4" \t "_blank) "static member를 instance를 경유하여 참조해서는 안 된다."

[6.2.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.2.5" \t "_blank) "super keyword는 부모의 this"

[6.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.3" \t "_blank) 상속에 관한 또 다른 문제

[6.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.4" \t "_blank) 그밖의 함정

[6.4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.4.1" \t "_blank) "생성자에 void 를 붙인다면?"

[6.4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-6.4.2" \t "_blank) "if / switch 의 함정"

**[7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-7" \t "_blank) Java 기능 적용 몇가지**

[7.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-7.1" \t "_blank) 대규모 개발에서 interface 분리하기

[7.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-7.1.1" \t "_blank) "interface 분리의 필요성"

[7.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-7.2" \t "_blank) Java에서의 열거형

[7.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-7.3" \t "_blank) Debug write

**[8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8" \t "_blank) Java 5.0 Tiger 에 대하여**

[8.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.1" \t "_blank) Working with java.util.Arrays

[8.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.2" \t "_blank) Using java.util.Queue interface

[8.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.3" \t "_blank) java.lang.StringBuilder 사용하기

[8.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.4" \t "_blank) Using Type-Safe Lists

[8.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.5" \t "_blank) Writing Generic Types

[8.6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.6" \t "_blank) 새로운 static final enum

[8.7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.7" \t "_blank) Using java.util.EnumMap

[8.8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.8" \t "_blank) Using java.util.EnumSet

[8.9](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.9" \t "_blank) Convert Primitives to Wrapper Types

[8.10](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.10" \t "_blank) Method Overload resolution in AutoBoxing

[8.11](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.11" \t "_blank) 가변적인 argument 개수 **...**

[8.12](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.12" \t "_blank) The Three Standard Annotation

[8.13](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-8.13" \t "_blank) Creating Custom Annotation Types

**[9](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-9" \t "_blank) The for/in Statement**

[9.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-9.1" \t "_blank) for/in 의 자주 사용되는 형태

**[10](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-10" \t "_blank) Static Import**

[10.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-10.1" \t "_blank) static member/method import

**[11](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "s-11" \t "_blank) References**

===============================================================

**[1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 객체지향의 구멍 static**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-1)

**[1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java는 객체지향 언어이다?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-1.1)

"Java는 완전한 객체지향 언어이다" 라는 주장을 자주 접하게 된다. 만일 이것이 사실이라면 Java를 사용하는 한 "기존의 절차지향 프로그래밍을 전혀 할수 없을것 같지만 그렇지 않다. 빠져나갈 구멍이 있는 것이다. static을 이용하면 비 객체지향 언어처럼 코딩할 수 있다.

static method는 instance가 아닌 클래스에 속하는 method로, class method라고 부른다. 반대로 static이 아닌 method는 instance method라고 부른다.

static method는 this가 없다. instance method에는 숨겨진 파라미터로 this가 건네진다. (아래 "객체지향에 흔희 있는 오해" 참고) 하지만 static method는 절차지향의 함수와 동일하므로 숨겨진 파라미터 this는 없다. 그래서 static method에서는 전달한 this가 없으므로 instance method를 호출하거나 instance field를 참조할 수 없는 것이다.

(참고) 객체지향에 흔히 있는 오해

* 오해1. "객체지향에서는 객체끼리 서로 메세지를 주고 받으며 동작한다." 라는 말을 듣고 다음과 같이 생각할 수 있다. "객체지향에서는 객체가 각각 독립하여 움직인다는 것인가, 그러면 각 객체에 독립된 thread가 할당되어 있단 말인가?" 그렇지 않다. "메세지를 보낸다"라는 것은 단순히 각 객체의 함수 호출에 불과하다.
* 오해2. "객체지향에서는 method가 class에 부속되어 있다"는 말을 듣고 다음과 같이 생각할 수 있다. "그러면 instance별로 method의 실행코드가 복제되고 있는 것이 아닌가?" 물론 이것도 오해다. method의 실행코드는 종래의 함수와 동일한 어딘가 다른곳(JVM의 class area)에 존재하며 그 첫번째 파라미터로 객체의 포인터 this가 건네질 뿐이다.
* 오해3. "그렇다면 각 instance가 method의 실행코드를 통째로 갖고 있지 않는 것은 확실하지만, method의 실행 코드의 포인터는 각 instance별로 보관하고 있는것이 아닌가?" 이것은 약가 애매한 오해이긴 하다. JVM 스펙에서는 class영역에 실행코드를 갖고 있으며, method 호출시 별도의 stack frame이 생성되어 실행되고 실행 완료시 복귀 주소를 전달한다.

**[1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 전역변수**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-1.2)

static에서 public field는 전역변수(global variable, 글로벌 변수)이다. 여기서 "글로벌 변수는 왜 안 되는가"에 대해 잠깐 생각해 본다. 우리는 흔히 "글로벌 변수는 될수있는한 사용하지 않는 것이 좋다"라고 한다. 그 이유는 글로벌 변수는 어디서든 참조할 수 있고 값을 변경할 수 있기 때문이다.

또한 파라미터나 리턴값으로 교환해야 할 정보를 글로별 변수를 경유(사용)하여 건네주면 함수의 역할이 불분명 해지고 흐름도 애매해 진다. 마지막 이유로는 "글로벌 변수는 하나밖에 없다"는 것이다. 이는 어디서 이값을 변경했는지 알 수 없게 하는 지름길이고 실무에서도 간혹 발생하긴 하지만, 이 하나밖에 없는 변수가 버전업으로 두개가 필요하게 되었을때 확장도 대형 프로젝트에서는 힘들어진다.

따라서 static에서 public은 final을 붙여 상수로 사용해야지 그 외의 용도는 자제하는 것이 좋을 것이다.

* (참고) final 초기화에서의 주의점. 예를 들어 다음과 같은 코드를 보았을때 우려되는 점은 무엇인가?

public final static Color WHITE = new Color(255, 255, 255);

위의 코드는 java.awt.Color에서 발췌한 것인데, final 변수는 한번 초기화 되면 변경이 불가능한데 object로 초기화 할 경우 WHITE라는 필드가 변경될 수 없는 것이지 그것이 가리키는 객체는 아니라는 점이다.

과거 신규 서비스 개발시 final 변수 필드에 설정파일을 읽어 cache하는 singleton class의 특정 member 를 이용하여 초기화 할 경우 이 멤버값이 변경되면 final 변수의 값이 변경되었는데 프로그램에서는 이상한 짓을 하는 원인을 찾기가 상당히 어려웠던 경험을 하고 난 후 부터 이런 코드는 냄새나는 코드로 여겨지게 되었다.

static은 글로벌변수와 동일하므로 남발해서는 안된다. static을 사용할 경우 다음 두가지는 최소한 기억한다.

1. static field는 final의 경우와 달리 정말 "하나여도 되는지" 여부를 잘 생각해야 한다.
2. static method는 주저하지 말고 쓰되 다음 두가지의 경우 매우 활용적이다.
   1. 다른 많은 클래스에서 사용하는 Utility Method 군을 만드는 경우. (주로 Utility Class의 method)
   2. 클래스 안에서만 사용하는 "하청 메소드(private method)". 이유를 예를 들어 설명하면, 아래와 같은 조금은 과장된 클래스가 있다고 하자.

public class T ..

private int a;

private int b;

private int c;

private int calc(){

c = a + b;

return c \* c;

}

....other method or getter/setter...

위의 클래스 T의 경우 내부에서 calc라는 instance 함수를 사용하게 되면 c 의 값이 매번 변하게 된다. 이는 무심코 하는 실수로 클래스내에서 private method는 모든 멤버 instance 변수에 접근 가능하게 되면서 발생한다. c의 값이 변하지 않기를 바랄수 있다. 이때 안전한 방법은 다음과 같이 calc 하청 메소드를 static method로 수정하면 안전하다.

private static int calc(int a, int b){

int c = a + b;

return c \* c;

}

여기서 a와 b는 멤버 변수를 접근할수 없어 전달해야한다.(static에는 this가 없어 instance field를 참조할 수 없다는 것은 이미 위에서 설명했다.) 또한 c도 같은 이유로 사용할 수 없어 로컬 변수로 선언하고 사용하고 있다. 이럴 경우 메소드가 약간 커질수 있지만 instance member 변수를 안전하게 사용할 수 있다는 장점이 있다. 이것은 static을 다시한번 생각하게 하는 좋은 예가 되었을 것이다.

**[2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java는 Pointer언어이다? (Java에는 Pointer밖에 없다?)**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2)

**[2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java는 primitive형을 제외하곤 모두 Pointer이다**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.1)

"Java에는 포인터가 없다" 라고 Java의 장점 이라고 생각하는 것은 입문자도 외우고 있다. 하지만 이 부분은 의외로 Java를 혼란스럽게 하는 주범이라고 생각한다. Java에 포인터가 없기는 커녕 primitive(int, short, char, long...등 JVM의 Heap에 object로 생성되지 않는것들)를 제외하면 "포인터 밖에 없는 언어이다"라는 명제가 성립되게 된다. 사실 여기서 포인터라고 함은 C 의 그것과는 조금 다른 reference(참조)이긴 하지만...

"즉, 자바의 클래스형의 변수는 모두 포인터이다."

**[2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) null은 객체인가?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.2)

Java에서 공참조(힙에 실제로 참조되는 object가 없는 참조)의 경우는 당연히 객체가 붙어 있지 않다. 그러나, Java API 레퍼런스의 NullPointerException 항에는 다음과 같이 기술되어 있다.

"object가 필요한 경우 application이 null을 사용하려고 하면 throw된다. 가령 다음과 같은 경우이다."

* null object의 instance method 호출
* null object의 field(member variables)에 대한 액세스 또는 그 값의 변경
* null의 길이를 배열처럼 취득할 경우
* null의 slot을 배열처럼 액세스 또는 수정
* null을 Throwable처럼 throw 할 경우

위에서 null object라는 말이 등장하는데 이는 공참조에 객체가 붙어 있지 않은 것이 아니라 null을 가리키는 객체라고 볼수 있다. 즉, 공참조라는 것은 JVM에서 봤을때 아무것도 참조하지 않는것이 아니라 null이라고 하는 object를 참조하고 있는것이다. 그러나 JSL 4.3.1에서는 다음과 같이 나와있다.

"참조값(reference)은 이러한 객체의 포인터나 어떤 객체도 참조하지 않는 특수한 null참조가 된다"

즉, 공참조는 어떤 객체도 참조하지 않는다고 단정하고 있다. 하지만 '==' 연산에 있어 두개의 객체가 모두 null이거나 동일한 객체 또는 배열 참조의 경우 true라고 되어있는것으로 봐서 서로 다른 두 객체가 동일한 null을 참조하고 있으므로 true가 된것이 아닌가 하는 생각을 할 수 있다.

즉, null이 Object의 instance 형태는 아니지만 개념적으로 봤을때 null도 object라고 봐야 하지 않을까?

**[2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) String에 대하여**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.3)

String Object에 대한 생각.

String str = "111222";

String a = "111";

String b = "222";

String c = "111";

String d = b;

String t = str.substring(0,3); //111

위의 소스를 보고 다음이 참인지 생각해 보자. (==연산자는 포인터의 비교이지 값의 비교가 아님)

1. str == (a + b) ==> 이것은 두개의 참조와 하나의 참조를 비교했으므로 당연히 false이다.
2. a == b ==> 이것은 당연히 false
3. d == b ==> 이것은 동일한 reference이므로 true
4. a == t ==> a 와 t 는 둘다 값이 "111"이다. 하지만 이것은 서로 다른 참조를 가져 false이다. 그렇다면 다음 5번도 false일까?
5. a == c ==> 이것은 true이다. 아.. 4번과 혼란스럽다. 이것이 참인 이유는? ==> 이것의 해답은 JSR 3.10.5에 다음과 같이 나와 있기 때문이다.

"동일한 내용의 문자열 리터럴에 관해서는 인스턴스를 하나밖에 생성하지 않는다."

즉, 위의 a와 c 는 '=' 를 이용하여 문자열 리터럴을 생성하게 되는데 a 에서 이미 만들어 놓았으므로 c에서는 그것을 참조한다.

**[2.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 객체지향의 캡슐화 파괴 주의**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.4)

"object pointer를 반환하는 getter method는 객체지향의 캡슐화가 파괴될 가능성이 있다." 이는 object형의 field(member variable)의 getter에서 이 object를 그냥 반환하면 이 object를 받은쪽이나 참조하고 있는 다른쪽에서 이 object의 내용을 변경하게 되므로 사실 캡슐화(은닉)는 이루어 지지 않았다고 봐야한다.

"이럴 경우 object를 clone(복제) 하여 반환하지 않아도 되는지를 반드시 생각해 본다."

object의 복사에는 shallow copy와 deep copy가 있다.

//(참고)Member에는 두개의 field(Identity Class 형의 ID와 Family Class 형의 family)가 있다.

/\*\* shallow copy \*/

Member shallowCopy(){

Member newer = new Member();

newer.id = this.id;

newer.family = this.family;

return newer;

}

/\*\* deep copy \*/

Member deepCopy(){

Member newer = new Member();

newer.id = new Idetity(this.id.getId(), this.id.getName());

newer.family = new Family(this.family.getFamilyName(), this.family.getFamilyInfo());

return newer;

}

위 소스에서 보듯이 shallowCopy 는 object를 복사하여 반환한것 처럼 보이지만, 사실은 Member object만 새로 생성되었을뿐 Member의 field는 newer와 this 둘다 서로같은 힙의 id와 family를 참조한다. 하지만 두번째 method인 deepCopy의 경우 Member의 field를 새로 생성하여 복사하므로 서로 다른 id와 family이다.

"Java에서는 clone이라는 method가 준비되어 사용되는데 이는 기본이 shallow copy임을 명심해야 한다. deep copy를 사용하기 위해서는 clone method를 overload하거나 따로 만들어 직접 기술해야 한다."

(참고) object를 immutable(변하지 않는, 불변의 객체)로 만드는 요령

1. 모든 field(member variable)를 생성자(constructor)를 이용하여 초기화 한다.
2. 모든 field는 private으로 선언하고, getter method는 만들되 setter는 기술하지 않는다.

즉, 값을 변경하기 위해서는 object를 다시 만들어야만 하는 불편은 있지만 안전하게 사용하려 할때 유용하다.

**[2.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 배열에 대하여**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.5)

**[2.5.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 배열은 object 인가?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.5.1)

JVM에서 배열은 object로 취급되어 object와 같이 aload, astore와 같이 bytecode로 기술되어 진다. int[] iarr = new int10; 에서 보는것과 같이 new로 Heap 영역에 object를 생성하므로 object임을 알 수 있다.

**[2.5.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 배열의 length는 왜 field(member variable)인가?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.5.2)

String의 길이를 구할때는 length()와 같이 method를 이용하는데 배열은 object임에도 불구하고 legth와 같이 필드로 되어있다. '이래도 Java가 완전한 객체지향 언어인가' 라는 의심이 들게 한다. 그렇다면 length가 public이므로 array.length = 100; 과 같이 하면 배열 크기가 변경되나?

이것은 컴파일 오류가 난다. length는 final이라 값을 변경 할 수 없다는 것이다. 그렇다면 final field로 한 이유는 무엇이냐는 Java News Group에 찾아보면 대부분이 "효율을 위해서"라고 되어 있다. JIT compiler를 사용하지 않는한은 method보다는 field가 빠른건 당연한 것이다.

그런데 정말 알수 없는것은 byte code에서는 arraylength라는 전용명령으로 컴파일 된다. 즉, length는 Java의 문법이 어찌되었든 JVM레벨에서는 field가 아닌것이 분명하다. 그렇다면 효율을 위해서 field로 했다는 것은 도데체 무슨 소리인가?

전문가들의 대답에는 이것은 Java의 수수께끼 중 하나라고 대답하는 사람이 많다고 한다.^^;

**[2.5.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) final과 배열에 대하여...**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.5.3)

우리가 흔희 앞에서도 나온바 있지만 final은 값을 변경할 수 없는 것이라고만 생각하지 object로 되어 있을 경우 그 object는 변경 가능하다는 것을 잊곤한다. 배열도 object이므로 마찬가지다.

final int[] iarr = new int[5]; 일경우 iarr = null; 은 에러가 나지만 iarr[3] = 5; 는 에러가 나지 않는다. 즉, final이 지정되어 있는것은 iarr이지 iarr이 가리키는 곳 배열의 요소가 아닌 것이다.

**[2.5.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Java에서의 다차원 배열은 존재하지 않는다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.5.4)

가령 2차원 배열 처럼 보이는 int[][] iarr 또는 int[] iarr[] 은 일차원 배열 두개이지 2차원 행열 구조가 아닌것이다. 즉, 두개의 배열은 각각이 배열로 되어 있는 것이지 테이블(행열)형태가 아니다.

**[2.6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 인수(parameter/argument)전달의 개념**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.6)

**[2.6.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Java에서 parameter(argument) 전달은 무조건 'call by value' 이다"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.6.1)

primitive type의 경우 호출한 쪽의 변수값은 호출 받은 method내에서 값이 변경되어도 변경되지 않는다. reference type의 경우도 reference되는 object에 대해서는 함께 변경되지만 reference pointer는 call by value이다. object를 가리키는 pointer는 call by value로 변경되지만 Heap의 실제 object내용은 변경되지 않는다.

**[2.6.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "C와 같은 언어는 static linking이지만, Java는 dynamic linking이다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.6.2)

따라서 Java는 Class 파일이 처음에 한꺼번에 memory에 읽혀지는 것이 아니라 런타임시에 그것이 필요해 졌을때 읽혀지고 링킹된다. static field의 영역도 Class가 읽혀지는 시점에 비로서 확보된다. 이렇게 되면 최초 가동시간이 단축되고 끝까지 사용하지 않는 Class의 경우 신경쓸 필요가 없어지게 된다.

따라서 static field는 프로그램이 시작되어 해당 Class가 필요해 졌을때 JVM이 알아서 load/link 해 준다. 즉, static field는 프로그램이 실행되기 시작할 때부터 끝날때까지 계속해서 존재하는 것이라고 보면 된다.

(참고) 링킹(linking)의 의미

link된다는 것은 Class가 memory에 loading될 때 특정 메모리 번지에 loading되는데 이 메모리 번지는 loading될때 마다 다른 번지수에 loading된다. 이때의 메모리 주소값(Java에서는 실제 메모리 값이 아닐 수 있다)을 현재 실행중인 프로그램에서 알 수 있도록 하여 해당 Class에 대한 참조가 가능하도록 연결하는 과정이다.

정적(static) link라는 것은 이러한 메모리에 대한 주소 정보를 컴파일 시에 compiler가 미리 결정하는 것이고, 동적(dynamic) link라는 것은 프로그램 수행 중 결정되는 것을 의미한다. 정적인 link의 경우 직접적으로 메모리의 번지값이 할당 되는 것이 아니라 offset값(기준위치로 부터의 index값)으로 연결시킨다.

**[2.7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) GC 에 대하여 잠깐!**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.7)

**[2.7.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Garbage Collection은 만능이 아니다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.7.1)

Java에는 free가 없다. GC가 알아서 해준다. 하지만 GC 수행중에는 프로그램이 멈추는 것과 동일한 증상이 나타나기 때문에 GC가 자주 발생하지 않도록 프로그램 해야 한다. 서비스 되고 있는 시스템에서도 가끔 시스템이 응답이 늦어지는 시점이 있는데, 이는 GC가 수행되고 있는 중이 대부분이다.

그렇다면 GC가 자주 발생하지 않도록 해야 하는데 가장좋은 방법은 무엇일까? 그것은 바로 불필요한 객체를 생성하지 않는 것이 아닐까?

개인적으로 Java에 free가 없는것이 너무나 든든하게 느껴진다. 이유는 두개의 변수가 Heap내의 하나의 object를 reference하고 있을 경우 실수로 하나의 변수만 free해 버리면 나머지 하나는 dangling pointer라하여 reference pointer가 모르는 사이데 사라져 버려 곤경에 처하는 것을 예방해 주기 때문이다.

참고로 Object class에는 finalizer라는 method가 있어 GC 수행시점에 호출되는 method가 있지만 이것은 GC가 언제 수행될지 알 수 없으므로 과신하지 말아야 할 것이다.

**[2.8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java Pointer 결론**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.8)

**[2.8.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "결국 Java에는 pointer가 있는 것인가, 없는 것인가?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-2.8.1)

Java는 Heap내의 Object를 참조(reference)하고 있고, 참조는 결국 개념이 포인터와 같은 것이므로, "Java에는 pointer가 없다"는 것은 어불성설이다.  
// 이부분에 대해 Object를 이해하시면 족히 이런 문제는 사라질것으로 봅니다.  
// 클래스에 대한 인스턴스(object)들은 reference로 밖에 가질(참조될)수 없기 때문입니다.  
// 컴파일러 입장이 아닌 언어 자체의 사상을 가지고 쉽게 이해시키는 것이 좋을것 같습니다.

JSR 4.3.1을 보면 다음과 같은 말이 나온다.

"참조값(reference)은 객체의 pointer이거나, 또는 어떠한 객체도 참조하지 않는 특수한 null 참조가 된다"

또한 java.sun.com의 Java programmer's FAQ에 "Java는 pointer가 없다고 하는데, linked list는 어떻게 만들어야 하나?"라는 질문에 다음과 같이 답변이 나와있다.

(답변) Java에 관한 많은 오해중에서 이것이 가장 심각한 것이다. 포인터가 없기는 커녕 Java에 있어 객체지향 프로그래밍은 오로지 pointer에 의해 행해진다. 다시 말새 객체는 항상 포인터를 경유해서만 access되며 결코 직접적으로 access되지 않는다. pointer는 reference(참조)라고 불리며 당신을 위해 자동으로 참조된다.

"Java에는 pointer가 없고 주장하는 모든 서적과 글들은 Java의 reference사양에 모순된다고 봐야 할 것이다."

**[3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속과 interface의 문제점**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3)

**[3.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.1)

**[3.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속에 있어서의 생성자(constructor)**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.1.1)

"child의 default 생성자가 있으면 그 생성자에는 parent의 생성자(super()) 호출이 compile시 자동 삽입된다." 따라서 super() 이전에 다른 코드가 있으면 object가 생성되기 이전이므로 오류를 발생하게 된다.

**[3.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "down cast는 본질적으로 매우 위험하다"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.1.2)

down cast - child의 type으로 parent를 casting - 는 parent 형태의 type이 정말 child type인지 compile시에는 알 수 없다. 실행시에 type check가 이루어 지므로 runtime시에 ClassCastException이 발생할 가능성이 커진다.

"프로그래밍시 오류는 가능한한 compile시에 처리하는것이 좋다."

**[3.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "추상클래스에 final이 있으면 compile error이다"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.1.3)

abstract method가 있는 클래스는 추상 클래스이고 추상클래스는 상속되지 않으면 아무런 의미가 없기 때문이다.

**[3.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) interface**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.2)

**[3.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "interface는 interface일뿐 다중 상속의 대용품이 아니다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.2.1)

interface를 method signature - 추상클래스와 같이 구현부는 없이 선언부만 있는 method - 의 용도로 생각하는것이 Java에서는 옳다. 즉, interface는 final field와 abstract method가 있는 클래스와 동일하긴 하지만 상속의 의미와는 그 용도가 다르다. 공통된 type을 정의하는것으로 보는것이 맞는 의미일 것이다.

또한 interface는 클래스를 재이용하기 위해 상속을 사용하여 캡슐화의 파괴를 수반하는 것을 방지하는 기능이있다. 상속을 사용하면 모두 구현후 마치 소스 코드가 여기저기 천 조각을 주워 모아 만든 '누더기'같이 보이는 것에 한숨을 쉰 경험이 있을 것이다. 이 부분을 interface로 구현하면 보다 깔끔한 코드가 나오게 된다. 물론 public과 protected를 적절히 잘 사용해도 되긴 하지만 말이다.

하지만 상속은 메소드 오버라이드한 경우 클래스를 마음대로 개조해 버린 셈이 되므로 어디선가 묘한 모순이 발생하게 될 가능성도 높아질뿐 아니라 추상클래스의 경우 실제 구현부가 어디에 위치하는지도 에매하게 느껴질 수 있어 불안한 코드가 되고 만다.

**[3.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속 제대로 사용하기**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-3.3)

"그렇다면 제대로 된 상속은 어떻게 판단할 수 있을까?"

상속은 'is a'관계가 성립해야 올바르다. 즉 '서브클래스(자식) is a 슈퍼클래스(부모)'가 성립해야 한다. 예를 들면 Red is a Color는 올바른 명제이지만 Engine is a Car는 'has a'관계이므로 상속이라고 볼 수 없다.

"따라서 'has a'관계는 상속이 아니므로 composition과 delegation을 이용하면 된다."

composition은 '객체를 field가 갖게 하는 방법'을 의하므로 'has a'관계가 정확히 성립한다.

"상속 대신 composition과 delegation(조작이나 처리를 다른 객체에 위임)을 사용하면 다음과 같은 이점이 있다."

1. 상속에서는 슈퍼클래스가 허용하고 있는 조작을 서브클래스에서 모두 허용해야 하지만, composition과 delegation에서는 조작을 제한할 수 있다.
2. 클래스는 결코 변경할 수 없지만, composition하고 있는 객체는 자유롭게 변경할 수 있다. 예를 들면 학생 클래스가 영원이 학생이 아니라 나중에 취직을 하여 직장인 클래스가 될수 있다.

상속을 composition과 delegation으로 변경하는 요령은? 여기서 Shape를 상속한 Polyline과 Circle을 변경한다면 다음과 같다.

1. Shape(부모)의 공통된 내용을 구현한 구현 클래스(ShapeImpl)를 만든다.
2. Polyline과 Circle 클래스에서 ShapeImpl을 composition하고 부모와 공통되지 않는 method를 각각 위임 받는다.
3. ShapeImpl 클래스의 method를 추출한 ShapeIF interface를 작성하고 Polyline과 Circle에서는 implements 한다.

**[4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) package와 access 제어에 관한 이해**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4)

**[4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) package**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.1)

**[4.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "package는 '계층구조' 인가?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.1.1)

처음 Java를 접하면서 package에 대해 이해할때 마치 파일시스템과 같은 계층구조라고 이해하게 되어 ' import /test/\*.class '는 왜 안되는지 의아해 했던 기억이 있다. 그리고 부모 directory에 있는 클래스에서 왜 자식 directory에 있는 Class를 import없이 사용할 수 없는지도 이상했다.

즉, package에서 동일 부모라도 서로 다른 package는 완전히 별개의 package였던 것이다. 이 부분에 관해서는 JLS 7.1 에서 다음과 같이 기술되어 있다고 한다.

"package가 계층적인 이름 구조로 되어 있는 것은 관련된 package를 일정 규약에 따라 체계화하기 위해서이고, package 안에서 선언되어 있는 top-level형과 동일한 이름을 가진 서브 package를 갖는 것이 금지되어 있는 점을 제외하면 특별한 의미는 없다."

즉, Java에서는 package이름을 계층적으로 명명할 수 있을뿐 package구조 자체에는 어떤 계층적인 의미 부여도 할 수 없는 것이다. 다시 말해서 Java에서는 package이릉을 계층적으로 명명할 수 있을 뿐 구조자체는 평평한 것이다. 실제로 바이트 코드의 내용을 보면 깨어진 내용중에 java/lang/String과 같이 완전한 한정이름을 class이름으로 사용됨을 알 수 있다.

**[4.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "compiler 가 인식하는 class검색 순서(소스코드내 클래스가 발견될 경우 그 클래스의 위치를 찾는 순서)"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.1.2)

1. 그 class자신
2. 단일형식으로 임포트된 class
3. 동일한 패키지에 존재하는 다른 class
4. 온디멘드 형식(..\* 형태) 임포트 선언된 class

**[4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) access 제어**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.2)

public은 다른 package에서 참조 가능하고, 무지정할 경우 동일한 package내에서만 참조 가능하다.

**[4.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "interfacde member의 access 제어"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.2.1)

interface의 member field/method는 모두 public이다. interface member에 protected나 private을 지정할 수는 없다. 또한 public을 지정할 필요도 없다. 지정해도 상관없지만 JLS 9.4에서는 다음과 같이 명시되어 있다.

"interface의 method에 대하여 public 수식자를 지정하는 것이 허용되고 있지만, style로서는 전혀 권장할 수 없다."

즉, interface member는 모두 public이라 되어 있는 것이다. 또한 James Gosling도 집필에 참가한 '프로그래밍 언어 Java 3판'에서는 다음과 같이 기술되어 있다고 한다.

"public이 아닌 member를 interface에게 갖게 하는 것은 아무런 의미가 없다. interface의 member에 대한 access제어에 interface 자신의 access 제한을 적용하는 것이므로 이것은 아무런 의미가 없다."

**[4.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 그렇다면 interface를 다른 package에 대하여 숨기고 싶으면 어떻게 하는가?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-4.2.2)

그것은 interface 자체 선언에 public이 아닌 것을 적용하면 되는 것이다. member별로 제어할 수 없어 불편한 면도 있지만, 나름대로 그럴 듯한 규칙이다. 하지만 이것은 정말 이상한 논리가 아닐수 없다. public이 아닌 interface에 public method가 무슨 의미가 있는지 알 수 없기 때문이다. 이 interface를 구현한 클래스에서도 method는 모두 public이 되어야 하는데, 이것도 아무래도 이상하다.

**[5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 기타 Java 기능**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5)

**[5.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Thread**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.1)

**[5.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Multi Thread에서는 흐름은 복수이지만 data는 공유될 수 있다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.1.1)

Multi processing에서는 흐름은 복수이지만 data는 독립되어 있다. 하지만 Multi Thread에서는 Heap과 static영역에 관한 한 2개 이상의 Thread 사이에 공유가 이루어 진다. 따라서 2개 이상의 Thread에서는 동일한 static field를 참조할 수 있고, 동일한 객체에 접근할 수도 있다. 그러나 stack만은 Thread별로 독립되어 있다. stack은 method에 들어가는 시점에 확보되고 빠져 나오는 시점에 확보되고 빠져 나오는 시점에 Free 되므로 2개 이상의 Thread에서 공유할 수는 없는 것이다.

**[5.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Thread는 객체와 직교하는 개념이다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.1.2)

Multi Thread는 어디까지나 Thread라는 처리 흐름이 여러 개 존재할 수 있다는 의미이다. 요약하면 다음 3가지 이다.

1. Multi Thread에서는 Thread라는 처리 흐름이 2개 이상 존재할 수 있다.
2. 어떤 Thread에서 움직이기 시작한 method가 다른 method를 호출 했을때 호출된 측의 method는 호출한 측의 method와 동일한 Thread에서 동작한다.
3. Thread의 경계와 객체의 경계는 전혀 관계가 없다. 즉, Thread와 객체는 직교하고 있다.

**[5.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Synchronized 의 이해"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.1.3)

Multi Thread 기반의 programming시에 synchronized를 놓쳐 자주는 일어나지 않으나 뭔가 잘못되어 가는것을 경험한 적이 있다. 즉, 이것이 원인이 되어 버그가 발생한 경우 그 버그는 재현성이 지극히 낮아지기 때문에 꽤 고생을 하게 된다. 이런 사태가 발생하게 되면 버그의 원인을 찾기가 어렵게 되고 해당 application은 언제 발생할지도 모르는 오류가 있는 상태 그대로 운영되기 때문에 심각성이 내포되어 있다고 할 수 있다.

이러한 사태를 방지하기 위해서는 critical section을 2개 이상의 Thread가 동시에 실행되지 않도록 '배타 제어'를 해야한다. 그 키워드가 바로 synchronized이다.

synchronized에는 synchronized(obj){} 형태와 method에 synchronized 를 붙이는 두가지 방법이 있는데, 이 둘은 범위만 같다면 같은 의미이다. 예를 들어 설명하면, 아래의 소스에서 method1()과 method2()는 동일하다.

synchronized void method1(){

...

}

void method2(){

synchronized(this){

...

}

}

이렇게 동일한 의미를 두가지로 만든것은 method단위로 synchronized를 걸 일이 그만큼 많다는 것을 의미한다. 많이들 오해하고 있는 부분이 위의 소스에서 알수 있듯이 method에 synchronized를 사용한다는 것은 '그 객체에 해한 조작은 동시에 하나의 Thread라는 것이지 method 호출이 하나의 Thread가 아닌것이다'

그렇다면, Thread A가 obj의 Lock을 설정하고 있는 상태에서 다시 한번 Thread A 자신이 obj의 Lock을 설정하면 어떻게 될까? 이 경우 Thread A는 이미 이 obj에 대하여 Lock을 보유하고 있으므로 기다리지는 않아도 된다. 위의 소스에서 method1에서 method2를 호출한다면?

method1에서 이미 obj의 Lock을 보유 했으므로 method2의 synchronized(this) 부분에서는 Lock을 기다리지 않아도 된다.

즉, Lock의 기준이 특정Thread에 있어서 Lock의 기준이 method가 아닌 object인 것이다. 이 규칙 덕분에 synchronized method도 재귀호출이 가능해지고, synchronized method가 동일한 instance의 synchronized method를 호출할 수 있는 것이다.

주의할 점은 static method에 synchronized가 있다면 static은 this참조가 없다고 위에서 설명하였으므로, 이 클래스의 Class 객체를 Lock하게 된다. 기준이 xx.Class가 되는 것이다.

**[5.1.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Thread 사용법의 정석은?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.1.4)

Thread 사용법에는 다음 두가지의 정석이 있다.

1. Runnable을 implements하고 Thread의 참조를 보유(composition) 하는 방법. 이경우는 단지 Runnable만 implement함으로서 해결되는 경우가 대부분이긴 하지만, 그 class 내에서 해당 class의 Thread를 조작하게 된다면 composition한 Thread 객체에 delegation하면 된기 때문이다.
2. Thread class를 상속하는 방법. JDK의 소스를 보면 Thread class에는 Runnable을 implements 하고 있다. 그리고 run method는 native method이다. 따라서 Thread를 상속한 모든 클래스는 사실 Runnable을 implements하고 있는 것이다. run method는 abstract가 아니므로 구현되어 있고 우리는 이를 오버라이드하여 사용하고 있다. 이 방식을 사용하면 Thread의 method를 안팍으로 자유롭게 호출할 수 이지만, 이미 다른 class를 상속하고 있다면 이 방식을 사용할 수는 없다.

JDK API Reference의 Runnable에 과한 설명중에 다음과 같은 내용이 있다.

"Thread class의 method중 run method만을 오버라이드하여 사용하는 경우는 Runnable interface만 implements하여 사용하면 된다. 왜냐하면, class의 기본적인 동작을 수정 또는 확장하지 않는한 그 class를 sub class화 하는 것은 바람직하지 않기 때문이다."

그렇다면 위에서 언제나 1)번 방식을 사용하면 되는 것 아닌가 라는 의문이 생기게 된다. 왜 귀찮게 2)의 방법을 고민하는 것인가, 극단적이긴 하지만 만일에 사태에 이 클래스가 다른 클래스를 상속받게 되는 경우도 있을수 있는데.

하지만 이것은 아닐것이다. 만약 이렇다면 Thread class가 Runnable을 implements할 필요가 없었을 것이기 때문이다. 또한 Thread는 생성자의 인수로 Runnable의 reference를 취득한 후 계속해서 그것을 보유한다는 것도 이상하다. Thread에 있어 Runnable이 필요한 것은 start() 때 뿐이므로 start()의 인수로 Runnable을 건네줘도 좋을 것이다.

그럼에도 불구하고 굳이 Thread에서 계속적으로 Runnable을 보유하고 있는 것은 Runnable객체와 Thread를 강하게 결합시키려는 의도 때문이다. 이것은 의도적으로 위의 2)의 방법을 권장하는 듯한 느낌을 받게 하는듯 하다.

그렇다면 API Reference의 말은 단지 상속을 피하라는 의미만 있는 것인가? 마지막으로 한가지 추정이 되는 부분은 Thread에는 suspend()나 stop()등과 같은 method가 현재 모두 deprecate되었다. 또한 sleep()이나 yield()는 모두 static method이므로 굳이 Thread 객체를 보유할 필요가 없다.

그렇다면 위의 1)의 방법에서 Thread객체를 composition할 필요가 없어진다.

"그렇다면 Thread를 아무도 보유하지 않고 Runnable만 implements한 방식이 최선인가?"

무엇이 정답인지 도무지 알길이 없다. ^^;

**[5.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Exception**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2)

**[5.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "finally 절은 반드시 어떠한 경우에도 실행되는가?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.1)

try ~ catch 문의 finally 절은 'loop라면 break, method라면 return 절'을 만나도 뒤에 있는 finally절은 수행된다. 하지만 다음의 경우는 그렇지 않다.

try{

...

System.exit(1);

}catch(...){

}finally{

... //이 부분은 실행되지 않는다.

}

**[5.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "예외의 종류 3가지 (Error, RuntimeException, 그밖의 Exception)"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.2)

**[5.2.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Error**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.2.1)

이에 관해선 JLS 11.2.1에 다음과 같이 기술되어 있다. "체크되지 않는 예외 클래스(Error와 그 Sub class)는 프로그램안의 다양한 위치에서 발생할 가능성이 있으며, 회복이 불가능하기 때문에 컴파일시 체크되지 않는 것이다. 이러한 예외를 프로그램에서 선언한다고 해도 난잡하고 무의미한 것이 될 뿐이다."

Java의 클래스 librury에서 Error의 sub class를 살펴봐도 AWTError, LinkageError, ThreadDeath, VirtualMachineError 등 'catch해도 소용 없을 것' 들 뿐이다. (OutOfMemoryError는 VirtualMachineError 아래에 위치한다.)

**[5.2.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) RuntimeException**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.2.2)

위의 Error 이외의 Exception들은 application에서 catch할 가능성이 있는 예외들이다.(버그가 없으면 발생하지 않는 예외들) 그리고 RuntimeException은 '어디서든 발생할 가능성이 있는 예외'이다. RuntimeException의 sub class로는 NullPointerException, ArrayIndexOutOfBoundException, ClassCastException 등을 들 수 있다. '이러한 예외는 버그가 없는 한 발생하지 않으므로 일일이 throws 를 작성하지 않아도 된다.'

프로그램에 버그가 없는 한 발생할 수 없는 예외가 발생한 경우 C 언어와 같이 영역 파괴가 일어나기 쉬운 언어라면 프로그램 전체를 종료시키는 것이 정답이겠지만, Java와 같이 영역파괴가 일어나지 않도록 실행시 체크(JVM Classloader의 formal verification process)를 하고 동적으로 프로그램을 load하는 언어에서는 국소적인 NullPointerException 때문에 프로그램 전체를 중지시켜서는 안 될 것이다.

따라서, RuntimeException은 catch하지 않는 것이 바람직하다고 볼 수 있다. 버그가 있는 프로그램은 신속히 종료시키는 것이 대부분의 경우 최선의 방책이라 생각하기 때문이다.

**[5.2.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 그밖의 Exception**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.2.3)

위의 RuntimeException이외의 Exception의 sub class는 사용자의 잘못된 조작 등으로 인해 프로그램에 버그가 없어도 발생할 가능성이 있고 그에 대하여 프로그램이 확실히 대응해야 하는 경우에 사용된다. 예를 들면 FileNotFoundException등이다.

그런데 개발하다 보면 이상하고 의아한 것이 하나 있다. 숫자 부분에 문자를 넣었을때 발생하는 NumberFormatException이다. 이것은 이상하게도 RuntimeException의 sub class이다. 이것은 RuntimeException이 아니었으면 하는데 NumberFormat체크는 Runtime시에만 가능한 모양이다.

**[5.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "OutOfMemoryError는 어떻게 처리해야 하는가?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.2.3)

예전에 Swing에서 Tree구조를 이용하는 프로젝트를 한적이 있다. 이때 Tree에 branch와 node가 무수히 생기자 JVM은 OutOfMemoryError를 내뱉었다. 이에 급한 마음에 OutOfMemoryError를 catch하여 사용자에게 재시작을 요청하는 Dialog를 띄우도록 수정하였다면 이 Dialog가 과연 떳을까? 현재 메모리가 부족한 판에 Dialog를 띄울 메모리가 남아있질 않았던 것이다. 다행히 Dialog가 떴어도 작업은 계속되지 못했을 것이다. NullPointerException가 나기 때문이다.

원인은 나중에 찾았는데, Tree구조에서 부모부터 자식들을 붙이는 순으로 Tree를 구성하는데 자식들을 줄줄이 붙여나가다가 메모리 부족현상이 발생하였고 NullPointerException은 자식이 없으니 클릭하는 순간 null을 반환하여 발생하였던 것이다.

OutOfMemoryError의 가장 좋은 해결책은 불필요한 객체를 만들지 않는 것이었다. 그리고 Tree생성시에도 자식부터 만들고 부모를 만드는 순서로 프로그램을 수정하여 프로젝트를 정상적으로 마칠수 있었다.

마지막에 드는 심정은 프로그램이 OutOfMemoryError를 일으키는 원인이 과연 이렇게 구성되어 발생했는지 어떻게 알수 있을까 하는 의문이다.

**[5.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Object Serialize**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.3)

Java에서는 ObjectOutputStream의 writeObject() method에 데이타 구조 저장소의 참조만 건네주기만 하면 그 안에 있는 모든 객체를 1차원 stream으로 출력해 준다. (파일이나 ByteArrayOutputStream을 이용한 메모리로) 단, static field는 Serialize되지 않는데 이는 Serialize의 대상이 instance 객체뿐이기 때문이다.

**[5.3.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Serialize를 위해서는 marker interface인 java.io.Serializable interface를 implements해야한다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.3.1)

여기서 marker interface는 java.lang.Cloneable과 같이 method와 field의 정의는 없지만 객체 Type을 위한 interface이다. 예전에 Serialize를 이용하여 데이타를 유지하는 프로젝트를 한 적이 있는데 그때 생각했던것이 '모든 class들이 기본적으로 Serializable을 implements하고 있으면 편할텐데..'라는 생각이었다. 하지만 이것은 상당히 위험한 발상이었다.

Serializable이 기본으로 implements되어 잇으면 엉뚱한 객체까지 Serialize되고 그것을 알아채지도 못하는 사태가 일어날 가능성이 높다. Serializable이 optional인 이유는 이러한 이유 때문이리라..

**[5.3.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "super class는 Serializable이 아닌데 sub class만 Serializable인 경우의 문제점"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.3.2)

Serialize을 이용하여 프로젝트를 할때 한번쯤 실수할 수 있는 부분이 상속된 class의 Serialize이다. 컴파일 에러도 없고 Deserialize도 잘 되었다. 하지만 키가 되는 값이 null과 0이었다. 영문을 몰라 다른곳을 헤매여도 보다가 결국 찾은 원인은 부모의 field는 Serialize되지 않는다는 것을 알게 되었다. transient와 마찬가지로 형식별 default 값으로 채워졌었다. 이는 컴파일과 실행시 아무런 오류없이 실행되어 나를 힘들게 하였기에 Java가 원망스러웠던 기분좋은 추억이다. ^^;

**[5.3.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "transient field의 복원(?)관련"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.3.3)

Serialize를 이용한 프로젝트를 할때는 writeObject와 readObject를 이용하여 기본적으로 제공하는 Serialize를 customizing할수있다.

Serializable에 대한 API reference에도 다음과 같이 나와있다.

"Serialize와 Deserialize에 대한 특별한 handling을 위해서는 다음 두개의 특별한 메소드를 구현하면 된다."

private void writeObject(java.io.ObjectOutputStream out) throws IOException;

private void readObject(java.io.ObjectInputStream in) throws IOException, ClassNotFoundException;

이 두 method가 private으로 되어 있는 것을 보고 처음에는 의아해 했었던 기억이 있다. 이를 protected나 public으로 하면 제대로 동작하지 않는다. 이는 override가 이니기 때문이다. 사실은 속에서 reflectiond을 이용하여 강제적으로 호출되고 있는것이다. reflection에서는 private method까지 찾을 수 있기 때문이다.

또한 private으로 한 가장 큰 이유는 Serialize를 객체자신이 직접 해야 안전하다는 의미도 있지 않을까 하는 생각도 든다. 다시 본론으로 들어가서 transient를 복원하는 것에 얘기를 하자면, 사실 transient는 Serialize대상에서 제외되는 것인데 복원을 할 수 있다는 말이 안된다. 하지만 프로젝트를 진행하다 보면 logic상 가능한 경우가 많이 있다.

즉, 모든 field를 Serialize하지 않고 필요한 것만 하고 특정 field는 Serialize한 field들을 이용하여 복원하는 방법이다. 또한 Serialize당시의 객체 상태와 Deserialize시의 객체상태가 서로 다를 수 있는 field도 그것에 해당된다. cafeid만으로 나머지 field는 DB에서 읽어오게 한다면 나머지 field는 transient로 처리하고 Deserialize시 readObject()에서 복원하는 것이다.

**[5.3.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Stack Overflow에 주의하라!"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.3.4)

Serialize를 하다보면 참조로 연결된 객체를 recursive하게 거슬러 올라가며 이것이 너무 깊어지면 Stack Overflow가 발생한다. 가령 linked list같은 경우이다. 이것을 Serialize하면 그 요소수만큼 recursive 호출이 발생한다. 과거(JDK1.3.0시절) 프로젝트 당시 JVM이 5111에서 Stack Overflow가 발생했던 기억이 있다.

물론 실행시 java option에 -Xss 를 이용하여 statck 크키를 조절할 수 있지만 이것은 개발자가 아닌 실행하는 사람들에게 부담이었다. JDK의 LinkedList class의 소스를 보면 writeObject()와 readObject()를 다음과 같이 변경하고 있다.

private synchronized void writeObject(java.io.ObjectOutputStream s) throws IOException {

s.defaultWrtieObject(); //이 코드는 무조건 들어가게 되는데 이곳 소스의 System.arraycopy()에서 overflow발생한다.

s.writeInt(size); //이부분이 실제 추가되어 Stack Overflow를 예방한다.

for(Entry e = ...)

s.writeObject(e.element);

}

...

}

//readObject()도 이와 같은 개념으로 변경되어 있다.

**[5.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "nested class / inner class / 중첩클래스"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.4)

**[5.4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "중첩클래스의 개념"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.4.1)

개인적으로 중첩클래스를 어떠한 경우는 사용하지 않으려 한다. 사용하기가 만만치 않고 코드 읽기가 힘들어 지기때문이다. 하지만 '어떤 클래스 내에서 은폐할 목적으로 사용하는 클래스가 있다면 이것을 사용해야 한다' 실제로 Java의 AWT 클래스 Event Handler를 비롯하여 많은 클래스에서 중첩클래스를 사용하고 있다. 또한 내부 class는 그것을 둘러싸는 class의 instance(enclosing object라고 하는)의 field를 참조 할수 있는것도 장점이다. 하지만 이는 내부클래스가 아닐경우 부부 클래스를 new해서 사용하는것과 별반 다를께 없지 않은가.

**[5.4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "내부클래스는 부모의 참조를 몰래 보유하고 있다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.4.2)

내부 클래스의 instance는 부모의 instance에 대한 참조를 몰래 보유하고 있기 대문에 위에서 얘기한 부모의 field를 참조할 수 있는 것이다. 그러므로 static method에서는 내부클래스를 생성할 수 없다. 다음 예를 보면 바로 알수 있다.

class Test{

class InnerClass {

int i;

...

}

public static void main(String[] args){

InnerClass icls = new InnerClass();

...

}

}

이 소스를 compile하면 다음의 오류가 발생한다. "non-static variable this cannot be referenced from a static context..." main method는 static이므로 this를 참조할수 없다는 것이다. 이는 InnerClass가 new 되면서 외부 클래스 Test의 this를 보유해야 하는데 여기서 static을 만나니 오류를 표출시킨것이다. 물론 일반 instance method에서는 오류가 나지 않는다.

**[5.4.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "local inner class에 대하여"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.4.3)

local inner class라 함은 method내에서 선언된 inner class이다.

public class OuterClass {

public int get(){

int i = 9;

int id = 99;

int id2 = 99;

final int id3 = 100000;

class LocalInnerClass {

int id = 100;

LocalInnerClass(){

System.out.println("LocalInnerClass");

}

int getId(){

return id3 + id;

}

}

LocalInnerClass lic = new LocalInnerClass();

return id + lic.getId();

}

public static void main(String[] args){

OuterClass outer = new OuterClass();

System.out.println("id = " + outer.get());

//결과 값은 "100000(id3) + 100(LocalInnerClass.id) + 99(OuterClass.get())" 인 100199가 나온다.

}

}

위 소스의 LocalInnerClass는 get() 이라는 method에서만 보이는 class이다. 그리고 특이할 만한 부분이 OuterClass의 get() method에서 final로 선언된 id3이 LocalInnerClass에서 참조 가능해 진다. id2를 참조하면 compile error가 나지만 final로 선언된 것은 오류가 나지 않는다.

이는 local variable은 method에서 나오는 순간 사라지는데, local inner class는 local variable보다 수명이 조금더 길기 때문에 final만 허용한 것이다.

**[5.4.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "anonymous class(무명클래스)에 대하여"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-5.4.4)

무명 클래스는 말그대로 이름이 없는 클래스이다.

class AnonymousTest {

private interface Printable {

void print();

}

static void doPrint(Printable p){

p.print();

}

public static void main(String[] args){

doPrint( new Printable(){

public void print(){

System.out.println("this is new Printable print()");

}

});

}

}

위 소스의 "doPrint( new Printable(){" 부분이 무명클래스 이다. compile을 수행하면 AnonymousTest$Printable.class, AnonymousTest$1.class, AnonymousTest.class 세개의 클래스가 생긴다. 여기서 AnonymousTest$Printable.class는 Printable interface이고 AnonymousTest$1.class이 무명클래스이다.

이 소스를 보면 처음에 드는 의심이 Printable interface를 new 했다는 것이다. 여기서 굳이super class(이 소스에서는 interface)를 저정해야 하는 이유는 아무것도 상속하지 않는 무명 클래스의 instance를 만들어 봐야 의미가 없기 때문에 이렇게 한듯하다.

"무명클래스는 어떤 class나 interface를 상속/구현 해야만 그 instance를 사용할 수 있는 것이다"

이처럼 무명 클래스를 사용하면 어떤 절차(수행)를 다른 method의 인수로 건네줄 수 있게 된다. 하지만 간단한 로직만 구현처리해야 한다.

"무명클래스는 조금만 복잡해져도 급격히 소스의 가독성이 떨어지게 되므로 남용하지 않는 것이 바람직하다"

**[6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 이래도 Java가 간단한가?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6)

**[6.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) method overload 에서의 혼란?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.1)

**[6.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "overload란 이름이 가고 인수가 다른 method에 compiler가 다른 이름을 붙이는 기능"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.1.1)

overload를 구현하면 bytecode로 변환시 다른 이름으로 method가 변환되어 별개의 method로 처리된다. 이를 JVM에서 method descripter라 하여 Oolong asembler로 변화시 다른 형태의 method가 된다. 예를 들어 "void get(double d, long l)" 은 "get(DJ)V"로 변경된다. 여기서 D는 double, J는 long, V는 void를 의미한다.

그런데 여기서 "get(DJ)" 부분만 method 이름이므로 return type이 다른 동일 method는 overload 할 수 없다. 따라서 overload는 정적(compile시 결정)이라는 명제가 성립니다. 그래서 동적으로 사용되면 compile시 오류를 표출한다. 아래의 소스를 보자. 여기에는 IFS라는 interface와 이를 implements한 Impl1, Impl2 라는 class가 있다.

//IFS.java

interface IFS {

public String getName();

}

//Impl1.java

class Impl1 implements IFS {

public String getName(){

return "Impl1";

}

}

//Impl2.java

class Impl2 implements IFS {

public String getName(){

return "Impl2";

}

}

//main이 있는 OverloadTest.java

public class OverloadTest {

static void pr(int i){

System.out.println("pr\_int : " + i);

}

static void pr(String s){

System.out.println("pr\_string : " + s);

}

static void pr(IFS ifs){

System.out.println("pr\_string : " + ifs.getName());

}

static void pr\_run(Impl1 i1){

System.out.println("pr\_run : " + i1.getName());

}

static void pr\_run(Impl2 i2){

System.out.println("pr\_run : " + i2.getName());

}

public static void main(String[] args){

OverloadTest test = new OverloadTest();

test.pr(10);

test.pr("Jeid");

IFS ifs1 = new Impl1();

test.pr(ifs1);

IFS ifs2 = new Impl2();

test.pr(ifs2);

//pr\_run(ifs1);

//pr\_run(ifs2);

}

}

위의 소스를 수행하면 정상적으로 compile이 될것인가?

당연히 잘 된다. pr()은 overload를 잘 구현했다. 하지만 소스 하단의 두 주석문을 풀면 어떻게 될까? 이는 compile오류를 낸다.

OverloadTest.java:36: cannot resolve symbol

symbol : method pr\_run (IFS)

location: class OverloadTest

pr\_run(ifs1);

^

OverloadTest.java:37: cannot resolve symbol

symbol : method pr\_run (IFS)

location: class OverloadTest

pr\_run(ifs2);

^

2 errors

실제 위 둘의 pr\_run method는 bytecode로 변환시 "pr\_run(Lpackage\_name.IFS)V"로 동일하게 생성된다. 따라서 compile시에 오류를 표출한다. 이 소스를 보면 알 수 있듯이 "method overload는 정적(compile시)으로 미리 결정되며, 동적(실행시판단)으로 사용할수 없다."

**[6.1.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "그렇다면 overload에서 실제로 혼동되는 부분은 무엇인가?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.1.2)

다음 소스를 보고 실제로 수행되는 method를 찾아보라.

class OverloadTest2 {

static int base(double a, double b){ ... } //method A

static int count(int a, int b){ ... } //method B

static int count(double a, double b){ ... } //method C

static int sum(int a, double b){ ... } //method D

static int sum(double a, int b){ ... } //method E

}

* base(3,4) 를 호출했을때 수행되는 method는? => 당연히 method A (3과 4는 정수라도 double이 되므로 정상적으로 수행됨)
* count(3,4) 를 호출했을때 수행되는 method는? => B와 C중 갈등이 생긴다. 이럴경우 JVM은 가장 한정적(more specific)한 method를 찾는다. 여기서 3과 4는 정수형에 가까우므로 method B 가 호출된다.
* count(3, 4.0) 을 호출했을때 수행되는 method는? => 이것은 4.0 이 double이므로 method C 가 더 한정적이므로 method C 가 호출된다.

* sum(3,4.0) 을 호출했을때 수행되는 method는? => 이것은 당연히 type이 일치하는 method D.

* sum(3,4) 를 호출했을때 수행되는 method는?? => 이런 코드가 소스내에 있으면 다음과 같은 compile 오류를 표출한다.

OverloadTest.java:48: reference to sum is ambiguous, both method sum(int,double)

in OverloadTest and method sum(double,int) in OverloadTest match

System.out.println("sum(3,4) = " + sum(3,4));

^

1 error

method D와 method E가 애매하다는 compile 오류이다. 이것은 둘중 어느것이 더 한정적인지 찾을 수 없으므로 bytecode 를 생성 할 수 없다는 것이다.

"이렇듯 compiler에게 불필요한 오해(혼동)를 초래하는 overload는 사용하지 않는 것이 좋다. 개인적으로 overload를 가능한 사용하지 않으려 하고 필요하다면 인수의 개수가 다른 overload를 사용하는 편이다."

**[6.1.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) (참고) 또다른 혼동, overload한 method를 override 하면?**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.1.3)

overload란 compiler가 bytecode변환시 다른 이름을 붙이는 기능이라는 것을 위에서 설명했다. 따라서 super class에서 overload한 method를 상속하여 override하면 완전 별개의 method를 override한것처럼 JVM은 판단한다. 즉, overload와 override는 직교(전혀상관없는)하는 개념이다.

**[6.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속/override/은폐 에서의 복잡함**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2)

**[6.2.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "Java class의 member 4 종류"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2.1)

1. instance field
2. instance method
3. static field
4. static method

여기서 상속을 하였을 경우 runtime시 객체의 형식에 따라 선택되는 것은? 2번 instance method 뿐이다. 즉, 동명의 member를 sub class에서 선언했을 때 instance method만 override 되고 나머지는 완전 별개의 member가 된다. 따라서 위의 1,3,4는 sub class에서 동일하게 선언했을 경우 별개의 것으로 인식되며 compile시에 무엇을 access 할지 결정된다.

즉, instance method는 override되지만 instance field/static field는 은폐된다. override는 실행시 객체의 형식에 따라 처리 할당되지만, 은폐의 경우는 compile시에 결정되고 만다.

**[6.2.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "override시 method 이름에 대한 함정"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2.2)

과거에 코딩을 하던중 정말이지 어처구니 없는 경우를 당했다. override 하는 method이름을 잘못써서 황당한(?) 고생을 한적이 있다. super class의 writable()이라는 method를 writeable()이라고 override(?)하였는데 프로그램 수행 중에 writable()이 항상 false가 나오는 것이 아닌가? 그래서 소스를 추적추적 하다 몇시간을 허비했었던 기억이 있다.

java를 접한지 얼마되지 않았고 요즘같이 eclipse같은 에디터도 없이 메모장에서 코딩하던 시절이라 더욱 고생했던것 같다. 한참 후에야 우연히 스펠링이 잘못된걸 알고 얼마나 황당했던지... 지금 생각하면 이것도 좋은 추억이리라.

무조건 override 잘 되었을거라 생각 했던 나의 불찰도 있었지만 compile때나 runtime시 아무런 반응을 보이지 않던 Java도 원망스러웠다. 2003년도에 C#으로 프로젝트를 했는데 C#은 상속의 override에 대하여 "override void writalbe().."과 같이 정의시 override를 명시해야 된다는 것을 보고 상당히 마음에 들어 했던 기억이 있다. 가독성도 뛰어날 뿐더러 나의 몇시간동안의 헤메임도 없을 것이기 때문다. Java도 이렇게 확실한 명세였으면 정말 좋겠다.

**[6.2.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "또다른 나의(?) 실수 - 말도 안되는 오타"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2.3)

위의 method이름을 잘못써서 고생하기 이전에 아주 비슷한 고생을 한적이 있다.

'난 정말 바보인가'라는 생각을 들게 했던 문제였다. 초보 시절에는 왜이리도 오타가 많이 나던지... 요즘은 대충 키보드 두드려도 오타가 잘 안나는데 그 시절에 오타 때문에 느린 CPU에서 컴파일을 몇번을 했는지...

기억을 되살리면 소스는 다음과 같다.

public class Member {

private int memberNo;

public int getMemberNo(){

return this.memberNo;

}

public void setMemberNo(int menberNo){

this.memberNo = memberNo;

}

......

}

위 소스의 Member에는 다른 여러가지 member field가 있는데 DB의 member table에 memberid 컬럼이 memberno로 변경되면서 Member class의 memberId를 memberNo로 변경하게 되었다. 위와 같이 수정하여 배포해놓고 테스트를 하는데 시스템이 완전히 뒤죽박죽으로 돌아버리는 것이 아닌가. 이 경우도 method 이름처럼 몇시간을 헤매었다.

이번에 argument의 오타로 인한 어처구니 없는 실수였다. setMemberNo(int menberNo)에서 문제가 발생되었던 것이다. 인수의 memberNo를 menberNo로 잘못친것이다. 그래서 memberNo에는 해당 member의 memberno가 아닌 0이 모두 들어갔어던 것이다. 시스템은 memberno를 기준으로 도는 부분이 너무나 많았기에 오류나는 부분도 많았으며 DB에서는 제대로 된 memberno을 읽어 왔으며, compile과 runtime시 아무런 반응도 없었기에, 초보자를 그렇게도 고생시켰나 보다.

이것도 member field면 무조건 this를 붙이도록 하던지 Java가 인수는 'm\_'와 prefix를 붙이도록 Coding Style을 정의- SUN사이트의 Java Coding 규약에는 "Variable names should not start width underscore\_ or dollar sign $ characters, even though both are allowed." 와 같이 명시되어 있다 - 했더라면 발생하지 않았을 문제이다.

또한 C언어나 C#에서 처럼 compile 경고레벨을 높여놓으면 "menberNo는 어디서도 사용하지 않습니다."와 같은 메세지를 보여 줬더라면 고생을 덜 하지 않았을까?

**[6.2.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "static member를 instance를 경유하여 참조해서는 안 된다."**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2.4)

예를 들어 ClassA 에 public static int AA 라는 static field가 있을 경우 ClassA.AA 로 접근해야 하는데, 다음과 같이 사용하는 실수를 범한다.(물론 오류는 없지만)

ClassA a = new ClassA();

int i = a.AA; //instance를 경유하여 접근

int j = ClassA.AA; //올바르게 접근

그럼 왜 굳이 ClassA.AA와 같이 instance가 아닌 class이름을 붙여야 할까?

static member(static field/static method)는 compile시에 이미 어느것을 호출할 지 결정하기 때문에 위의 a.AA와 같은 것은 static이 아닌것 같은 오해와 혼란만 가져오기 때문이다. 심지어 개인적으로는 동일 class 내 - 위 소스에서 ClassA의 member method - 에서 ClassA.AA라고 사용하는 편이다.

이는 local variable과 혼동될 염려도 없을뿐더러 AA라는 변수가 static이라는 것도 확실히 알 수 있기 때문이다. 물론 private static 의 경우는 ClassA.BB 와 같이 하지 않고 BB 라고 해도 무방하겠지만 말이다.

**[6.2.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "super keyword는 부모의 this"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.2.5)

Java 개발자 대부분은 'super' 에 대하여 그렇게 민감하지 않을 것이다. 그거 super() 나 super.method1() 과 같이 사용되지 그 이상에 대해선 깊이 생각하지 않게 된다. super를 한마디로 정리하면 다음과 같다.

"super keyword는 instance method등에서 this를 사용할 수 있는 곳에서만 쓸 수 있다. this의 자리에 super라고 쓰면 현재 class의 member가 참조되는 대신 부모 class의 member가 참조되는 것이다."

**[6.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 상속에 관한 또 다른 문제**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.3)

**[6.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 그밖의 함정**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.4)

**[6.4.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "생성자에 void 를 붙인다면?"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.4.1)

생성자에 void를 붙인다면 그 class가 new 될때 그 생성자(?)가 실행될까?? 아래의 'Constuctor'라는 문자열은 출력될까?

public class ConstructorTest{

void ConstructorTest(){

System.out.println("Constuctor");

}

.....

}

출력되지 않는다. 물론 compile시 아무런 경고도 없었다. 즉, void가 붙은 ConstructorTest()는 생성자가 아니라 instance method일 뿐이었고 new시에는 default constructor가 실행 되었던 것이다.

**[6.4.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "if / switch 의 함정"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-6.4.2)

Java 개발자라면 대부분이 초보시절에 if 조건절에 '==' 대신 '='을 써본 기억이 있을것이다. 예를 들어 "if( isListenLecture == Student.STUDENT )" 를 "if( isListenLecture = Student.STUDENT )" 로 잘못 쓴 경우이다. 여기서 Student.STUDENT는 boolean type이다. 여기서 isListenLecture는 항상 Student.STUDENT 값을 갖게 되는 버그가 생긴다. 이는 compile시에 아무런 경고도 없다. 이렇게 한번 당하고 나면 앞으로는 '=='를 정확히 쓰게 되거나 아니면 다음과 같이 쓴다.

"if( isListenLecture )" 또는 "if( !isListenLecture )" 라고 말이다. 이것이 더욱 간결하고 의미도 분명해 지기 때문이다. 또한 다음 소스와 같은 오류도 범하는 경우가 있다. 이는 잘못된 indentation으로 빚어지는 초보의 함정이다.

이글을 읽는 분께 한가지 당부드리고 싶은것은 여기서 초보라고 다 그런건 아니라는 것이다.

....

if( a < 5 )

b = 3;

c = 10; //이부분은 나중에 추가된 라인이다.

if( isStudent )

if( isFemale )

sayHello("Hi~~");

else

sayHello("Hello Professor~");

위의 소스중 c = 10; 이 if( a < 5 )의 참일때 수행된다고 오해할 수도 있고, sayHello("Hello Professor~"); 부분이 if( isStudent )의 else 부분이라고 오해 할 수도 있다. 이것은 전적으로 indentation(들여쓰기)의 불찰로 개발자가 잘못 읽을 수 있는 부분이다. Java Coding Style에서는 if문 다음에 한줄의 코드가 있더라도 {} 를 사용하길 권고한다. 그러면 첫번째 if문과 같은 오류를 방지할 수 있고 두번째 if문에서도 보다 가독성이 생길 것이다.

이와 유사한 것으로 switch문의 case 절에서 break를 쓰지 않아 항상 동일하게 처리되는 버그도 경험해 보았을 것이다.

**[7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java 기능 적용 몇가지**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-7)

**[7.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 대규모 개발에서 interface 분리하기**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-7.1)

**[7.1.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) "interface 분리의 필요성"**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-7.1.1)

Java와 같은 객체지향언어에서는 공개해야 할 method만을 public으로 하고, 공개할 필요가 없는 것은 private으로 하여 class의 상세한 내용을 은폐할 수 있게 되어 있다. 그런데 private 부분이 은폐되어 있는것 처럼 보이는가?

소스를 보면 훤히 들여다 보이는데?

대규모 개발은 하부 class부터 bottom-up으로 진행하는 것이 이상적인 형태일 것이다. 그런 형태로 개발하면 임의의 시점에서 테스트를 할 수도 있다. 그러나 현실적으로 단기간에 많은 수의 개발자가 붙어서 단시간에 개발을 진행하는 경우가 많다. 또한 서로 호응하는 관계에 있는 class들은 어느쪽이 하부인지 정의하기가 난감할때가 많다. 이런경우 우리는 흔히 package단위로 나누어 개발한다. 하지만 이럴경우 어느정도 코딩이 종료될때까지 테스트하기가 상당히 힘들어 진다. Java에서는 private member와 method 구현까지 하나의 파일에 코딩하는데 개발 중간에 공개하여 다른 개발자가 이용해야 하는 class를 배포할 수 없으므로 동시 개발이 까칠해 진다.

이 상황에서 다른 package(개발자)에 공개해야 하는 class 부분을 interface로 공개하면 많은 부분 유연하게 된다. 이 interface를 다른 개발자는 개발을 하고 테스트가 필요하다면 TestImpl class를 만들어 하면된다. RMI나 CORBA에서도 Stub은 이런식으로 IDL을 정의한다.

**[7.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java에서의 열거형**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-7.2)

Java에서는 열거형-C의 구조체, 공용체-이 없다. 열거형이 왜 필요하냐고 반문하는 개발자도 있을 것이다.

하지만 열거형이 없어 곤란을 경험한 개발자도 꽤 있으리라 본다. 최근언어(특히 객체지향 언어) - Java, Eiffel, Oberon등 - 에는 열거형은 포함되어 있지 않다. C#에는 있긴 하지만.

이런 이유로 Java AWT의 Label class는 다음과 같이 구현되어 있다.(텍스트의 정렬값관련)

public static final int LEFT = 0;

public static final int CENTER = 1;

public static final int RIGHT = 2;

...

label.setAlignment(Label.CENTER);

...

하지만 위의 소스에는 문제가 있다. setAlignment() method의 인자가 int인 것이다. 만약 위에 정의한 0, 1, 2가 아닌 다른 int 값이 들어가도 compile/runtime시 알수가 없다. 그래서 주석을 달게 되는데, 주석이라 함은 정말이지 최후의 수단이라고 봐야 한다.

실제로 우리가 개발해 놓은 소스에도 이런부분이 있으리라 예상된다. 이 문제를 어떻게 하면 해결할 수 있을까?

Java에서 열거형을 한번 만들어 보자.

//LabelAlignment.java

public class LabelAlignment {

private LabelAlignment() {} //이는 생성자를 private으로 하여 다른데서는 만들지 못하도록 하기위함이다.

public static final LabelAlignment LEFT = new LabelAlignment():

public static final LabelAlignment CENTER = new LabelAlignment():

public static final LabelAlignment RIGHT = new LabelAlignment():

}

//변형된 Label.java 의 일부..

public synchronized void setAlignment(LabelAlignment alignment){

if( alignment == LabelAlignment.LEFT ){

...//왼쪽으로 맞추기..

}else if( ...

...

}

}

...

위에서 작성한 소스는 잘 작동한다. 서로 다른 3개의 instance이므로 reference가 달라 '==' 연산도 가능하고, 훌륭하다.

하지만 한가지 문제가 있다. LabelAlignment가 Serializable한 class에서 serialize되었다 deserialize 된다면?

LabelAlignment alignment 는 새로운 instance가 되고 serialize전의 reference와 다른 참조 위치를 갖게 되어 '==' 연산은 버그를 발생시킨다.

그럼 이것만 해결하면 되겠는데, 어떻게 refactoring하면 될 것인가? '==' 연산 대신 equals로 변형하면 되겠는데.

//LabelAlignment.java

public class LabelAlignment {

private int flag;

private LabelAlignment(int flag){

this.flag = flag;

}

public static final LabelAlignment LEFT = new LabelAlignment(0):

public static final LabelAlignment CENTER = new LabelAlignment(1):

public static final LabelAlignment RIGHT = new LabelAlignment(2):

public boolean equals(Object obj){

return ((LabelAlignment)obj).flag == this.flag;

}

}

//변형된 Label.java 의 일부..

public synchronized void setAlignment(LabelAlignment alignment){

if( LabelAlignment.LEFT.equals(alignment) ){

...//왼쪽으로 맞추기..

}else if( ...

...

}

}

...

하하, Serialize까지 잘 작동한다. ^^;

여기서 Debug를 고려한다면 0, 1, 2 대신 문자열로 "LEFT", "CENTER", "RIGHT"로 한다면 더욱 명확하지 않을까?

(주의) 위에서처럼 LabelAlignment.LEFT 라고 쓰기 싫어서 상수 interface를 만들어 그걸 implements 하여 그냥 LEFT 라고 쓰는 것을 뿌듯해 하며 쓰는 개발자들이 있다. 물론 Swing의 소스들을 보다보면 SwingConstants라는 interface에 LEFT를 비롯하여 온갖 잡다한 상수를 집어넣어놓고 여기 저기서 implements해서 사용하고 있다. 이런 코딩 스타일은 '내 스타일이야~' 가 아니라 냄새나는 코드이다.

LEFT라는 것이 구현한 class에 이미 있을 수 있을 수 있을뿐아니라 구현한 모든 클래스에서 LEFT를 보유하여 SwingConstants.LEFT뿐 아니라 Impl.LEFT로도 사용되게 되어 온갖 혼란을 초래하게 된다. 입력량을 줄이기 위해 interface를 implements 해서는 안되지 않을까?

**[7.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Debug write**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-7.3)

C에서는 다음과 같이 pre-process로 정의하면 DEBUG라는 식별자를 #define하지 않으면 컴파일후 해당 소스의 부분이 삭제된다.

#ifdef DEBUG

fprintf(stderr, "error...%d\n", error);

#endif /\* DEBUG \*/

그럼 Java에서는?

Java에서는 Pre-process가 없지만 다음과 같이 작성했을때 Debug.isDebug 가 final로 선언되어 있으면 compile후 아래 3줄 모두 삭제 된다.(단 Debug.isDebug 가 false 로 초기화 되었다면 제거된다.)

if( Debug.isDebug ){

System.out.println("error..." + error);

}

Java는 compile시 byte code 생성시 final은 정적으로 판단하여 미리 정의하기 때문에 위의 3줄은 삭제될 수 있다. if문과 함께 없어지게 되므로 처리 속도에 피해를 주지 않는다. 단, 주의해야 할 점은 Debug.isDebug 값이 변경되면 이 것을 사용하고 있는 측도 모두 함께 다시 compile해야 한다. bytecode를 다시 만들어야 하기 때문이다.

그런데, 이 소스를 Debug.write()와 같이 static 으로 하여 이 method내에서 판단하게 하면 편리할텐데. 그리고 class별로 ON/OFF 처리를 할 수 있으면 좋을텐데, 어찌 하면 가능할 것인가?

그럼 먼저 호출한 쪽의 class이름을 찾아보자. 접근은 Exception의 printStackTrace()로 부터 시작되었다. 하지만 이 소스에는 Exception 객체를 new한 시점에 결정되어 있다. 그래서 부모인 Throwable의 생성자를 확인해 보니 fillInStackTrace() 로 되어있는데 이 method는 native method였다.

API Reference를 보면 Thread class에서는 dumpStackTrace()라는 method가 있었다. 소스를 보니, 그것도 생성시점이었다. 아무래도 예외방면에서 찾는건 무리인듯 했다.

그래서 class의 호출계층을 나타내는 java.lang.SecurityManager의 getClassContext() method로 접근하였다. sample 소스는 다음과 같다.

// 1. GetCallerSecurityManager.java

public final class GetCallerSecurityManager extends SecurityManager {

public Class[] getStackTrace(){

return this.getClassContext();

}

}

// 2. GetCallerClass.java

public final class GetCallerClass {

private static GetCallerSecurityManager mgr;

static{

mgr = new GetCallerSecurityManager();

System.setSecurityManager(mgr);

}

public static void writeCaller(String str){

Class[] stk = mgr.getStackTrace();

int size = stk.length;

for(int i = 0; i < size; i++){

System.out.println("stk[" + i + "] = " + stk[i]);

}

String className = stk[2].getName();

System.out.println("className is " + className + " : " + str);

}

}

// 3. GetCallerClassMain1 : 호출하는 클래스 예제 1

public class GetCallerClassMain1 {

public static void main(String[] args){

GetCallerClass.writeCaller(", real is 1.");

}

}

// 4. GetCallerClassMain1 : 호출하는 클래스 예제 2

public class GetCallerClassMain2 {

public static void main(String[] args){

GetCallerClass.writeCaller(", real is 2.");

}

}

위의 3번 주석과 4번 주석 부분을 수행하면 다음과 같은 결과가 나온다.

className is GetCallerClassMain1 : , real is 1.

className is GetCallerClassMain2 : , real is 2.

정확히 호출한 클래스를 표현하고 있다. 이것을 비교해서 클래스별 ON/OFF를 구현하면 된다.

**[8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Java 5.0 Tiger 에 대하여**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8)

Tiger에서는 새로운 개념의 적용이 많은 부분 시도 되었다. 이중 가장 기본이 되는 몇가지를 살펴보자.

**[8.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Working with java.util.Arrays**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.1)

Tiger에서는 무엇보다도 Collection class들에 대해 많은 부분 정비하였다. 예를 들면 for/in 구문 지원과 Generic Type member와 Arrays Utility class 등이다. 그럼 Collection에 대한 static method들을 담고 있는 Arrays 에 대해 다음 example로 한눈에 살펴보자.

package com.jeid.tiger;

import java.util.Arrays;

import java.util.Comparator;

import java.util.List;

public class ArraysTester {

private int[] arr;

private String[] strs;

public ArraysTester(int size) {

arr = new int[size];

strs = new String[size];

for (int i = 0; i < size; i++) {

if (i < 10) {

arr[i] = 100 + i;

} else if (i < 20) {

arr[i] = 1000 - i;

} else {

arr[i] = i;

}

strs[i] = "str" + arr[i];

}

}

public int[] getArr() {

return this.arr;

}

public String[] getStrs() {

return this.strs;

}

public static void main(String[] args) {

int size = 50;

ArraysTester tester = new ArraysTester(size);

int[] testerArr = tester.getArr();

int[] cloneArr = tester.getArr().clone();

String[] testerStrs = tester.getStrs();

String[] cloneStrs = tester.getStrs().clone();

// clone test

if (Arrays.equals(cloneArr, testerArr)) {

System.out.println("clonse int array is same.");

} else {

System.out.println("clonse int array is NOT same.");

}

if (Arrays.equals(cloneStrs, testerStrs)) {

System.out.println("clonse String array is same.");

} else {

System.out.println("clonse String array is NOT same.");

}

// 2부터 10까지 값 셋팅

Arrays.fill(cloneArr, 2, 10, new Double(Math.PI).intValue());

testerArr[10] = 98;

testerStrs[10] = "corea";

testerStrs[11] = null;

List<String> listTest = Arrays.asList(testerStrs);

System.out.println("listTest[10] = " + listTest.get(10));

System.out.println("------- unsorted arr -------");

System.out.println("Arrays.toString(int[]) = " + Arrays.toString(testerArr));

System.out.println("Arrays.toString(String[]) = " + Arrays.toString(testerStrs));

Arrays.sort(testerArr);

// Arrays.sort(testerStrs); //NullPointerException in sort method..(null이 없더라도 길이에 대한 크기 체크는 못함)

Arrays.sort(testerStrs, new Comparator<String>() {

public int compare(String s1, String s2) {

if (s1 == null && s2 == null) {

return 0;

} else if (s1 == null && s2 != null) {

return -1;

} else if (s1 != null && s2 == null) {

return 1;

} else if (s1.length() < s2.length()) {

return -1;

} else if (s1.length() > s2.length()) {

return 1;

} else if (s1.length() == s2.length()) {

return 0;

} else {

return s1.compareTo(s2);

}

}

});

System.out.println("------- sorted arr -------");

System.out.println("Arrays.toString(int[]) = " + Arrays.toString(testerArr));

System.out.println("Arrays.toString(String[]) = " + Arrays.toString(testerStrs));

System.out.println("------------------------------------------------");

String[][] mstrs1 = { { "A", "B" }, { "C", "D" } };

String[][] mstrs2 = { { "a", "b" }, { "c", "d" } };

String[][] mstrs3 = { { "A", "B" }, { "C", "D" } };

System.out.println("Arrays.deepToString(mstrs1) = " + Arrays.deepToString(mstrs1));

System.out.println("Arrays.deepToString(mstrs2) = " + Arrays.deepToString(mstrs2));

System.out.println("Arrays.deepToString(mstrs3) = " + Arrays.deepToString(mstrs3));

if( Arrays.deepEquals(mstrs1, mstrs2)) {

System.out.println("mstrs1 is same the mstrs2.");

}else {

System.out.println("mstrs1 is NOT same the mstrs2.");

}

if( Arrays.deepEquals(mstrs1, mstrs3)) {

System.out.println("mstrs1 is same the mstrs3.");

}else {

System.out.println("mstrs1 is NOT same the mstrs3.");

}

System.out.println("mstrs1's hashCode = " + Arrays.deepHashCode(mstrs1));

System.out.println("mstrs2's hashCode = " + Arrays.deepHashCode(mstrs2));

System.out.println("mstrs3's hashCode = " + Arrays.deepHashCode(mstrs3));

}

}

**[8.2](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Using java.util.Queue interface**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.2)

Queue를 이용하여 First In First OutOrdering한 Queue를 구현 가능하다.

package com.jeid.tiger;

import java.util.LinkedList;

import java.util.PriorityQueue;

import java.util.Queue;

public class QueueTester {

public static void main(String[] args) {

System.out.println("---------- testFIFO ----------");

testFIFO();

System.out.println("---------- testOrdering ----------");

testOrdering();

}

private static void testFIFO() {

Queue<String> q = new LinkedList<String>();

q.add("First");

q.add("Second");

q.add("Third");

String str;

while ((str = q.poll()) != null) {

System.out.println(str);

}

}

private static void testOrdering() {

int size = 10;

Queue<Integer> qi = new PriorityQueue<Integer>(size);

Queue<String> qs = new PriorityQueue<String>(size);

for (int i = 0; i < size; i++) {

qi.offer(10 - i);

qs.offer("str" + (10 - i));

}

for (int i = 0; i < size; i++) {

System.out.println("qi[" + i + "] = " + qi.poll() + ", qs[" + i + "] = " + qs.poll());

}

}

}

**[8.3](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) java.lang.StringBuilder 사용하기**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.3)

StringBuffer가 synchronize하지 않은 method들로 구성된 듯한 StringBuilder를 사용하므로 성능 향상을 도모할수 있다. 사용법은 StringBuffer와 동일하다.

package com.jeid.tiger;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class StringBuilderTester {

public static void main(String[] args) {

List<String> list = new ArrayList<String>();

list.add("str1");

list.add("str2");

list.add("str3");

String ret = appendItems(list);

System.out.println("ret = " + ret);

}

private static String appendItems(List<String> list) {

StringBuilder sb = new StringBuilder();

for (Iterator<String> iter = list.iterator(); iter.hasNext();) {

sb.append(iter.next()).append(" ");

}

return sb.toString();

}

}

**[8.4](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Using Type-Safe Lists**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.4)

Collection에 type을 명시하여 type-safe 하게 처리 가능. 아래에서 type을 명시하지 않을 경우 compile error가 남을 보여준다. tip으로 Number를 이용하여 byte, short, int, long, double, float 동시 사용하는 부분 참조.

package com.jeid.tiger;

import java.util.Iterator;

import java.util.LinkedList;

import java.util.List;

public class ListTester {

public static void main(String[] args) {

List<String> list = new LinkedList<String>();

list.add("str1");

list.add("str2");

list.add(new Integer(123)); // <-- String이 아니므로 compile error!!

//Iterator에 String type을 명시하므로 정삭작동됨.

for (Iterator<String> iter = list.iterator(); iter.hasNext();) {

String str = iter.next();

System.out.println("srt = " + str);

}

//Iterator에 String type을 명시하지 않았으므로 아래 A 부분에서 compile 오류 발생!!

for (Iterator iter = list.iterator(); iter.hasNext();) {

String str = iter.next(); //A

System.out.println("srt = " + str);

}

//byte, short, int, long, double, float 동시 사용

List<Number> lstNum = new LinkedList<Number>();

lstNum.add(1);

lstNum.add(1.2);

for (Iterator<Number> iter = lstNum.iterator(); iter.hasNext();) {

Number num = iter.next();

System.out.println("num = " + num);

}

}

}

**[8.5](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Writing Generic Types**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.5)

class 나 interface keyword에 type을 명시하여 동일 타입 명시 가능. 주의 할 점은 any type은 static 일 수 없다.(동적으로 type이 정해지므로)

class AnyTypeList<T> {

//class AnyTypeList<T extends Number> { // <-- 이는 Number를 상속한 type은 허용하겠다는 의미.

private List<T> list;

//private static List<T> list; // <-- 이는 정적이므로 compile error 발생!!!

public AnyTypeList(){

list = new LinkedList<T>();

}

public boolean isEmpty(){

return list == null || list.size() == 0;

}

public void add(T t){

list.add(t);

}

public T grap(){

if (!isEmpty() ) {

return list.get(0);

} else {

return null;

}

}

}

**[8.6](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 새로운 static final enum**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.6)

예제를 통해 알아보자.

package com.jeid.tiger;

import com.jeid.BaseObject;

import com.jeid.MyLevel;

public class EnumTester extends BaseObject {

private static long start = System.currentTimeMillis();

public static void main(String[] args) {

try {

test();

enum1();

} catch (Exception e) {

e.printStackTrace();

}

printEllapseTime();

}

private static void test() throws Exception {

byte[] b = new byte[0];

System.out.println(b.length);

}

private static void enum1() {

//enum TestEnum { A, B }; //enum cannot be local!!!

for(MyVO.TestEnum te: MyVO.TestEnum.values()){

System.out.println("Allow TestEnum value : " + te);

}

System.out.println("---------------------------------------");

MyVO vo = new MyVO();

vo.setName("enum1");

vo.setLevel(MyLevel.A);

System.out.println(vo);

System.out.println("isA = " + vo.isA() + ", isGradeA = " + vo.isLevelA()+ ", isValueOfA = " + vo.isValueOfA());

System.out.println("getLevelInKorean = " + vo.getLevelInKorean());

}

private static void printEllapseTime() {

System.out.println("==> ellapseTime is " + (System.currentTimeMillis() - start) + " ms.");

}

}

package com.jeid.tiger;

import com.jeid.BaseObject;

import com.jeid.MyLevel;

public class MyVO extends BaseObject {

enum TestEnum {

A, B

}; // this is same public static final

private int id;

private String name;

private MyLevel grade;

// private List<T> list;

public MyLevel getLevel() {

return grade;

}

public void setLevel(MyLevel grade) {

this.grade = grade;

}

public boolean isA() {

return "A".equals(this.grade);

}

public boolean isValueOfA() {

return MyLevel.valueOf("A").equals(grade);

}

public boolean isLevelA() {

return MyLevel.A.equals(this.grade);

}

//A,B,C..대신 0,1,2... 도 동일함.

public String getLevelInKorean() {

switch(this.grade){

case A:

return "수";

case B:

return "우";

case C:

return "미";

case D:

return "양";

case E:

return "가";

default:

return "없음";

}

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**[8.7](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Using java.util.EnumMap**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.7)

java.util.Map과 동일하나 key가 enum type이어 한다. 예제로 살펴보자.

package com.jeid.tiger;

import java.util.EnumMap;

public class EnumMapTester {

private enum MyEnum {

A, B, C

}; // this is same the static final..

public static void main(String[] args) {

MyEnum[] enums = MyEnum.values();

System.out.println("MyEnum is " + enums[0] + ", " + enums[1] + ", " + enums[2]);

EnumMap<MyEnum, String> em = new EnumMap<MyEnum, String>(MyEnum.class);

em.put(MyEnum.A, "수");

em.put(MyEnum.B, "우");

em.put(MyEnum.C, "미");

em.put(MyEnum.B, "가"); //key 중복은 HashMap과 동일하게 overwrite임.

for (MyEnum myEnum : MyEnum.values()) {

System.out.println(myEnum + " => " + em.get(myEnum));

}

}

}

**[8.8](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Using java.util.EnumSet**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.8)

java.util.Set과 동일하나 value가 enum type이어 한다. 예제로 살펴보자.

package com.jeid.tiger;

import java.util.EnumSet;

public class EnumSetTester {

private enum MyEnum {

A, B, C, a, b, c

}; // this is same the static final..

public static void main(String[] args) {

MyEnum[] enums = MyEnum.values();

System.out.println("MyEnum is " + enums[0] + ", " + enums[1] + ", " + enums[2]);

EnumSet<MyEnum> es1 = EnumSet.of(MyEnum.A, MyEnum.B, MyEnum.C);

EnumSet<MyEnum> es2 = EnumSet.of(MyEnum.a, MyEnum.b, MyEnum.c);

EnumSet<MyEnum> es3 = EnumSet.range(MyEnum.a, MyEnum.c);

if (es2.equals(es3)) {

System.out.println("e2 is same e3.");

}

for (MyEnum myEnum : MyEnum.values()) {

System.out.println(myEnum + " contains => " + es1.contains(myEnum));

}

}

}

**[8.9](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Convert Primitives to Wrapper Types**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.9)

int, short, char, long, double등 primitive와 이들의 Object Wrapper 인 Integer, Shrt, Char등 간의 converting에 있어 자동으로 처리해주는 boxing과 unboxing이 지원 됨에 따라 type에 대한 유연한 처리가 가능해졌다. 예제로 살펴보자.

package com.jeid.tiger;

public class AutoBoxingTester {

public static void main(String[] args) {

int i = 0;

Integer ii = i; // boxing. JDK 1.4에서는 incompatible type error가 발생 했었으나 Tiger에서는 괜찮다.

int j = ii; // unboxing

for (ii = 0; ii < 5; ii++) { // Integer인데도 ++ 연산자 지원.

}

i = 129;

ii = 129;

if (ii == i) {

System.out.println("i is same ii.");

}

// -128 ~ 127 사이의 수는 unboxing이 되어 == 연산이 허용되지만,

// 그 범위 외의 경우 Integer로 boxing된 상태므로 equals를 이용해야함.

// 이는 버그가 발생했을 경우 찾기 쉽지 않은 단점도 내포하고 있다.!!

checkIntegerSame(127, 127); // same

checkIntegerSame(128, 128); // Not same

checkIntegerEquals(128, 128); // equals

checkIntegerSame(-128, -128); // same

checkIntegerSame(-129, -129); // Not same

checkIntegerEquals(-129, -129); // equals

System.out.println("--------------------------------------------");

Boolean arriving = false;

Boolean late = true;

String ret = arriving ? (late ? "도착했지만 늦었네요." : "제시간에 잘 도착했군요.") :

(late ? "도착도 못하고 늦었군요." : "도착은 못했지만 늦진 않았군요.");

System.out.println(ret);

StringBuilder sb = new StringBuilder();

sb.append("appended String");

String str = "just String";

boolean mutable = true;

CharSequence chSeq = mutable ? sb : str;

System.out.println(chSeq);

}

private static void checkIntegerSame(Integer ii, Integer jj) {

if (ii == jj) {

System.out.println("ii = " + ii + ", jj = " + jj + " ==> jj is same ii.");

} else {

System.out.println("ii = " + ii + ", jj = " + jj + " ==> jj is NOT same ii!!");

}

}

private static void checkIntegerEquals(Integer ii, Integer jj) {

if (ii.equals(jj)) {

System.out.println("ii = " + ii + ", jj = " + jj + " ==> jj is equals ii.");

} else {

System.out.println("ii = " + ii + ", jj = " + jj + " ==> jj is NOT equals ii!!");

}

}

}

**[8.10](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Method Overload resolution in AutoBoxing**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.10)

int가 127을 초과할 경우 boxing이 이루어 질듯 하지만, method overload에 있어서는 boxing이 이루어 지지 않아 JDK1.4와 동일한 결과를 얻는다. 예제로 살펴보자.

package com.jeid.tiger;

public class OverloadTester {

public static void main(String[] args) {

double d = 10;

Integer ii = new Integer(10);

doSomething(10);

doSomething(1000);

doSomething(ii);

doSomething(d);

}

private static void doSomething(Integer ii) {

System.out.println("This is doSomething(Integer)");

}

private static void doSomething(double d) {

System.out.println("This is doSomething(double)");

}

}

**[8.11](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) 가변적인 argument 개수 ...**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.11)

인수가 가변적일 경우 인수의 개수가 없는것 부터 다수개까지 모두 지원. 예제로 살펴보자.

package com.jeid.tiger;

public class VarArgsTester {

public static void main(String[] args) {

setNumbers(1, 2);

setNumbers(1, 2, 3, 4);

setNumbers(1);

// setNumbers(); //해당 되는 method가 없어 compile error!!

System.out.println("==============================================");

setNumbers2(1, 2, 3, 4);

setNumbers2(1);

setNumbers2();

}

// this is same setNumbers(int first, int[] others)

private static void setNumbers(int first, int... others) {

System.out.println("-----------setNumbers()----------- : " + first);

for (int i : others) {

System.out.println("i = " + i);

}

}

// this is same setNumbers(int[] others)

private static void setNumbers2(int... others) {

System.out.println("-----------setNumbers2()----------- : "

+ (others != null && others.length > 0 ? others[0] : "null"));

for (int i : others) {

System.out.println("i = " + i);

}

}

}

**[8.12](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) The Three Standard Annotation**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.12)

@Override - sign the override from superclass.

//정상적인 사용

@Override

public int hashCode(){

return toString().hashCode();

}

//스펠링이 틀려 compile error!!

@Override

public int hasCode(){ //misspelled => method does not override a method from its superclass error!!

return toString().hashCode();

}

@Deprecated deprecated 주석과 동일하나 부모의 method가 deprecated되면 자식의 method를 사용해도 deprecated로 나온다.

package com.jeid.tiger;

public class AnnotationDeprecateTester {

public static void main(String[] args){

DeprecatedClass dep = new DeprecatedTester();

dep.doSomething(10); //deprecated

}

}

class DeprecatedClass {

@Deprecated

public void doSomething(int ii){ //deprecated

System.out.println("This is DeprecatedClass's doSomething(int)");

}

public void doSomethingElse(int ii){

System.out.println("This is DeprecatedClass's doSomethingElse(int)");

}

}

class DeprecatedTester extends DeprecatedClass {

@Override

public void doSomething(int ii){

System.out.println("This is DeprecatedTester's doSomething(int)");

}

}

@SuppressWarnings SuppressWarnings에 인자는 String[] type으로 여러개를 배열형태로 쓸수 있다.

package com.jeid.tiger;

import java.util.ArrayList;

import java.util.List;

public class AnnotationSuppressWarningsTester {

@SuppressWarnings({"unchecked", "fallthrough"} )

private static void test1(){

List list = new ArrayList();

list.add("aaaaaa");

}

@SuppressWarnings("unchecked")

private static void test2(){

List list = new ArrayList();

list.add("aaaaaa");

}

//warning이 없는 소스.

private static void test3(){

List<String> list = new ArrayList<String>();

list.add("aaaaaa");

}

}

**[8.13](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Creating Custom Annotation Types**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-8.13)

나만의 annotation을 정의할 수 있는데 키워드는 @interface이 각 method정의가 member라고 보면 된다. 간단한 예를 보면 다음과 같다.

package com.jeid.tiger;

import java.lang.annotation.Documented;

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Documented

@Target( { ElementType.TYPE, ElementType.FIELD, ElementType.METHOD, ElementType.ANNOTATION\_TYPE })

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation {

String columnName();

String methodName() default "";

}

//사용하는 쪽..

public class AnnotationTester {

@MyAnnotation(columnName = "test", methodName = "setTest")

private String test;

@MyAnnotation(columnName = "grpid")

public String grpid;

....

}

//위의 test 멤버의 경우 다음과 같이 접근 가능하다.

Field testField = cls.getDeclaredField("test");

if (testField.isAnnotationPresent(MyAnnotation.class)) {

Annotatioin anno = testField.getAnnotation(MyAnnotation.class);

System.out.println(anno.columnName() + ", method = " + anno.methodName());

}

**[9](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) The for/in Statement**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-9)

**[9.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) for/in 의 자주 사용되는 형태**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-9.1)

for/in은 무엇보다 다양한 유형의 예제를 보는것이 제일 빠를것이다. 형태별 사용 예제를 살펴보면 다음과 같다.

//1. 가장 단순한 형태인 배열(array)

String[] strs = { "aaa", "bbb", "ccc" };

for (String str : strs) {

System.out.println(str);

}

//2. List by using Iterator

List<Number> lstNum = new LinkedList<Number>();

lstNum.add(1);

lstNum.add(1.2);

for (Iterator<Number> iter = lstNum.iterator(); iter.hasNext();) {

Number num = iter.next();

System.out.println("num = " + num);

}

//3. List를 바로 사용

List<String> lst = new LinkedList<String>();

lst.add("aaaaa");

lst.add("bbbbb");

lst.add("ccccc");

lst.add("ddddd");

for (String str : lst) {

System.out.println("str = " + str);

}

// 4. List of List

List[] lists = { lst, lst };

for (List<String> l : lists) {

for (String str : l) {

System.out.println("str = " + str);

}

}

**[10](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) Static Import**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-10)

**[10.1](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost" \l "toc" \t "_blank) static member/method import**[**#**](http://dna.daum.net/technote/java/PrincipleOfJavaInternalForDeveloperEasyToLost#s-10.1)

Tiger에서는 다른 클래스의 member와 method를 import 할수 있다. 단, static 일 경우만 가능하다.

//예를 들어 System.out.println() 이라는 것을 사용하기 위해서는 다음의 import 문이 필요하다.

import java.lang.System; //물론 java.lang 이기에 import 문이 필요없지만 예를 들자면 그렇다는 것이다.&^^

//허나, Tiger에서는 다음과 같이 사용할수 있다.

import static java.lang.System.out;

...

out.println(...);

// method를 import 한다면..

import static java.lang.System.out.println;

...

println(...);

출처: <http://aesthetica7.tistory.com/entry/개발자가-놓치기-쉬운-자바의-개념-기본-원리> [WWW.오왕근.ORG]