이 문서의 저작권은 이덕하(본명, 주민 등록 번호 7\*0\*20-\*46\*5\*4, 메일 계정 [SF1856@hotmail.com](mailto:SF1856@hotmail.com))에게 있습니다. 이 문서를 배포할 수 있는 유일한 형식은 Microsoft Word 파일입니다. 문서를 배포할 때 이 문서의 파일명(BavaLang.doc)을 포함하여 어떤 부분도 삭제하거나 바꾸거나 어떤 것도 추가해서는 안됩니다(소스 코드의 글자 크기, 줄바꿈, 들여쓰기는 바꿀 수 있습니다). 저작권자의 서면 동의 없이 이 문서의 전체 또는 일부를 다른 어떤 매체로 옮기는 것을 금합니다. 특히 출판하기 위해 인쇄하거나 학교 또는 학원에서 강의 교재로 사용하기 위해 인쇄하는 것을 금합니다(빠르면 한 두 달 후면 출판될 것입니다. 프린트하지 마시고 기다려 주십시오). 이 문서로부터 인용을 할 때는 글쓴이 이덕하, 제목 자바 프로그래밍 언어 정복, 파일명 BavaLang.doc을 반드시 언급해야 합니다. 위의 조건이 지켜진다면 이 문서를 어디에 올리셔도 됩니다.

글쓴이의 말

저는 이덕하라고 합니다

이제 책이 완성되었습니다. 약간 수정할 것을 제외하면 이대로 출판할 생각입니다. 처음 계획했던 원대한 계획에는 조금 못 미치지만 그런대로 쓸만한 책이라고 자부합니다. 아직 출판사와 계약이 되진 않았는데 빠른시간 내에 출판되도록 노력하겠습니다.

이 책에 대해 궁금한 점이나 지적할 점이 있으면 제가 만든 이 책에 대한 카페인 <http://cafe.daum.net/BavaLang>에 들려주십시오.

**자바 프로그래밍 언어 정복**

최종본

2002-05-15

글쓴이 : 이 덕하

1. 차례

[1 차례 3](#_Toc9250762)

[2 머리말 24](#_Toc9250763)

[3 용어 25](#_Toc9250764)

[4 준비하기 69](#_Toc9250765)

[4.1 JDK와 에디터 69](#_Toc9250766)

[4.2 JavaTM 2 Platform, Standard Edition, v 1.x API Specification 69](#_Toc9250767)

[4.3 JavaTM 2 Platform, Standard Edition, v 1.x Source Code 69](#_Toc9250768)

[4.4 Hello World 69](#_Toc9250769)

[4.5 이 책에 쓰이는 관례 70](#_Toc9250770)

[4.5.1 소스 파일 이름 70](#_Toc9250771)

[4.5.2 예제의 결과 70](#_Toc9250772)

[4.5.3 식별자 이름 71](#_Toc9250773)

[4.5.4 들여쓰기(indentation) 71](#_Toc9250774)

[4.6 deprecated 71](#_Toc9250775)

[5 언어의 기본 요소 72](#_Toc9250776)

[5.1 유니코드 72](#_Toc9250777)

[5.1.1 유니 코드 버전 72](#_Toc9250778)

[5.1.2 유니코드 이스케이프(Unicode escape) 72](#_Toc9250779)

[5.1.3 주의할 점 73](#_Toc9250780)

[5.2 line terminators 74](#_Toc9250781)

[5.3 주석(comment) 74](#_Toc9250782)

[5.4 토큰(token) 76](#_Toc9250783)

[5.5 식별자(identifier) 78](#_Toc9250784)

[5.6 구분자(separator, punctuator) 79](#_Toc9250785)

[5.7 연산자(operator) 79](#_Toc9250786)

[5.8 키워드 80](#_Toc9250787)

[5.9 simple name과 qualified name 80](#_Toc9250788)

[6 제어문 81](#_Toc9250789)

[6.1 if 81](#_Toc9250790)

[6.1.1 if-then-else문의 기초 81](#_Toc9250791)

[6.1.2 if-then문의 기초 84](#_Toc9250792)

[6.1.3 if문의 조건식 84](#_Toc9250793)

[6.1.4 들여쓰기의 예외 85](#_Toc9250794)

[6.1.5 실수를 줄이는 법 86](#_Toc9250795)

[6.2 switch 91](#_Toc9250796)

[6.2.1 switch문의 기초 91](#_Toc9250797)

[6.2.2 조건식과 case 식의 제한 94](#_Toc9250798)

[6.2.3 실수를 줄이는 법 98](#_Toc9250799)

[6.3 while, do-while 101](#_Toc9250800)

[6.3.1 while문의 기초 102](#_Toc9250801)

[6.3.2 do-while문의 기초 104](#_Toc9250802)

[6.3.3 continue 105](#_Toc9250803)

[6.4 for 106](#_Toc9250804)

[6.4.1 for문의 기초 106](#_Toc9250805)

[6.4.2 for문의 유연성 107](#_Toc9250806)

[6.4.3 실수를 줄이는 법 111](#_Toc9250807)

[6.4.4 증감하는 숫자를 되도록이면 정수를 사용하라 – 좌표값 구하는 예제 113](#_Toc9250808)

[6.5 제어문의 중첩 116](#_Toc9250809)

[6.5.1 구구단 116](#_Toc9250810)

[6.5.2 팩토리얼 : 100! 구하기 117](#_Toc9250811)

[6.5.3 숫자 지그재그 120](#_Toc9250812)

[6.5.4 숫자 사선 122](#_Toc9250813)

[6.5.5 숫자 소용돌이 123](#_Toc9250814)

[6.5.6 레이블이 없는 break와 continue 125](#_Toc9250815)

[6.6 레이블 126](#_Toc9250816)

[6.6.1 레이블의 필요성 126](#_Toc9250817)

[6.6.2 레이블이 있는 break문의 기초 128](#_Toc9250818)

[6.6.3 레이블이 있는 continue문의 기초 130](#_Toc9250819)

[6.6.4 레이블과 이름 충돌 131](#_Toc9250820)

[6.7 도달할 수 없는 문장(unreachable statement) 132](#_Toc9250821)

[6.7.1 도달할 수 없는 문장이란? 132](#_Toc9250822)

[6.7.2 if 문 133](#_Toc9250823)

[6.7.3 while, do-while, for 134](#_Toc9250824)

[6.7.4 switch 문 136](#_Toc9250825)

[7 기본형 137](#_Toc9250826)

[7.1 기본형(primitive type)이란 137](#_Toc9250827)

[7.1.1 기본형과 참조형(reference type) 137](#_Toc9250828)

[7.1.2 자바 기본형의 특징 137](#_Toc9250829)

[7.2 정수형 137](#_Toc9250830)

[7.2.1 byte 137](#_Toc9250831)

[7.2.2 short 139](#_Toc9250832)

[7.2.3 int 139](#_Toc9250833)

[7.2.4 long 140](#_Toc9250834)

[7.2.5 정수형 리터럴 140](#_Toc9250835)

[7.2.6 char 141](#_Toc9250836)

[7.2.7 문자형 리터럴 141](#_Toc9250837)

[7.3 실수형 142](#_Toc9250838)

[7.3.1 float 142](#_Toc9250839)

[7.3.2 double 142](#_Toc9250840)

[7.3.3 실수형 리터럴(floating-point literal) 142](#_Toc9250841)

[7.4 논리형 144](#_Toc9250842)

[7.4.1 boolean 144](#_Toc9250843)

[7.5 기본형의 형변환 145](#_Toc9250844)

[7.5.1 넓히는 기본형 형변환(widening privitive conversion) 145](#_Toc9250845)

[7.5.2 좁히는 기본형 변환(narrowing privitive conversion)과 캐스팅(casting) 145](#_Toc9250846)

[7.5.3 형변환(conversion)의 예외 148](#_Toc9250847)

[7.6 포장 클래스(wrapper class) 149](#_Toc9250848)

[7.6.1 포장 클래스란 149](#_Toc9250849)

[7.6.2 Number 클래스 151](#_Toc9250850)

[7.6.3 정수형 151](#_Toc9250851)

[7.6.4 실수형 157](#_Toc9250852)

[7.6.5 논리형 157](#_Toc9250853)

[7.6.6 NumberFormatException 157](#_Toc9250854)

[7.6.7 포장 클래스의 쓰임 159](#_Toc9250855)

[7.7 FP-strict 159](#_Toc9250856)

[7.7.1 float value set, double value set 159](#_Toc9250857)

[7.7.2 float-extended-exponent value set, double-extended-exponent value set 159](#_Toc9250858)

[7.7.3 오버플로우(overflow)와 언더플로우(underflow) 160](#_Toc9250859)

[7.7.4 xxx-extended-exponent value set의 존재 이유 160](#_Toc9250860)

[7.7.5 strictfp 키워드 161](#_Toc9250861)

[7.7.6 strictfp의 존재 이유 161](#_Toc9250862)

[7.7.7 StrictMath 162](#_Toc9250863)

[7.7.8 strictfp와 상속 162](#_Toc9250864)

[7.8 기타 162](#_Toc9250865)

[7.8.1 Random 163](#_Toc9250866)

[7.8.2 Math 164](#_Toc9250867)

[7.8.3 BigInteger 165](#_Toc9250868)

[7.8.4 BigDecimal 166](#_Toc9250869)

[7.8.5 BitSet 167](#_Toc9250870)

[8 연산자 168](#_Toc9250871)

[8.1 사칙연산 168](#_Toc9250872)

[8.1.1 더하기 + 168](#_Toc9250873)

[8.1.2 빼기 – 172](#_Toc9250874)

[8.1.3 곱하기 \* 172](#_Toc9250875)

[8.1.4 나누기 / 173](#_Toc9250876)

[8.1.5 나머지 % 176](#_Toc9250877)

[8.1.6 단항(unary) +,- 178](#_Toc9250878)

[8.2 증감 연산자 179](#_Toc9250879)

[8.2.1 ++, -- 179](#_Toc9250880)

[8.3 논리 연산자 181](#_Toc9250881)

[8.3.1 ! 181](#_Toc9250882)

[8.3.2 &, | 182](#_Toc9250883)

[8.3.3 &&, || 183](#_Toc9250884)

[8.4 비트(bit)연산자 187](#_Toc9250885)

[8.4.1 ~,&, |, ^ 187](#_Toc9250886)

[8.5 비교 연산자 189](#_Toc9250887)

[8.5.1 ==, != 189](#_Toc9250888)

[8.5.2 <, >, <=, >= 190](#_Toc9250889)

[8.6 대입(assignment) 연산자 191](#_Toc9250890)

[8.6.1 = 191](#_Toc9250891)

[8.6.2 복합 대입(compound assignment) 연산자 +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=, >>>= 194](#_Toc9250892)

[8.7 쉬프트(shift) 연산자 195](#_Toc9250893)

[8.7.1 >>, >>>, << 195](#_Toc9250894)

[8.8 조건(conditional) 연산자 200](#_Toc9250895)

[8.8.1 ? : 200](#_Toc9250896)

[8.9 기타 201](#_Toc9250897)

[8.9.1 문자열 연견(string concatenation) 연산자 + 201](#_Toc9250898)

[8.9.2 instanceof 201](#_Toc9250899)

[8.10 연산자의 우선순위 201](#_Toc9250900)

[8.10.1 연산자의 우선 순위 201](#_Toc9250901)

[8.10.2 괄호 202](#_Toc9250902)

[8.11 promotion과 캐스팅(casting) 203](#_Toc9250903)

[8.11.1 promotion 203](#_Toc9250904)

[8.11.2 binary promotion 203](#_Toc9250905)

[8.11.3 unary promotion 204](#_Toc9250906)

[8.11.4 캐스팅(casting) 207](#_Toc9250907)

[8.11.5 캐스팅에서 주의할 점 209](#_Toc9250908)

[9 배열의 기초 210](#_Toc9250909)

[9.1 기본형의 배열 210](#_Toc9250910)

[9.1.1 기본형의 1차원 배열 210](#_Toc9250911)

[9.1.2 기본형의 다차원 배열 211](#_Toc9250912)

[9.2 참조형의 배열 213](#_Toc9250913)

[9.2.1 클래스의 배열 213](#_Toc9250914)

[9.2.2 수퍼 클래스의 배열 214](#_Toc9250915)

[9.2.3 인터페이스의 배열 216](#_Toc9250916)

[9.3 배열의 초기화 217](#_Toc9250917)

[9.3.1 기본형 배열의 디폴트 초기값 217](#_Toc9250918)

[9.3.2 참조형 배열의 디폴트 초기값 218](#_Toc9250919)

[9.3.3 배열의 초기화 218](#_Toc9250920)

[9.4 final 배열 220](#_Toc9250921)

[9.5 크기가 0인 배열 221](#_Toc9250922)

[9.6 배열에서의 예외상황 222](#_Toc9250923)

[9.6.1 NullPointerException 222](#_Toc9250924)

[9.6.2 ArrayIndexOutOfBoundsException과 배열의 크기 223](#_Toc9250925)

[10 배열의 해부 225](#_Toc9250926)

[10.1 배열은 객체다 225](#_Toc9250927)

[10.1.1 자바 배열은 객체다 225](#_Toc9250928)

[10.1.2 배열의 멤버 226](#_Toc9250929)

[10.1.3 배열 클래스의 이름 226](#_Toc9250930)

[10.2 배열의 구조 228](#_Toc9250931)

[10.2.1 배열 원소(element of the array)와 배열 요소 (component of the array) 228](#_Toc9250932)

[10.2.2 배열의 배열 229](#_Toc9250933)

[10.2.3 들쭉날쭉한 배열 233](#_Toc9250934)

[10.2.4 배열의 배열과 다차원 배열의 장단점 234](#_Toc9250935)

[10.2.5 배열의 복제 235](#_Toc9250936)

[10.3 배열의 형변환(Casting) 243](#_Toc9250937)

[10.3.1 기본형 배열의 형변환 243](#_Toc9250938)

[10.3.2 참조형 배열의 형변환 244](#_Toc9250939)

[10.3.3 ArrayStoreException 244](#_Toc9250940)

[10.4 배열에 관련된 클래스들 245](#_Toc9250941)

[10.4.1 System 클래스의 배열 복사 245](#_Toc9250942)

[10.4.2 Arrays 클래스 248](#_Toc9250943)

[10.4.3 Array 클래스와 리플렉션(reflection mechanism) 260](#_Toc9250944)

[10.4.4 범용 깊은 복제 클래스 만들기 264](#_Toc9250945)

[11 클래스의 기초 273](#_Toc9250946)

[11.1 클래스와 객체 273](#_Toc9250947)

[11.1.1 클래스(class)와 객체(object) 273](#_Toc9250948)

[11.1.2 객체(인스턴스instance)의 생성과 사용 273](#_Toc9250949)

[11.1.3 클래스의 정의 274](#_Toc9250950)

[11.2 메쏘드 275](#_Toc9250951)

[11.2.1 메쏘드(method)의 정의 275](#_Toc9250952)

[11.2.2 값에 의한 호출(call by value, copy by value, pass by value) 276](#_Toc9250953)

[11.2.3 final 매개변수(final parameter) 278](#_Toc9250954)

[11.2.4 return문과 반환값 279](#_Toc9250955)

[11.2.5 메쏘드의 중복정의 (overloading) 285](#_Toc9250956)

[11.3 필드 288](#_Toc9250957)

[11.3.1 필드(field)와 지역 변수(local variable) 288](#_Toc9250958)

[11.3.2 필드의 초기화와 디폴트(default) 초기값 289](#_Toc9250959)

[11.3.3 final필드 292](#_Toc9250960)

[11.4 생성자 293](#_Toc9250961)

[11.4.1 생성자(constructor) 정의와 사용 293](#_Toc9250962)

[11.4.2 디폴트 생성자(default consturctor)와 no-argument 생성자(no-arg constructor) 295](#_Toc9250963)

[11.4.3 초기화는 생성자에서 하자. 295](#_Toc9250964)

[11.5 this 297](#_Toc9250965)

[11.5.1 this의 필요성 297](#_Toc9250966)

[11.5.2 메쏘드 또는 생성자에서의 this 298](#_Toc9250967)

[11.5.3 생성자를 부르는 this 300](#_Toc9250968)

[11.6 정적(static)과 비정적(non-static) 302](#_Toc9250969)

[11.6.1 정적(static) 필드와 인스턴스(instance) 필드 302](#_Toc9250970)

[11.6.2 인스턴스 메쏘드 305](#_Toc9250971)

[11.6.3 정적 메쏘드(static method) 309](#_Toc9250972)

[11.7 public과 private 311](#_Toc9250973)

[11.7.1 인터페이스와 구현(interface and implimentation) 311](#_Toc9250974)

[11.7.2 필드는 private으로, 메쏘드는 public으로 312](#_Toc9250975)

[11.7.3 데이터를 감추어야 하는 이유 315](#_Toc9250976)

[11.7.4 필드를 public으로 해야 할 때가 있다. 322](#_Toc9250977)

[11.7.5 메쏘드도 private으로 해야 할 때가 있다. 322](#_Toc9250978)

[11.7.6 접근 변경자는 왜 있어야만 하나? 322](#_Toc9250979)

[11.8 클래스의 멤버 323](#_Toc9250980)

[12 클래스의 상속 324](#_Toc9250981)

[12.1 클래스 상속하기 324](#_Toc9250982)

[12.1.1 수퍼 클래스(superclass)와 서브 클래스(subclass) 324](#_Toc9250983)

[12.1.2 단일 상속(single inheritance) 324](#_Toc9250984)

[12.1.3 final 클래스 325](#_Toc9250985)

[12.2 멤버의 상속 326](#_Toc9250986)

[12.2.1 필드와 메쏘드의 상속 326](#_Toc9250987)

[12.2.2 public 멤버 327](#_Toc9250988)

[12.2.3 private 멤버 327](#_Toc9250989)

[12.2.4 protected 멤버 328](#_Toc9250990)

[12.2.5 생성자는 상속되지 않는다 330](#_Toc9250991)

[12.2.6 생성자를 부르는 super 331](#_Toc9250992)

[12.3 메쏘드의 재정의(overriding) 339](#_Toc9250993)

[12.3.1 기초 339](#_Toc9250994)

[12.3.2 동적 다형성(dynamic polymorphism) 341](#_Toc9250995)

[12.3.3 메쏘드를 부르는 super 345](#_Toc9250996)

[12.3.4 필드는 재정의되지 않는다 345](#_Toc9250997)

[12.3.5 정적 메쏘드는 재정의되지 않는다. 347](#_Toc9250998)

[12.3.6 final 메쏘드 348](#_Toc9250999)

[12.3.7 수퍼 클래스에서 서브 클래스의 메쏘드 부르기 348](#_Toc9251000)

[12.4 추상 메쏘드와 추상 클래스 354](#_Toc9251001)

[12.4.1 추상 메쏘드(abstract method) 354](#_Toc9251002)

[12.4.2 추상 클래스(abstract class) 356](#_Toc9251003)

[12.4.3 추상 클래스를 남용하지 말자 358](#_Toc9251004)

[12.4.4 abstract 키워드의 제한 358](#_Toc9251005)

[12.5 클래스의 contract와 구현 359](#_Toc9251006)

[12.5.1 contract와 구현 359](#_Toc9251007)

[12.5.2 contract와 인터페이스 359](#_Toc9251008)

[12.5.3 사용자(user)에 대한 contract 361](#_Toc9251009)

[12.5.4 구현자(implementer)에 대한 contract 362](#_Toc9251010)

[12.5.5 contract의 상속과 implementation의 상속 362](#_Toc9251011)

[12.6 상속에 대한 조언 362](#_Toc9251012)

[12.6.1 어떤 경우에 상속을 사용할까? 362](#_Toc9251013)

[12.6.2 상속보다는 composition을 이용하라 364](#_Toc9251014)

[12.6.3 가능하면 멤버를 계층(hierarchy)의 위쪽으로 365](#_Toc9251015)

[12.6.4 고수는 protected와 private을 구별해서 사용한다 366](#_Toc9251016)

[13 인터페이스 371](#_Toc9251017)

[13.1 인터페이스란? 371](#_Toc9251018)

[13.1.1 클래스가 할 수 있는 것과 인터페이스가 할 수 있는 것 371](#_Toc9251019)

[13.1.2 추상 클래스 – 인터페이스로 가는 길 372](#_Toc9251020)

[13.2 인터페이스의 구현 373](#_Toc9251021)

[13.2.1 구현하기 374](#_Toc9251022)

[13.2.2 다중 상속(multiple inheritance) 374](#_Toc9251023)

[13.2.3 추상 클래스 – 인터페이스의 미완성 구현 378](#_Toc9251024)

[13.3 인터페이스의 사용 379](#_Toc9251025)

[13.3.1 인터페이스의 구현 379](#_Toc9251026)

[13.3.2 컬렉션 클래스(collection classes)에서의 예 382](#_Toc9251027)

[13.3.3 상수 모음 386](#_Toc9251028)

[13.3.4 marker 인터페이스 387](#_Toc9251029)

[13.3.5 콜백(call back)과 인터페이스 388](#_Toc9251030)

[13.4 인터페이스의 멤버 393](#_Toc9251031)

[13.4.1 인터페이스의 필드 393](#_Toc9251032)

[13.4.2 인터페이스의 메쏘드 - public abstract 394](#_Toc9251033)

[13.4.3 멤버 클래스와 멤버 인터페이스 395](#_Toc9251034)

[14 패키지 396](#_Toc9251035)

[14.1 패키지의 필요성 396](#_Toc9251036)

[14.1.1 이름 충돌 396](#_Toc9251037)

[14.1.2 grouping 396](#_Toc9251038)

[14.2 package 사용하기 396](#_Toc9251039)

[14.2.1 package문 396](#_Toc9251040)

[14.2.2 package와 디렉토리 396](#_Toc9251041)

[14.2.3 컴파일 단위(compilation unit) 397](#_Toc9251042)

[14.2.4 디폴트 패키지(unnamed package) 397](#_Toc9251043)

[14.2.5 패키지와 컴파일 단위의 저장 398](#_Toc9251044)

[14.2.6 유일한 이름 붙이기 398](#_Toc9251045)

[14.3 import 사용하기 398](#_Toc9251046)

[14.3.1 import문의 의미 398](#_Toc9251047)

[14.3.2 single-type-import 400](#_Toc9251048)

[14.3.3 type-import-on-demand 400](#_Toc9251049)

[14.3.4 import의 충돌 402](#_Toc9251050)

[14.4 패키지의 의미 403](#_Toc9251051)

[14.4.1 패키지의 멤버 403](#_Toc9251052)

[14.4.2 패키지와 접근 403](#_Toc9251053)

[14.4.3 패키지 계층의 특권 – 없다 404](#_Toc9251054)

[14.4.4 package의 인터페이스와 구현 405](#_Toc9251055)

[15 중첩 타입 406](#_Toc9251056)

[15.1 중첩 타입이란? 406](#_Toc9251057)

[15.1.1 최상위 타입(top level tpye) 과 중첩 타입(nested type) 406](#_Toc9251058)

[15.1.2 중첩 타입의 종류 406](#_Toc9251059)

[15.1.3 용어에 대해 407](#_Toc9251060)

[15.1.4 중첩 타입의 접근 지정자(access modifier) 407](#_Toc9251061)

[15.1.5 중첩 타입의 유용성 408](#_Toc9251062)

[15.1.6 중첩 타입은 멤버이다 408](#_Toc9251063)

[15.2 인터페이스 안에서의 중첩 타입 409](#_Toc9251064)

[15.2.1 중첩 인터페이스(nested interface) 409](#_Toc9251065)

[15.2.2 중첩 클래스(nested class) 409](#_Toc9251066)

[15.3 클래스 안에서의 중첩 타입 410](#_Toc9251067)

[15.3.1 중첩 인터페이스(nested interface) 410](#_Toc9251068)

[15.3.2 정적 중첩 클래스(static nested class) 410](#_Toc9251069)

[15.3.3 내부 클래스(inner class) 410](#_Toc9251070)

[15.4 내부 클래스 411](#_Toc9251071)

[15.4.1 이너 클래스(inner class)와 외부 클래스(outer class, enclosing class)의 관계 411](#_Toc9251072)

[15.4.2 내부 클래스의 구현 414](#_Toc9251073)

[15.4.3 내부 클래스의 문제점 – 보안의 구멍과 복잡성 419](#_Toc9251074)

[15.4.4 지역 클래스(local class) 420](#_Toc9251075)

[15.4.5 익명 내부 클래스(anonymous inner class) 423](#_Toc9251076)

[15.5 중첩 타입 사용에 대한 조언 425](#_Toc9251077)

[15.5.1 익명 내부 클래스를 남용하지 말라 425](#_Toc9251078)

[15.5.2 중첩을 많이 하지 말라 425](#_Toc9251079)

[16 클래스와 인터페이스 – 못다한 이야기 426](#_Toc9251080)

[16.1 main 메쏘드 426](#_Toc9251081)

[16.1.1 entry point 426](#_Toc9251082)

[16.1.2 public static void main(String[] args) 426](#_Toc9251083)

[16.2 상속 427](#_Toc9251084)

[16.2.1 상속의 순환(cyclic inheritance) 427](#_Toc9251085)

[16.2.2 여러 갈래로 상속받는 인터페이스 428](#_Toc9251086)

[16.3 필드 428](#_Toc9251087)

[16.3.1 지정자 428](#_Toc9251088)

[16.3.2 필드 초기화문 428](#_Toc9251089)

[16.3.3 정적 초기화 블록(static initializer, static initialization block) 429](#_Toc9251090)

[16.3.4 인스턴스 초기화 블록(instance initializer, instance initialization block) 432](#_Toc9251091)

[16.3.5 final필드를 사용해야 하는 경우 436](#_Toc9251092)

[16.3.6 blank final 438](#_Toc9251093)

[16.4 필드의 초기화 순서 440](#_Toc9251094)

[16.4.1 필드의 초기화에서는 코드상의 순서가 중요할 수 있다 440](#_Toc9251095)

[16.4.2 정적 필드의 교차 초기화 442](#_Toc9251096)

[16.4.3 정적 필드와 인스턴스 필드 445](#_Toc9251097)

[16.4.4 중첩 클래스의 초기화 순서 445](#_Toc9251098)

[16.5 이름 충돌 447](#_Toc9251099)

[16.5.1 이름 충돌의 종류 447](#_Toc9251100)

[16.5.2 메쏘드의 서명(signature) 충돌 448](#_Toc9251101)

[16.5.3 다중 상속된 필드의 충돌 449](#_Toc9251102)

[16.5.4 다중 상속된 중첩 타입 451](#_Toc9251103)

[16.5.5 금지된 이름 충돌 452](#_Toc9251104)

[16.5.6 지역 변수와 매개변수 452](#_Toc9251105)

[16.5.7 허용된 이름 충돌 453](#_Toc9251106)

[16.5.8 필드를 가리는(shadow) 지역변수 453](#_Toc9251107)

[16.5.9 지역 변수를 가리는 필드 454](#_Toc9251108)

[16.5.10 필드를 가리는 중첩타입의 필드 457](#_Toc9251109)

[16.5.11 필드의 감추기(hiding) 458](#_Toc9251110)

[16.5.12 중첩 타입의 이름 충돌 462](#_Toc9251111)

[16.5.13 외부 클래스의 감춰진(hidden) 필드에 접근하기 464](#_Toc9251112)

[16.5.14 감추기(hiding)와 재정의(overriding)의 차이 465](#_Toc9251113)

[16.5.15 멤버 타입 감추기 466](#_Toc9251114)

[16.5.16 정적 메쏘드의 감추기 468](#_Toc9251115)

[16.5.17 정적 메쏘드 감추기의 제한 471](#_Toc9251116)

[16.5.18 감추기는 왜 허용되었나 471](#_Toc9251117)

[16.6 메쏘드 472](#_Toc9251118)

[16.6.1 header와 몸체(body) 472](#_Toc9251119)

[16.6.2 지정자 472](#_Toc9251120)

[16.6.3 메쏘드의 매개변수의 개수는 가변일 수 없다 473](#_Toc9251121)

[16.6.4 메쏘드는 최대 하나만 반환(return)할 수 있다 474](#_Toc9251122)

[16.6.5 final 메쏘드 476](#_Toc9251123)

[16.6.6 final 메쏘드와 performance 477](#_Toc9251124)

[16.6.7 final 매개변수(parameter) 478](#_Toc9251125)

[16.6.8 중복정의(overloading)와 상속 479](#_Toc9251126)

[16.6.9 중복정의와 서명 – 다른 제한(반환형, static 여부, throws절)이 없다 480](#_Toc9251127)

[16.6.10 중복정의된 메쏘드의 선택 481](#_Toc9251128)

[16.6.11 재정의의 제한 486](#_Toc9251129)

[16.6.12 재정의와 접근 제한 487](#_Toc9251130)

[16.6.13 추상 메쏘드가 추상 메쏘드를 재정의할 수 있다 490](#_Toc9251131)

[16.6.14 접근할 수 없는 조상의 메쏘드 490](#_Toc9251132)

[16.6.15 private 메쏘드와 디폴트 접근 메쏘드 493](#_Toc9251133)

[16.6.16 정적 메쏘드를 부르는 이상한 방법(super 키워드) 497](#_Toc9251134)

[16.6.17 구식 선언 방식 498](#_Toc9251135)

[16.7 접근(access) 499](#_Toc9251136)

[16.7.1 타입의 접근과 멤버의 접근 500](#_Toc9251137)

[16.7.2 디폴트 생성자의 접근 500](#_Toc9251138)

[16.7.3 private은 per-object가 아니라 per-class로 작동한다 502](#_Toc9251139)

[16.7.4 protected의 진정한 의미는? 503](#_Toc9251140)

[16.7.5 디폴트 접근 클래스를 다른 패키지에서 사용하기 504](#_Toc9251141)

[16.8 인터페이스 506](#_Toc9251142)

[16.8.1 여러 인터페이스에서 상속된 메쏘드 506](#_Toc9251143)

[16.8.2 같은 서명(signature)의 메쏘드가 이미 있는 경우 507](#_Toc9251144)

[16.9 객체의 생성 508](#_Toc9251145)

[16.9.1 생성자의 변경자 제한 508](#_Toc9251146)

[16.9.2 객체가 생성되는 방법 508](#_Toc9251147)

[16.9.3 생성자를 부르는 this와 super에서의 제한 512](#_Toc9251148)

[16.9.4 복제 생성자(copy constructor) 514](#_Toc9251149)

[16.9.5 생성자를 부르는 this의 순환 514](#_Toc9251150)

[16.9.6 객체 생성에서의 무한 루프 516](#_Toc9251151)

[16.9.7 객체가 생성될 때의 초기화 순서 516](#_Toc9251152)

[16.9.8 생성자에서 부르는 메쏘드 519](#_Toc9251153)

[16.9.9 new 문과 쓰레기 수거(garbage collection) 520](#_Toc9251154)

[16.10 중첩 타입 520](#_Toc9251155)

[16.10.1 simple name과 qualified name 520](#_Toc9251156)

[16.10.2 내부 클래스의 정적 멤버 522](#_Toc9251157)

[16.10.3 blank final의 초기화 524](#_Toc9251158)

[16.10.4 정적 멤버 클래스 524](#_Toc9251159)

[16.10.5 지역 클래스에서 지역 변수의 접근하기 525](#_Toc9251160)

[16.10.6 static context에 선언된 내부 클래스 529](#_Toc9251161)

[16.10.7 중첩타임의 상속 533](#_Toc9251162)

[16.10.8 상속되진 않지만 접근할 수는 있는 경우 533](#_Toc9251163)

[16.10.9 자신의 외부 클래스를 상속받는 중첩 클래스 536](#_Toc9251164)

[16.10.10 중첩 클래스와 신뢰 관계 537](#_Toc9251165)

[16.10.11 내부 클래스와 상속 541](#_Toc9251166)

[16.10.12 중첩 타입의 중첩 547](#_Toc9251167)

[17 변수 550](#_Toc9251168)

[17.1 변수(variable)의 종류 550](#_Toc9251169)

[17.1.1 필드(field) 550](#_Toc9251170)

[17.1.2 배열 요소 (array component) 550](#_Toc9251171)

[17.1.3 매개변수 (parameter) 550](#_Toc9251172)

[17.1.4 지역변수 550](#_Toc9251173)

[17.2 변수의 선언과 초기화 551](#_Toc9251174)

[17.2.1 변수의 선언 551](#_Toc9251175)

[17.2.2 변수의 초기화 552](#_Toc9251176)

[17.2.3 final 553](#_Toc9251177)

[17.2.4 blank final 554](#_Toc9251178)

[17.2.5 definite assignment 554](#_Toc9251179)

[17.2.6 definite unassignment 555](#_Toc9251180)

[17.3 타입(type) 555](#_Toc9251181)

[17.3.1 타입 555](#_Toc9251182)

[17.3.2 기본형(primitive type) 556](#_Toc9251183)

[17.3.3 참조형(reference type) 557](#_Toc9251184)

[17.3.4 컴파일 타임 타입(compile time type)과 런 타임 타입(run time type) 557](#_Toc9251185)

[17.4 객체(object)와 참조(reference) 558](#_Toc9251186)

[17.4.1 객체와 참조 558](#_Toc9251187)

[17.5 참조형의 형변환 560](#_Toc9251188)

[17.5.1 넓히는 참조형 형변환(widening reference conversion) 560](#_Toc9251189)

[17.5.2 좁히는 참조형 형변환(narrowing reference conversion)과 캐스팅(casting) 561](#_Toc9251190)

[17.5.3 넓히는 형변환과 캐스팅 570](#_Toc9251191)

[17.5.4 캐스팅에서 주의할 점 571](#_Toc9251192)

[17.5.5 금지된 형변환 574](#_Toc9251193)

[17.6 Assignment Compatibility 574](#_Toc9251194)

[17.6.1 대입 호환성(assignment compatibility)이란 574](#_Toc9251195)

[17.6.2 대입 호환성이 적용되는 경우 576](#_Toc9251196)

[17.7 영역(scope) 579](#_Toc9251197)

[17.7.1 영역 580](#_Toc9251198)

[17.7.2 금지된 이름 충돌 581](#_Toc9251199)

[17.7.3 가리기(shawdowing) 582](#_Toc9251200)

[17.7.4 지역 변수 선언에 대한 조언 583](#_Toc9251201)

[18 Object 클래스 588](#_Toc9251202)

[18.1 Object클래스에 대하여 588](#_Toc9251203)

[18.1.1 Object 클래스의 소스 코드 588](#_Toc9251204)

[18.1.2 Object 클래스는 모든 클래스의 조상이다. 589](#_Toc9251205)

[18.2 equals()와 ==의 차이 592](#_Toc9251206)

[18.2.1 참조형에서 ==의 의미 592](#_Toc9251207)

[18.2.2 equals()와 동등성(equality) 595](#_Toc9251208)

[18.3 hashCode() 601](#_Toc9251209)

[18.3.1 hashCode 메쏘드는 어디에 쓰이나 601](#_Toc9251210)

[18.3.2 hashCode와 equals의 관계 602](#_Toc9251211)

[18.3.3 hashCode() 재정의하기 603](#_Toc9251212)

[18.4 toString() 606](#_Toc9251213)

[18.4.1 toString 메쏘드 606](#_Toc9251214)

[18.4.2 Object.toString() 607](#_Toc9251215)

[18.4.3 toString 메쏘드 재정의하기 608](#_Toc9251216)

[18.5 clone()와 복제 608](#_Toc9251217)

[18.5.1 복제(clone)이란? 608](#_Toc9251218)

[18.5.2 객체를 복제하는 여러가지 방법 609](#_Toc9251219)

[18.5.3 Object.clone()을 재정의해보자. 613](#_Toc9251220)

[18.5.4 Cloneable 인터페이스 618](#_Toc9251221)

[18.5.5 Object.clone()이 하는 일 – 얕은 복제(shallow copy) 619](#_Toc9251222)

[18.5.6 깊은 복제(deep copy) 622](#_Toc9251223)

[18.5.7 clone()과 blank final 624](#_Toc9251224)

[18.5.8 상속과 복제 625](#_Toc9251225)

[18.6 기타 626](#_Toc9251226)

[18.6.1 finalize() 626](#_Toc9251227)

[18.6.2 getClass() 626](#_Toc9251228)

[18.6.3 notify(), notifyAll() 626](#_Toc9251229)

[18.6.4 wait(), wait(long), wait(long, int) 627](#_Toc9251230)

[19 예외 상황 628](#_Toc9251231)

[19.1 예외 상황(exception condition)과 예외 상황 메커니즘(exception mechanism) 628](#_Toc9251232)

[19.1.1 예외 상황이란? 628](#_Toc9251233)

[19.1.2 이전의 처리 방식 628](#_Toc9251234)

[19.1.3 예외 상황 메커니즘(exception mechanism) 629](#_Toc9251235)

[19.2 예외 상황 처리하기 629](#_Toc9251236)

[19.2.1 try-catch 문 629](#_Toc9251237)

[19.2.2 try-catch-finally문 633](#_Toc9251238)

[19.2.3 처리되지 않은 예외 상황 633](#_Toc9251239)

[19.3 예외 상황 발생시키기 635](#_Toc9251240)

[19.3.1 왜 예외 상황을 발생시켜야 하는가? 635](#_Toc9251241)

[19.3.2 throw와 throws 636](#_Toc9251242)

[19.4 예외 상황 클래스 638](#_Toc9251243)

[19.4.1 Throwable 클래스 638](#_Toc9251244)

[19.4.2 검사되지 않는 예외 상황(unchecked exception) 638](#_Toc9251245)

[19.4.3 검사되는 예외 상황(checked exception) 638](#_Toc9251246)

[19.4.4 컴파일러는 왜 검사되지 않는 예외 상황에 관대한가? 639](#_Toc9251247)

[19.4.5 사용자 정의형 예외 상황 클래스 640](#_Toc9251248)

[19.5 예외 상황 메커니즘의 올바른 사용법 640](#_Toc9251249)

[19.5.1 예외 상황이 발생하면 적절한 처리를 해 주어야 한다. 640](#_Toc9251250)

[19.5.2 예외 상황 클래스의 수퍼 클래스의 사용을 자제해야 한다. 641](#_Toc9251251)

[19.5.3 누가 처리할지를 잘 판단해야 한다. 642](#_Toc9251252)

[19.6 try-catch-finally문의 해부 642](#_Toc9251253)

[19.6.1 try 642](#_Toc9251254)

[19.6.2 무엇을 catch할 수 있고 무엇을 catch 해야 하나? 643](#_Toc9251255)

[19.6.3 catch의 순서 645](#_Toc9251256)

[19.6.4 catch 블럭 또는 finally 블럭에서 예외 상황이 발생하면? 646](#_Toc9251257)

[19.6.5 finally와 break, return, continue 649](#_Toc9251258)

[19.6.6 try 문의 중첩 650](#_Toc9251259)

[19.7 throw와 throws의 해부 651](#_Toc9251260)

[19.7.1 throw 651](#_Toc9251261)

[19.7.2 throw와 return 652](#_Toc9251262)

[19.7.3 throws 653](#_Toc9251263)

[19.8 상속과 예외 상황 653](#_Toc9251264)

[19.8.1 클래스의 상속과 예외 상황 654](#_Toc9251265)

[19.8.2 생성자와 예외 상황 655](#_Toc9251266)

[19.8.3 인터페이스의 구현 658](#_Toc9251267)

[19.8.4 두개의 인터페이스에서 똑같은 메쏘드를 구현하는 경우 658](#_Toc9251268)

[19.9 필드 초기화와 예외 상황 660](#_Toc9251269)

[19.9.1 정적 필드의 초기화와 예외상황 660](#_Toc9251270)

[19.9.2 인스턴스 필드의 초기화와 예외상황 661](#_Toc9251271)

[19.9.3 익명 클래스의 필드의 초기화와 예외상황 662](#_Toc9251272)

[20 문자열 664](#_Toc9251273)

[20.1 문자열 리터럴(string literal) 664](#_Toc9251274)

[20.2 문자열 연결 연산자 + 664](#_Toc9251275)

[20.2.1 숫자를 문자열(string)로 변환하는 방법 664](#_Toc9251276)

[20.2.2 + 연산자 666](#_Toc9251277)

[20.2.3 + 연산자의 연산 순서 667](#_Toc9251278)

[20.2.4 객체와 + 연산자 668](#_Toc9251279)

[20.3 String 클래스 669](#_Toc9251280)

[20.3.1 생성자 669](#_Toc9251281)

[20.3.2 String 객체는 변할 수 없다(immutable) 672](#_Toc9251282)

[20.3.3 ==와 equals 673](#_Toc9251283)

[20.3.4 intern 메쏘드 677](#_Toc9251284)

[20.4 StringBuffer 클래스 681](#_Toc9251285)

[20.4.1 StringBuffer 객체는 변할 수 있다. 682](#_Toc9251286)

[21 리플렉션 683](#_Toc9251287)

[21.1 리플렉션이란 683](#_Toc9251288)

[21.1.1 리플렉션의 의미 683](#_Toc9251289)

[21.1.2 리플렉션 클래스들 683](#_Toc9251290)

[21.1.3 리플렉션은 어디에 사용되나 684](#_Toc9251291)

[21.2 instanceof 연산자 684](#_Toc9251292)

[21.2.1 instanceof 연산자란? 684](#_Toc9251293)

[21.2.2 instanceof와 null 684](#_Toc9251294)

[21.2.3 컴파일타임 오류 686](#_Toc9251295)

[21.2.4 instanceof와 상속 687](#_Toc9251296)

[21.2.5 instanceof의 그럴듯한 사용 예 689](#_Toc9251297)

[21.3 class 리터럴 691](#_Toc9251298)

[21.4 Class 클래스 692](#_Toc9251299)

[21.4.1 Class 객체 얻기 692](#_Toc9251300)

[21.4.2 타입에 대해 알아보기 694](#_Toc9251301)

[21.4.3 수퍼 타입 얻기 698](#_Toc9251302)

[21.4.4 멤버 타입 얻기 698](#_Toc9251303)

[21.5 Modifier 클래스 700](#_Toc9251304)

[21.6 Package 클래스 701](#_Toc9251305)

[21.7 Field 클래스 702](#_Toc9251306)

[21.7.1 필드 이름 알아내기 702](#_Toc9251307)

[21.7.2 필드의 값 703](#_Toc9251308)

[21.8 Method 클래스 704](#_Toc9251309)

[21.8.1 메쏘드의 이름 알아내기 705](#_Toc9251310)

[21.8.2 메쏘드 실행하기 705](#_Toc9251311)

[21.8.3 InvocationTargetException 707](#_Toc9251312)

[21.9 Constructor 클래스 708](#_Toc9251313)

[21.9.1 객체 생성하기 708](#_Toc9251314)

[21.9.2 중첩 클래스의 객체 생성하기 710](#_Toc9251315)

[21.10 Array 클래스 713](#_Toc9251316)

[21.11 리플렉션과 접근 제한 713](#_Toc9251317)

[21.11.1 IllegalAccessException 713](#_Toc9251318)

[21.11.2 AccessibleObject와 접근 제한의 우회 714](#_Toc9251319)

[21.12 Proxy 클래스 715](#_Toc9251320)

[21.12.1 Proxy 에 대해 715](#_Toc9251321)

[21.12.2 Proxy 클래스 715](#_Toc9251322)

[21.12.3 InvocationHandler 인터페이스 716](#_Toc9251323)

[21.13 연습 프로그램 719](#_Toc9251324)

[21.13.1 간단한 디컴파일러 719](#_Toc9251325)

[22 쓰레드(thread) 727](#_Toc9251326)

[22.1 쓰레드란 727](#_Toc9251327)

[22.1.1 프로세스(process)와 쓰레드 727](#_Toc9251328)

[22.2 쓰레드 만들기 727](#_Toc9251329)

[22.2.1 Thread 클래스 727](#_Toc9251330)

[22.2.2 Runnable 인터페이스 730](#_Toc9251331)

[22.3 Daemon 쓰레드 734](#_Toc9251332)

[22.3.1 daemon 쓰레드 734](#_Toc9251333)

[22.4 sleep, yield, jion 736](#_Toc9251334)

[22.4.1 seep 736](#_Toc9251335)

[22.4.2 yield 737](#_Toc9251336)

[22.4.3 join 737](#_Toc9251337)

[22.5 ThreadGroup 클래스 739](#_Toc9251338)

[22.5.1 현재 돌고 있는 쓰레드들 알아내기 739](#_Toc9251339)

[22.6 priority 740](#_Toc9251340)

[22.6.1 priority란 740](#_Toc9251341)

[22.6.2 setPriority, getPriority 740](#_Toc9251342)

[22.7 경합상태 743](#_Toc9251343)

[22.7.1 경합상태(race condition)이란 743](#_Toc9251344)

[22.7.2 경합상태(race condition)의 예 743](#_Toc9251345)

[22.8 synchronized와 lock 748](#_Toc9251346)

[22.8.1 lock 748](#_Toc9251347)

[22.8.2 synchronized 문 748](#_Toc9251348)

[22.8.3 synchronized 메쏘드 751](#_Toc9251349)

[22.8.4 this가 아닌 다른 객체를 잠그는 synchronized 문 757](#_Toc9251350)

[22.8.5 잠기는 것은 객체이지 클래스가 아니다. 760](#_Toc9251351)

[22.8.6 클래스를 잠그기 761](#_Toc9251352)

[22.8.7 synchronized의 ‘중첩’ 765](#_Toc9251353)

[22.9 교착 상태 765](#_Toc9251354)

[22.9.1 교착 상태(dead lock)란 766](#_Toc9251355)

[22.9.2 교착 상태(dead lock)의 예 766](#_Toc9251356)

[22.9.3 교착 상태(dead lock)를 어떻게 막을 것인가 768](#_Toc9251357)

[22.10 쓰레드와 UI 768](#_Toc9251358)

[22.10.1 반응하지 않는 UI 768](#_Toc9251359)

[22.10.2 반응하는 UI 777](#_Toc9251360)

[22.11 wait, notify, notifyAll 780](#_Toc9251361)

[22.11.1 Object 클래스 780](#_Toc9251362)

[22.11.2 wait와 synchronized 780](#_Toc9251363)

[22.11.3 notify, notifyAll 784](#_Toc9251364)

[22.11.4 wait의 필요성 784](#_Toc9251365)

[22.11.5 wait – notify의 사용 789](#_Toc9251366)

[22.12 deprecated – stop, suspend, resume 792](#_Toc9251367)

[22.12.1 deprecated 792](#_Toc9251368)

[22.12.2 stop 792](#_Toc9251369)

[22.12.3 stop이 deprecated 된 이유는 793](#_Toc9251370)

[22.12.4 stop의 대체 – 변수 사용 797](#_Toc9251371)

[22.12.5 stop의 대체 – interrupt 메쏘드 799](#_Toc9251372)

[22.12.6 suspend, resume 801](#_Toc9251373)

[22.12.7 suspend, resume이 deprecated 된 이유는 802](#_Toc9251374)

[23 스트림 803](#_Toc9251375)

[23.1 스트림(stream)이란 803](#_Toc9251376)

[23.1.1 스트림이란 803](#_Toc9251377)

[23.1.2 스트림의 종류 803](#_Toc9251378)

[23.1.3 열기와 닫기 803](#_Toc9251379)

[23.1.4 차단(block) 804](#_Toc9251380)

[23.2 InputStream 클래스 805](#_Toc9251381)

[23.2.1 InputStream의 서브클래스 805](#_Toc9251382)

[23.2.2 InputStream의 소스 805](#_Toc9251383)

[23.2.3 InputStream.read() 808](#_Toc9251384)

[23.2.4 InputStream.read(byte[] b, int off, int len), InputStream.read(byte[] b) 811](#_Toc9251385)

[23.2.5 InputStream.close() 812](#_Toc9251386)

[23.2.6 InputStream.skip(long n) 812](#_Toc9251387)

[23.2.7 InputStream.available() 812](#_Toc9251388)

[23.2.8 InputStream.mark(int readlimit), InputStream.reset(), InputStream.markSupported() 812](#_Toc9251389)

[23.3 OutputStream 클래스 813](#_Toc9251390)

[23.3.1 OutputStream의 서브 클래스 813](#_Toc9251391)

[23.3.2 OutputStream의 소스 코드 813](#_Toc9251392)

[23.3.3 OutputStream.write(int b) 814](#_Toc9251393)

[23.3.4 OutputStream.write(byte[] b, int off, int length), OutputStream.write(byte[] b) 815](#_Toc9251394)

[23.3.5 OutputStream.close() 816](#_Toc9251395)

[23.3.6 OutputStream.flush() 816](#_Toc9251396)

[23.4 Reader 클래스 816](#_Toc9251397)

[23.4.1 Reader의 서브 클래스 816](#_Toc9251398)

[23.4.2 Reader의 소스 코드 816](#_Toc9251399)

[23.4.3 Reader.read(char[] cbuf, int off, int len), Reader.read(char[] cbuf), Reader.read() 818](#_Toc9251400)

[23.4.4 Reader.close() 819](#_Toc9251401)

[23.4.5 Reader.skip(long n) 819](#_Toc9251402)

[23.4.6 Reader.ready() 819](#_Toc9251403)

[23.4.7 Reader.mark(int readAheadLimit), Reader.reset(), Reader.markSupported() 819](#_Toc9251404)

[23.5 Writer 클래스 819](#_Toc9251405)

[23.5.1 Writer의 서브 클래스 819](#_Toc9251406)

[23.5.2 Writer의 소스 코드 820](#_Toc9251407)

[23.5.3 Writer.write(char[] cbuf, int off, int len), Writer.write(char[] cbuf), Writer.write(int c) 821](#_Toc9251408)

[23.5.4 Writer.write(String str, int off, int len), Writer.write(String str) 821](#_Toc9251409)

[23.5.5 Writer.close() 822](#_Toc9251410)

[23.5.6 Writer.flush() 822](#_Toc9251411)

[23.6 필터(filter) 822](#_Toc9251412)

[23.6.1 필터란 822](#_Toc9251413)

[23.6.2 FilterInputStream 822](#_Toc9251414)

[23.6.3 FilterOutputStream 828](#_Toc9251415)

[23.6.4 FilterReader 829](#_Toc9251416)

[23.6.5 FilterWriter 830](#_Toc9251417)

[23.6.6 여러 개의 필터의 연결 830](#_Toc9251418)

[23.6.7 FilterXXX를 상속하지 않는 필터 스트림 834](#_Toc9251419)

[23.7 버퍼(buffer) 835](#_Toc9251420)

[23.7.1 버퍼란 835](#_Toc9251421)

[23.7.2 버퍼의 사용 835](#_Toc9251422)

[23.7.3 flush 836](#_Toc9251423)

[23.8 파이프(pipe) 838](#_Toc9251424)

[23.8.1 파이프란 838](#_Toc9251425)

[23.8.2 파이프의 연결 838](#_Toc9251426)

[23.8.3 파이프의 사용 838](#_Toc9251427)

[23.8.4 blocking 841](#_Toc9251428)

[23.9 스트림과 쓰레드 842](#_Toc9251429)

[23.9.1 byte 스트림 842](#_Toc9251430)

[23.9.2 character 스트림 849](#_Toc9251431)

[24 직렬화 853](#_Toc9251432)

[24.1 직렬화(serialization)란 853](#_Toc9251433)

[24.1.1 직렬화와 desirialization 853](#_Toc9251434)

[24.1.2 직렬화의 쓰임 854](#_Toc9251435)

[24.2 Default Serialization 854](#_Toc9251436)

[24.2.1 ObjectInputStream.readObject(), ObjectOutputStream.writeObject(Object obj) 854](#_Toc9251437)

[24.2.2 Serializable 인터페이스 856](#_Toc9251438)

[24.2.3 object graph(web of objects) 858](#_Toc9251439)

[24.2.4 transient 필드와 static 필드 862](#_Toc9251440)

[24.2.5 직렬화와 생성자 865](#_Toc9251441)

[24.2.6 상속과 직렬화 867](#_Toc9251442)

[24.3 Custom Serialization 870](#_Toc9251443)

[24.3.1 custom serialization의 필요성 870](#_Toc9251444)

[24.3.2 readObject(ObjectInputStream in), writeObject(ObjectOutputStream out) 871](#_Toc9251445)

[24.3.3 Externalizable 인터페이스 872](#_Toc9251446)

[24.3.4 readObject, writeObject 대 readExternal, writeExternal 873](#_Toc9251447)

[24.4 Serialization을 이용한 깊은 복제 877](#_Toc9251448)

[24.4.1 깊은 복제 877](#_Toc9251449)

[25 메모리 관리와 쓰레기 수거 882](#_Toc9251450)

[25.1 쓰레기 수거(Garbage Collection)란 882](#_Toc9251451)

[25.1.1 쓰레기 수거란 882](#_Toc9251452)

[25.1.2 쓰레기 수거의 장단점 882](#_Toc9251453)

[25.1.3 쓰레기(garbage)란 무엇인가 883](#_Toc9251454)

[25.1.4 cycle(circular reference) 884](#_Toc9251455)

[25.1.5 잘 안보이는 참조 885](#_Toc9251456)

[25.1.6 garbage collection 알고리즘 886](#_Toc9251457)

[25.2 Object.finalize() 887](#_Toc9251458)

[25.2.1 쓰레기 수거기 와 finalize() 887](#_Toc9251459)

[25.2.2 finalize() 재정의하기 887](#_Toc9251460)

[25.2.3 cleanup 888](#_Toc9251461)

[25.2.4 finalize()를 부르는 Thread 890](#_Toc9251462)

[25.2.5 finalize()의 예외 상황은 무시된다. 892](#_Toc9251463)

[25.2.6 부활(ressurection) 892](#_Toc9251464)

[25.3 Runtime, System 클래스와 메모리 관리 893](#_Toc9251465)

[25.3.1 System.gc(), System.runFinalization() 893](#_Toc9251466)

[25.3.2 Runtime.freeMemory(), Runtime.totalMemory() 894](#_Toc9251467)

[25.4 쓰레드와 Garbage Collection 894](#_Toc9251468)

[25.4.1 쓰레드와 Garbage Collection 895](#_Toc9251469)

[25.4.2 Thread 객체는 언제부터 쓰레기 취급을 받나 896](#_Toc9251470)

[25.5 스택(stack)과 힙(heap) 898](#_Toc9251471)

[25.5.1 스택에 만들어지는 것들 898](#_Toc9251472)

[25.5.2 힙에 만들어지는 것들 900](#_Toc9251473)

[25.5.3 메모리는 어떻게 할당되나 901](#_Toc9251474)

[25.5.4 메모리 할당은 가변적일 수 있다 902](#_Toc9251475)

[26 Assertion(단언) 906](#_Toc9251476)

[26.1 assert의 문법 906](#_Toc9251477)

[26.1.1 assert 키워드 906](#_Toc9251478)

[26.1.2 AssertionError 클래스 906](#_Toc9251479)

[26.2 assert와 관련된 설정 906](#_Toc9251480)

[26.2.1 버전 문제 906](#_Toc9251481)

[26.2.2 호환성과 컴파일 옵션 906](#_Toc9251482)

[26.2.3 실행할 때의 assert의 on/off 907](#_Toc9251483)

[26.2.4 클래스 파일에서 assert를 아예 없애버리기 907](#_Toc9251484)

[26.2.5 assert를 무조건 실행되게 하기 909](#_Toc9251485)

[26.3 assertion 의 사용에 대한 조언 909](#_Toc9251486)

[26.3.1 assert를 사용해야 할 곳 910](#_Toc9251487)

[26.3.2 assert를 사용해서는 안되는 곳 912](#_Toc9251488)

1. 머리말

이 책은 자바 프로그래밍 언어에 대한 책이다. 오직 자바의 언어적 측면만을 다루었다.

이 책은 두가지를 목표로 해서 쓰여졌다. 첫째는 아주 복잡한 프로그래밍도 할 수 있을 정도로 자바 언어를 충분히 깊이 있게 다루는 것이다. 둘째는 최대한 많은 사람들이 볼 수 있도록 최대한 쉽게 쓰는 것이다. 그리고 두 가지 목표는 상당 부분 이루어졌다고 생각한다.

그러나 문제점도 있는데 첫째는 너무 쉽게 쓰려고 노력하다보니 중급자 이상에게는 장황한 예제 때문에 짜증이 날 수도 있겠다. 둘째는 그 많은 예제의 질이 그리 높지 않다는 것이다. 그저 언어의 한 측면을 설명하기 위한 예제들로 받아들여주기 바란다. 셋째는 난이도가 점진적으로 높아지지 않고 상당 부분 오락가락하는데 초보자로서는 한 번 읽어서는 소화하기 힘들 것이다. 넷째는 곳곳에 나오는 유머가 너무 썰렁하다는 것이다.

하여튼 이 책에 여러분들이 자바 프로그래밍 언어를 공부하고 더 나아가 훌륭한 프로그래머가 되는데 도움이 되었으면 한다.

이 책은 기본적으로

<The Java Programming Language> Third Edition by Ken Arnold, James Gosling, David Holmes  
<The Java Language Specification> Second Edition by Gilad Bracha, James Gosling, Bill Joy, Guy Steele

JavaTM 2 Platform, Standard Edition, v 1.4.0 API Specification

을 바탕으로 쓰여졌다. 이 책에 쓰이는 용어도 위의 두 책에 따랐다.

그리고 다음의 책 또는 문서등이 도움이 되었다.

<Core Java 2 : Volume 1 Fundamentals> Fifth Edition by Cay S. Horstmann, Gary Cornell

<Core Java 2, Volume 2: Advanced Features> Fourth Edition by Cay S. Horstmann, Gary Cornell

<Thinking in Java> Second Edition by Bruce Eckel

<http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html> 의 Assertion Facility

<http://java.sun.com/j2se/1.4/docs/guide/misc/threadPrimitiveDeprecation.html> 의 Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?

<http://www.starcraft.co.kr>

<http://www.thecomedylab.com>

1. 용어

abrupt completion

이 책에 쓰이는 번역어 : 비정상적 종료

다른 번역어 : 갑작스런 종료

다른 용어 참조 : normal completion

이 책의 내용 참조 :

abstract class

이 책에 쓰이는 번역어 : 추상 클래스

다른 번역어 :

다른 용어 참조 : concrete class, abstract method, interface

이 책의 내용 참조 :

abstract method

이 책에 쓰이는 번역어 : 추상 메쏘드

다른 번역어 :

다른 용어 참조 : concrete method, abstract class

이 책의 내용 참조 :

access

이 책에 쓰이는 번역어 : 접근

다른 번역어 : 접근 속성

다른 용어 참조 : public, protected, private, default access, access control, access modifier

이 책의 내용 참조 :

access control

이 책에 쓰이는 번역어 : 접근 제어

다른 번역어 : 접근 속성 제어

다른 용어 참조 :

이 책의 내용 참조 :

access modifier

이 책에 쓰이는 번역어 : 접근 지정자

다른 번역어 : 접근 제한자, 접근 변경자, 접근 수정자, 접근 지시자

다른 용어 참조 : public, protected, private, default access, package access

이 책의 내용 참조 :

accessibility

이 책에 쓰이는 번역어 : 접근성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

algorithm

이 책에 쓰이는 번역어 : 알고리즘

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

ambiguity

이 책에 쓰이는 번역어 : 모호함

다른 번역어 : 모호성

다른 용어 참조 : name

이 책의 내용 참조 :

anonymous inner class

이 책에 쓰이는 번역어 : 익명 내부 클래스, 익명 클래스(anonymouse class)

다른 번역어 : 익명 안긴 클래스, 무명 안긴 클래스

다른 용어 참조 : named class, local class, inner class, nested type

이 책의 내용 참조 :

argument

이 책에 쓰이는 번역어 : 인자

다른 번역어 : 실질 매개 변수(actual parameter), 실인수, 실인자, 아규먼트, 인수

다른 용어 참조 : parameter, method

이 책의 내용 참조 :

array

이 책에 쓰이는 번역어 : 배열

다른 번역어 :

다른 용어 참조 : index, component, element

이 책의 내용 참조 :

ASCII

이 책에 쓰이는 번역어 : 아스키

다른 번역어 :

다른 용어 참조 : Unicode

이 책의 내용 참조 :

assertion

이 책에 쓰이는 번역어 : 단언

다른 번역어 : 어서션

다른 용어 참조 :

이 책의 내용 참조 :

assignment

이 책에 쓰이는 번역어 : 대입

다른 번역어 : 할당, 지정

다른 용어 참조 : compound assignment

이 책의 내용 참조 :

assignment compatibility

이 책에 쓰이는 번역어 : 대입 호환성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

associativity

이 책에 쓰이는 번역어 : 결합 법칙

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

asynchronous exception

이 책에 쓰이는 번역어 : 비동기 예외

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

backslash(\)

이 책에 쓰이는 번역어 : 백 슬래시

다른 번역어 : 역 슬래시

다른 용어 참조 :

이 책의 내용 참조 :

backspace

이 책에 쓰이는 번역어 : 백 스페이스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

base

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

binary compatibility

이 책에 쓰이는 번역어 : 이진 호환성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

binary operator

이 책에 쓰이는 번역어 : 이항 연산자

다른 번역어 :

다른 용어 참조 : unary operator, ternary operator, operator

이 책의 내용 참조 :

blank final

이 책에 쓰이는 번역어 : blank final

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

block

이 책에 쓰이는 번역어 : 블록

다른 번역어 : 구역

다른 용어 참조 : statement, method, local variable

이 책의 내용 참조 :

block

이 책에 쓰이는 번역어 : 막히다, 막힘

다른 번역어 : 차단, 차단하다

다른 용어 참조 : buffer, stream, thread

이 책의 내용 참조 :

body

이 책에 쓰이는 번역어 : 몸체

다른 번역어 : 본체, 본문

다른 용어 참조 :

이 책의 내용 참조 :

boolean

이 책에 쓰이는 번역어 : 논리형

다른 번역어 : 불형

다른 용어 참조 :

이 책의 내용 참조 :

buffer

이 책에 쓰이는 번역어 : 버퍼

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

bytecode

이 책에 쓰이는 번역어 : 바이트코드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

call

이 책에 쓰이는 번역어 : 호출, 부르다

다른 번역어 : 콜

다른 용어 참조 : method, invocation

이 책의 내용 참조 :

callback

이 책에 쓰이는 번역어 : 콜백

다른 번역어 :

다른 용어 참조 : interface, method

이 책의 내용 참조 :

call by value

이 책에 쓰이는 번역어 : 값에 의한 호출

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

carriage return(CR)

이 책에 쓰이는 번역어 :

다른 번역어 : 복귀

다른 용어 참조 :

이 책의 내용 참조 :

casting

이 책에 쓰이는 번역어 : 캐스팅

다른 번역어 : 형식 변환, 캐스트, 명시적 형변환(explicit conversion)

다른 용어 참조 :

이 책의 내용 참조 :

catch

이 책에 쓰이는 번역어 : 잡기, 잡다, catch

다른 번역어 : 잡아내기

다른 용어 참조 :

이 책의 내용 참조 :

character

이 책에 쓰이는 번역어 : 문자

다른 번역어 :

다른 용어 참조 : string

이 책의 내용 참조 :

check

이 책에 쓰이는 번역어 :

다른 번역어 : 체크, 검사

다른 용어 참조 :

이 책의 내용 참조 :

checked exception

이 책에 쓰이는 번역어 : 검사되는 예외

다른 번역어 : 검사된 예외

다른 용어 참조 :

이 책의 내용 참조 :

circular declaration

이 책에 쓰이는 번역어 : 순환 선언

다른 번역어 : 원형 선언

다른 용어 참조 :

이 책의 내용 참조 :

class

이 책에 쓰이는 번역어 : 클래스

다른 번역어 :

다른 용어 참조 : interface, member, package

이 책의 내용 참조 :

class literal

이 책에 쓰이는 번역어 : 클래스 리터럴

다른 번역어 :

다른 용어 참조 : constant, reflection

이 책의 내용 참조 :

class loader

이 책에 쓰이는 번역어 : 클래스 로더

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

clone

이 책에 쓰이는 번역어 : 복제

다른 번역어 : 클론

다른 용어 참조 : deep cloning, shallow cloning, copy constructor

이 책의 내용 참조 :

code

이 책에 쓰이는 번역어 : 코드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

collection (classes)

이 책에 쓰이는 번역어 : 컬렉션

다른 번역어 : 모음, 수집, 집합

다른 용어 참조 :

이 책의 내용 참조 :

command line argument

이 책에 쓰이는 번역어 : 명령행 인자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

comment

이 책에 쓰이는 번역어 : 주석

다른 번역어 : 주석문

다른 용어 참조 :

이 책의 내용 참조 :

comparison operator

이 책에 쓰이는 번역어 : 비교 연산자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

compatibility

이 책에 쓰이는 번역어 : 호환성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

compile

이 책에 쓰이는 번역어 : 컴파일

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

compile-time constant

이 책에 쓰이는 번역어 : 컴파일타임 상수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

compile-time error

이 책에 쓰이는 번역어 : 컴파일타임 오류

다른 번역어 : 컴파일 오류, 컴파일 에러, 컴파일타임 에러

다른 용어 참조 : run-time error

이 책의 내용 참조 :

completion

이 책에 쓰이는 번역어 : 종료

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

component of the array

이 책에 쓰이는 번역어 : 배열 요소

다른 번역어 : 배열 컴포넌트, 배열 소자, 배열의 구성요소

다른 용어 참조 : element of the array

이 책의 내용 참조 :

주의 : 많은 사람들이 element of the array를 배열 요소라고 번역한다.

compound assignment operator

이 책에 쓰이는 번역어 : 복합 대입 연산자

다른 번역어 : 연산후 대입 연산자, 복합 지정 연산자, 복합 할당 연산자

다른 용어 참조 : assignment, operator

이 책의 내용 참조 :

concatenation

이 책에 쓰이는 번역어 : (문자열) 연결

다른 번역어 : (문자열) 병합

다른 용어 참조 : string

이 책의 내용 참조 :

concrete class

이 책에 쓰이는 번역어 : 구체 클래스

다른 번역어 : 실체 클래스, 구체적 클래스

다른 용어 참조 : abstract class

이 책의 내용 참조 :

concrete method

이 책에 쓰이는 번역어 : 구체 메쏘드

다른 번역어 : 구체적 메쏘드, 실체 메쏘드

다른 용어 참조 : abstract method, concrete class

이 책의 내용 참조 :

concurrency

이 책에 쓰이는 번역어 : 동시

다른 번역어 : 병행

다른 용어 참조 : thread

이 책의 내용 참조 :

constant

이 책에 쓰이는 번역어 : 상수

다른 번역어 :

다른 용어 참조 : variable, final

이 책의 내용 참조 :

constructor

이 책에 쓰이는 번역어 : 생성자

다른 번역어 : 구성자

다른 용어 참조 : class, method

이 책의 내용 참조 :

context

이 책에 쓰이는 번역어 : 컨텍스트

다른 번역어 : 문맥

다른 용어 참조 :

이 책의 내용 참조 :

contract

이 책에 쓰이는 번역어 : contract

다른 번역어 : 계약, 협정

다른 용어 참조 :

이 책의 내용 참조 :

control flow

이 책에 쓰이는 번역어 : 제어 흐름

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

conversion

이 책에 쓰이는 번역어 : 형변환

다른 번역어 : 변환

다른 용어 참조 :

이 책의 내용 참조 :

copy constructor

이 책에 쓰이는 번역어 : 복제 생성자

다른 번역어 : 복사 생성자

다른 용어 참조 :

이 책의 내용 참조 :

creation

이 책에 쓰이는 번역어 : 생성

다른 번역어 : 만들기

다른 용어 참조 :

이 책의 내용 참조 :

daemon

이 책에 쓰이는 번역어 : 데몬

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

data

이 책에 쓰이는 번역어 : 데이터

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

deadlock

이 책에 쓰이는 번역어 : 교착 상태

다른 번역어 : 교착, 데드락

다른 용어 참조 :

이 책의 내용 참조 :

decimal

이 책에 쓰이는 번역어 : 십지수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

declaration

이 책에 쓰이는 번역어 : 선언

다른 번역어 : 선언문

다른 용어 참조 :

이 책의 내용 참조 :

deep cloning

이 책에 쓰이는 번역어 : 깊은 복제

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

default

이 책에 쓰이는 번역어 : 디폴트

다른 번역어 : 기본

다른 용어 참조 :

이 책의 내용 참조 :

default access

이 책에 쓰이는 번역어 : 디폴트 접근

다른 번역어 : 패키지 보호(package access), 기본 접근 제한

다른 용어 참조 : package access

이 책의 내용 참조 :

definite assignment

이 책에 쓰이는 번역어 : definite assignment

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

definite unassignment

이 책에 쓰이는 번역어 : definite unassignment

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

deprecated

이 책에 쓰이는 번역어 : deprecated

다른 번역어 : 배척해야 하는, 폐기 준비, 사용금지(된)

다른 용어 참조 :

이 책의 내용 참조 :

deserialization

이 책에 쓰이는 번역어 : 역직렬화

다른 번역어 :

다른 용어 참조 : serialization, stream, transient

이 책의 내용 참조 :

diamond inheritance

이 책에 쓰이는 번역어 : 다이아몬드 상속

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

digit

이 책에 쓰이는 번역어 : 숫자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

directory

이 책에 쓰이는 번역어 : 디렉토리

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

double quote

이 책에 쓰이는 번역어 : 큰 따옴표

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

downcast

이 책에 쓰이는 번역어 : 다운 캐스팅

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

dynamic binding

이 책에 쓰이는 번역어 : 동적 바인딩

다른 번역어 : 동적 결합, 동적 설정

다른 용어 참조 :

이 책의 내용 참조 :

element of the array

이 책에 쓰이는 번역어 : 배열 원소

다른 번역어 : 배열 요소, 배열 소자

다른 용어 참조 : component of the array

이 책의 내용 참조 :

주의 : 많은 사람들이 element of the array를 배열 요소라고 번역한다. 사실 component of the array와 element of the array를 구분하는 사람이 별로 없다. 둘다 그냥 배열 요소라고 하는 것이다.

empty array

이 책에 쓰이는 번역어 : 빈 배열

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

empty statement

이 책에 쓰이는 번역어 : 빈 문장

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

enclosing class

이 책에 쓰이는 번역어 : 외부 클래스

다른 번역어 : 외곽 클래스, 감싸는 클래스

다른 용어 참조 :

이 책의 내용 참조 :

enclosing object

이 책에 쓰이는 번역어 : 외부 객체

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

encoding

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

equality

이 책에 쓰이는 번역어 : 같음, equality

다른 번역어 : 동등

다른 용어 참조 : equivalence, identity

이 책의 내용 참조 :

equivalence

이 책에 쓰이는 번역어 : 동등, equivalence

다른 번역어 : 등가, 등치

다른 용어 참조 :

이 책의 내용 참조 :

error

이 책에 쓰이는 번역어 : 오류

다른 번역어 : 에러

다른 용어 참조 : exception, compile-time error, run-time error

이 책의 내용 참조 :

escape sequence

이 책에 쓰이는 번역어 : 이스케이프 시퀀스

다른 번역어 : 이스케이프 문자열, 탈출 문자열

다른 용어 참조 :

이 책의 내용 참조 :

evaluation

이 책에 쓰이는 번역어 : 평가

다른 번역어 : 계산

다른 용어 참조 :

이 책의 내용 참조 :

evaluation order

이 책에 쓰이는 번역어 : 평가 순서

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

event

이 책에 쓰이는 번역어 : 이벤트

다른 번역어 : 사건

다른 용어 참조 :

이 책의 내용 참조 :

exception

이 책에 쓰이는 번역어 : 예외, 예외 상황(exception condition)

다른 번역어 :

다른 용어 참조 : throw, try, catch, error

이 책의 내용 참조 :

exception-handler

이 책에 쓰이는 번역어 : 예외 처리기

다른 번역어 : 예외 처리기

다른 용어 참조 :

이 책의 내용 참조 :

excution

이 책에 쓰이는 번역어 : 실행

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

expression

이 책에 쓰이는 번역어 : 식

다른 번역어 : 표현, 표현식

다른 용어 참조 :

이 책의 내용 참조 :

extend

이 책에 쓰이는 번역어 : 상속하다

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

field

이 책에 쓰이는 번역어 : 필드

다른 번역어 : 멤버 변수(member variable)

다른 용어 참조 : member, method

이 책의 내용 참조 :

file

이 책에 쓰이는 번역어 : 파일

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

filter

이 책에 쓰이는 번역어 : 필터

다른 번역어 :

다른 용어 참조 : pipe, stream

이 책의 내용 참조 :

final

이 책에 쓰이는 번역어 : final

다른 번역어 : 종단

다른 용어 참조 :

이 책의 내용 참조 :

finalization

이 책에 쓰이는 번역어 : 종결, finalization

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

floating-point

이 책에 쓰이는 번역어 : 부동 소수점

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

flush

이 책에 쓰이는 번역어 : flush

다른 번역어 :

다른 용어 참조 : buffer, stream

이 책의 내용 참조 :

form feed

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

forward reference

이 책에 쓰이는 번역어 : 전방 참조

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

FP-strict

이 책에 쓰이는 번역어 : FP-strict

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

garbage collection

이 책에 쓰이는 번역어 : 쓰레기 수거

다른 번역어 : 가비지 콜렉션, 이삭 수거, 쓰레기 정리, 가비지 콜렉팅, 쓰레기 수집

다른 용어 참조 : finalization, heap

이 책의 내용 참조 :

heap

이 책에 쓰이는 번역어 : 힙

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

hexadecimal

이 책에 쓰이는 번역어 : 16진수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

hiding

이 책에 쓰이는 번역어 : 감추기

다른 번역어 : 은폐, 은닉

다른 용어 참조 :

이 책의 내용 참조 :

hierarchy

이 책에 쓰이는 번역어 : 계층

다른 번역어 : 계층형, 계서

다른 용어 참조 :

이 책의 내용 참조 :

horizontal tap

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

identifier

이 책에 쓰이는 번역어 : 식별자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

identity

이 책에 쓰이는 번역어 : 같음, identity

다른 번역어 : 독자성, 동치, 항등

다른 용어 참조 :

이 책의 내용 참조 :

immutable

이 책에 쓰이는 번역어 :

다른 번역어 : 불변적

다른 용어 참조 :

이 책의 내용 참조 :

implementation

이 책에 쓰이는 번역어 : 구현

다른 번역어 :

다른 용어 참조 : interface

이 책의 내용 참조 :

indentation

이 책에 쓰이는 번역어 : 들여쓰기

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

index(배열에서)

이 책에 쓰이는 번역어 : 인덱스

다른 번역어 : 첨자, 색인

다른 용어 참조 :

이 책의 내용 참조 :

infinity

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

inheritance

이 책에 쓰이는 번역어 : 상속

다른 번역어 : 계승, 물림, 전승, 상속성

다른 용어 참조 : superclass, subclass, superinterface, subinterface, supertype, subtype

이 책의 내용 참조 :

initialization

이 책에 쓰이는 번역어 : 초기화

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

initializer

이 책에 쓰이는 번역어 : 초기화 블록

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

initializer for class variable

이 책에 쓰이는 번역어 : 클래스 변수 초기화문

다른 번역어 : 스태틱 필드 초기화문(intializer for static field), static variable initializer

다른 용어 참조 : static intializer

이 책의 내용 참조 :

initializer for instance variable

이 책에 쓰이는 번역어 : 인스턴스 변수 초기화문

다른 번역어 : 인스턴스 필드 초기화문(initializer for instance field), instance variable initializer

다른 용어 참조 : instance initializer

이 책의 내용 참조 :

inner class

이 책에 쓰이는 번역어 : 내부 클래스

다른 번역어 : 안긴 클래스, 이너 클래스

다른 용어 참조 : nested class, nested type, enclosing class, outer class

이 책의 내용 참조 :

instance

이 책에 쓰이는 번역어 : 인스턴스

다른 번역어 :

다른 용어 참조 : class, creation

이 책의 내용 참조 :

instance initializer

이 책에 쓰이는 번역어 : 인스턴스 초기화 블록

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

instantiation

이 책에 쓰이는 번역어 : 인스턴스 생성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

integer

이 책에 쓰이는 번역어 : 정수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

integral type

이 책에 쓰이는 번역어 : 정수형

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

interface

이 책에 쓰이는 번역어 : 인터페이스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

internationalization

이 책에 쓰이는 번역어 : 국제화

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

introspection

이 책에 쓰이는 번역어 : introspection

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

invocation

이 책에 쓰이는 번역어 : 호출

다른 번역어 : 인보크

다른 용어 참조 :

이 책의 내용 참조 :

iteration

이 책에 쓰이는 번역어 : 반복

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

JavaBeans

이 책에 쓰이는 번역어 : 자바 빈즈

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

keyword

이 책에 쓰이는 번역어 : 키워드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

label

이 책에 쓰이는 번역어 : 레이블

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

labeled statement

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

left-hand side

이 책에 쓰이는 번역어 : 좌변

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

line terminator

이 책에 쓰이는 번역어 : 개행 문자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

linefeed(LF)

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

linking

이 책에 쓰이는 번역어 : 링크

다른 번역어 : 연결

다른 용어 참조 :

이 책의 내용 참조 :

literal

이 책에 쓰이는 번역어 : 리터럴

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

loading

이 책에 쓰이는 번역어 : 로딩

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

local class

이 책에 쓰이는 번역어 : 지역 클래스

다른 번역어 : 로컬 클래스, 국소적 클래스

다른 용어 참조 :

이 책의 내용 참조 :

local inner class

이 책에 쓰이는 번역어 : 지역 내부 클래스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

local variable

이 책에 쓰이는 번역어 : 지역 변수

다른 번역어 : 로컬 변수

다른 용어 참조 :

이 책의 내용 참조 :

lock

이 책에 쓰이는 번역어 : 잠금, 잠그다

다른 번역어 : 락, 자물쇠

다른 용어 참조 :

이 책의 내용 참조 :

loop

이 책에 쓰이는 번역어 : 루프

다른 번역어 : 순환, 반복

다른 용어 참조 :

이 책의 내용 참조 :

lvalue

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

marker interface

이 책에 쓰이는 번역어 : marker interface

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

member

이 책에 쓰이는 번역어 : 멤버

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

member class

이 책에 쓰이는 번역어 : 멤버 클래스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

member interface

이 책에 쓰이는 번역어 : 멤버 인터페이스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

member type

이 책에 쓰이는 번역어 : 멤버 타입

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

memory

이 책에 쓰이는 번역어 : 메모리

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

method

이 책에 쓰이는 번역어 : 메쏘드

다른 번역어 : 함수(function), 멤버 함수(member function)

다른 용어 참조 :

이 책의 내용 참조 :

modifier

이 책에 쓰이는 번역어 : 지정자

다른 번역어 : 수정자, 제한자, 수직자, 변경자

다른 용어 참조 :

이 책의 내용 참조 :

monitor

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

multiple inheritance

이 책에 쓰이는 번역어 : 다중 상속

다른 번역어 : 다중 계승

다른 용어 참조 :

이 책의 내용 참조 :

multithreading

이 책에 쓰이는 번역어 : 다중 쓰레딩

다른 번역어 : 멀티 쓰레딩

다른 용어 참조 :

이 책의 내용 참조 :

name

이 책에 쓰이는 번역어 : 이름

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

named class

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

named type

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

narrowing conversion

이 책에 쓰이는 번역어 : 좁히는 형변환

다른 번역어 : 협소화 형변환, 축소 변환

다른 용어 참조 :

이 책의 내용 참조 :

narrowing primitive conversion

이 책에 쓰이는 번역어 : 좁히는 기본형 형변환

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

narrowing reference conversion

이 책에 쓰이는 번역어 : 좁히는 참조형 형변환

다른 번역어 : 협소화 참조 형변환

다른 용어 참조 :

이 책의 내용 참조 :

native code

이 책에 쓰이는 번역어 : 네이티브 코드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

native method

이 책에 쓰이는 번역어 : 네이티브 메쏘드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

nested class

이 책에 쓰이는 번역어 : 중첩 클래스

다른 번역어 : 내포된 클래스, 내포 클래스

다른 용어 참조 :

이 책의 내용 참조 :

nested interface

이 책에 쓰이는 번역어 : 중첩 인터페이스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

nested type

이 책에 쓰이는 번역어 : 중첩 타입

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

non-final class

이 책에 쓰이는 번역어 : non-final 클래스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

non-public

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

non-static field

이 책에 쓰이는 번역어 : 비정적 필드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

non-static member

이 책에 쓰이는 번역어 : 비정적 멤버

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

normal completion

이 책에 쓰이는 번역어 : 정상적 종료

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

number

이 책에 쓰이는 번역어 : 숫자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

object

이 책에 쓰이는 번역어 : 객체

다른 번역어 : 개체

다른 용어 참조 :

이 책의 내용 참조 :

object-oriented

이 책에 쓰이는 번역어 : 객체지향

다른 번역어 : 개체지향

다른 용어 참조 :

이 책의 내용 참조 :

obscured

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

observable

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

octal

이 책에 쓰이는 번역어 : 8진수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

operand

이 책에 쓰이는 번역어 : 피연산자

다른 번역어 : 연산 대상

다른 용어 참조 :

이 책의 내용 참조 :

operator

이 책에 쓰이는 번역어 : 연산자

다른 번역어 : 조작원

다른 용어 참조 :

이 책의 내용 참조 :

optimization

이 책에 쓰이는 번역어 : 최적화

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

outer class

이 책에 쓰이는 번역어 : 외부 클래스

다른 번역어 : 바깥 클래스

다른 용어 참조 :

이 책의 내용 참조 :

overflow

이 책에 쓰이는 번역어 : 오버 플로우

다른 번역어 : 오버 플로

다른 용어 참조 :

이 책의 내용 참조 :

overloading

이 책에 쓰이는 번역어 : 중복 정의

다른 번역어 : 오버로딩, 중복, 다중 정의

다른 용어 참조 :

이 책의 내용 참조 :

overriding

이 책에 쓰이는 번역어 : 재정의

다른 번역어 : 오버라이딩, 치환

다른 용어 참조 :

이 책의 내용 참조 :

package

이 책에 쓰이는 번역어 : 패키지

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

package access

이 책에 쓰이는 번역어 : 패키지 접근

다른 번역어 :

다른 용어 참조 : default access

이 책의 내용 참조 :

parameter

이 책에 쓰이는 번역어 : 매개변수

다른 번역어 : 파라미터, 형식 매개변수(formal parameter), 인자, 형식 인자, 가인수, 인수, formal method parameter

다른 용어 참조 : argument

이 책의 내용 참조 :

pass by reference

이 책에 쓰이는 번역어 : 참조에 의한 호출(call by value)

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

performance

이 책에 쓰이는 번역어 :

다른 번역어 : 성능

다른 용어 참조 :

이 책의 내용 참조 :

pipe

이 책에 쓰이는 번역어 : 파이프

다른 번역어 :

다른 용어 참조 : filter, stream

이 책의 내용 참조 :

platform

이 책에 쓰이는 번역어 : 플랫폼

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

pointer

이 책에 쓰이는 번역어 : 포인터

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

polymorphism

이 책에 쓰이는 번역어 : 다형성

다른 번역어 : 폴리모피즘

다른 용어 참조 :

이 책의 내용 참조 :

portability

이 책에 쓰이는 번역어 : 이식성

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

postfix operator

이 책에 쓰이는 번역어 : 접미 연산자

다른 번역어 : 후치형 연산자, 후위 연산자

다른 용어 참조 :

이 책의 내용 참조 :

precedence

이 책에 쓰이는 번역어 : 우선 순위

다른 번역어 : 우선 순위

다른 용어 참조 :

이 책의 내용 참조 :

prefix operator

이 책에 쓰이는 번역어 : 접두 연산자

다른 번역어 : 전치형 연산자, 전위 연산자

다른 용어 참조 :

이 책의 내용 참조 :

primitive type

이 책에 쓰이는 번역어 : 기본형

다른 번역어 : 원시형, 기초 자료형, 원시 타입, 기본 데이터형, 기본 데이터 유형, 원형

다른 용어 참조 : reference type

이 책의 내용 참조 :

priority

이 책에 쓰이는 번역어 :

다른 번역어 : 우선 순위

다른 용어 참조 :

이 책의 내용 참조 :

private

이 책에 쓰이는 번역어 : private

다른 번역어 : 전용, 사적

다른 용어 참조 :

이 책의 내용 참조 :

process

이 책에 쓰이는 번역어 : 프로세스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

program

이 책에 쓰이는 번역어 : 프로그램

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

promotion

이 책에 쓰이는 번역어 :

다른 번역어 : 승급, 할당변환

다른 용어 참조 :

이 책의 내용 참조 :

protected

이 책에 쓰이는 번역어 : protected

다른 번역어 : 보호, 보호된

다른 용어 참조 :

이 책의 내용 참조 :

public

이 책에 쓰이는 번역어 : public

다른 번역어 : 공용

다른 용어 참조 :

이 책의 내용 참조 :

qualified

이 책에 쓰이는 번역어 : 수식된

다른 번역어 : 한정된

다른 용어 참조 :

이 책의 내용 참조 :

qualifying type

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

queue

이 책에 쓰이는 번역어 : 큐

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

race condition

이 책에 쓰이는 번역어 : 경합 상태

다른 번역어 : 경주 조건

다른 용어 참조 :

이 책의 내용 참조 :

reachable

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

recursive call

이 책에 쓰이는 번역어 : 재귀 호출

다른 번역어 : 순환 호출

다른 용어 참조 :

이 책의 내용 참조 :

reference

이 책에 쓰이는 번역어 : 참조

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

reference type

이 책에 쓰이는 번역어 : 참조형

다른 번역어 : 레퍼런스 데이트 형

다른 용어 참조 :

이 책의 내용 참조 :

reflection

이 책에 쓰이는 번역어 : 리플렉션

다른 번역어 : 클래스 발견 메커니즘, 반사

다른 용어 참조 :

이 책의 내용 참조 :

resource

이 책에 쓰이는 번역어 : 자원

다른 번역어 : 리소스

다른 용어 참조 :

이 책의 내용 참조 :

return

이 책에 쓰이는 번역어 : 반환

다른 번역어 : 리턴, 복귀

다른 용어 참조 :

이 책의 내용 참조 :

right-hand side

이 책에 쓰이는 번역어 : 우변

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

rounding

이 책에 쓰이는 번역어 : 반올림

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

run-time

이 책에 쓰이는 번역어 : 실행 시간

다른 번역어 : 런타임, 실행

다른 용어 참조 :

이 책의 내용 참조 :

run-time error

이 책에 쓰이는 번역어 : 실행 시간 에러

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

scheduling

이 책에 쓰이는 번역어 : 스케줄링

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

scope

이 책에 쓰이는 번역어 : 영역

다른 번역어 : 범위, 가시 구역, 유효범위, 스코프

다른 용어 참조 :

이 책의 내용 참조 :

security

이 책에 쓰이는 번역어 : 보안

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

semicolon(;)

이 책에 쓰이는 번역어 : 세미 콜론

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

separator

이 책에 쓰이는 번역어 : 구분자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

serialization

이 책에 쓰이는 번역어 : 직렬화

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

shadowing

이 책에 쓰이는 번역어 : 가리기

다른 번역어 : 가림, 은닉

다른 용어 참조 : hiding

이 책의 내용 참조 :

shallow cloning

이 책에 쓰이는 번역어 : 얕은 복제

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

side effect

이 책에 쓰이는 번역어 : 부수적 효과

다른 번역어 : 부수 효과, 부작용, 부차적 작용

다른 용어 참조 :

이 책의 내용 참조 :

signature

이 책에 쓰이는 번역어 : 서명

다른 번역어 : 사인, 시그너춰

다른 용어 참조 :

이 책의 내용 참조 :

simple name

이 책에 쓰이는 번역어 : 간단한 이름

다른 번역어 : 약식 이름

다른 용어 참조 : qualified name, fully qualified name

이 책의 내용 참조 : 언어의 기본요소 – simple name과 qualified name

single inheritance

이 책에 쓰이는 번역어 : 단일 상속

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

single quote(‘)

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

sorting

이 책에 쓰이는 번역어 : 정렬

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

source code

이 책에 쓰이는 번역어 : 소스 코드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

statement

이 책에 쓰이는 번역어 : 문, 문장

다른 번역어 : 명령문

다른 용어 참조 :

이 책의 내용 참조 :

static

이 책에 쓰이는 번역어 : 정적, static

다른 번역어 : 스태틱

다른 용어 참조 :

이 책의 내용 참조 :

static binding

이 책에 쓰이는 번역어 : 정적 바인딩

다른 번역어 : 정적 결합, 컴파일 타임 바인딩(compile-time binding)

다른 용어 참조 :

이 책의 내용 참조 :

static context

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

static field

이 책에 쓰이는 번역어 : 정적 필드

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

static initializer

이 책에 쓰이는 번역어 : 정적 초기화 블록

다른 번역어 : 스태틱 초기화 블록

다른 용어 참조 :

이 책의 내용 참조 :

static member

이 책에 쓰이는 번역어 : 정적 멤버

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

static method

이 책에 쓰이는 번역어 : 정적 메쏘드

다른 번역어 : 스태틱 메쏘드

다른 용어 참조 :

이 책의 내용 참조 :

static nested class

이 책에 쓰이는 번역어 : 정적 중첩 클래스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

store

이 책에 쓰이는 번역어 : 저장

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

stream

이 책에 쓰이는 번역어 : 스트림

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

string

이 책에 쓰이는 번역어 : 문자열

다른 번역어 : 스트링

다른 용어 참조 :

이 책의 내용 참조 :

subclass

이 책에 쓰이는 번역어 : 서브 클래스

다른 번역어 : 자식 클래스(child class), 파생 클래스(derived class), 파생된 클래스, 하위 클래스

다른 용어 참조 : superclass

이 책의 내용 참조 :

subpackage

이 책에 쓰이는 번역어 : 서브 패키지

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

subtype

이 책에 쓰이는 번역어 : 서브 타입

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

superclass

이 책에 쓰이는 번역어 : 수퍼 클래스

다른 번역어 : 부모 클래스(parent class), 기초 클래스(base class), 기반 클래스, 기저 클래스, 상위 클래스

다른 용어 참조 :

이 책의 내용 참조 :

superinterface

이 책에 쓰이는 번역어 : 수퍼 인터페이스

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

supertype

이 책에 쓰이는 번역어 : 수퍼 타입

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

synchronization

이 책에 쓰이는 번역어 : 동기화

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

tag

이 책에 쓰이는 번역어 : 태크

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

ternary operator

이 책에 쓰이는 번역어 : 삼항 연산자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

thread

이 책에 쓰이는 번역어 : 쓰레드

다른 번역어 :

다른 용어 참조 : synchronization, deadlock, lock, race condition

이 책의 내용 참조 :

throw

이 책에 쓰이는 번역어 : 던지기, throw

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

time slice

이 책에 쓰이는 번역어 : 시분할

다른 번역어 : 시간 조각화

다른 용어 참조 :

이 책의 내용 참조 :

token

이 책에 쓰이는 번역어 : 토큰

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

top-level class

이 책에 쓰이는 번역어 : 최상위 클래스

다른 번역어 :

다른 용어 참조 : nested class

이 책의 내용 참조 :

top-level interface

이 책에 쓰이는 번역어 : 최상위 인터페이스

다른 번역어 :

다른 용어 참조 : nested interface

이 책의 내용 참조 :

top-level type

이 책에 쓰이는 번역어 : 최상위 타입

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

type

이 책에 쓰이는 번역어 : 타입, 형, 데이터형

다른 번역어 : 자료형, 형식

다른 용어 참조 :

이 책의 내용 참조 :

unary operator

이 책에 쓰이는 번역어 : 단항 연산자

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

unchecked exception

이 책에 쓰이는 번역어 : 검사되지 않는 예외

다른 번역어 : 검사되지 않은 예외

다른 용어 참조 :

이 책의 내용 참조 :

underflow

이 책에 쓰이는 번역어 : 언더 플로우

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

Unicode

이 책에 쓰이는 번역어 : 유니 코드

다른 번역어 : 문자표준

다른 용어 참조 : character, string

이 책의 내용 참조 :

unloading

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

unnamed package

이 책에 쓰이는 번역어 : 익명 패키지

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

unqualified

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

unreachable

이 책에 쓰이는 번역어 :

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

value

이 책에 쓰이는 번역어 : 값

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

variable

이 책에 쓰이는 번역어 : 변수

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

virtual machine

이 책에 쓰이는 번역어 : 가상 머신

다른 번역어 : 가상 기계, 버추얼 머신

다른 용어 참조 :

이 책의 내용 참조 :

visible

이 책에 쓰이는 번역어 :

다른 번역어 : 가시성

다른 용어 참조 :

이 책의 내용 참조 :

widening

이 책에 쓰이는 번역어 : 넓히는

다른 번역어 : 광역화, 확장

다른 용어 참조 :

이 책의 내용 참조 :

widening primitive conversion

이 책에 쓰이는 번역어 : 넓히는 기본형 형변환

다른 번역어 :

다른 용어 참조 :

이 책의 내용 참조 :

widening reference conversion

이 책에 쓰이는 번역어 : 넓히는 참조형 형변환

다른 번역어 : 광역화 참조 형변환

다른 용어 참조 :

이 책의 내용 참조 :

wrapper class

이 책에 쓰이는 번역어 : 포장 클래스

다른 번역어 : 래퍼 클래스, 타입 포장 클래스

다른 용어 참조 :

이 책의 내용 참조 :

1. 준비하기
   1. JDK와 에디터

자바 프로그래밍을 하기 위해서는 최소한으로 필요한 것이 에디터(editor)와 JDK이다. 현재 JDK 는 버젼 1.3까지(1.4는 베타 버전이 있다) 나와 있다. 이 책의 대부분의 코드는 1.1이상에서 돈다. 그러나 몇몇 부분은 1.3 또는 1.4버전이 필요하다. 최신 버전을 사용하는 것이 좋다. 에디터는 아무거나 사용해도 된다.

JDK는 JavaTM 2 Platform, Standard Edition라고도 불린다. 선마이크로시스템사(http://java.sun.com)에 가서 받아오면 된다.

사실 간단한 에디터와 컴파일러와 가상 머신만을 이용해서 할 수도 있지만 통합 개발 환경(Intergrated Developement Environment)를 사용하는 것이 편리하다. 여러가지 자바 개발환경 중 하나를 선택해서 그것을 깔고 도움말을 보면 될 것이다.

여기서는 JDK를 사용하는 것을 기준으로 한다.

이 책의 예제는 Windows운영체제에서 테스트 해 보았다. JDK는 항상 똑 같은 버전을 사용한 것이 아니다. 그러므로 결과가 약간 다를 수도 있다.

* 1. JavaTM 2 Platform, Standard Edition, v 1.x API Specification

자바 프로그래밍을 할 때 가장 기본이 되는 도움말이라고 보면 된다. API의 도움말이다. 선 마이크로시스템 사에서 Doc이라고 된 것을을 받아오면 된다. 여러 버전의 도움말을 받아서 API가 어떻게 발전했는지 보는 것도 좋다.

* 1. JavaTM 2 Platform, Standard Edition, v 1.x Source Code

자바를 깊이 공부할수록 기본적인 API소스가 어떻게 생겼는지 보는 것이 도움이 된다. 사실 자바 실력이 늘수록 API 소스 코드를 보고 싶은 마음이 더 많이 생길 것이다. JDK를 깔때 같이 온다. src.jar라는 이름(아니면 그 비슷한)의 파일에 있으니 압축을 풀면 된다. 버전에 따라 약간 다르다. 어떤 경우에는 API Specification의 문서가 부족해서 소스 코드를 봐야만 의문이 풀리는 경우도 있다. 별로 보고 싶은 마음이 생기지 않아도 억지로 보는 것도 좋다. 당신보다 훨씬 고수인 사람들인 만든 것이기 때문에 거기에 있는 코드를 모방하는 것도 좋은 공부 방법이다.

* 1. Hello World

가장 기본적인 프로그램을 만들어 보자. 다음과 같다.

public class Example {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

위의 코드를 Example.java라는 파일명의 파일에 써넣자. 파일명에 대소문자도 구별해서 써야 한다.

그 다음에 자바 컴파일러(javac.exe)를 실행시킨다.

javac Example.java

그러면 Example.class라는 파일이 생성될 것이다.

자바 가상 머신을 실행한다.

java Example

그럼 다음과 갈은 결과가 나올 것이다.

Hello World

뭔가 잘 안된다면 선마이크로시스템사의 자바 튜토리얼(Java Tutorial)을 참조하라.

위의 예에서 main()메쏘드가 entry point이다. 즉 프로그램은 메인 메쏘드의 내용을 실행한다.

당장은 왜 메인 메쏘드에 public static void가 붙는지 또는 왜 String[] args가 필요한지는 무시하라. 천천히 알게 될 것이다. 그냥 main 메쏘드 안에 넣은 코드가 실행된다고만 알면 된다.

* 1. 이 책에 쓰이는 관례
     1. 소스 파일 이름

특별한 언급이 없는 이 책에 쓰이는 모든 소스 코드는 Example.java파일에 저장해서 컴파일 하고 실행해 보면 된다. 패키지가 명시적으로 사용되는 경우에는 디렉토리 구조를 패키지에 맞춰 주어야 한다. 자세한 것은 패키지를 다룬 부분을 참조하라.

* + 1. 예제의 결과

예제의 결과를 보일 때 보통 두가지 표현이 쓰인다.

첫번째로는 “결과는 다음과 같을 것이다” 라는 표현은 특별한 일이 없는 한 결과가 책에 있는 그대로 즉 글자 하나 안 틀리고 나올 때를 말한다. 여기서 특별한 일이란 컴퓨터가 외계인에게 납치되어 뇌수술을 받거나 누군가 전원을 뽑아 버리거나 어떤 이유로 컴퓨터가 다운되는 경우이다. 아주 희귀하게는 당신이 예제를 잘못 쳐넣은 경우도 있다.

두번째로는 “결과는 다음과 비슷할 것이다” 라는 표현은 상황에 따라 결과가 약간 또는 상당히 다르게 나타날 수 있는 경우이다. 여기서 상황이란 시스템 시간, 운영체제의 종류, JDK 버전 등등이다. 결과가 어떤 점에서 비슷한 것인가는 내용을 이해하면 대충 알 수 있을 것이다.

어떤 예제들은 두 가지 표현 중 하나를 선택하기 애매한 경우도 있었다.

* + 1. 식별자 이름

이 책에 쓰이는 식별자 이름(변수명, 클래스명, 메쏘드명등)은 일관성이 없다. 어떤 때는 a, b등으로 쓰고 어떤 때는 MFC(Visual C++ 참조)식으로 m\_iValue 같이 쓰기도 하고 자바 API에서 쓰는 방식을 쓰기도 한다. 어떤 경우에는(예를 들어 이름 충동을 다룰 때) 불가피하게 나쁜 방식을 쓸 수 밖에 없는 경우도 있다.

한마디로 말하면 본받을 만한 사용도 있지만 결코 따라하면 안되는 사용도 있으니 잘 판단하기 바란다. 대부분의 경우에는 본받을 만한 사용이 아니다. 식별자 이름 붙이는 것의 모범은 이 책이 아니라 API 소스코드에서 찾아야 한다.

* + 1. 들여쓰기(indentation)

식별자 이름과는 달리 들여쓰기는 일관성이 있다. 여기서 사용한 방식은 표준 API가 쓰는 방식과 거의 비슷하다. 물론 때에 따라 설명을 위해서 일부러 이상하게 들여쓰기를 할 때도 있다.

* 1. deprecated

자바를 조금 공부하다 보면 deprecated라는 컴파일 경고가 나올 때가 있다. 또는 API Specification을 보다보면 deprecated라고 된 것을 볼 수 때가 있을 것이다. 에러(Error)가 아니라 경고(warning)이기 때문에 실행은 된다.

자바는 1995년에 태어나서 급속하게 발전했다. 게다가 초기에는 너무 급하게 만들어져서 여러가지 문제가 있었다. 그 문제점을 고치는 과정에서 만들어지는 과정에서 나온 것이 deprecated이다. 주로 메쏘드가 deprecated되었는데 때에 따라서는 클래스 전체가 deprecated될 수도 있다.

deprecated는 비난받는다는 뜻이 있다. 배척해야 하는, 폐기 준비, 사용 금지된 등등으로 번역되기도 한다. 이런 메쏘드들은 미래의 버전에서는 삭제될 수도 있다. 그 이유뿐 아니라 왜 deprecated로 만들었는지를 생각해 보자. 문제가 있기 때문에 다른 것으로 대체한 것이다. 그러므로 정말 다른 방법이 없는 경우가 아니면 쓰지 않는 것이 좋다.

deprecated에 대한 해결책으로 컴파일러의 deprecated 옵션을 끄면 된다는 말을 들은 적이 있는데 정말 어이가 없는 해결책이다. 그건 해결이 아니라 문제를 덮어두는 것이다.

1. 언어의 기본 요소
   1. 유니코드
      1. 유니 코드 버전

자바 프로그램은 유니 코드로 쓰여진다. 현재 쓰이고 있는 자바 언어의 유니 코드 버젼은 Character.java에 문서화되어 있다. 다음은 Character.java에서 인용한 것이다.

…

\* Character information is based on the Unicode Standard, version 3.0.

…

유니 코드에 대한 자세한 것은 http://www.unicode.org을 참조하라.

* + 1. 유니코드 이스케이프(Unicode escape)

만약 유니코드가 지원되지 않는 시스템에서 프로그래밍을 한다면 유니코드 이스케이프를 사용하여 ASCII코드만으로도 유니코드 문자를 표현할 수 있다.

유니코드 이스케이프는 다음과 같이 구성된다.

\u네자리의16진수

예를 들면 다음과 같다.

\u004E

유니코드가 어느 정도 지원되더라도 예를 들어 아랍어나 히브리어를 입력할 수 있는 편집기를 구하기 어려울 것이다. 이럴 때 불편하지만 유니코드 이스케이프를 사용할 수 있는 것이다.

다음 프로그램(프로그램 맞다)을 컴파일해서 실행해 보라. 파일 이름은 Example.java가 되어야 한다. 주의할것은 공백문자(스페이스, 탭, 줄바꿈 등)가 전혀 없어야 된다는 것이다.

\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0063\u006c\u0061\u0073\u0073\u0020\u0045\u0078\u0061\u006d\u0070\u006c\u0065\u0020\u007b\u0020\u0070\u0075\u0062\u006c\u0069\u0063\u0020\u0073\u0074\u0061\u0074\u0069\u0063\u0020\u0076\u006f\u0069\u0064\u0020\u006d\u0061\u0069\u006e\u0028\u0053\u0074\u0072\u0069\u006e\u0067\u005b\u005d\u0020\u0061\u0072\u0067\u0073\u0029\u0020\u007b\u0020\u0053\u0079\u0073\u0074\u0065\u006d\u002e\u006f\u0075\u0074\u002e\u0070\u0072\u0069\u006e\u0074\u006c\u006e\u0028\u0022\u0048\u0065\u006c\u006c\u006f\u0020\u0057\u006f\u0072\u006c\u0064\u0022\u0029\u003b\u0020\u007d\u0020\u007d

결과는 다음과 같을 것이다.

Hello World

어떤 통합 개발 환경에서는 컴파일이나 실행이 안 될 수도 있다. 그럴 때는 도스창 등에서 JDK를 사용하면 된다. 위의 유니 코드 이스케이프를 변환하면 다음과 같은 프로그램이 된다(들여쓰기, 줄바꿈 등은 약간 바뀜).

public class Example {

public static void main(String[] args) {

System.out.println("Hello World");

}

}

그리고 위의 유니 코드 이스케이프는 다음 프로그램을 실행하면 얻을 수 있다.

public class Example {

public static void main(String[] args) {

String strCode = "public class Example { public static void main(String[] args) { System.out.println(\"Hello World\"); } }";

for (int i = 0; i < strCode.length(); i++) {

char ch = strCode.charAt(i);

System.out.print("\\u");

String str = Integer.toHexString(ch);

if (str.length() == 3) {

str = "0" + str;

} else if (str.length() == 2) {

str = "00" + str;

} else if (str.length() == 1) {

str = "000" + str;

}

System.out.print(str);

}

}

}

* + 1. 주의할 점

자바 언어의 표준은 유니코드를 사용하고 유니코드 이스케이프를 이용해 ASCII 코드만으로도 유니 코드를 지원하게 했다. 현재 쓰이고 있는 유니 코드는 2바이트 유니코드이다. 이론상으로는 어떤 언어로도 프로그래밍을 할 수 있다. 예를 들어 한글로 변수 이름을 지정할 수도 있으며 취향에 따라 아랍어나 히브리어로 프로그래밍을 할 수도 있을 것이다.

그러나 이것은 표준이 그렇다는 얘기일 뿐이다. 실제로는 다음과 같은 여러가지 문제가 있다.

* 모든 운영 체제 또는 에디터가 유니 코드를 지원하지 않는다.
* 모든 자바 컴파일러가 유니코드와 유니코드 이스케이프 표준을 따르는 것은 아닐 수도 있다.
* 유니 코드 표준 자체가 조금씩 바뀌고 있다.
* 현재 2바이트 유니코드로는 6만여 문자 밖에 표현할 수 없다. 이것은 한자 같은 언어로 고려한다면 터무니 없이 작다. 중국의 한자만 하더라도 제대로 표현하려면 수십만 개의 문자가 필요할 것이다. 그래서 4바이트 유니코드가 있고 장래에는(언제가 될지 모르겠지만) 4바이트 유니코드를 사용할 것이다.
* 영어 이외의 문자를 사용한다면 다른 나라 프로그래머가 알아보기 힘들 것이다.
* 영어 이외(또는 ASCII 이외)의 문자를 사용한다면 유니 코드 표준이 바뀐 후 변수명이 제대로 남아 있으라는 보장이 없다. 변수명의 일부가 조금이라도 바뀌면 프로그래머에게는 재앙이다.

따라서 식별자 이름을 영어로 사용할 것을 강력히 권한다.

* 1. line terminators

소스 코드에서 줄바꿈은 (컴파일러에게는) 두가지 정도의 의미가 있다. 첫째, //주석은 그 줄의 끝까지를 주석문으로 생각한다. 둘째, 컴파일러의 에러 메시지에 몇 라인에서 에러가 났는지가 나온다.

줄바꿈을 나타내는 것을 라인 종결자(Line Terminator)라고 하는데 다음 셋은 하나의 줄바꿈을 나타낸다.

* LF(Line Feed)
* CR(Carrige Return)
* LF와 바로 다음에 오는 CR : 이것은 둘이 아니라 하나의 줄바꿈이다.

물론 사람에게는 엄청나게 중요하다. 줄바꿈이 전혀 안된 프로그램을 한번 상상해 보라.

* 1. 주석(comment)

자바 주석에는 세가지가 있다.

/\*\* javadoc 을 위한 주석 \*/

public class Example { // 한줄 주석

public static void main(String[] args) {

/\* 여러

줄

주석 \*/

System.out.println("Hello World");

}

}

먼저 //로 시작해서 그 줄 끝까지 가는 한 줄짜리 주석이 있다.

/\*로 시작해서 \*/까지 가는 여러 줄 주석이 있다.

/\*\*로 시작해서 \*/까지 가는 여러 줄 주석이 있다. 이 주석은 javadoc을 위한 주석이다. 자세한 것은 javadoc을 다루는 책이나 문서를 보기 바란다.

위의 예제에서 다음과 같이 하면 더 예쁘다고 믿는 사람들이 많다.

/\*\*

\* javadoc 을 위한 주석

\*/

public class Example { // 한줄 주석

public static void main(String[] args) {

/\*

\* 여러 줄 주석

\*/

System.out.println("Hello World");

}

}

주석은 순전히 프로그래머를 위한 것이다(약간의 예외가 있긴 하다). 컴파일러는 주석 부분을 무시한다. 주석을 쓰는 원칙이 있긴 하지만 초보자에게는 다음 원칙으로 충분하다. “무조건 덕지덕지 주석을 달아라.”

자바의 주석을 중첩될 수 없다. 예를 들어 다음 프로그램을 컴파일해 보라.

파일 이름은Example.java로 한다.

class Example {

/\* /\* \*/ A\*/

}

컴파일타임 오류가 난다. 컴파일러는 /\* /\* \*/까지를 주석으로 생각한다. /\*가 시작하면 그곳부터 \*/를 처음 만나는 곳까지를 주석으로 보는 것이다. 그러므로 컴파일러는 주석을 제외한 소스 코드를 다음과 같다고 생각한다.

class Example {

A\*/

}

하나의 주석이 시작되고 아직 그 주석이 끝나지 않았다면 그 안에 있는 주석은 무시된다. 다음 예를 보자.

class Example {

// /\*

\*/

}

컴파일 에러가 난다. 컴파일러는 주석을 제외한 소스 코드를 다음과 같이 생각할 것이다.

class A {

\*/

}

문자열 안에 있는 것은 주석이 아니다. 예를 들어 다음 코드에는 주석문이 없다. 다음 코드를 실행시켜 보라.

public class Example {

public static void main(String[] args) {

System.out.println("aa/\* ee \*/");

}

}

결과는 다음과 같을 것이다.

aa/\* ee \*/

즉 컴파일러는 /\* ee \*/를 주석으로 생각하지 않기 때문에 무시하지 않는다.

/\* \*/ 는 다음과 같이 쓰일 수도 있다. 그러나 보면 알겠지만 권할 만한 것은 아니다.

class Example {

public /\* 모두가 접근할 수 있다.\*/ int i;

}

* 1. 토큰(token)

컴파일러는 일단 공백과 주석을 걸러낸다. 그러면 토큰이 남는다. 아래의 예를 보자.

public class Example { // 주석

public static void main(String[] args) {

int a = 0;

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

}

}

컴파일러는 위의 소스 코드를 아래와 같이 여러 개의 토큰의 연속으로 인식할 것이다.

public

class

Example

{

public

static

void

main

(

String

[

]

args

)

{

int

a

=

0

;

System

.

out

.

println

(

"a = "

+

a

)

;

}

}

토큰은 언어의 기본 요소이다. 위에 쓰인 토큰을 분류하면 다음과 같다.

* 리터럴 : "a = ", 0
* 식별자(identifier) : Example, a, main, args, System, out, println, String
* 키워드(keyword) : public, class, static, void, int
* 연산자(operator) : =, +, [, ], (, )
* 구분자(separator) : ;, {, }, .

주석은 공백문자처럼 토큰을 나눠주는 역할을 한다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int val;

}

}

위의 프로그램에서 val이라는 int형 변수가 선언된다.

public class Example {

public static void main(String[] args) {

int va l;

}

}

위의 예에서는 공백 문자 때문에 va와 l이라는 두 개의 토큰으로 나뉜다. 그러므로 컴파일 에러가 난다.

public class A {

public static void main(String[] args) {

int va/\*\*/l;

}

}

위의 예에서도 주석 때문에 va와 l이라는 두개의 토큰으로 나뉜다. 그러므로 컴파일 에러가 난다.

* 1. 식별자(identifier)

식별자(identifier)가 되기 위해서는 다음의 조건이 성립해야 한다.

* 자바의 예약어(reserved word)와 일치하면 안된다 : 즉 키워드, null, true, false등과 일치하면 안된다.
* 첫문자에 제한이 있다. 그 문자가 첫문자에 쓰이는 것을 정확히 알고 싶으면 Character.isJavaIdenifierStart 메쏘드를 실행시켜보면 된다. 첫문자에 쓰일 수 없는 것은 숫자, 특수문자 등이다. 쓸 수 있는 특수 문자는 밑줄(\_)과 달러($)이다. 그러나 $는 컴파일러가 자동으로 생성하는 곳에만 쓰일 수 있다. 즉 프로그래머가 직접 쳐 넣으면 에러가 난다. $에 대한 자세한 것은 중첩 타입을 다루는 부분을 참조하라.
* 첫문자가 아닌 문자에도 제한이 있다. 그것을 정확히 알고 싶으면 Character.isJavaIdentifierPart 메쏘드를 실행시켜보면 된다. 첫문자가 아닌 문자에는 숫자도 쓰일 수 있다는 것을 제외하면 첫문자와 제한이 같다.
* 길이에는 제한이 없다. 그러나 너무 길면 예를 들어 길이가 1000정도 되는 변수를 만든다면 좋아할 사람이 없을 것이다. 모든 컴파일러와 가상 머신이 무한 길이 문자열을 지원하는지는 잘 모르겠다.

두 식별자는 완전히 같을 때만 동일한 것으로 취급된다. 예를 들어 대소문자도 구별한다. 그러므로 val과 Val과 VAL은 모두 각각 다른 식별자이다.

비슷하게 생긴 문자라도 유니코드 코드 값이 다르다면 다르게 취급된다. 그러나 영어로만 식별자를 만든다면 이것은 별로 중요하지 않다. 그러나 영어와 러시아어를 같이 쓰거나 하면 주의해야 할 것이다. 영어와 러시아를 같이 쓴다는 것 자체가 문제지만.

다음에 나열된 것은 올바른 식별자이다.

True : 컴파일은 되겠지만 true와 헷갈리므로 안쓰는 것이 좋다.

VAL

val

minMax

\_minMax

A$B : 이것은 컴파일러가 자동으로 생성할 수는 있지만 프로그래머가 직접 쳐 넣을 수는 없다. 내부 클래스를 참조하라.

다음에 나열된 것은 잘못된 식별자이다.

true : 예약어이다.

goto : 자바에서는 goto문이 없지만 식별자로 사용하는 것이 금지되어 있다. 이것은 실수를 막기 위한 것이다.

9ee : 숫자가 앞에 올 수 없다.

* 1. 구분자(separator, punctuator)

구분자에는 다음과 같은 것이 있다.

( ) { } [

] ; , .

* 1. 연산자(operator)

연산자에는 다음과 같은 것이 있다. 자세한 것은 연산자를 다루는 부분을 참조하라.

= > < ! ~

? : == <= >=

!= && || ++ --

+ - \* / &

| ^ % << >>

>>> += -= \*= /=

&= |= ^= %= <<=

>>= >>>=

* 1. 키워드

자바 키워드에는 다음과 같은 것이 있다.

abstract default if private this

boolean do implements protected throw

break double import public throws

byte else instanceof return transient

case extends int short try

catch final interface static void

char finally long strictfp volatile

class float native super while

const for new switch

continue goto package synchronized

자바에서는 const와 goto를 사용하지 않지만 식별자로 쓸 수 없다. 이것은 프로그래머의 실수를 줄이기 위해서이다. 실수로 const등을 사용했다면 컴파일러 에러가 날 것이다.

엄밀히 말해서 null은 null 리터럴이고 true와 false는 불형(boolean) 리터럴이다.

* 1. simple name과 qualified name

간단한 이름(simple name)은 하나의 토큰으로 된 이름이다. 이에 반해서 수식된 이름(qualified name)은 여러 개의 토큰으로 된 이름이다.

예를들어 Button은 simple name이고 java.awt.Button은 qualified name이다. 다른 예를 들면 Point 클래스에서 x라고 쓰면 simple name이고 p.x라고 쓰면 qualified name이다(이 때 p는 Point형 변수이다). 그리고 java.awt.Button처럼 전체적으로 이름에 필요한 정보를 다 쓰는 것을 fully qualified name이라고 한다.

1. 제어문
   1. if
      1. if-then-else문의 기초

if-then-else문은 둘중에 하나를 실행시킬 때 쓰인다. 조건식(if 키워드 다음에 오는 괄호 안의 식, 아래 예제에서 a >= 0)이 참이면 if 다음에 있는 것을 거짓일 때는 else다음에 있는 문장을 실행한다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = -23;

if (a >= 0)

System.out.println("a는 0 이상이다.");

else

System.out.println("a는 0 미만이다.");

}

}

결과는 다음과 같다.

a는 0 미만이다.

문장은 널 문장일 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

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

if (a >= 0)

; // null 문장

else

System.out.println("a는 0 미만이다.");

}

}

결과는 다음과 같다.

a = 10

그러나 세미콜론까지 빠지면 안된다. 다음 예제를 컴파일 해 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

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

if (a >= 0)

else

System.out.println("a는 0 미만이다.");

}

}

컴파일 에러가 날 것이다.

문장 대신 블럭이 올 수도 있다.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0) {

System.out.println("a는 0 이상이다.");

} else {

System.out.println("a는 0 미만이다.");

}

}

}

결과는 다음과 같다.

a는 0 이상이다.

위의 예에서는 블럭을 사용하는 것이 시간과 공간의 낭비라는 생각이 들 것이다. 그러나 시간과 공간을 낭비하더라도 이렇게 해야 한다고 믿는 사람들이 있다. 그것에 대해서는 아래의 “실수를 줄이는법”을 참조하라. 그리고 블럭을 꼭 사용해야 할 때가 있다. 여러 문장을 하나의 단위로 묶어주기 위해서이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = -10;

if (a >= 0) {

System.out.println("a는 0 이상이다.");

System.out.println("정말이다.");

} else {

System.out.println("a는 0 미만이다.");

System.out.println("진짜라니까!!!");

}

}

}

결과는 다음과 같다.

a는 0 미만이다.

진짜라니까!!!

블럭에는 아무것도 안들어가도 된다.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0) {

System.out.println("a는 0 이상이다.");

System.out.println("정말이다.");

} else {

}

}

}

결과는 다음과 같다.

a는 0 이상이다.

정말이다.

한쪽은 문장을 사용하고 한쪽은 블럭을 사용할 수도 있다.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0)

System.out.println("a는 0 이상이다.");

else {

System.out.println("a는 0 미만이다.");

System.out.println("진짜라니까!!!");

}

}

}

결과는 다음과 같다.

a는 0 이상이다.

이건 미관상 안좋다. 게다가 실수를 할 확률을 늘릴 것이다.

* + 1. if-then문의 기초

if-then문은 if-then-else문에서 else가 생략된 것이다. 즉 조건식이 참일 때는 어떤 문장 또는 블럭을 실행하고 거짓일 때는 그 문장 또는 블럭을 실행하지 않는다. 나머지는 if-then-else문과 같다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0)

System.out.println("a는 0 이상이다.");

}

}

결과는 다음과 같다.

a는 0 이상이다.

* + 1. if문의 조건식

if문의 조건식에는 boolean형만 올 수 있다. C 언어에서는 정수도 쓰일 수 있는데 자바에서는 그것이 허용되지 않는다. 이렇게 함으로써 유연성을 희생해서 실수할 가능성을 줄였다.

다음 예제는 C 프로그래머들이 밥먹듯이 하는 실수이다.

public class Example {

public static void main(String[] args) {

int a = -10;

if (a = 0) // 원래 의도는 (a == 0)

System.out.println("a는 0 이다.");

else

System.out.println("a는 0 이 아니다.");

}

}

만약 C였다면 무사히 컴파일되어서 원하지 않는 결과가 출력될 것이다. 자바에서는 컴파일 에러가 난다.

조건식은 반드시 ()로 묶여야 한다. 즉 다음 예제는 컴파일되지 않는다.

public class Example {

public static void main(String[] args) {

int a = 10;

boolean booleanValue = (a >= 0);

if booleanValue

System.out.println("a는 0 이상이다.");

}

}

* + 1. 들여쓰기의 예외

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = 9999;

if (a < 0)

System.out.println("a는 음수이다.");

else

if (a == 0)

System.out.println("a는 0 이다.");

else

if (a < 100)

System.out.println("a는 100 보다 작은 양수이다.");

else

if (a < 10000)

System.out.println("a는 100 이상이고 10000보다 작은 양수이다.");

else

System.out.println("a는 10000 이상인 양수이다.");

}

}

결과는 다음과 같다.

a는 100 이상이고 10000보다 작은 양수이다.

이럴 경우에는 예외적으로 들여쓰기를 다음과 같이 한다.

public class Example {

public static void main(String[] args) {

int a = 9999;

if (a < 0)

System.out.println("a는 음수이다.");

else if (a == 0)

System.out.println("a는 0 이다.");

else if (a < 100)

System.out.println("a는 100 보다 작은 양수이다.");

else if (a < 10000)

System.out.println("a는 100 이상이고 10000보다 작은 양수이다.");

else

System.out.println("a는 10000 이상인 양수이다.");

}

}

훨씬 알아보기 쉬워졌다.

* + 1. 실수를 줄이는 법

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

if (10 < 1);

System.out.println("10 은 1보다 작다");

}

}

결과는 다음과 같을 것이다.

10 은 1보다 작다

흔히 하는 실수 중에 하나이다. 세미 콜론 때문에 컴파일러는 위의 예제를 다음과 같이 해석할 것이다.

if (10 < 1)

;

System.out.println("10 은 1보다 작다");

다음 예를 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

boolean booleanValue = (a >= 0);

if (true == booleanValue)

System.out.println("a는 0 이상이다.");

else

System.out.println("a는 0 미만이다.");

}

}

결과는 다음과 같다.

a는 0 이상이다.

여기서 조건식을 (true == booleanValue)라고 썼다. 많은 사람들이 (booleanValue == true)가 더 미관상 보기 좋다고 생각할 것이다. 그럼에도 위와 같이 쓰기를 고집하는 사람이 있다면 아래에 나오는 실수 때문에 디버깅하느라고 밤을 새본 경험이 있는 사람일 것이다.

다음 예제는 사람들이 흔히 하는 실수이다.

public class Example {

public static void main(String[] args) {

int a = -10;

boolean booleanValue = (a >= 0);

if (booleanValue = true) // 여기를 주목하라.

System.out.println("a는 0 이상이다.");

else

System.out.println("a는 0 미만이다.");

}

}

결과는 다음과 같다.

a는 0 이상이다.

(booleanValue == true)를 쓴다는게 =을 하나 빼먹어서 (booleanValue = true)라고 쓴 것이다. 만약 거꾸로 썼다면 즉 (true = booleanValue)라고 썼다면 이 실수를 컴파일러가 잡아줄 것이다. true에 무언가를 대입할 수는 없기 때문이다. 실제로 이렇게 바꾸어서 컴파일해보자.

앞에서 필요없는 블럭을 사용하기를 고집하는 사람이 있다고 했다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0) {

System.out.println("a는 0 보다 크거나 같다.");

} else {

System.out.println("a는 0 보다 작다.");

}

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

if (a >= 0)

System.out.println("a는 0 보다 크거나 같다.");

else

System.out.println("a는 0 보다 작다.");

}

}

결과는 다음과 같다.

a는 0 보다 크거나 같다.

-----------------------

a는 0 보다 크거나 같다.

프로그램을 다음과 같이 고쳤다고 생각해 보자.

public class Example {

public static void main(String[] args) {

int a = 10;

if (a >= 0) {

System.out.println("a는 0 보다 크거나 같다.");

} else {

System.out.println("a는 0 보다 작다.");

System.out.println("즉 음수이다.");

}

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

if (a >= 0)

System.out.println("a는 0 보다 크거나 같다.");

else

System.out.println("a는 0 보다 작다.");

System.out.println("즉 음수이다.");

}

}

결과는 다음과 같다.

a는 0 보다 크거나 같다.

-----------------------

a는 0 보다 크거나 같다.

즉 음수이다.

프로그래머의 의도는

if (a >= 0)

System.out.println("a는 0 보다 크거나 같다.");

else

System.out.println("a는 0 보다 작다.");

System.out.println("즉 음수이다.");

였지만 컴파일러는

if (a >= 0)

System.out.println("a는 0 보다 크거나 같다.");

else

System.out.println("a는 0 보다 작다.");

System.out.println("즉 음수이다.");

로 받아들인다. 이것은 정말 흔한 실수이다. 만약 블럭을 편집증적으로 사용한다면 이런 실수를 막을 수 있을 것이다. 그렇지만 블럭을 그런 식으로 사용하면 코드가 한줄이 더 늘어난다. 더욱이

if (a >= 0)

{

System.out.println("a는 0 보다 크거나 같다.");

}

else

{

System.out.println("a는 0 보다 작다.");

}

이런 식의 스타일을 선호하는 사람들에게는 네 줄이나 늘어난다. 게다가 이런 스타일에서는 가독성도 약간 떨어진다. 실수를 줄이느냐 줄수를 줄이느냐 그것이 문제로다! 선택은 당신에게 달렸다.

이번에는 dangling else problem이라고 이름붙여진 것에 대해 알아보자.

public class Example {

public static void main(String[] args) {

int a = 10;

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

if (a >= 0)

if (a > 100)

System.out.println("a는 100 보다 큰 양수이다.");

else

System.out.println("a는 음수이다.");

}

}

결과는 다음과 같다.

a = 10

a는 음수이다.

프로그래머의 의도는

if (a >= 0)

if (a > 100)

System.out.println("a는 100 보다 큰 양수이다.");

else

System.out.println("a는 음수이다.");

였지만 컴파일러는

if (a >= 0)

if (a > 100)

System.out.println("a는 100 보다 큰 양수이다.");

else

System.out.println("a는 음수이다.");

로 받아들인다. else는 가장 가까운 짝지어지지 않은 if와 짝을 맺는다. 이 문제를 해결하는 것도 블럭이다. 예제는 다음과 같이 고쳐져야 한다.

public class Example {

public static void main(String[] args) {

int a = 10;

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

if (a >= 0) {

if (a > 100)

System.out.println("a는 100 보다 큰 양수이다.");

} else

System.out.println("a는 음수이다.");

}

}

결과는 다음과 같다.

a = 10

만약 블럭을 편집증적으로 사용했다면 이런 실수도 막을 수 있었을 것이다.

* 1. switch
     1. switch문의 기초

switch문에서는 여러 개의 문장 중에서 선택할 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int number = 2;

switch (number) { // 여기서 ()안에 있는 number가 조건식이다.

case 1 :

System.out.println("number는 1 입니다.");

break;

case 2 :

System.out.println("number는 2 입니다.");

break;

case 3 :

System.out.println("number는 3 입니다.");

break;

case 4 :

System.out.println("number는 4 입니다.");

break;

case 5 :

System.out.println("number는 5 입니다.");

break;

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

break;

}

}

}

결과는 다음과 같다.

number는 2입니다.

switch문은 조건식(switch키워드 다음에 오는 괄호안의 식)의 값과 같은 case 다음의 문장으로 제어흐름(control flow)을 옮긴다. 만약 어떤 case에도 일치하는 값이 없다면 default다음의 문장을 실행한다. 만약 그 때 default문이 없다면 아무것도 실행하지 않는다. break는 switch문에서 빠져나오게 한다(제어문이 중첩된다면 문제가 복잡해진다). 만약 위의 예제에서 break를 모두 없애면 어떻게 될까. 그것이 다음 예제이다.

public class Example {

public static void main(String[] args) {

int number = 2;

switch (number) {

case 1 :

System.out.println("number는 1 입니다.");

case 2 :

System.out.println("number는 2 입니다.");

case 3 :

System.out.println("number는 3 입니다.");

case 4 :

System.out.println("number는 4 입니다.");

case 5 :

System.out.println("number는 5 입니다.");

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

}

}

}

결과는 다음과 같다.

number는 2 입니다.

number는 3 입니다.

number는 4 입니다.

number는 5 입니다.

number는 1보다 작고 5보다 큽니다.

break가 없으면 끝까지 계속 실행하는 것을 볼 수 있다. break가 없다면 그것은 보통 실수를 했다는 것을 뜻한다. 그러나 이런 것을 이용할 수도 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int number = 2;

switch (number) {

case 1 :

System.out.println("number는 0 보다 큽니다.");

case 2 :

System.out.println("number는 1 보다 큽니다.");

case 3 :

System.out.println("number는 2 보다 큽니다.");

case 4 :

System.out.println("number는 3 보다 큽니다.");

case 5 :

System.out.println("number는 4 보다 큽니다.");

break;

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

break;

}

}

}

결과는 다음과 같다.

number는 1 보다 큽니다.

number는 2 보다 큽니다.

number는 3 보다 큽니다.

number는 4 보다 큽니다.

* + 1. 조건식과 case 식의 제한

조건식에는 다음 타입만 올 수 있다.

byte, char, short, int

long이 올 수 없다는 것에 주의하라. 물론 실수형과 객체형은 더더욱 안된다. “Core Java”라는 책에서는 switch문의 유연성을 더 높게 만들었어야 했다고 말한다. 하지만 자바를 만든 사람은 C와 거의 똑같이 만들었다. 자바를 만든 James Gosling의 입장이 무엇인지는 잘 모르겠다. 그렇기 때문에 실수형과 객체형에 대해서는 좀더 거추장스러운 여러개의 if문을 사용해야 한다.

case식에는 상수 또는 상수식만 쓸 수 있다. 즉 컴파일 타임에 그 값이 결정되는 식만 쓸 수 있다. 다음 예제를 컴파일해보자. 두 군데에서 에러가 난다.

public class Example {

public static void main(String[] args) {

int number = 4;

int one = 1;

final int SIX = 6;

switch (number) {

case one : // 에러

System.out.println("number is 1.");

break;

case 2 : // OK

System.out.println("number is 2.");

break;

case 2 + 1 : // OK

System.out.println("number is 3.");

break;

case (int)(Math.PI + 1) : // OK

System.out.println("number is 4.");

break;

case getFive() : // 에러

System.out.println("number is 5.");

break;

case SIX : // OK

System.out.println("number is 6.");

break;

}

}

public static int getFive() {

return 5;

}

}

case식에 쓸 수 있는 값은 조건식의 타입에 의해 결정된다. case식에는 조건식의 타입으로 표현될 수 있는 수만 쓰일 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte number = 4;

final byte ONE = 1;

final short TWO = 2;

final int THREE = 3;

final char FOUR = 4;

number = THREE; // OK

switch (number) {

case ONE : // OK

System.out.println("number is 1.");

break;

case TWO : // OK

System.out.println("number is 2.");

break;

case THREE : // OK

System.out.println("number is 3.");

break;

case FOUR : // OK

System.out.println("number is 4.");

break;

}

}

}

결과는 다음과 같다.

number is 4.

컴파일 에러가 안나는 것에 의아해하는 사람도 있을 것이다. 예를 들어 THREE는 int형이다. 그리고 number는 byte형이다. int형은 byte형에 대입될 수 없다. 그러나 주목할 것은 THREE의 값이 컴파일 타임에 정해진다는 것이다. 그래서 위의 예에서

number = THREE; // OK

와 같은 것이 가능했던 것이다. 마찬가지로

case THREE : // OK

도 에러 없이 컴파일된다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte number = 4;

final long ONE = 1;

switch (number) {

case ONE : // Error

System.out.println("number is 1.");

break;

case 2L : // Error

System.out.println("number is 2.");

break;

case 3f : // Error

System.out.println("number is 3.");

break;

case 4d : // Error

System.out.println("number is 4.");

break;

case 1000 : // Error

System.out.println("number is 4.");

break;

}

}

}

1000은 byte형의 범위를 넘어서기 때문에 에러가 난다. 1은 byte형으로 표현 가능한 수이지만 long, float,double형을 쓰면 에러가 난다.

case에는 blank final 변수가 올 수 없다. 선언과 함께 값을 정해주지 않는 final 변수를 blank final 변수라 한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte number = 1;

final byte ONE; // blank final

ONE = 1;

switch (number) {

case ONE : // Error

System.out.println("number is 1.");

break;

}

}

}

컴파일 에러가 난다. 많은 사람이 이상하다고 생각할 것이다. 분명히 ONE은 final변수고 ONE에 1이 대입되었기 때문이다. 그러나 문제는 그렇게 간단하지가 않다. 다음 예제를 보면 왜 컴파일러가 그런식으로 판단하는지 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

byte number = 1;

final byte ONE; // blank final

System.out.println("Enter One Digit Number.");

char in = '1';

try {

in = (char)System.in.read();

} catch (Exception e) {

}

if (in == '1')

ONE = 1;

else

ONE = 2;

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

}

}

결과는 다음과 비슷하게 진행될 것이다. 여기서는 사용자가 ‘4’ ‘엔터키’를 누른 경우이다.

Enter One Digit Number.

4

ONE = 2

blank final 변수는 컴파일 타임에 값이 정해지지 않을 수도 있다.

* + 1. 실수를 줄이는 법

다음 예제는 위에서 한 번 본 예제에 간단한 주석을 붙인 것이다.

public class Example {

public static void main(String[] args) {

int number = 2;

switch (number) {

case 1 :

System.out.println("number는 0 보다 큽니다.");

// fall through

case 2 :

System.out.println("number는 1 보다 큽니다.");

// fall through

case 3 :

System.out.println("number는 2 보다 큽니다.");

// fall through

case 4 :

System.out.println("number는 3 보다 큽니다.");

// fall through

case 5 :

System.out.println("number는 4 보다 큽니다.");

break;

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

break;

}

}

}

일부러 break를 빼버린 곳에 (// fall through)라는 주석을 붙였다. break를 빼먹는 것은 일반적으로 실수를 의미하기 때문에 실수가 아님을 이렇게 명시적으로 보여주는 것이 좋다. 다음과 같이 명백한 경우에는 굳이 주석을 붙이지 않는 것이 좋을 것이다. 예제를 보자.

public class Example {

public static void main(String[] args) {

final int SUNDAY = 0;

final int MONDAY = 1;

final int TUESDAY = 2;

final int WEDNESDAY = 3;

final int THURSDAY = 4;

final int FRIDAY = 5;

final int SATURDAY = 6;

int dayOfTheWeek = THURSDAY;

switch (dayOfTheWeek) {

case MONDAY :

case TUESDAY :

case WEDNESDAY :

case THURSDAY :

case FRIDAY :

System.out.println("안 노는 날.");

break;

case SATURDAY :

System.out.println("반만 노는 날.");

break;

case SUNDAY :

System.out.println("노늘 날.");

break;

default :

System.out.println("요일을 잘못 넣으셨습니다.");

break;

}

}

}

결과는 다음과 같다.

안 노는 날.

마지막 case또는 default의 break는 없어도 문법적으로는 상관 없다. 어짜피 거기서 switch문은 끝나니까. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int number = 10;

switch (number) {

case 1 :

System.out.println("number는 1 입니다.");

break;

case 2 :

System.out.println("number는 2 입니다.");

break;

case 3 :

System.out.println("number는 3 입니다.");

break;

case 4 :

System.out.println("number는 4 입니다.");

break;

case 5 :

System.out.println("number는 5 입니다.");

break;

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

}

}

}

결과는 다음과 같다.

number는 1보다 작고 5보다 큽니다.

아무 문제 없이 잘 돌아간다. 그러나 다음 예제와 같은 실수를 할 확률을 높인다.

public class Example {

public static void main(String[] args) {

int number = 10;

switch (number) {

case 1 :

System.out.println("number는 1 입니다.");

break;

case 2 :

System.out.println("number는 2 입니다.");

break;

case 3 :

System.out.println("number는 3 입니다.");

break;

case 4 :

System.out.println("number는 4 입니다.");

break;

case 5 :

System.out.println("number는 5 입니다.");

break;

default :

System.out.println("number는 1보다 작고 5보다 큽니다.");

// 여기에 처음부터 break;를 넣었다면 이런 실수는 안할 것이다.

case 6 :

System.out.println("number는 6 입니다.");

}

}

}

그러므로 문법적으로는 필요가 없더라도 맨 마지막에 break를 넣어주는 것이 좋다. 그리고 위에서처럼 default를 case중간에 넣는 것도 좋지 않다. default는 맨 위나 맨 밑에 넣는 것이 좋다.

* 1. while, do-while
     1. while문의 기초

while문은 어떤 일을 반복해서 하기 위해 쓰인다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int count = 1;

while (count <= 5) {

System.out.println("while문은 반복에 쓰인다. " + count);

count++;

}

}

}

결과는 다음과 같다.

while문은 반복에 쓰인다. 1

while문은 반복에 쓰인다. 2

while문은 반복에 쓰인다. 3

while문은 반복에 쓰인다. 4

while문은 반복에 쓰인다. 5

같은 일을 break를 이용해서 할 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int count = 1;

while (true) {

System.out.println("while 문은 반복에 쓰인다. " + count);

if (count >= 5)

break;

count++;

}

}

}

반복의 개수는 사전에 정해지지 않을 수도 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int count = 1;

while (true) {

System.out.println("소문자 y를 치면 끝납니다." + count);

char in = 'n';

try {

in = (char)System.in.read();

System.in.skip(System.in.available());

} catch (Exception e) {

}

if (in == 'y')

break;

count++;

}

System.out.println("끝");

}

}

결과는 다음과 비슷할 것이다.

소문자 y를 치면 끝납니다.1

1234

소문자 y를 치면 끝납니다.2

',.p

소문자 y를 치면 끝납니다.3

aoeu

소문자 y를 치면 끝납니다.4

y

끝

if 문과 마찬가지로 while문의 조건식에도 boolean형만 사용할 수 있다.

다음 예제는 무한 루프를 보여준다. 무한 루프는 대부분의 경우 실수이다. 프로그램을 끝내려면 강제로 프로그램을 종료시켜야 한다.

public class Example {

public static void main(String[] args) {

int count = 1;

while (true) {

System.out.println("무한 루프 " + count);

count++;

}

}

}

또 다른 무한 루프 예제.

public class Example {

public static void main(String[] args) {

while (true)

;

}

}

똑같은 무한 루프 예제.

public class Example {

public static void main(String[] args) {

while (true) {

}

}

}

while에 반드시 블럭을 사용해야 하는 것은 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int count = 1;

while (count <= 5)

count++;

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

}

}

결과는 다음과 같다.

count = 6

* + 1. do-while문의 기초

do-while문은 while문과 거의 같다. 다른 점은 조건 테스트가 끝에 있기 때문에 반드시 한번 이상은 실행된다는 점이다. while문에서는 조건식이 처음부터 false값이면 한번도 실행되지 않을 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int count = 1;

do {

System.out.println("do-while문도 반복에 쓰인다. " + count);

count++;

} while (count <= 5); // 여기에 세미콜론이 있다는 것에 주의하라.

}

}

결과는 다음과 같다.

do-while문도 반복에 쓰인다. 1

do-while문도 반복에 쓰인다. 2

do-while문도 반복에 쓰인다. 3

do-while문도 반복에 쓰인다. 4

do-while문도 반복에 쓰인다. 5

do-while문은 while문으로 쉽게 대체될 수 있어서 글쓴이는 do-while문을 전혀 사용하지 않는다.

* + 1. continue

continue가 while또는 do-while문에 사용되면 바로 조건식 테스트로 넘어간다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int count = 0;

while (count < 5) {

count++;

System.out.println(count);

if (count % 2 == 0)

continue;

System.out.println(count + "는 홀수이다");

}

}

}

결과는 다음과 같다.

1

1는 홀수이다

2

3

3는 홀수이다

4

5

5는 홀수이다

continue문이 실행되면 그 아래에 있는 문장들은 건너뛰는 것이다.

* 1. for
     1. for문의 기초

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i;

for (i = 1; i <= 5; i++)

System.out.println("for문도 반복에 쓰인다. " + i);

}

}

결과는 다음과 같다.

for문도 반복에 쓰인다. 1

for문도 반복에 쓰인다. 2

for문도 반복에 쓰인다. 3

for문도 반복에 쓰인다. 4

for문도 반복에 쓰인다. 5

같은 일을 while문에 비해 라인 수도 줄고 알아보기도 더 쉽다. for문의 기본 구조는 다음과 같다.

for (초기화; 조건 ; 증감) {

실행 부분;

}

조건부분에 오는 식은 결과가 boolean이어야 한다.

for문이 실행되는 순서는 다음과 같다.

1. 먼저 초기화 부분을 실행한다.
2. 조건을 테스트한다.
3. 조건이 거짓이면 for문에서 빠져나온다.
4. 조건이 참이면 실행부분을 실행한다.
5. break를 만나면 for문에서 빠져나온다(제어문이 중첩될 때는 문제가 복잡해진다).
6. continue를 만나면 그 아래의 문장을 무시하고 7번으로.
7. 증감을 실행한다.
8. 2번으로.

for문의 break, continue는 while과 비슷하다. 다음 예제는 break와 continue의 사용법을 보여준다.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10 ; i++) {

System.out.println(i);

if (i > 4)

break;

if (i > 2)

continue;

System.out.println(i + "는 2보다 작거나 같다.");

}

}

}

결과는 다음과 같다.

0

0는 2보다 작거나 같다.

1

1는 2보다 작거나 같다.

2

2는 2보다 작거나 같다.

3

4

5

* + 1. for문의 유연성

초기화, 조건, 증감, 실행부분 중 어떤 것도 생략될 수 있다. 조건이 생략되면 조건이 참인 것으로 간주된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a;

for (a = 1; a <= 3; a++)

System.out.println("for문도 반복에 쓰인다. " + a);

for (int b = 1; b <= 3; b++)

System.out.println("변수를 for문 초기화 부분에 선언할 수도 있다. " + b);

int c = 1;

for (; c <= 3; c++)

System.out.println("초기화 부분이 생략될 수도 있다. " + c);

for (int d = 1; ; d++) {

System.out.println("조건 부분이 생략될 수도 있다. " + d);

if (d >= 3)

break;

}

for (int e = 1; e <= 3; ) {

System.out.println("증감 부분이 생략될 수도 있다. " + e);

e++;

}

int f = 1;

for ( ; f <= 3; ) {

System.out.println("차라리 while문을 써라. " + f);

f++;

}

int g = 1;

for ( ; ; ) {

System.out.println("while(true)와 같다. " + g);

g++;

if (g > 3)

break;

}

for (int h = 1; h <= 3; System.out.println("사회에 불만 있냐? " + h), h++)

;

}

}

결과는 다음과 같다.

for문도 반복에 쓰인다. 1

for문도 반복에 쓰인다. 2

for문도 반복에 쓰인다. 3

변수를 for문 초기화 부분에 선언할 수도 있다. 1

변수를 for문 초기화 부분에 선언할 수도 있다. 2

변수를 for문 초기화 부분에 선언할 수도 있다. 3

초기화 부분이 생략될 수도 있다. 1

초기화 부분이 생략될 수도 있다. 2

초기화 부분이 생략될 수도 있다. 3

조건 부분이 생략될 수도 있다. 1

조건 부분이 생략될 수도 있다. 2

조건 부분이 생략될 수도 있다. 3

증감 부분이 생략될 수도 있다. 1

증감 부분이 생략될 수도 있다. 2

증감 부분이 생략될 수도 있다. 3

차라리 while문을 써라. 1

차라리 while문을 써라. 2

차라리 while문을 써라. 3

while(true)와 같다. 1

while(true)와 같다. 2

while(true)와 같다. 3

사회에 불만 있냐? 1

사회에 불만 있냐? 2

사회에 불만 있냐? 3

어떤 식으로 해도 문법적으로 틀리지 않지만 알아보기 쉬운 것으로 하는 것이 좋다.

초기화 부분에는 여러 변수를 만들어서 초기화 할 수 있다. 그리고 증감 부분에는 1씩 증가시키는 것 이외에도 여러가지를 할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 1, j = 2; i <= 5 ; i++, j \*= j) {

System.out.println(i + " : " + j);

}

}

}

결과는 다음과 같다.

1 : 2

2 : 4

3 : 16

4 : 256

5 : 65536

초기화, 조건, 증감 부분에 메쏘드를 부를 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = 0;

for (double d = Math.sqrt(2); lessThanSix(i) ; i++, d = Math.sqrt(d)) {

System.out.println(i + " : " + d);

}

}

public static boolean lessThanSix(int number) {

return (number < 6);

}

}

결과는 다음과 같다.

0 : 1.4142135623730951

1 : 1.189207115002721

2 : 1.0905077326652577

3 : 1.0442737824274138

4 : 1.0218971486541166

5 : 1.0108892860517005

위의 예제들은 for문을 유연하게 사용할 수 있음을 보여주는 예제들이었다. 그 중의 일부는 좋은 사용법이 아니다. 중요한 것은 어떻게 하면 알아보기 쉽게 하는가이다. 사람들이 많은 경우에 while문보다 for문을 선호하는 이유는 그렇게 하는 것이 알아보기 쉽기 때문이다. 만약 for문의 유연성을 백분 활용한다는 것이 더 알아보기 힘든 코드가 된다면 그것은 유연성의 활용이 아니라 남용이다. 예를 들어 다음과 같이 쓸 수도 있지만 당신의 코드를 보는 모든 사람들이 싫어할 것이다.

public class Example {

public static void main(String[] args) {

int i = 0;

for ( ; ; System.out.println("for문의 남용 " + i)) {

i++;

if (i > 3)

break;

}

}

}

결과는 다음과 같다.

for문의 남용 1

for문의 남용 2

for문의 남용 3

초기화, 조건, 증감 부분은 각 의미에 맞게 사용하는 것이 좋다.

* + 1. 실수를 줄이는 법

for문에서는 초기화 부분에서 변수를 선언 할 수 있다. 그것을 적극 이용하는 것이 좋다. 때에 따라 그렇게 할 수 없는 경우도 있지만 가능하다면 거기서 선언을 하도록 하자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i, j;

for (i = 0; i < 4; i++) {

for (j = 0; j < 5; j++) {

System.out.print("\*");

}

System.out.println();

}

}

}

결과는 다음과 같다.

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

만약 위의 예제를 만든다는 것이 실수로 다음과 같이 했다면

public class Example {

public static void main(String[] args) {

int i, j;

for (i = 0; i < 4; i++) {

for (i = 0; i < 5; i++) { // 실수

System.out.print("\*");

}

System.out.println();

}

}

}

결과는 다음과 같을 것이다.

\*\*\*\*\*

위의 예제를 다음과 같이 바꿔보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 4; i++) {

for (int i = 0; i < 5; i++) { // 실수를 컴파일러가 잡아준다.

System.out.print("\*");

}

System.out.println();

}

}

}

컴파일타임 오류가 난다. 이 예제를 보면 어떤 바보가 이런 실수를 할까라는 생각이 들것이다. 그러나 위에서는 프로그램이 간단했지만 복잡한 알고리즘을 구현하는 수백 라인이 넘는 메쏘드에서는 충분히 할 만한 실수다. 위의 예에서(버그 없는 첫번째) i는 바깥쪽 for문 안에서만 쓰인다. 그리고 j는 안쪽 for문 안에서만 쓰인다. 그럼에도 i와 j는 메쏘드 전체에 알려져 있다. 즉 알 필요가 없는 곳에서도 i와 j의 존재를 알고 있는 것이다. for문의 초기화 하는 부분에 그 변수를 선언했다면 for 문 내에서만 그 존재를 알 수 있다. 이것은 객체 지향 프로그래밍을 배우다 보면 계속 나오는 원리와 연결된다. 프로그램 내에서 각 존재들을 최대한 다른 존재에 대해서 모르는 것이 좋다. 클래스가 하나의 객체를 캡슐화(encapsulation)하듯이 작은 for문의 블록도 최대한 독립된 존재가 되게 하는 것이 좋다. 이것은 다른 지역 변수를 선언할 때도 마찬가지다. 최대한 안쪽의 블록에 지역 변수를 선언하라.

* + 1. 증감하는 숫자를 되도록이면 정수를 사용하라 – 좌표값 구하는 예제

증감하는 숫자는 가능하면 실수보다는 정수를 사용하는 것이 좋다. 왜냐하면 실수는 오차가 있을 수 있기 때문이다.

다음 예제를 실행해 보자. 이것은 바둑판을 그리는 예제이다.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(400, 400);

}

public static void main(String[] args) {

new Example().setVisible(true);

}

public void paint(Graphics g) {

double x1 = 50.0;

double x2 = 351.0;

double y1 = 50.0;

double y2 = 351.0;

int num = 10;

for (double x = x1; x <= x2; x += (x2 - x1) / num) {

g.drawLine((int)x, (int)y1, (int)x, (int)y2); // 세로선

}

for (double y = y1; y <= y2; y += (y2 - y1) / num) {

g.drawLine((int)x1, (int)y, (int)x2, (int)y); // 가로선

}

}

}

오른쪽 끝줄과 왼쪽 끝줄이 안 그려진다. 그 이유를 알아보자.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(400, 400);

}

public static void main(String[] args) {

new Example().setVisible(true);

}

public void paint(Graphics g) {

double x1 = 50.0;

double x2 = 351.0;

double y1 = 50.0;

double y2 = 351.0;

int num = 10;

System.out.println("(x2 - x1) / num = " + (x2-x1)/num);

double x;

for (x = x1; x <= x2; x += (x2 - x1) / num) {

g.drawLine((int)x, (int)y1, (int)x, (int)y2); // 세로선

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

}

System.out.println(x);

for (double y = y1; y <= y2; y += (y2 - y1) / num) {

g.drawLine((int)x1, (int)y, (int)x2, (int)y); // 가로선

}

}

}

결과는 다음과 비슷할 것이다.

(x2 - x1) / num = 30.1

x = 50.0

x = 80.1

x = 110.19999999999999

x = 140.29999999999998

x = 170.39999999999998

x = 200.49999999999997

x = 230.59999999999997

x = 260.7

x = 290.8

x = 320.90000000000003

351.00000000000006

(x2 - x1) / num = 30.1

x = 50.0

x = 80.1

x = 110.19999999999999

x = 140.29999999999998

x = 170.39999999999998

x = 200.49999999999997

x = 230.59999999999997

x = 260.7

x = 290.8

x = 320.90000000000003

351.00000000000006

351.00000000000006가 351보다 0.00000000000006만큼 크기 때문이었다. 이럴 때 정수를 사용했으면 아무런 문제가 발생하지 않았을 것이다. 증감하는 변수를 정수로 바꾼 예제이다.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(400, 400);

}

public static void main(String[] args) {

new Example().setVisible(true);

}

public void paint(Graphics g) {

int x1 = 50;

int x2 = 351;

int y1 = 50;

int y2 = 351;

int num = 10;

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

int x = (int)(x1 + (double)(x2-x1)/ num\*i);

g.drawLine(x, y1, x, y2); // 세로선

}

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

int y = (int)(y1 + (double)(y2-y1)/ num\*i);

g.drawLine(x1, y, x2, y); // 가로선

}

}

}

바둑판이 잘 그려질 것이다. 이 예제중

int x = (int)(x1 + (double)(x2-x1)/ num\*i);

에서 (double)을 빼고 실행해 보자. 가장자리가 좀 삐져나온다. 왜 그렇게 되는지는 그 이유는 당신이 알아보기 바란다.

* 1. 제어문의 중첩
     1. 구구단

여러 제어문은 마음대로 중첩될 수 있다. 이것은 강력한 프로그래밍 수단이다.

고전적인(classical) 예제 구구단을 만들어보자. for문이나 while문을 중첩해서 만들 수 있다. 여기서는 for문을 이용한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 2; i <= 4; i++) { // 지면 관계상 4단까지만.

for (int j = 1; j <= 9; j++) {

System.out.println(i + " \* " + j + " = " + (i\*j));

}

System.out.println();

}

}

}

결과는 다음과 같다.

2 \* 1 = 2

2 \* 2 = 4

2 \* 3 = 6

2 \* 4 = 8

2 \* 5 = 10

2 \* 6 = 12

2 \* 7 = 14

2 \* 8 = 16

2 \* 9 = 18

3 \* 1 = 3

3 \* 2 = 6

3 \* 3 = 9

3 \* 4 = 12

3 \* 5 = 15

3 \* 6 = 18

3 \* 7 = 21

3 \* 8 = 24

3 \* 9 = 27

4 \* 1 = 4

4 \* 2 = 8

4 \* 3 = 12

4 \* 4 = 16

4 \* 5 = 20

4 \* 6 = 24

4 \* 7 = 28

4 \* 8 = 32

4 \* 9 = 36

* + 1. 팩토리얼 : 100! 구하기

Factorial이란 숫자를 연속해서 곱하는 것을 말한다. 예를 들어

10! = 1 \* 2 \* 3 \* 4 \* 5 \* 6 \* 7 \* 8 \* 9 \* 10

이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

long factorial = 1;

for (int i = 1; i <= 100; i++)

factorial \*= i;

System.out.println(factorial);

}

}

결과는 다음과 같을 것이다.

0

왜 이런 희한한 결과가 나왔는지는 예제를 다음과 같이 고쳐서 실행해 보면 대충 알 수 있다. 문제는 long형이 감당하기에는 100!이 너무 크다는데 있다.

public class Example {

public static void main(String[] args) {

long factorial = 1;

for (int i = 1; i <= 100; i++) {

factorial \*= i;

System.out.print(factorial + ", ");

}

}

}

결과는 다음과 같을 것이다.

1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800, 479001600, 6227020800, 87178291200, 1307674368000, 20922789888000, 355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000, -4249290049419214848, -1250660718674968576, 8128291617894825984, -7835185981329244160, 7034535277573963776, -1569523520172457984, -5483646897237262336, -5968160532966932480, -7055958792655077376, -8764578968847253504, 4999213071378415616, -6045878379276664832, 3400198294675128320, 4926277576697053184, 6399018521010896896, 9003737871877668864, 1096907932701818880, 4789013295250014208, 2304077777655037952, -70609262346240000, -2894979756195840000, 7538058755741581312, -7904866829883932672, 2673996885588443136, -8797348664486920192, 1150331055211806720, -1274672626173739008, -5844053835210817536, 8789267254022766592, -3258495067890909184, -162551799050403840, -8452693550620999680, -5270900413883744256, -7927461244078915584, 6711489344688881664, 6908521828386340864, 6404118670120845312, 2504001392817995776, 162129586585337856, -8718968878589280256, 3098476543630901248, 7638104968020361216, 1585267068834414592, -9223372036854775808, -9223372036854775808, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

해결책은 double형이다.

public class Example {

public static void main(String[] args) {

double factorial = 1;

for (int i = 1; i <= 100; i++)

factorial \*= i;

System.out.print(factorial);

}

}

결과는 다음과 비슷할 것이다.

9.33262154439441E157

그런데 만약 근사값이 아니라 정확한 값을 원한다면 어떨까? double로는 안된다.

다음 예제는 int 배열을 이용해서 그 값을 구한다. 코드를 보기 전에 스스로 한 번 짜보기 바란다.

public class Example {

public static void main(String[] args) {

int[] factorial = new int[200];

factorial[0] = 1;

for (int i = 1; i < factorial.length; i++)

factorial[i] = -1;

for (int i = 1; i <= 100; i++) {

for (int j = 0; j < factorial.length; j++) {

if (factorial[j] == -1)

break;

factorial[j] \*= i;

}

for (int j = 0; j < factorial.length; j++) {

if (factorial[j] == -1)

break;

if (factorial[j] / 10 != 0) {

if (factorial[j+1] == -1)

factorial[j+1] = 0;

factorial[j+1] += factorial[j] / 10;

factorial[j] %= 10;

}

}

}

int top = -1;

for (int i = 0; i < factorial.length; i++)

if (factorial[i] == -1) {

top = i - 1;

break;

}

for (int i = top; i >= 0; i--)

System.out.print((char)(factorial[i] + '0'));

}

}

결과는 다음과 같을 것이다.

93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

사실 이런 상황을 위해서 BigDecimal, BigInteger, BitSet등의 클래스가 있다. 그것을 다루는 부분을 참조하라.

* + 1. 숫자 지그재그

코드를 보기 전에 결과를 보고 스스로 만들어 보기 바란다.

public class Example {

public static void main(String[] args) {

int number = 1;

for (int i = 0; i < 10; i++) {

if (i % 2 == 1)

number += 9;

else

if (number != 1)

number += 11;

for (int j = 0; j < 10; j++) {

if (number < 10)

System.out.print(" ");

System.out.print(number + " ");

if (i % 2 == 0)

number++;

else number--;

}

System.out.println();

}

}

}

결과는 다음과 같을 것이다.

1 2 3 4 5 6 7 8 9 10

20 19 18 17 16 15 14 13 12 11

21 22 23 24 25 26 27 28 29 30

40 39 38 37 36 35 34 33 32 31

41 42 43 44 45 46 47 48 49 50

60 59 58 57 56 55 54 53 52 51

61 62 63 64 65 66 67 68 69 70

80 79 78 77 76 75 74 73 72 71

81 82 83 84 85 86 87 88 89 90

100 99 98 97 96 95 94 93 92 91

또는 다음과 같은 좀더 알아보기 쉬운 코드로 짤 수도 있다.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

int number;

if (i % 2 == 0)

number = i \* 10 + j + 1;

else

number = (i + 1) \* 10 - j;

if (number < 10)

System.out.print(" ");

System.out.print(number + " ");

}

System.out.println();

}

}

}

* + 1. 숫자 사선

코드를 보기 전에 결과를 보고 스스로 만들어 보기 바란다.

public class Example {

public static void main(String[] args) {

for (int i = 0, start = 1; i < 10; i++, start += i + 1) {

int number = start;

for (int j = 0, k = i; j < 10; j++) {

if (j != 0)

number += k;

if (i + j < 10) {

k++;

if (number < 10)

System.out.print(" ");

System.out.print(" " + number);

}

}

System.out.println();

}

}

}

결과는 다음과 같을 것이다.

1 2 4 7 11 16 22 29 37 46

3 5 8 12 17 23 30 38 47

6 9 13 18 24 31 39 48

10 14 19 25 32 40 49

15 20 26 33 41 50

21 27 34 42 51

28 35 43 52

36 44 53

45 54

55

* + 1. 숫자 소용돌이

이번에는 조금 복잡한 프로그램을 보자. 숫자로 소용돌이 모양을 만드는 것이다. 먼저 실행 결과를 보기 바란다. 그리고 한번 시도해 본 다음 소스를 봐도 늦지 않다. 다음 예제는 if, switch, for, while문을 중첩한 예이다.

public class Example {

public static void main(String[] args) {

final int RIGHT = 0;

final int DOWN = 1;

final int LEFT = 2;

final int UP = 3;

int[][] spiral = new int[10][10];

int x = 0;

int y = 0;

int direction = RIGHT;

int count = 1;

while (count <= 100) {

spiral[x][y] = count;

switch (direction) {

case RIGHT :

x++;

if ( (x>9) || (spiral[x][y]!=0) ) {

x--;

y++;

direction++;

direction %= 4;

}

break;

case DOWN :

y++;

if ((y>9) || (spiral[x][y]!=0) ) {

y--;

x--;

direction++;

direction %= 4;

}

break;

case LEFT :

x--;

if ( (x<0) || (spiral[x][y]!=0) ) {

x++;

y--;

direction++;

direction %= 4;

}

break;

case UP :

y--;

if ( (y<0) || (spiral[x][y]!=0) ) {

y++;

x++;

direction++;

direction %= 4;

}

break;

}

count++;

}

for (int i = 0; i < 10; i++) {

for (int j = 0; j < 10; j++) {

if (spiral[j][i] < 100)

System.out.print(" ");

if (spiral[j][i] < 10)

System.out.print(" ");

System.out.print(spiral[j][i] + " ");

}

System.out.println();

}

}

}

결과는 다음과 같다.

1 2 3 4 5 6 7 8 9 10

36 37 38 39 40 41 42 43 44 11

35 64 65 66 67 68 69 70 45 12

34 63 84 85 86 87 88 71 46 13

33 62 83 96 97 98 89 72 47 14

32 61 82 95 100 99 90 73 48 15

31 60 81 94 93 92 91 74 49 16

30 59 80 79 78 77 76 75 50 17

29 58 57 56 55 54 53 52 51 18

28 27 26 25 24 23 22 21 20 19

* + 1. 레이블이 없는 break와 continue

레이블이 없는 break와 연결될 수 있는 제어문에는 for, while, do-while, switch이 있다. if문은 break와 연결되지 않는다. 그러므로 다음 예제는 컴파일 에러가 난다.

public class Example {

public static void main(String[] args) {

if (true) {

break;

}

}

}

break를 만나면 그 break를 포함하는 가장 안쪽의 for, while, do-while, switch를 찾아서 그 제어문에서 빠져나온다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 1; i < 10; i++) {

for (int j = 1; j < 10; j++) {

System.out.print(j);

if (j >= i)

break; // j for문에서 빠져나온다.

}

System.out.println();

}

}

}

결과는 다음과 같다.

1

12

123

1234

12345

123456

1234567

12345678

123456789

continue는 반복문(for, while, do-while)과만 연결된다. continue의 의미를 생각할 때 당연하다.

* 1. 레이블
     1. 레이블의 필요성

옛날엔 goto라는 키워드가 있었다. 만약 이 키워드를 좋아하게 된다면 스파게티 프로그램(제어흐름이 스파케티처럼 이리저리 꼬인 프로그램)을 잘 만들수 있을 것이다. 그러므로 사람들은 이 키워드를 별로 좋아하지 않는다. 그러나 C언어에서 goto의 쓰임이 정당화될 수 있는 상황이 있었으니 바로 다음과 같은 경우이다. 예제를 보자.

public class Example {

public static void main(String[] args) {

boolean bOverFifty = false;

for (int i = 1; i <= 5; i++) {

for (int j = 1; j <= 5; j++) {

for (int k = 1; k <= 5; k++) {

System.out.print(i + "\*" + j + "\*" + k + "=" + (i\*j\*k) + " ");

if (i \* j \* k > 50) {

bOverFifty = true;

break;

}

}

if (bOverFifty)

break;

System.out.println();

}

if (bOverFifty)

break;

}

System.out.println("the end");

}

}

결과는 다음과 같다.

1\*1\*1=1 1\*1\*2=2 1\*1\*3=3 1\*1\*4=4 1\*1\*5=5

1\*2\*1=2 1\*2\*2=4 1\*2\*3=6 1\*2\*4=8 1\*2\*5=10

1\*3\*1=3 1\*3\*2=6 1\*3\*3=9 1\*3\*4=12 1\*3\*5=15

1\*4\*1=4 1\*4\*2=8 1\*4\*3=12 1\*4\*4=16 1\*4\*5=20

1\*5\*1=5 1\*5\*2=10 1\*5\*3=15 1\*5\*4=20 1\*5\*5=25

2\*1\*1=2 2\*1\*2=4 2\*1\*3=6 2\*1\*4=8 2\*1\*5=10

2\*2\*1=4 2\*2\*2=8 2\*2\*3=12 2\*2\*4=16 2\*2\*5=20

2\*3\*1=6 2\*3\*2=12 2\*3\*3=18 2\*3\*4=24 2\*3\*5=30

2\*4\*1=8 2\*4\*2=16 2\*4\*3=24 2\*4\*4=32 2\*4\*5=40

2\*5\*1=10 2\*5\*2=20 2\*5\*3=30 2\*5\*4=40 2\*5\*5=50

3\*1\*1=3 3\*1\*2=6 3\*1\*3=9 3\*1\*4=12 3\*1\*5=15

3\*2\*1=6 3\*2\*2=12 3\*2\*3=18 3\*2\*4=24 3\*2\*5=30

3\*3\*1=9 3\*3\*2=18 3\*3\*3=27 3\*3\*4=36 3\*3\*5=45

3\*4\*1=12 3\*4\*2=24 3\*4\*3=36 3\*4\*4=48 3\*4\*5=60 the end

세 수의 곱을 표시하다가 그 곱이 50이 넘으면 끝나는 프로그램이다. 문제는 세개의 for문에서 빠져나오는데 너무 고생을 한다는 것이다. 빠져나오는데 하나의 boolean형 변수, 세개의 break, 두개의 if문이 필요했다. 이럴때 goto문 하나면 해결될 수 있다. 그러나 불행하게도(?) 자바에는 goto문이 없다. 그러나 이 문제를 해결할 방법이 있으니 그것이 바로 레이블이 있는 break문(labeled break)이다. 위의 예제를 다음과 바꿀 수 있다. 물론 실행 결과는 똑같다.

public class Example {

public static void main(String[] args) {

ForI :

for (int i = 1; i <= 5; i++) {

for (int j = 1; j <= 5; j++) {

for (int k = 1; k <= 5; k++) {

System.out.print(i + "\*" + j + "\*" + k + "=" + (i\*j\*k) + " ");

if (i \* j \* k > 50)

break ForI;

}

System.out.println();

}

}

System.out.println("the end");

}

}

코드가 훨씬 간결해졌다.

* + 1. 레이블이 있는 break문의 기초

레이블은 break와 연결될 수 있는 for, while, do-while, switch문에 붙일 수 있다. 거기다가 if문에도 붙일 수 있다. 심지어 블럭에도 붙일 수가 있다. 이 예제는 if문에 붙인 레이블의 예이다.

public class Example {

public static void main(String[] args) {

LabelIf :

if (true) {

for (int i = 1; i < 10; i++) {

System.out.println(i);

if (i > 3)

break LabelIf;

}

System.out.println("End of if"); // 실행되지 않는다.

}

}

}

결과는 다음과 같다.

1

2

3

4

이번에는 블럭에 레이블을 붙인 예이다.

public class Example {

public static void main(String[] args) {

LabelBlock :

{

for (int i = 1; i < 10; i++) {

System.out.println(i);

if (i > 3)

break LabelBlock;

}

System.out.println("End of block"); // 실행되지 않는다.

}

}

}

결과는 위와 같다.

다음 예제는 레이블이 있는 break문을 사용하는 희한한 방법이다. 이 예제의 목적은 레이블의 문법을 익히자는 것이지 결코 이런식으로 코딩하라는 것은 아니다.

public class Example {

public static void main(String[] args) {

A :

break A; // 이런 어이없는 labeled break문도 가능하다.

for (int i = 0; i < 5; i++)

B : {

System.out.println(i);

if (i > 2)

break B; // for문에서 빠져나오지 못한다.

System.out.println(i + "는(은) 2보다 크지 않다.");

}

}

}

결과는 다음과 같다.

0

0는(은) 2보다 크지 않다.

1

1는(은) 2보다 크지 않다.

2

2는(은) 2보다 크지 않다.

3

4

* + 1. 레이블이 있는 continue문의 기초

continue문에서는 for, while, do-while에 연결된 레이블만 쓸 수 있다. 그러므로 다음 예제는 당연히 컴파일 에러가 난다.

public class Example {

public static void main(String[] args) {

Label :

if (true) {

continue Label;

}

}

}

다음 예제는 레이블이 있는 continue문의 사용법을 보여준다.

public class Example {

public static void main(String[] args) {

ForI :

for (int i = 1; i < 10; i++)

for (int j = 1; j < 10; j++) {

System.out.print(j);

if (j >= i) {

System.out.println();

continue ForI;

}

}

}

}

결과는 다음과 같다.

1

12

123

1234

12345

123456

1234567

12345678

123456789

* + 1. 레이블과 이름 충돌

레이블을 선언할 때는 콜론을 붙인다. 그리고 break와 continue 키워드 다음에만 쓰인다. 그렇기 때문에 다른 이름과 같아도 충돌이 일어나지 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

i :

for (int i = 1; i < 10; i++)

for (int j = 1; j < 10; j++) {

System.out.print(j);

if (j >= i) {

System.out.println();

continue i;

}

}

}

}

i가 레이블 이름과 변수명으로 다 쓰였음에도 무사히 컴파일되어 실행되는 것을 볼 수 있다.

다음 프로그램을 컴파일해 보자.

public class Example {

public static void main(String[] args) {

I :

for (int i = 0; i < 10 ; i++)

break I;

I :

for (int i = 0; i < 10; i++)

break I;

}

}

에러 없이 컴파일된다. 두개의 break문이 무엇을 할지 모호하지 않다. 이것은 i가 두개의 for문에 쓰였지만 모호하지 않은것과 비슷하다. i를 이런식으로 재활용(?)하는 것은 공인된 프로그래밍 스타일이다. 레이블 I를 재활용하는 것이 정당화될 수 있을지는 의문이다.

* 1. 도달할 수 없는 문장(unreachable statement)
     1. 도달할 수 없는 문장이란?

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

return;

System.out.println("나는 뭐야!!!"); // unreachable statement

}

}

컴파일타임 오류가 날 것이다.

return다음에 있는 문장은 절대로 실행될 수 없는 것이 분명하다. 위와 같은 프로그램은 분명 프로그래머가 실수를 한 것이다. 이런 실수를 컴파일 타임에 더 많이 고치기 위해서 컴파일러는 컴파일을 거부한다. 여기까지는 당연해 보인다. 그러나 문제는 그렇게 단순하지 않다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

if (true)

return;

System.out.println("나는 뭐야!!!"); // unreachable statement ???

}

}

위의 예에서도 if문 아래의 문장이 실행되지 않는 것은 자명하다. 그러나 컴파일이 잘 된다. 이제부터 컴파일러는 어떤 것을 도달할 수 없는 문장이라고 생각하는지 알아보자.

* + 1. if 문

다음 예를 보자.

public class Example {

public static void main(String[] args) {

if (false)

System.out.println("나는 절대 실행 안돼 !!!");

}

}

잘 컴파일 된다. 왜 이렇게 만들었을까? 거기에는 이유가 있다. 프로그램을 만들다 보면 테스트할 때는 필요하지만 실제 실행할 때는 없어야 하는 코드가 있다. 프로그램이 크다 보면 그런 코드의 양도 많아진다. 그것을 일일이 주석으로 만들어서 없애는 것(comment out)은 너무 힘들다. 그럴 때는 이런식으로 해결하는 것이 좋다. 예제를 보자.

public class Example {

public static void main(String[] args) {

if (ProgramState.TEST) {

System.out.println("나는 테스트에 필요한 코드.");

}

}

}

class ProgramState {

//public static final boolean TEST = true;

public static final boolean TEST = false;

}

이런 식의 코딩이 가능하게 하기 위해 버그를 잡을 가능성을 약간 줄이지만 도달할 수 없는 문장이 있는 if문은 예외적으로 컴파일이 된다.

if문이 이런 면에서 예외이기 때문에 다음 프로그램은 잘 컴파일된다.

public class Example {

public static void main(String[] args) {

if (true)

return;

System.out.println("실행이 안되더라도 컴파일 에러는 아님");

for ( ; ; ) {

if (true)

break;

System.out.println("실행이 안되더라도 컴파일 에러는 아님");

}

}

}

* + 1. while, do-while, for

다음 예제는 컴파일이 안될 것이다.

public class Example {

public static void main(String[] args) {

while (false)

System.out.println("나는 실행이 안돼"); // Error

for ( ; false; )

System.out.println("나는 실행이 안돼"); // Error

}

}

물론 다음 예제도 컴파일이 안된다.

public class Example {

public static void main(String[] args) {

while (-1 > 0)

System.out.println("나는 실행이 안돼"); // Error

for ( ; -1 > 0; )

System.out.println("나는 실행이 안돼"); // Error

}

}

그러나 다음 예제는 컴파일이 된다. 왜냐하면 변수를 사용했기 때문이다.

public class Example {

public static void main(String[] args) {

int i = -1;

while (i > 0)

System.out.println("나는 실행이 안돼");

for ( ; i > 0; )

System.out.println("나는 실행이 안돼");

}

}

다음 예제는 컴파일이 안된다. break아래에 있는 문장이 도달할 수 없기 때문이다.

public class Example {

public static void main(String[] args) {

while (true) {

break;

System.out.println("나는 실행이 안돼"); // Error

}

do {

break;

System.out.println("나는 실행이 안돼"); // Error

} while (true);

for ( ; ; ) {

break;

System.out.println("나는 실행이 안돼"); // Error

}

}

}

다음 예제도 컴파일이 안된다. 무한 루프가 돌아서 그 밑에 있는 문장이 실행되지 않기 때문이다.

public class Example {

public static void main(String[] args) {

while (true)

;

System.out.println("나는 실행이 안돼"); // Error

do

;

while(true);

System.out.println("나는 실행이 안돼"); // Error

for ( ; ; )

;

System.out.println("나는 실행이 안돼"); // Error

}

}

* + 1. switch 문

다음 예제는 컴파일이 잘 된다.

public class Example {

public static void main(String[] args) {

switch (1) {

case 2 :

System.out.println("실행이 안되더라도 컴파일 에러는 아님");

}

}

}

1. 기본형
   1. 기본형(primitive type)이란
      1. 기본형과 참조형(reference type)

자바의 형(type)에는 기본형과 참조형이 있다. 참조형 변수는 객체를 가리키고 기본형 변수는 정수, 실수, 논리값을 저장한다. 기본형과 참조형은 엄격히 구분된다.

* + 1. 자바 기본형의 특징

자바의 기본형의 크기는 일정하다. 어떤 운영체제 또는 CPU에서 돌더라도 예를 들어 int 형의 크기는 4 바이트(32비트)이다. 이렇게 함으로써 속도에서 약간은 느릴 수 있으나 귀찮은 porting 문제가 사라졌다. 자바의 철학인 플랫폼 독립성에 부합하는 특징이다.

* 1. 정수형
     1. byte

byte형의 범위를 알아보자. 최소값은 –128이고 최대값은 127이다. 크기는 1바이트(8비트)이다.

비트 패턴과 그것의 해석(비트 패턴에 해당하는 정수값)의 관계에 대해 알아보자. 2의 보수 (two’s complement)에 대해서는 다른 책을 참조하도록 하고 다음 예제의 실행 결과를 보면 대충 이해가 갈 것이다. 다음 예제를 실행해보자. 코드가 이해가 가지 않으면 다음 기회에 이해하도록 하고 여기서 중요한 것은 실행 결과다. 실행 결과가 길어서 중간부분을 생략했다. 전체를 보려면 직접 실행해 보기 바란다. char를 제외한 다른 정수형도 크기가 다를뿐 모두 마찬가지이다.

public class Example {

public static void main(String[] args) {

for (int i = -128; i < 128; i++) {

byte b = (byte)i;

String strHex = Integer.toHexString(b);

if (strHex.length() >= 2)

strHex = strHex.substring(strHex.length() - 2, strHex.length());

else

strHex = "0" + strHex;

strHex = strHex.toUpperCase();

System.out.print(strHex + " : ");

String strBinary = Integer.toBinaryString(b);

if (strBinary.length() >= 8) {

strBinary = strBinary.substring(strBinary.length() - 8, strBinary.length());

} else {

int length = strBinary.length();

for (int j = 0; j < 8 - length; j++)

strBinary = "0" + strBinary;

}

System.out.print(strBinary + " : ");

System.out.println(b);

}

}

}

결과는 다음과 같을 것이다.

80 : 10000000 : -128

81 : 10000001 : -127

82 : 10000010 : -126

83 : 10000011 : -125

- 중간 생략 -

FD : 11111101 : -3

FE : 11111110 : -2

FF : 11111111 : -1

00 : 00000000 : 0

01 : 00000001 : 1

02 : 00000010 : 2

03 : 00000011 : 3

- 중간 생략 -

7C : 01111100 : 124

7D : 01111101 : 125

7E : 01111110 : 126

7F : 01111111 : 127

byte형의 최대값인 127에 1을 더하면 어떻게 될까? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 127;

b++;

System.out.println(b);

b = -128;

b--;

System.out.println(b);

b = 100;

b += 100;

System.out.println(b);

}

}

결과는 다음과 같을 것이다.

-128

127

-56

정수형에서는 표현범위가 넘으면 예외상황이 발생하지 않고 나름대로의 규칙으로 변환한다. 이런 경우는 대부분 버그이다. 정수형을 사용할 때 이런 상황이 발생하지 않도록 조심해야 한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (byte b = -128; b <= 127; b++)

System.out.println(b);

}

}

원래 의도는 –128부터 127까지 한번 보여주려고 한 것이다. 그러나 무한 루프가 돌 것이다.

* + 1. short

short 형의 범위를 알아보자. 최소값은 – 32767이고 최대값은 32767이다. 크기는 2바이트(16비트)이다.

* + 1. int

int형의 범위를 알아보자. 최소값은 – 2147483647이고 최대값은 2147483647이다. 크기는 4바이트(32비트)이다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = Integer.MIN\_VALUE;

System.out.println(i);

i = -i;

System.out.println(i);

}

}

결과는 다음과 같을 것이다.

-2147483648

-2147483648

-2147483648에 –1을 곱하면 2147483648이지만 int형으로는 표현할 수 없기 때문에 좀 이상하지만 그냥 –2147483648이 되는 것이다.

* + 1. long

long형의 범위를 알아보자. 최소값은 – 9223372036854775807이고 최대값은 9223372036854775807이다. 크기는 8바이트(64비트)이다.

* + 1. 정수형 리터럴

정수형 리터럴에는 10진수, 16진수, 8진수가 쓰인다. 2진수는 있을 법도 한데 쓰이지 않는다.

16진수 표시는 0x또는 0X로 시작하는 숫자이다(0은 숫자임). 16진수를 표현하기 위해 A, B, C, D, E, F가 쓰인다. 소문자를 써도 된다.

8진수는 0으로 시작하는 숫자이다.

16진수나 8진수가 아니라면 10진수 표기법이다.

0의 10진수, 8진수, 16진수 표기법은 다음과 같으나 실제로는 구별이 의미가 없다. 0, 00, 0X0.

public class Example {

public static void main(String[] a) {

System.out.println(10); // 10진수

System.out.println(010); // 8진수

System.out.println(0x10); // 16진수

System.out.println(0x1F); // 16진수

}

}

위의 예제를 실행시키면 다음과 같은 결과가 나올 것이다. 결과로 나온 값은 10진수이다.

10

8

16

31

정수형 리터럴은 숫자 다음에 대문자L 또는 소문자 l을 쓰면 long형이 되고 아니면 int형이 된다. 소문자 l은 숫자1과 구별하기 힘들기 때문에 대문자 L을 쓰는 것이 좋다.

L(또는 l)을 붙이지 않은 수가 int형의 범위를 초과하면 컴파일타임 오류가 난다.

public class Example {

public static void main(String[] a) {

long l = 2147483648;

}

}

위의 예에서 숫자 2147483648이 int형의 범위를 초과했기 때문에 컴파일타임 오류가 난다. 그 문장을 다음과 같이 고치면 무사히 컴파일될 것이다.

long l = 2147483648L;

L(또는 l)을 붙인 숫자가 long형의 범위를 초과하면 마찬가지로 컴파일되지 않는다.

* + 1. char

char형의 범위를 알아보자. 최소값은 0이고 최대값은 65535이다. 크기는 2바이트(16비트)이다. char형은 유니 코드를 나타내기 위한 목적으로 만들어졌다. 32비트 유니코드가 쓰여진다면 char형의 운명은 어떻게 될 것인가?

* + 1. 문자형 리터럴

문자형 리터럴은 홑따옴표로 표시한다. 예를 들어

'e'

'\n'

'\u0061'

등이 그 예이다.

여기서 주의할 것이 있다. 다음 프로그램을 컴파일해 보자.

public class Example {

public static void main(String[] args) {

char c = '\u000a';

}

}

컴파일타임 오류가 날 것이다. 왜 그럴까? 유니코드 이스케이프는 먼저 유니코드로 변환되기 때문이다. \u000a는 LineFeed(줄바꿈)를 뜻하기 때문에 줄 바꿈으로 해석된다. 그래서 위의 프로그램은 다음과 같이 변환될 것이다.

public class Example {

public static void main(String[] args) {

char c = '

';

}

}

그래서 위와 같은 컴파일타임 오류가 나는 것이다. 이럴 때는 '\n'과 같이 이스케이프 시퀀스를 사용하면 된다.

이스케이프 시퀀스(escape sequence)에는 다음과 같은 것들이 있다.

\b : \u0008 : backspace BS

\t : \u0009 : horizontal tab HT

\n : \u000a : linefeed LF

\f : \u000c : form feed FF

\r : \u000d : carriage return CR

\" : \u0022 : double quote "

\' : \u0027 : single quote '

\\ : \u005c : backslash \

* 1. 실수형
     1. float

가장 큰 양의 float 값은 3.40282347e+38f이다. 가장 작은 양의 float 값은 1.40239846e-45f이다. float 형의 크기는 4바이트(32비트)이다.

* + 1. double

가장 큰 양의 double 값은 1.79769313486231570e+308이다. 가장 작은 양의 double 값은 4.94065645841246544e-324이다. double 형의 크기는 8바이트(64비트)이다.

* + 1. 실수형 리터럴(floating-point literal)

실수형 리터럴은 다음과 같이 구성된다.

정수부분, 소수점, 소수부분, exponent, 타입 지정

예를 들면

1.23E4d

에서1은 정수부분, .은 소수점, 23은 소수부분, E4는 exponent, d는 타입지정이다.

여러 부분 중 여러가지가 생략될 수 있다. 그러나 최소한 필요한 것이 있다. 예를 들어 234는 정수로 취급될 것이다. 이것을 실수로 바꾸려면 234.과 같이 .를 붙이면 된다. 또는 234d와 같이 명시적으로 타입을 지정해 주면 된다. 또는 234e1과 같이 exponent로 표현하는 방법도 있다.

타입 지정에는 float를 나타내는 F또는 f, double을 나타내는 D또는 d가 있다. 아무것도 지정하면 디폴트로 double형이 된다.

double tooBigDouble = 1.79769313486231570e+309;

위의 코드를 컴파일하면 numeric overflow같은 컴파일타임 오류가 날 것이다.

double tooSmallDouble = 4.94065645841246544e-325;

위의 코드를 컴파일하면 numeric underflow같은 컴파일타임 오류가 날 것이다.

그러나 다음과 같이하면 무한대를 표현할 수 있다.

public class Example {

public static void main(String[] a) {

double infiniteFloat = 1f / 0f;

double infiniteDouble = 1d / 0d;

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

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

infiniteFloat = -1f / 0f;

infiniteDouble = -1d / 0d;

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

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

infiniteFloat = Float.POSITIVE\_INFINITY;

infiniteDouble = Double.POSITIVE\_INFINITY;

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

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

infiniteFloat = Float.NEGATIVE\_INFINITY;

infiniteDouble = Double.NEGATIVE\_INFINITY;

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

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

}

}

다음과 같은 결과가 나올 것이다.

infiniteFloat = Infinity

infiniteDouble = Infinity

infiniteFloat = -Infinity

infiniteDouble = -Infinity

infiniteFloat = Infinity

infiniteDouble = Infinity

infiniteFloat = -Infinity

infiniteDouble = -Infinity

1f / 0f와 같은 식이 컴파일타임 오류 없이 실행되는 것에 주의하라.

Not-A-Number를 표현할 수도 있다. 다음 프로그램을 실행해 보자.

public class Example {

public static void main(String[] a) {

double notANumber = 0d / 0d;

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

notANumber = Double.NaN;

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

}

}

다음과 같은 결과가 나올 것이다.

notANumber = NaN

notANumber = NaN

0d / 0d가 컴파일타임 오류 없이 실행되는 것에 주의하라.

* 1. 논리형
     1. boolean

boolean형은 true, false 두 개의 값만 가질 수 있다.

자바에서는 논리형과 정수형 사이에서는 형변환이 절대 안된다. 명시적으로 캐스팅(casting)을 해도 안된다. 이것은 어떤 사람들에게는 약간 불편할지도 모르지만 확실히 실수를 줄이는 효과가 있다.

* 1. 기본형의 형변환
     1. 넓히는 기본형 형변환(widening privitive conversion)

넓히는 기본형 형변환은 더 큰 타입으로 변환하는 것이다. 다음은 그 변환들이다.

* byte에서 short, int, long, float, double로
* short에서 int, long, float, double로
* char에서 int, long, float, double로
* int에서 long, float, double로
* long에서 float, double로
* float에서 double로

넓히는 기본형 형변환에서는 값의 전체 크기는 희생되지 않는다. 이것은 무엇을 뜻하는 것일까? 예를 들어 byte에서 int등으로 형변환될 때는 값이 전혀 유실되지 않는다. 그러나 예를 들어 long에서 float으로 변환될 때는 값이 손상되기는 하지만 그 값이 그리 크지 않다. 다음 프로그램을 실행해 보자.

public class Example {

public static void main(String[] args) {

long l = Long.MAX\_VALUE;

float f = l;

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

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

}

}

다음과 같은 결과가 나올 것이다.

l = 9223372036854775807

f = 9.223372E18

정보가 유실된것을 알수 있다. 즉 9223372036854775807이 9223372000000000000으로 바뀐 것을 알 수 있다. 그러나 큰 범위에서 보면 대체로 정보는 양호하게 유지된 것을 알 수 있다. 즉 정보가 유실되더라도 크지는 않다. 넓히는 기본형 형변환에서 정보가 유실될 수 있다는 것에 주의하라.

정보가 유실될 수 있음에도 넓히는 기본형 형변환은 예외상황이 발생하지 않는다. 즉 넓히는 기본형 형변환은 안전한 것으로 취급된다. 다시한번 말하지만 항상 안전한 것은 아니다.

* + 1. 좁히는 기본형 변환(narrowing privitive conversion)과 캐스팅(casting)

좁히는 기본형 형변환은 넓히는 기본형 형변환의 반대로 생각하면 된다. 다음과 같은 것들이 있다.

* byte에서 char로
* short에서 byte, char로
* char에서 byte,r short로
* int에서 byte, short, char로
* long에서 byte, short, char, int로
* float에서 byte, short, char, int, long로
* double에서 byte, short, char, int, long, float로

좁히는 기본형 형변환은 넓히는 기본형 형변환에 비해 더 심각하게 정보의 손실이 있을 수 있다. 자리수나 양수, 음수 여부까지도 바뀔 수 있다.

byte에서 char로 형변환된다면 char가 음수를 표현할 수 없기 때문에 음수가 양수로 바뀐다. 다음 프로그램을 실행해보자.

public class Example {

public static void main(String[] args) {

byte b = -10;

char c = (char)b;

int i = c;

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

}

}

다음과 같은 결과가 나올 것이다.

i = 65526

char c = (char)b;를 한번 거치더니 숫자가 영 이상하게 바뀌었다. 프로그램은 나름대로 규칙을 가지고 변환했지만 사용자의 입장에서는 음수가 양수로 바뀌고 10이 65526으로 바뀌었다. 이것은 넓히는 기본형 형변환의 예에서 나오는 정보 손실과는 비교도 할 수 없는 심각한 손실이다.

그러므로 좁히는 기본형 형변환은 위험한 형변환이다. 그러므로 명시적으로 프로그래머가 캐스팅(casting)을 해주지 않으면 컴파일타임 오류가 난다. 명시적인 캐스팅은 프로그래머가 정보의 손실을 각오하고 형변환을 한다는 의미이므로 컴파일러는 에러 없이 컴파일한다. 좁히는 기본형 형변환을 할 때에는 안전한지 확인하고 사용해야 한다. 그렇지 않으면 위의 예에서와 같이 재앙이 일어날 것이다.

이제 부터 좁히는 기본형 형변환이 어떻게 일어나는지 알아보자. 대충 나누면 다음과 같이 정리할 수 있을 것이다.

* 정수에서 정수로
* 실수에서 정수로
* 실수에서 실수로

먼저 정수형에 대해 알아보자. 정수형의 좁히는 기본형 형변환에서는 만약 바이트수가 줄어든다면 least siginificant bits만 그대로 살아남는다. 다음 프로그램을 실행해보자. 비트 패턴은 16진수와 2진수로 볼 때 편리하므로 같이 볼 수 있도록 했다.

public class Example {

public static void main(String[] args) {

convertAndPrint(1L);

convertAndPrint(-1L);

convertAndPrint(10000000000L);

convertAndPrint(-10000000000L);

}

public static void convertAndPrint(long l) {

int i = (int)l;

System.out.print("l = " + l + ", ");

fillZeros(Long.toHexString(l), 16);

System.out.println();

System.out.print("i = " + i + ", ");

fillZeros(Integer.toHexString(i), 8);

System.out.println();

System.out.println();

}

public static void fillZeros(String s, int magnitude) {

for (int i = 0; i < magnitude - s.length(); i++)

System.out.print("0");

System.out.print(s);

}

}

다음과 같은 결과가 나올 것이다.

l = 1, 0000000000000001

i = 1, 00000001

l = -1, ffffffffffffffff

i = -1, ffffffff

l = 10000000000, 00000002540be400

i = 1410065408, 540be400

l = -10000000000, fffffffdabf41c00

i = -1410065408, abf41c00

10진수로 볼때는 규칙이 없는 듯이 마주잡이로 바꾸는 것 같지만 16진수나 2진수로 보면 정확하게 어떻게 바뀌는지 알 수 있을 것이다.

실수에서 정수 또는 실수에서 실수로 바뀌는 과정은 좀더 복잡하다.

결론적으로 말하면 좁히는 기본형 형변환은 안전할 때만 사용해야 한다. 안전할 때란 좁혀졌음에도 정보를 충분히 표현할 수 있을 때이다. 예를 들어 short에서 byte로 변환될 때 실제 숫자가 -128에서 127까지만 있다는 것을 확신할 때만 안전하다. 그 범위를 넘어가면 예측하기 어려운 변환이 일어날 수 있다.

* + 1. 형변환(conversion)의 예외

정수형 리터럴 12는 int 형이다. 그리고 int 형을 byte 형으로 형변환하는 것은 위험하기 때문에 명시적으로 캐스팅을 해주어야 한다. 그러나 여기에 예외가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 12;

short sh = 1234;

char ch = 6123;

}

}

무사히 컴파일된다. 리터럴인 경우에는 특별히 위와 같은 사용법이 허용된다. 리터럴인 경우는 컴파일러가 안전함을 100퍼센트 보장할 수 있기 때문에 편의상 허용한 것이다. 그것이 허용되지 않았다면

byte b = (byte)12;

와 같이 좀 짜증나는 캐스팅을 해야만 했을 것이다.

이것은 리터럴 뿐만 아니라 컴파일 타임 상수식(compile-time contant expression)에도 해당한다. 컴파일 타임 상수식이란 컴파일할 때에 그 값을 알 수 있는 상수식을 말한다. 아래 예제를 컴파일해 보자.

public class Example {

public static void main(String[] args) {

final byte b = 12 \* 2; // OK. 상수식도 확실히 보장할 수 있다.

short sh = 1234 + b; // OK. b 가 final 이기 때문에 보장된다.

char ch = 123 + sh; // Error. sh 는 final 이 아니다.

}

}

만약 범위가 벗어나면 즉

byte b = 1234;

와 같은 경우가 생기면 컴파일타임 오류가 난다. 이것은 메쏘드에 인수로 넘길 때는 통하지 않는데 그것은 메쏘드 중복정의(method overloading) 과 관련이 있다. ‘클래스와 인터페이스 - 못다한 이야기’를 다룬 장의 ‘중복정의된 메쏘드의 선택’을 참조하라.

* 1. 포장 클래스(wrapper class)
     1. 포장 클래스란

자바에서는 모든 것이 객체인데 기본형만 예외다. 그래서 문제가 생길 때가 있다. 그것을 해결하기 위해 만들어진 것이 포장 클래스이다. 포장 객체(wrapper object)는 기본형의 값을 포장(저장)하는 객체이다. 그 값은 바뀔 수 없다. 이런 점에서 String 객체와 비슷하다.

포장 클래스에는 이 외에도 여러가지 다른 기능도 한다. 예를 들어 Character클래스에는 여러가지 기능이 있다.

각 포장 클래스에는 TYPE라는 Class 형 상수가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(Byte.TYPE == byte.class);

System.out.println(Character.TYPE == char.class);

System.out.println(Double.TYPE == double.class);

}

}

결과는 다음과 같을 것이다.

true

true

true

위의 예제가 이해가 잘 안되는 사람은 리플렉션을 다룬 장을 참조하라.

포장 클래스에는 그 형(type)의 최대값과 최소값을 나타내는 상수가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("Byte.MIN\_VALUE = " + Byte.MIN\_VALUE);

System.out.println("Byte.MAX\_VALUE = " + Byte.MAX\_VALUE);

System.out.println("Short.MIN\_VALUE = " + Short.MIN\_VALUE);

System.out.println("Short.MAX\_VALUE = " + Short.MAX\_VALUE);

System.out.println("Integer.MIN\_VALUE = " + Integer.MIN\_VALUE);

System.out.println("Integer.MAX\_VALUE = " + Integer.MAX\_VALUE);

System.out.println("Long.MIN\_VALUE = " + Long.MIN\_VALUE);

System.out.println("Long.MAX\_VALUE = " + Long.MAX\_VALUE);

System.out.println("Character.MIN\_VALUE = " + Character.MIN\_VALUE); // 그 값에 해당하는 문자를 표시한다.

System.out.println("Character.MAX\_VALUE = " + Character.MAX\_VALUE); // 그 값에 해당하는 문자를 표시한다.

System.out.println("(int)Character.MIN\_VALUE = " + (int)Character.MIN\_VALUE);

System.out.println("(int)Character.MAX\_VALUE = " + (int)Character.MAX\_VALUE);

System.out.println("Float.MIN\_VALUE = " + Float.MIN\_VALUE); // 양의 최소값임에 주의하라.

System.out.println("Float.MAX\_VALUE = " + Float.MAX\_VALUE);

System.out.println("Float.NEGATIVE\_INFINITY = " + Float.NEGATIVE\_INFINITY);

System.out.println("Float.POSITIVE\_INFINITY = " + Float.POSITIVE\_INFINITY);

System.out.println("Float.NaN = " + Float.NaN);

System.out.println("Double.MIN\_VALUE = " + Double.MIN\_VALUE); // 양의 최소값임에 주의하라.

System.out.println("Double.MAX\_VALUE = " + Double.MAX\_VALUE);

System.out.println("Double.NEGATIVE\_INFINITY = " + Double.NEGATIVE\_INFINITY);

System.out.println("Double.POSITIVE\_INFINITY = " + Double.POSITIVE\_INFINITY);

System.out.println("Double.NaN = " + Double.NaN);

}

}

결과는 다음과 비슷할 것이다.

Byte.MIN\_VALUE = -128

Byte.MAX\_VALUE = 127

Short.MIN\_VALUE = -32768

Short.MAX\_VALUE = 32767

Integer.MIN\_VALUE = -2147483648

Integer.MAX\_VALUE = 2147483647

Long.MIN\_VALUE = -9223372036854775808

Long.MAX\_VALUE = 9223372036854775807

Character.MIN\_VALUE =

Character.MAX\_VALUE = ?(int)Character.MIN\_VALUE = 0

(int)Character.MAX\_VALUE = 65535

Float.MIN\_VALUE = 1.4E-45

Float.MAX\_VALUE = 3.4028235E38

Float.NEGATIVE\_INFINITY = -Infinity

Float.POSITIVE\_INFINITY = Infinity

Float.NaN = NaN

Double.MIN\_VALUE = 4.9E-324

Double.MAX\_VALUE = 1.7976931348623157E308

Double.NEGATIVE\_INFINITY = -Infinity

Double.POSITIVE\_INFINITY = Infinity

Double.NaN = NaN

* + 1. Number 클래스

Boolean을 제외한 모든 wrapper 클래스는 Number 클래스를 상속받는다.

* + 1. 정수형

정수형에는 byte, short, int, long, char가 있다. 이것에 대응하는 wrapper 클래스가 있는데 이름이 거의 비슷하다. Byte, Short, Integer, Long, Character가 그것이다.

Character 클래스에 대해 더 자세히 알고 싶으면 문자열을 다루는 장을 참조하라.

정수형 리터럴의 쓰임은 서로 거의 비슷하다. 여기서는 Integer 클래스를 중심으로 설명할 것이다. 다른 클래스도 거의 비슷하게 적용할 수 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Integer oi = new Integer(1234);

System.out.println(oi);

oi = new Integer("4321");

System.out.println(oi);

oi = Integer.decode("1111");

System.out.println(oi);

oi = Integer.decode("0X1111"); // 16 진수.

System.out.println(oi);

oi = Integer.decode("01111"); // 8 진수.

System.out.println(oi);

int i = Integer.parseInt("9999");

System.out.println(i);

for (int radix = Character.MIN\_RADIX; radix <= Character.MAX\_RADIX; radix++) {

i = Integer.parseInt("11", radix);

System.out.println("11(" + radix + "진법) = " + i + "(10진법)");

}

oi = Integer.valueOf("8888");

System.out.println(oi);

for (int radix = Character.MIN\_RADIX; radix <= Character.MAX\_RADIX; radix++) {

oi = Integer.valueOf("110", radix);

System.out.println("110(" + radix + "진법) = " + oi + "(10진법)");

}

oi = Integer.valueOf("Zealot", 36); // 36 진법은 숫자 10개와 알파벳 26개를 사용한다.

System.out.println(oi);

}

}

결과는 다음과 같을 것이다.

1234

4321

1111

4369

585

9999

11(2진법) = 3(10진법)

11(3진법) = 4(10진법)

11(4진법) = 5(10진법)

11(5진법) = 6(10진법)

11(6진법) = 7(10진법)

11(7진법) = 8(10진법)

11(8진법) = 9(10진법)

11(9진법) = 10(10진법)

11(10진법) = 11(10진법)

11(11진법) = 12(10진법)

11(12진법) = 13(10진법)

11(13진법) = 14(10진법)

11(14진법) = 15(10진법)

11(15진법) = 16(10진법)

11(16진법) = 17(10진법)

11(17진법) = 18(10진법)

11(18진법) = 19(10진법)

11(19진법) = 20(10진법)

11(20진법) = 21(10진법)

11(21진법) = 22(10진법)

11(22진법) = 23(10진법)

11(23진법) = 24(10진법)

11(24진법) = 25(10진법)

11(25진법) = 26(10진법)

11(26진법) = 27(10진법)

11(27진법) = 28(10진법)

11(28진법) = 29(10진법)

11(29진법) = 30(10진법)

11(30진법) = 31(10진법)

11(31진법) = 32(10진법)

11(32진법) = 33(10진법)

11(33진법) = 34(10진법)

11(34진법) = 35(10진법)

11(35진법) = 36(10진법)

11(36진법) = 37(10진법)

8888

110(2진법) = 6(10진법)

110(3진법) = 12(10진법)

110(4진법) = 20(10진법)

110(5진법) = 30(10진법)

110(6진법) = 42(10진법)

110(7진법) = 56(10진법)

110(8진법) = 72(10진법)

110(9진법) = 90(10진법)

110(10진법) = 110(10진법)

110(11진법) = 132(10진법)

110(12진법) = 156(10진법)

110(13진법) = 182(10진법)

110(14진법) = 210(10진법)

110(15진법) = 240(10진법)

110(16진법) = 272(10진법)

110(17진법) = 306(10진법)

110(18진법) = 342(10진법)

110(19진법) = 380(10진법)

110(20진법) = 420(10진법)

110(21진법) = 462(10진법)

110(22진법) = 506(10진법)

110(23진법) = 552(10진법)

110(24진법) = 600(10진법)

110(25진법) = 650(10진법)

110(26진법) = 702(10진법)

110(27진법) = 756(10진법)

110(28진법) = 812(10진법)

110(29진법) = 870(10진법)

110(30진법) = 930(10진법)

110(31진법) = 992(10진법)

110(32진법) = 1056(10진법)

110(33진법) = 1122(10진법)

110(34진법) = 1190(10진법)

110(35진법) = 1260(10진법)

110(36진법) = 1332(10진법)

2140325453

다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

Integer oi = new Integer("1234");

byte b = oi.byteValue(); // 이것의 결과에 주의하라.

System.out.println(b);

short sh = oi.shortValue();

System.out.println(sh);

int i = oi.intValue();

System.out.println(i);

long l = oi.longValue();

System.out.println(l);

float f = oi.floatValue();

System.out.println(f);

double d = oi.doubleValue();

System.out.println(d);

System.out.println(oi.toString(1234));

System.out.println(oi.toHexString(1234));

System.out.println(oi.toOctalString(1234));

System.out.println(oi.toBinaryString(1234));

for (int radix = Character.MIN\_RADIX; radix <= Character.MAX\_RADIX; radix++) {

System.out.println("1234(10진법) = " + Integer.toString(1234, radix) + "(" + radix + "진법)");

}

}

}

결과는 다음과 같을 것이다.

-46

1234

1234

1234

1234.0

1234.0

1234

4d2

2322

10011010010

1234(10진법) = 10011010010(2진법)

1234(10진법) = 1200201(3진법)

1234(10진법) = 103102(4진법)

1234(10진법) = 14414(5진법)

1234(10진법) = 5414(6진법)

1234(10진법) = 3412(7진법)

1234(10진법) = 2322(8진법)

1234(10진법) = 1621(9진법)

1234(10진법) = 1234(10진법)

1234(10진법) = a22(11진법)

1234(10진법) = 86a(12진법)

1234(10진법) = 73c(13진법)

1234(10진법) = 642(14진법)

1234(10진법) = 574(15진법)

1234(10진법) = 4d2(16진법)

1234(10진법) = 44a(17진법)

1234(10진법) = 3ea(18진법)

1234(10진법) = 37i(19진법)

1234(10진법) = 31e(20진법)

1234(10진법) = 2gg(21진법)

1234(10진법) = 2c2(22진법)

1234(10진법) = 27f(23진법)

1234(10진법) = 23a(24진법)

1234(10진법) = 1o9(25진법)

1234(10진법) = 1lc(26진법)

1234(10진법) = 1ij(27진법)

1234(10진법) = 1g2(28진법)

1234(10진법) = 1dg(29진법)

1234(10진법) = 1b4(30진법)

1234(10진법) = 18p(31진법)

1234(10진법) = 16i(32진법)

1234(10진법) = 14d(33진법)

1234(10진법) = 12a(34진법)

1234(10진법) = 109(35진법)

1234(10진법) = ya(36진법)

* + 1. 실수형

실수형 기본형 float, double에 대응하는 포장 클래스는 Float, Double가 있다. 정수형과 비슷하게 쓸 수 있는 메쏘드가 많이 있다. 정수형의 포장 클래스를 다룬 부분을 참조하라.

* + 1. 논리형

논리형 기본형 boolean에 대응하는 Boolean 클래스가 있다.

* + 1. NumberFormatException

스트링에서 숫자로 변환하다보면 잘못된 스트링 때문에 예외상황이 발생할 수 있다. 이 때 발생하는 것이 NumberFormateException이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Integer("12#4");

}

}

결과는 다음과 비슷할 것이다.

java.lang.NumberFormatException: 12#4

at java.lang.Integer.parseInt(Integer.java:438)

at java.lang.Integer.<init>(Integer.java:570)

at Example.main(Example.java:3)

Exception in thread "main"

이런일은 사용자가 숫자를 입력할 때 자주 일어나는 것이다. NumberFormatException은 RuntimeException이기 때문에 try 문으로 잡아 주지 않아도 컴파일타임 오류가 나지는 않는다. 그러나 try 문으로 처리해주는 것이 좋다. 사용자가 숫자 하나 잘못 입력했다고 프로그램이 종료되는 것은 문제가 있다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

while (true) {

System.out.println("숫자를 입력하세요");

String str = "";

try {

while (true) {

int i = System.in.read();

if (i == '\n')

break;

str += (char)i;

}

int value = Integer.parseInt(str);

System.out.println(value + "를 입력하셨습니다.");

break;

} catch (IOException e) {

System.out.println(e);

break;

} catch (NumberFormatException e) {

System.out.println("잘못 입력하셨습니다. 다시 한번 해 보세요.");

}

}

}

}

결과는 다음과 비슷할 것이다.

숫자를 입력하세요

5yf

잘못 입력하셨습니다. 다시 한번 해 보세요.

숫자를 입력하세요

7654

7654를 입력하셨습니다.

* + 1. 포장 클래스의 쓰임

포장 클래스가 어디에 쓰이는지 알고 싶은 사람은 리플렉션을 다루는 장을 참조하라. 그 외에도 여러가지로 쓰일 수 있다. 컬렉션 클래스(collection classes)에서 많이 쓰인다.

* 1. FP-strict
     1. float value set, double value set

float value set과 double value set은 IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985(IEEE 754)에 규정된 부동소수점 숫자들의 집합이다. float value set은 32비트이고 double value set은 64비트이다. 자바 프로그래밍 언어를 어떤 식으로 구현하든 float value set과 double value set을 반드시 제공해야 한다.

* + 1. float-extended-exponent value set, double-extended-exponent value set

실수 연산을 하다보면 오버플로우(overflow)나 언더플로우(underflow)가 발생할 때가 있다. 이것을 줄이기 위해서는 더 큰 value set을 사용하면 된다. 이것이 float-extended-exponent value set과 double-extended-exponent value set이다. extended-exponent라는 것이 덧붙여진 것에서도 알 수 있듯이 이것은 유효숫자가 아니라 지수의 범위를 더 넓혔다. 범위가 더 커졌기 때문에 비트수도 더 크다.

글쓴이는float-extended-exponent value set 또는 double-extended-exponent value set를 지원하는 실행 환경을 찾지 못했다.

Sun JDK 1.1.8, Sun JDK 1.2.2, Sun JDK 1.3.1, Sun JDK 1.4.0, IBM JDK 1.3.0, Microsoft SDK for Java 4.0, JBuilder의 JDK 1.3.0를 테스트해 보았으나 모두 지원하지 않는 듯하다.

* + 1. 오버플로우(overflow)와 언더플로우(underflow)

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(Double.MAX\_VALUE \* 10.0);

System.out.println(Double.MIN\_VALUE / 10.0);

}

}

결과는 다음과 같을 것이다.

Infinity

0.0

(Double.MAX\_VALUE \* 10.0)은 double이 나타낼 수 있는 숫자보다 더 크다. 이런 상황을 오버플로우라고 한다. 실수 연산에서 오버플로우가 발생하면 예외나 오류가 발생하지 않고 위의 결과에서 볼 수 있듯이 Infinity등의 값이 결과로 나온다. 마찬가지로 (Double.MIN\_VALUE / 10.0)은 double이 나타내기에는 너무 작은 수이다. 이런 경우는 언더플로우라고 한다. 결과에서 보듯이 0.0이 결과로 나왔다.

* + 1. xxx-extended-exponent value set의 존재 이유

float 형 필드, 매개변수, 지역 변수는 반드시 float value set에 있는 값만 가질 수 있다. 다시 말하면 float value set에는 없고 float-extended-exponent value set에만 있는 값을 가질 수 없다. 그것은 배열에서도 마찬가지이다. 그러므로 대입이 일어날 때나 인수가 매개변수에 설정될 때 등에는 값의 변환이 일어난다. 이것은 double형도 마찬가지이다.

하지만 float-extended-exponent value set 또는 double-extended-exponent value set의 값은 연산의 중간값(intermediate value)으로 쓰일 수 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(Double.MAX\_VALUE \* 10.0 / 100.0);

System.out.println(Double.MIN\_VALUE / 10.0 \* 100.0);

}

}

결과는 아마 다음과 같을 것이다.

Infinity

0.0

하지만 xxx-extended-exponent value set을 지원하는 실행 환경에서는 다음과 비슷한 결과가 나올 것이다.

1.7976931348623158E307

4.9E-323

double-extended-exponent value set을 사용한다면 연산 중간값인 (Double.MAX\_VALUE \* 10.0)을 표현할 수 있을 것이고 거기에 다시 100을 곱하면 double value set으로 표현할 수 있는 값이 나온다. 그러므로 Infinity가 아니라 (오차내에서) 정확한 값을 구할 수 있는 것이다. 정리하면 xxx-extended-exponent value set은 오버플로우와 언더플로우를 줄일 수 있다. 하지만 위에서 보았던 다음 예제에서는 안 통한다.

public class Example {

public static void main(String[] args) {

System.out.println(Double.MAX\_VALUE \* 10.0);

System.out.println(Double.MIN\_VALUE / 10.0);

}

}

(Double.MAX\_VALUE \* 10.0)은 println 메쏘드의 인수로 넘겨지고 매개변수는 double value set에 있는 값만 가질 수 있기 때문에 결국 Infinity가 될 수 밖에 없다.

그리고 xxx-extended-exponent value set은 속도를 좀더 빠르게 할 수도 있다. IEEE 754가 속도면에서 그리 좋은 평가를 받지 않는 듯하다. 자세한 것은 IEEE 754 또는 실수형 연산의 컴퓨터에서의 구현을 다루는 책이나 문서를 참조하라.

* + 1. strictfp 키워드

strictfp 키워드는 xxx-extended-exponent value set의 사용을 금지하고 표준인 IEEE 754만 사용하게 강제하는 것이다. 그것을 FP-strict라고 한다.

strictfp는 클래스, 인터페이스, 메쏘드에 쓰인다. 클래스에 strictfp를 사용하면 그 클래스의 모든 메쏘드, 멤버 타입, 초기화 블록, 필드 초기화문, 생성자는 strictfp이 적용된다. 생성자에는 사용할 수 없으며 인터페이스의 메쏘드에도 사용할 수 없다. 생성자는 클래스가 strictfp이면 strictfp이다. 인터페이스의 메쏘드에서는 구현에 대한 strictfp 여부를 정할 수 없기 때문이다.

* + 1. strictfp의 존재 이유

xxx-extended-exponent value set을 사용하면 오버플로우와 언더플로우를 줄임으로써 좀더 정확한 값을 얻을 수 있고 속도까지 빨라질 수 있는데 그것을 왜 strictfp라는 귀찮은 키워드를 사용해서 금지하려고 할까? 문제는 portability이다. IEEE 754가 속도면에서 문제가 있고 32비트와 64비트의 한계 때문에 오버플로우와 언더플로우가 있지만 그것은 표준이고 표준대로 하면 어떤 플랫폼에서 어떤 상황에 실행해도 똑같은 값을 얻을 수 있다. 즉 완전히 값은 비트패턴을 얻을 수 있는 것이다. 대부분의 경우에 이것은 중요하지 않다. 사실 오버플로우가 발생하는 대부분의 경우는 논리적 에러 즉 버그라고 할 수 있다. 그래서 대부분의 코드에서는 strictfp가 적용되지 않음을 볼 수 있을 것이다. 하지만 살다보면 실수형 연산을 할 때 결과값이 반드시 완전히 똑같아야 하는 경우가 있을지도 모른다. 그런 경우를 위해서 strictfp가 존재하는 것이다. 한마디로 하면 별로 중요한 키워드는 아니지만 혹시나 해서 만들어졌다고 생각하면 된다.

* + 1. StrictMath

StrictMath클래스는 Math클래스와 같지만 strictfp가 적용되기 때문에 모든 플랫폼의 모든 구현에서 똑 같은 값 즉 비트패턴이 결과로 나온다. 다음은 StrictMath.java에서 인용한 것이다.

…

public final strictfp class StrictMath {

…

}

strictfp로 선언되어 있다. 하지만 Math.java를 보면 Math 클래스 역시 strictfp로 선언되어 있다. 사실 글쓴이도 둘 사이에 차이가 있는지 잘 모르겠다.

…

public final strictfp class Math {

…

}

* + 1. strictfp와 상속

메쏘드를 재정의(overriding)할 때 strictfp 메쏘드를 strictfp가 아닌 메쏘드로 재정의할 수 있다.

추상(abstract) 메쏘드는 strictfp일 수 없다. strictfp는 구현에 관련된 것이기 때문이다. 하지만 그 메쏘드를 구현할 때에는 strictfp로 구현할 수 있다.

strictfp는 native와 공존할 수 없다.

* 1. 기타
     1. Random

Random 클래스는 의사 난수(pseudo random number)를 발생시킬 때 쓴다. Random 클래스가 작동하기 위해서는 seed를 설정해 주어야 한다. 그리고 발생하는 난수는 그 seed에 의해 결정된다. 다음 예제를 보면 발생되는 것이 진짜 난수가 아니라 의사 난수란 것을 알 수 있을 것이다. 심지어 다음 결과는 모든 플랫폼에서 같게 나올 것이다. 모든 자바 플랫폼에서 같은 알고리즘을 사용하게 함으로써 포팅(porting)의 문제를 최소화한 것이다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Random r = new Random(123); // seed 123

for (int i = 0; i < 4; i++) {

System.out.println(r.nextInt());

}

System.out.println();

r = new Random(123); // seed 123

for (int i = 0; i < 4; i++) {

System.out.println(r.nextInt());

}

System.out.println();

r = new Random(-123); // seed -123

for (int i = 0; i < 4; i++) {

System.out.println(r.nextInt());

}

System.out.println();

r = new Random(-123); // seed -123

for (int i = 0; i < 4; i++) {

System.out.println(r.nextInt());

}

System.out.println();

}

}

결과는 다음과 같을 것이다.

-1188957731

1018954901

-39088943

1295249578

-1188957731

1018954901

-39088943

1295249578

1188188232

1300478726

1853071797

-1486485370

1188188232

1300478726

1853071797

-1486485370

이렇게 예측 가능하다면 난수로서의 효용이 떨어질 것이다. seed를 예측하기 어려운 숫자를 넣어야 한다. 그것에 가장 가까운 것이 아마 시스템의 시간일 것이다. 시스템의 시간은 1000분의 1초 단위이기 때문에 이것을 예측하기는 거의 불가능하다. Random 클래스의 생성자중 하나가 실제로 자동으로 seed를 시스템 시간으로 설정한다. 다음은 Random.java에서 인용한 것이다.

...

public class Random implements java.io.Serializable {

...

public Random() {

this(System.currentTimeMillis());

}

...

}

double형 난수를 얻고 싶으면 간편하게 Math.random()을 사용할 수도 있다.

* + 1. Math

Math 클래스는 수학에 관련된 여러가지 값과 메쏘드가 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("Math.E = " + Math.E);

System.out.println("Math.PI = " + Math.PI);

}

}

결과는 다음과 비슷할 것이다.

Math.E = 2.718281828459045

Math.PI = 3.141592653589793

많이 보았던 숫자일 것이다. E는 로그에 쓰이고 PI는 원의 넓이를 구할 때나 삼각함수 등에서 많이 쓰인다.

삼각함수(sin, cos, tan등등), 절대값, 최대최소값 등등을 구할 수 있는 메쏘드들이 있다.

* + 1. BigInteger

정수형 중 가장 큰 long형의 한계를 넘어서는 값을 써야 할 때가 있다. 회계 프로그램 또는 천문학을 다루는 프로그램에서 그런 것이 필요할 수 있다. double을 써도 되는 경우도 있지만 만약에 근사값이 아니라 정확한 값을 구해야 한다면 어떻게 할까? 그 때 쓸 수 있는 것이 BigInteger이다.

팩토리얼을 구하는 예제이다.

import java.math.\*;

public class Example {

public static void main(String[] args) {

BigInteger bi = new BigInteger("1");

for (int i = 1; i <= 100; i++) {

bi = bi.multiply(new BigInteger("" + i));

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

}

}

}

결과는 다음과 같을 것이다.

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

…

100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

* + 1. BigDecimal

BigDecimal은 BigInteger의 실수판이라고 생각하면 된다.

팩토리얼(factorial)을 구하는 예제이다.

import java.math.\*;

public class Example {

public static void main(String[] args) {

BigDecimal bd = new BigDecimal("1");

for (int i = 1; i <= 100; i++) {

bd = bd.multiply(new BigDecimal("" + i));

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

}

}

}

결과는 다음과 같을 것이다.

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

…

100! = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286253697920827223758251185210916864000000000000000000000000

위의 예제는 굳이 BigDecimal을 쓰지 않아도 된다. BigInteger면 충분하다. 이번에는 BigDecimal만 할 수 있는 예제이다.

import java.math.\*;

public class Example {

public static void main(String[] args) {

BigDecimal bd = new BigDecimal("1.1");

String str = "1.1";

for (int i = 2; i <= 100; i++) {

bd = bd.multiply(new BigDecimal(i + "." + i));

str += " \* " + i + "." + i;

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

}

}

}

결과는 다음과 같을 것이다.

1.1 \* 2.2 = 2.42

1.1 \* 2.2 \* 3.3 = 7.986

1.1 \* 2.2 \* 3.3 \* 4.4 = 35.1384

1.1 \* 2.2 \* 3.3 \* 4.4 \* 5.5 = 193.26120

1.1 \* 2.2 \* 3.3 \* 4.4 \* 5.5 \* 6.6 = 1275.523920

1.1 \* 2.2 \* 3.3 \* 4.4 \* 5.5 \* 6.6 \* 7.7 = 9821.5341840

1.1 \* 2.2 \* 3.3 \* 4.4 \* 5.5 \* 6.6 \* 7.7 \* 8.8 = 86429.50081920

…

실제로 실행해 보라. 정말 큰 숫자도 잘 소화하는 것을 볼 수 있을 것이다. 즉 유효 숫자구 개수가 엄청나게 커질 수 있다. 메모리와 다른 것들이 든든하다면.

* + 1. BitSet

BitSet은 BigInteger의 비트판이다. 자세한 사용법은 API specification을 참조하라. java.util 패키지에 있다.

1. 연산자
   1. 사칙연산
      1. 더하기 +

교환법칙(commutative)에 대해 알아보자. 만약 피연잔사(operand)에 부수적 효과(side effect)가 없을 경우 + 연산자는 교환법칙이 성립한다. 다음 예제를 보면 교환 법칙이 성립함을 알 수 있다.

public class Example {

public static void main(String[] args) {

int a = 3;

int b = 4;

System.out.println(a + b);

System.out.println(b + a);

}

}

결과는 다음과 같을 것이다.

7

7

그럼 부수적 효과 란 무엇인가? 다음 예제를 보면 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

System.out.println(a() + b());

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

System.out.println(b() + a());

}

public static int a() {

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

return 3;

}

public static int b() {

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

return 4;

}

}

결과는 다음과 같을 것이다.

a()

b()

7

----------------

b()

a()

7

이제 결합법칙(associative)에 대해 알아보자. 정수형에서는 피연산자의 형이 같을 경우 +의 결합법칙이 성립한다.

public class Example {

public static void main(String[] args) {

int a = 3;

int b = 10;

int c = 20;

System.out.println( (a + b) + c );

System.out.println( a + (b + c) );

}

}

결과는 다음과 같을 것이다.

33

33

피연산자의 형이 다른 경우 어떤 일이 일어날 수 있는지 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = Integer.MAX\_VALUE;

int b = 1;

long c = Integer.MIN\_VALUE;

System.out.println( (a + b) + c );

System.out.println( a + (b + c) );

}

}

결과는 다음과 같을 것이다.

-4294967296

0

왜 결과가 이렇게 나왔는지 이해가 잘 안된다면 promotion을 다룬 부분을 읽은 다음에 다시 보기 바란다.

실수형에는 +의 결합법칙이 성립하지 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

double a = Double.MAX\_VALUE;

double b = a;

double c = -a;

System.out.println( (a + b) + c );

System.out.println( a + (b + c) );

}

}

결과는 다음과 같을 것이다.

Infinity

1.7976931348623157E308

이번에는 오버플로우(overflow)에 대해 알아보자. 순수한 수학과는 달리 컴퓨터에서는 수를 저장하기 위한 메모리가 제한되어 있기 때문에(예를 들어 int형인 경우에는 32비트에 정수를 저장한다) 오버플로우란 것이 있다. 저장할 수 있는 값의 한계를 초과한 것을 말한다. 먼저 오버플로우가 발생해도 예외 상황이 발생하지 않는다는 것을 기억해야 한다. 오버플로우가 발생했을 때 컴퓨터나는 나름대로 최선(?)의 값으로 변환한다. 먼저 정수형에 대해 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = Integer.MAX\_VALUE;

int b = a;

long la = a;

long lb = b;

System.out.println("a = " + Integer.toHexString(a));

System.out.println("b = " + Integer.toHexString(b));

System.out.println("la + lb = " + Long.toHexString(la+lb));

System.out.println("la + lb = " + Long.toBinaryString(la+lb));

System.out.println("la + lb = " + (la+lb));

System.out.println("a + b = " + Integer.toHexString(a+b));

System.out.println("a + b = " + Integer.toBinaryString(a+b));

System.out.println("a + b = " + (a+b));

}

}

결과는 다음과 같을 것이다.

a = 7fffffff

b = 7fffffff

la + lb = fffffffe

la + lb = 11111111111111111111111111111110

la + lb = 4294967294

a + b = fffffffe

a + b = 11111111111111111111111111111110

a + b = -2

a와 b를 더하면 16진수로 fffffffe가 된다. 이것의 마지막 32비트는 fffffffe이다. 이것은 int형으로 해석하면 –2가 된다.

실수형은 좀더 복잡한 규칙을 따른다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("Double.NaN + Double.NaN = " + (Double.NaN + Double.NaN));

System.out.println("Double.POSITIVE\_INFINITY + Double.NEGATIVE\_INFINITY = " + (Double.POSITIVE\_INFINITY + Double.NEGATIVE\_INFINITY));

System.out.println("Double.POSITIVE\_INFINITY + Double.POSITIVE\_INFINITY = " + (Double.POSITIVE\_INFINITY + Double.POSITIVE\_INFINITY));

System.out.println("Double.NEGATIVE\_INFINITY + Double.NEGATIVE\_INFINITY = " + (Double.NEGATIVE\_INFINITY + Double.NEGATIVE\_INFINITY));

System.out.println("Double.NEGATIVE\_INFINITY + 3 = " + (Double.NEGATIVE\_INFINITY + 3));

System.out.println("+0.0 = " + (+0.0));

System.out.println("-0.0 = " + (-0.0));

System.out.println("(+0.0) + (-0.0) = " + ((+0.0) + (-0.0)));

System.out.println("(-0.0) + (-0.0) = " + ((-0.0) + (-0.0)));

System.out.println("(+3.3) + (-3.3) = " + ((+3.3) + (-3.3)));

System.out.println("Double.MAX\_VALUE + Double.MAX\_VALUE = " + (Double.MAX\_VALUE + Double.MAX\_VALUE));

}

}

결과는 다음과 비슷할 것이다.

Double.NaN + Double.NaN = NaN

Double.POSITIVE\_INFINITY + Double.NEGATIVE\_INFINITY = NaN

Double.POSITIVE\_INFINITY + Double.POSITIVE\_INFINITY = Infinity

Double.NEGATIVE\_INFINITY + Double.NEGATIVE\_INFINITY = -Infinity

Double.NEGATIVE\_INFINITY + 3 = -Infinity

+0.0 = 0.0

-0.0 = -0.0

(+0.0) + (-0.0) = 0.0

(-0.0) + (-0.0) = -0.0

(+3.3) + (-3.3) = 0.0

Double.MAX\_VALUE + Double.MAX\_VALUE = Infinity

* + 1. 빼기 –

a – b 는 a + (-b)와 같다.

* + 1. 곱하기 \*

교환 법칙에 대해 알아보자. \* 연산자는 피연잔사에 부수적 효과가 없다면 교환 법칙이 성립한다. 이것에 대해서는 + 연산자를 참조하라.

결합 법칙에 대해 알아보자. \*의 결합법칙은 +의 결합 법칙과 마찬가지이다. 정수형에서는 모두 같은 형이면 결합 법칙이 성립한다. 실수형은 결합 법칙이 성립하지 않는다.

오버플로우에 대해 알아보자. 정수형에서는 + 연산자와 마찬가지로 연산한 값의 low bits가 해석된다. 실수형은 좀더 복잡한 규칙을 따른다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("Double.NaN \* 3 = " + (Double.NaN \* 3));

System.out.println("Double.POSITIVE\_INFINITY \* 0 = " + (Double.POSITIVE\_INFINITY \* 0));

System.out.println("Double.NEGATIVE\_INFINITY \* 3 = " + (Double.NEGATIVE\_INFINITY \* 3));

System.out.println("Double.NEGATIVE\_INFINITY \* Double.POSITIVE\_INFINITY = " + (Double.NEGATIVE\_INFINITY \* Double.POSITIVE\_INFINITY));

System.out.println("(+0.0) \* (-0.0) = " + ((+0.0) \* (-0.0)));

System.out.println("(-0.0) \* (-0.0) = " + ((-0.0) \* (-0.0)));

System.out.println("Double.MAX\_VALUE \* Double.MAX\_VALUE = " + (Double.MAX\_VALUE \* Double.MAX\_VALUE));

}

}

결과는 다음과 비슷할 것이다.

Double.NaN \* 3 = NaN

Double.POSITIVE\_INFINITY \* 0 = NaN

Double.NEGATIVE\_INFINITY \* 3 = -Infinity

Double.NEGATIVE\_INFINITY \* Double.POSITIVE\_INFINITY = -Infinity

(+0.0) \* (-0.0) = -0.0

(-0.0) \* (-0.0) = 0.0

Double.MAX\_VALUE \* Double.MAX\_VALUE = Infinity

* + 1. 나누기 /

먼저 정수형에 대해 알아보자. 정수형 나누기에서는 0으로 나눌 수 없다. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

int i = 1 / 0;

}

}

결과는 다음과 비슷할 것이다.

java.lang.ArithmeticException: / by zero

at Example.main(Example.java:3)

Exception in thread "main"

다음 예제는 정수의 나누기가 어떻게 되는가를 보여준다.

public class Example {

public static void main(String[] args) {

System.out.println("12 / 3 = " + (12 / 3));

System.out.println("12 / -3 = " + (12 / -3));

System.out.println("-12 / 3 = " + (-12 / 3));

System.out.println("-12 / -3 = " + (-12 / -3));

System.out.println("7 / 3 = " + (7 / 3));

System.out.println("7 / -3 = " + (7 / -3));

System.out.println("-7 / 3 = " + (-7 / 3));

System.out.println("-7 / -3 = " + (-7 / -3));

}

}

결과는 다음과 같을 것이다.

12 / 3 = 4

12 / -3 = -4

-12 / 3 = -4

-12 / -3 = 4

7 / 3 = 2

7 / -3 = -2

-7 / 3 = -2

-7 / -3 = 2

이번에는 실수형을 보자. 실수형에서는 0으로 나눌수 있다. 다음은 0으로 나누면 어떤 값들이 나오는지를 보여준다.

public class Example {

public static void main(String[] args) {

System.out.println("0.0 / 0.0 = " + (0.0 / 0.0));

System.out.println("-0.0 / 0.0 = " + (-0.0 / 0.0));

System.out.println("1.0 / 0.0 = " + (1.0 / 0.0));

System.out.println("10.3 / -0.0 = " + (10.3 / -0.0));

System.out.println("-Double.POSITIVE\_INFINITY / 0.0 = " + (-Double.POSITIVE\_INFINITY / 0.0));

System.out.println("-Double.MIN\_VALUE / -0.0 = " + (-Double.MIN\_VALUE / -0.0));

double d1 = 1.0;

double d2 = 0.0;

System.out.println("d1 / d2 = " + (d1 / d2)); // 변수를 사용해도 된다.

}

}

결과는 다음과 같을 것이다.

0.0 / 0.0 = NaN

-0.0 / 0.0 = NaN

1.0 / 0.0 = Infinity

10.3 / -0.0 = -Infinity

-Double.POSITIVE\_INFINITY / 0.0 = -Infinity

-Double.MIN\_VALUE / -0.0 = Infinity

d1 / d2 = Infinity

다음 예제는 여러가지 헷갈리는 경우를 정리했다.

public class Example {

public static void main(String[] args) {

System.out.println("Double.NaN / 3.0 = " + (Double.NaN / 3.0));

System.out.println("Double.POSITIVE\_INFINITY / Double.POSITIVE\_INFINITY = " + (Double.POSITIVE\_INFINITY / Double.POSITIVE\_INFINITY));

System.out.println("Double.POSITIVE\_INFINITY / -3.0 = " + (Double.POSITIVE\_INFINITY / -3.0));

System.out.println("-3.0 / Double.POSITIVE\_INFINITY = " + (-3.0 / Double.POSITIVE\_INFINITY));

}

}

결과는 다음과 같을 것이다.

Double.NaN / 3.0 = NaN

Double.POSITIVE\_INFINITY / Double.POSITIVE\_INFINITY = NaN

Double.POSITIVE\_INFINITY / -3.0 = -Infinity

-3.0 / Double.POSITIVE\_INFINITY = -0.0

* + 1. 나머지 %

나머지 연산자는 나머지를 구한다.

public class Example {

public static void main(String[] args) {

remainder(7, 3);

remainder(7, -3);

remainder(-7, 3);

remainder(-7, -3);

}

public static void remainder(int dividend, int divisor) {

System.out.println(dividend + " % " + divisor + " = " + (dividend%divisor));

System.out.println("(" + dividend + " / " + divisor + ") \* " + divisor + " + (" + dividend + " % " + divisor + ") = " + ((dividend/divisor)\*divisor + (dividend%divisor)));

}

}

결과는 다음과 같을 것이다.

7 % 3 = 1

(7 / 3) \* 3 + (7 % 3) = 7

7 % -3 = 1

(7 / -3) \* -3 + (7 % -3) = 7

-7 % 3 = -1

(-7 / 3) \* 3 + (-7 % 3) = -7

-7 % -3 = -1

(-7 / -3) \* -3 + (-7 % -3) = -7

위의 예에서 보듯이 정수형에 적용되었을 때는

(dividend/divisor)\*divisor + (dividend%divisor) == dividend

의 관계가 항상 성립한다. (7/3에서 7이 dividend, 3이 divisor)

정수형일 때는 0으로 나눌 수 없다. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

int i = 7 % 0;

}

}

결과는 다음과 비슷할 것이다.

java.lang.ArithmeticException: / by zero

at Example.main(Example.java:3)

Exception in thread "main"

나머지 연산자를 실수형에 사용할 수도 있다.

public class Example {

public static void main(String[] args) {

System.out.println("9.0 % 3.5 = " + (9.0 % 3.5));

System.out.println("9.0 % -3.5 = " + (9.0 % -3.5));

System.out.println("-9.0 % 3.5 = " + (-9.0 % 3.5));

System.out.println("-9.0 % -3.5 = " + (-9.0 % -3.5));

System.out.println("Math.IEEEremainder(9.0, 3.5) = " + Math.IEEEremainder(9.0, 3.5));

System.out.println("Math.IEEEremainder(9.0, -3.5) = " + Math.IEEEremainder(9.0, -3.5));

System.out.println("Math.IEEEremainder(-9.0, 3.5) = " + Math.IEEEremainder(-9.0, 3.5));

System.out.println("Math.IEEEremainder(-9.0, -3.5) = " + Math.IEEEremainder(-9.0, -3.5));

}

}

결과는 다음과 같을 것이다.

9.0 % 3.5 = 2.0

9.0 % -3.5 = 2.0

-9.0 % 3.5 = -2.0

-9.0 % -3.5 = -2.0

Math.IEEEremainder(9.0, 3.5) = -1.5

Math.IEEEremainder(9.0, -3.5) = -1.5

Math.IEEEremainder(-9.0, 3.5) = 1.5

Math.IEEEremainder(-9.0, -3.5) = 1.5

자바의 실수형 %는 IEEE 754 표준과 다르게 정의되었다. 그래서 IEEE 754 표준의 결과를 원하는 사람을 위해 Math.IEEEremainder 라는 메쏘드를 따로 만들어 놓았다.

다음은 실수형에서의 헷가리는 경우이다.

public class Example {

public static void main(String[] args) {

System.out.println("Double.NaN % 3.0 = " + (Double.NaN % 3.0));

System.out.println("Double.POSITIVE\_INFINITY % 3.0 = " + (Double.POSITIVE\_INFINITY % 3.0));

System.out.println("7.0 % 0.0 = " + (7.0 % 0.0));

System.out.println("7.0 % Double.NEGATIVE\_INFINITY = " + (7.0 % Double.NEGATIVE\_INFINITY));

System.out.println("0.0 % 3.0 = " + (0.0 % 3.0));

}

}

결과는 다음과 같을 것이다.

Double.NaN % 3.0 = NaN

Double.POSITIVE\_INFINITY % 3.0 = NaN

7.0 % 0.0 = NaN

7.0 % Double.NEGATIVE\_INFINITY = 7.0

0.0 % 3.0 = 0.0

* + 1. 단항(unary) +,-

모든 정수형의 값 x에 대해

-x == ~x + 1

가 성립한다. 예를 들어 다음과 같다.

public class Example {

public static void main(String[] args) {

int x = 1234;

System.out.println(-x);

System.out.println(~x + 1);

}

}

결과는 다음과 같을 것이다.

-1234

-1234

이것은 심지어 x가 Integer.MIN\_VALUE일 때도 성립한다. 이것은 특별한 경우인데 컴퓨터의 수의 표현(2의 보수)법에서 음수가 양수보다 하나 더 많기 때문에 생기는 현상이다. 다음 예제를 실행해 보자. long형도 마찬가지이다.

public class Example {

public static void main(String[] args) {

int x = Integer.MIN\_VALUE;

System.out.println(x);

System.out.println(-x);

System.out.println(toFullBinaryString(x));

System.out.println(toFullBinaryString(~x));

System.out.println(toFullBinaryString(~x + 1));

System.out.println(~x + 1);

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

-2147483648

-2147483648

10000000000000000000000000000000

01111111111111111111111111111111

10000000000000000000000000000000

-2147483648

* 1. 증감 연산자
     1. ++, --

증감 연산자 ++는 1을 증가시킨다. 마찬가지로 --는 1을 감소시킨다. 이 연산자는 변수에만 사용할 수 있다. 그러므로 다음과 같이 하면 컴파일타임 오류가 난다.

public class Example {

public static void main(String[] args) {

int a = 3++; // Error.

final int b = 23;

b++; // Error.

}

}

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte a = 123;

byte b = ++a; // int 로 promotion 되지 않는다.

System.out.println(b);

double c = 234.5;

System.out.println(++c); // 실수형에도 사용할 수 있다.

}

}

결과는 다음과 같을 것이다.

124

235.5

증감 연산자는 변수의 앞에도 붙을 수 있고 뒤에도 붙을 수 있다. 그 차이를 알아보자.

public class Example {

public static void main(String[] args) {

int a = 3;

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

int b = a++;

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

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

int c = ++a;

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

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

}

}

결과는 다음과 같을 것이다.

a = 3

b = 3

a = 4

c = 5

a = 5

연산자가 앞에 붙느냐 뒤에 붙느냐 하는 차이는 그것이 속한 식에서 연산전의 값이 쓰이느냐 연산후의 값이 쓰이느냐를 결정한다. 위의 예제를 잘 관찰해 보면 알 수 있을 것이다. 사실 속 편하게 다음과 같이 하는 것도 좋은 방법이다. 한 줄 더 많아지긴 했지만 적어도 이렇게 하면 실수할 가능성을 조금이라도 줄일 수 있다.

public class Example {

public static void main(String[] args) {

int x = 3;

int y = ++x; // 한줄로. 1 증가된 값이 대입된다.

System.out.println(y);

x = 3;

x++; // ++x로 써도 무방하다.

y = x; // 한줄이 두줄이 되었지만 실수할 가능성을 줄인다.

System.out.println(y);

}

}

결과는 다음과 같을 것이다.

4

4

* 1. 논리 연산자
     1. !

논리 연산자 !는 논리적 부정(negation)을 구한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

boolean b = true;

System.out.println(b);

System.out.println(!b);

System.out.println(true);

System.out.println(!true);

System.out.println(getTrue());

System.out.println(!getTrue());

}

public static boolean getTrue() {

return true;

}

}

결과는 다음과 같을 것이다.

true

false

true

false

true

false

* + 1. &, |

논리 연산자 &는 AND, |는 OR를 구한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(true & true);

System.out.println(true & false);

System.out.println(false & true);

System.out.println(false & false);

System.out.println();

System.out.println(true | true);

System.out.println(true | false);

System.out.println(false | true);

System.out.println(false | false);

}

}

결과는 다음과 같을 것이다.

true

false

false

false

true

true

true

false

* + 1. &&, ||

논리 연산자 &&는 AND, ||는 OR를 구한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(true && true);

System.out.println(true && false);

System.out.println(false && true);

System.out.println(false && false);

System.out.println();

System.out.println(true || true);

System.out.println(true || false);

System.out.println(false || true);

System.out.println(false || false);

}

}

결과는 다음과 같을 것이다.

true

false

false

false

true

true

true

false

그렇다면 &&와 &의 차이는 무엇인가? 또는 ||와 |의 차이는 무엇인가? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(getTrue() && getTrue());

System.out.println(getTrue() && getFalse());

System.out.println(getFalse() && getTrue());

System.out.println(getFalse() && getFalse());

System.out.println();

System.out.println(getTrue() & getTrue());

System.out.println(getTrue() & getFalse());

System.out.println(getFalse() & getTrue());

System.out.println(getFalse() & getFalse());

System.out.println();

System.out.println(getTrue() || getTrue());

System.out.println(getTrue() || getFalse());

System.out.println(getFalse() || getTrue());

System.out.println(getFalse() || getFalse());

System.out.println();

System.out.println(getTrue() | getTrue());

System.out.println(getTrue() | getFalse());

System.out.println(getFalse() | getTrue());

System.out.println(getFalse() | getFalse());

System.out.println();

}

public static boolean getTrue() {

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

return true;

}

public static boolean getFalse() {

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

return false;

}

}

결과는 다음과 같을 것이다.

getTrue()

getTrue()

true

getTrue()

getFalse()

false

getFalse()

false

getFalse()

false

getTrue()

getTrue()

true

getTrue()

getFalse()

false

getFalse()

getTrue()

false

getFalse()

getFalse()

false

getTrue()

true

getTrue()

true

getFalse()

getTrue()

true

getFalse()

getFalse()

false

getTrue()

getTrue()

true

getTrue()

getFalse()

true

getFalse()

getTrue()

true

getFalse()

getFalse()

false

위의 예제중 하나를 예를 들어 설명하겠다.

System.out.println(getFalse() && getTrue());

의 결과는 다음과 같다.

getFalse()

false

먼저 getFalse()를 부른다. getFalse()는 false를 리턴한다. 이미 하나가 false이기 때문에 나머지 하나를 보지 않아도 AND를 하면 false라는 것을 알 수 있다. 그 때 &&는 나머지 하나를 따져보지 않는다. 이것은 &와 다르다. &는 결과를 이미 알수 있어도 나머지 하나를 따져본다. 그러므로

System.out.println(getFalse() & getTrue());

의 결과는

getFalse()

getTrue()

false

와 같다. ||와 |의 관계도 마찬가지이다.

대부분의 경우에는 || 또는 &&를 사용하는 것이 좋다. 조금이라도 빨리 판단하기 때문에 속도가 조금이라도 빨라지기 때문이다. 그러나 만약 위예 예에서 getTrue() 와 getFalse() 가 반드시 실행되어야 하는 메쏘드라면 & 또는 |를 사용해야 한다.

이번에는 &&를 반드시 사용해야 하는 예를 보자.

public class Example {

public static void main(String[] args) {

A a = null;

if ((a != null) & (a.isTrue())) {

System.out.println("OK");

} else {

System.out.println("Not OK");

}

}

}

class A {

public boolean isTrue() {

return true;

}

}

결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

at Example.main(Example.java:4)

Exception in thread "main"

위의 예제에서 &를 &&로 고쳤다.

public class Example {

public static void main(String[] args) {

A a = null;

if ((a != null) && (a.isTrue())) { // &를 &&로 고침

System.out.println("OK");

} else {

System.out.println("Not OK");

}

}

}

class A {

public boolean isTrue() {

return true;

}

}

결과는 다음과 같을 것이다.

Not OK

* 1. 비트(bit)연산자
     1. ~,&, |, ^

비트 연산자 ~는 NOT, &는 AND, |는 OR, ^는 배타적(exclusive) OR를 나타낸다. 이것은 각각의 비트에 적용된다. 잘 이해가 가지 않으면 예제를 자세히 살펴보면 알 수 있을 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int a = 1234678901;

int b = 999999999;

System.out.println("a = " + toFullBinaryString(a));

System.out.println("b = " + toFullBinaryString(b));

System.out.println("(~a) = " + toFullBinaryString(~a));

System.out.println("(~a) = " + (~a));

System.out.println("(a&b) = " + toFullBinaryString(a&b));

System.out.println("(a&b) = " + (a&b));

System.out.println("(a|b) = " + toFullBinaryString(a|b));

System.out.println("(a|b) = " + (a|b));

System.out.println("(a^b) = " + toFullBinaryString(a^b));

System.out.println("(a^b) = " + (a^b));

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

a = 01001001100101111011010001110101

b = 00111011100110101100100111111111

(~a) = 10110110011010000100101110001010

(~a) = -1234678902

(a&b) = 00001001100100101000000001110101

(a&b) = 160596085

(a|b) = 01111011100111111111110111111111

(a|b) = 2074082815

(a^b) = 01110010000011010111110110001010

(a^b) = 1913486730

이 연산자에도 promotion이 적용된다.

이 연산자가 boolean 형에 쓰이면 비트 연산자가 아니라 논리 연산자가 된다. 이것에 대해 논리 연산자를 참조하라.

* 1. 비교 연산자
     1. ==, !=

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

double a = Double.NaN;

double b = a;

System.out.println("(a==b) = " + (a==b));

System.out.println("(a!=b) = " + (a!=b));

double positiveZero = +0.0;

double negativeZero = -0.0;

System.out.println("(positiveZero==negativeZero) = " + (positiveZero==negativeZero));

System.out.println("(positiveZero!=negativeZero) = " + (positiveZero!=negativeZero));

}

}

결과는 다음과 같을 것이다.

(a==b) = false

(a!=b) = true

(positiveZero==negativeZero) = true

(positiveZero!=negativeZero) = false

NaN는 특별대우를 받는다.

==, != 연산자를 실수에 사용할 때는 주의해야 한다. 컴퓨터의 실수 표현에는 항상 오차가 있을 수 있기 때문이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

float f = 1.0f / 3.0f;

double d = 1.0 / 3.0;

System.out.println(f == d);

System.out.println(f);

System.out.println(d);

}

}

결과는 다음과 비슷할 것이다.

false

0.33333334

0.3333333333333333

==, != 연산자는 boolean 형에도 사용할 수 있다. 그러나 boolean형과 숫자(정수,실수)는 비교될 수 없다. 즉 다음 코드를 컴파일되지 않는다.

public class Example {

public static void main(String[] args) {

boolean b = true;

int i = 1;

if (b == i) { // Error.

}

}

}

이것이 허용된다면 편리하게 사용할 수도 있겠지만 실수할 가능성이 높아진다.

참조형의 ==, !=에 대해서는 Object 클래스를 다룬 부분을 참조하라.

* + 1. <, >, <=, >=

이 연산자들은 대소를 비교하는 것이다. boolean형에는 사용할 수 없다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("(Double.NaN <= Double.NaN) = " + (Double.NaN <= Double.NaN));

System.out.println("(Double.POSITIVE\_INFINITY > 12345.543) = " + (Double.POSITIVE\_INFINITY > 12345.543));

System.out.println("(-0.0 < 0.0) = " + (-0.0 < 0.0));

System.out.println("(-0.0 <= 0.0) = " + (-0.0 <= 0.0));

System.out.println("Math.min(-0.0, 0.0) = " + Math.min(-0.0, 0.0));

System.out.println("Math.max(-0.0, 0.0) = " + Math.max(-0.0, 0.0));

}

}

결과는 다음과 같을 것이다.

(Double.NaN <= Double.NaN) = false

(Double.POSITIVE\_INFINITY > 12345.543) = true

(-0.0 < 0.0) = false

(-0.0 <= 0.0) = true

Math.min(-0.0, 0.0) = -0.0

Math.max(-0.0, 0.0) = 0.0

<, <=, >, >= 연산을 할 때 둘 중 하나 또는 둘 모두가 NaN이면 결과는 무조건 false이다. Infinity도 비교의 대상이 된다. 즉 양의 무한대는 가장 큰 것으로 음의 무한대는 가장 작은 것으로 취급된다. –0.0과 0.0은 같은 것으로 취급된다. 이것은 수학적으로 당연하다. 그러나 Math.min, Math.max 메쏘드는 위에서 보듯이 둘을 구분한다.

* 1. 대입(assignment) 연산자
     1. =

대입 연산자의 왼쪽에는 대입될 수 있는 변수가 와야 한다. 그렇지 않으면 컴파일타임 오류가 난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

301 = 302; // Error.

final int finalVariable = 3;

finalVariable = 4; // Error.

final int blankFinal; // 아직 초기화되지 않음.

blankFinal = 12; // OK.

method() = 432; // Error.

}

public static int method() {

int i = 10;

return i;

}

}

컴파일이 되지 않을 것이다.

오른쪽에 있는 값은 왼쪽 변수에 대입할 수 있어야 한다. 즉 두 타입이 같거나, 오른쪽의 값이 넓히는 형변환(widening conversion)으로 왼쪽 타입이 되거나, 아니면 명시적 캐스팅(casting)을 해야 한다. 그렇지 않으면 컴파일 에러가 난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte byteVariable;

byteVariable = 3.0; // Error.

byteVariable = (byte)3.0; // OK. 명시적 캐스팅

long longVariable = 3; // OK. 넓히는 형변환

byteVariable = (byte)true; // Error.

}

}

컴파일이 되지 않을 것이다.

아래에 나오는 예제들은 조금 어렵다. 잘 모르겠으면 클래스와 예외상황에 대해 배운 다음에 보면 된다. 왼쪽의 변수가 먼저 준비되고 그 다음에 오른쪽의 값이 평가(evalutate, 계산)된다. 다음 예제를 보면 그것을 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

int[] a = new int[4];

a[index()] = rightHand();

System.out.println();

A.getA().x = rightHand();

}

public static int index() {

System.out.println("Example.index()");

return 2;

}

public static int rightHand() {

System.out.println("Example.rightHand()");

return 123;

}

}

class A {

public int x;

public static A getA() {

System.out.println("A.getA()");

return new A();

}

}

결과는 다음과 같을 것이다.

Example.index()

Example.rightHand()

A.getA()

Example.rightHand()

대입 연산자를 사용한 식은 그 자체가 또 값이 되기도 한다. 다음을 보자.

public class Example {

public static void main(String[] args) {

int a, b, c;

a = b = c = 15;

System.out.println(a);

System.out.println(b);

System.out.println(c);

}

}

결과는 다음과 같을 것이다.

15

15

15

위의 예제에 있는 식을 이해 하기 쉽게 쓰면

a = (b = (c = 15));

와 같다. c = 15라는 대입식 자체도 대입된 15라는 값을 가지고 있기 때문에 그것을 b에 대입할 수 있었던 것이다. 그러나 이런 것을 이용하는 것이 좋은 코딩 스타일인지는 의문이다.

그리고 이런 것 때문에 버그가 생기기도 하는데 그것은 if문을 다룬 부분에서 다룬 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

boolean b = false;

if (b = true)

System.out.println("b는 참이다");

}

}

결과는 다음과 같을 것이다.

b는 참이다

b = true라는 식 자체가 대입된 값인 true값을 가지기 때문에 위와 같은 결과가 나온 것이다.

* + 1. 복합 대입(compound assignment) 연산자 +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=

복합 대입 연산자에 대해 알아보자. 예를 들어

x = x + 12;

는

x += 12;

로 바뀔 수 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int x = 10;

System.out.println(x);

x += 10;

System.out.println("(after +=) : " + x);

x -= 5;

System.out.println("(after -=) : " + x);

x \*= 2;

System.out.println("(after \*=) : " + x);

x /= 3;

System.out.println("(after /=) : " + x);

x %= 4;

System.out.println("(after %=) : " + x);

x &= 6;

System.out.println("(after &=) : " + x);

x |= 4;

System.out.println("(after |=) : " + x);

x ^= 4;

System.out.println("(after ^=) : " + x);

x <<= 4;

System.out.println("(after <<=) : " + x);

x >>= 2;

System.out.println("(after >>=) : " + x);

x >>>= 2;

System.out.println("(after >>>=) : " + x);

}

}

결과는 다음과 같을 것이다.

10

(after +=) : 20

(after -=) : 15

(after \*=) : 30

(after /=) : 10

(after %=) : 2

(after &=) : 2

(after |=) : 6

(after ^=) : 2

(after <<=) : 32

(after >>=) : 8

(after >>>=) : 2

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 10;

b = b + 1; // Error.

b += 1; // OK.

b++; // OK.

}

}

컴파일타임 오류가 왜 나는지 모르겠으면 promotion을 다룬 부분을 보면 알 수 있을 것이다.

* 1. 쉬프트(shift) 연산자
     1. >>, >>>, <<

쉬프트 연산자에 대해 알아보자. 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = 1234567890;

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

System.out.println(i + "<<" + 4 + " = " + toFullBinaryString(i << 4) + " = " + (i << 4));

System.out.println(i + ">>" + 4 + " = " + toFullBinaryString(i >> 4) + " = " + (i >> 4));

System.out.println(i + ">>>" + 4 + " = " + toFullBinaryString(i >>> 4) + " = " + (i >>> 4));

i = -1234567890;

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

System.out.println(i + "<<" + 4 + " = " + toFullBinaryString(i << 4) + " = " + (i << 4));

System.out.println(i + ">>" + 4 + " = " + toFullBinaryString(i >> 4) + " = " + (i >> 4));

System.out.println(i + ">>>" + 4 + " = " + toFullBinaryString(i >>> 4) + " = " + (i >>> 4));

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

1234567890 = 01001001100101100000001011010010

1234567890<<4 = 10011001011000000010110100100000 = -1721750240

1234567890>>4 = 00000100100110010110000000101101 = 77160493

1234567890>>>4 = 00000100100110010110000000101101 = 77160493

-1234567890 = 10110110011010011111110100101110

-1234567890<<4 = 01100110100111111101001011100000 = 1721750240

-1234567890>>4 = 11111011011001101001111111010010 = -77160494

-1234567890>>>4 = 00001011011001101001111111010010 = 191274962

2진수를 잘 보면 쉬프트가 어떻게 일어나는지 알수 있을 것이다. <<은 왼쪽으로 쉬프트하면서 오른쪽 끝을 0으로 채운다. >>>는 zero extension 방식을 취한다. 즉 오른쪽으로 쉬프트하면서 왼쪽을 0으로 채운다. >>는 sign extension 방식을 취한다. 음수이면 최상위 비트(제일 왼쪽 비트)가 1인데 이 때는 1로 채운다. 양수 또는 0이면 최상위 비트가 0인데 이 때는 0으로 채운다. 이렇게 하면 값의 부호가 보존된다.

쉬프트 연산자로 2를 제곱한 수(예를 들어 2, 4, 8, 16, 32…)의 곱하기, 나누기를 대체할 수 있다. 그렇게 하면 \*나 / 연산자를 쓰는 것보다 속도가 빠르다(그러나 \*, / 를 쓰더라도 영리한 컴파일러가 최적화해서 차이가 없을 수도 있다). 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int initValue = 3;

for (int i = 1; i < 7; i++) {

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

System.out.println(initValue + "\*" + "Math.pow(" + 2 + "," + i + ") = " + (initValue \* Math.pow(2 , i)));

}

System.out.println();

initValue = 192;

for (int i = 1; i < 7; i++) {

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

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

System.out.println(initValue + "\*" + "Math.pow(" + .5 + "," + i + ") = " + (initValue \* Math.pow(.5 , i)));

}

System.out.println();

initValue = -192;

for (int i = 1; i < 7; i++) {

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

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

System.out.println(initValue + "\*" + "Math.pow(" + .5 + "," + i + ") = " + (initValue \* Math.pow(.5 , i)));

}

}

}

결과는 다음과 비슷할 것이다.

3<<1 = 6

3\*Math.pow(2,1) = 6.0

3<<2 = 12

3\*Math.pow(2,2) = 12.0

3<<3 = 24

3\*Math.pow(2,3) = 24.0

3<<4 = 48

3\*Math.pow(2,4) = 48.0

3<<5 = 96

3\*Math.pow(2,5) = 96.0

3<<6 = 192

3\*Math.pow(2,6) = 192.0

192>>1 = 96

192>>>1 = 96

192\*Math.pow(0.5,1) = 96.0

192>>2 = 48

192>>>2 = 48

192\*Math.pow(0.5,2) = 48.0

192>>3 = 24

192>>>3 = 24

192\*Math.pow(0.5,3) = 24.0

192>>4 = 12

192>>>4 = 12

192\*Math.pow(0.5,4) = 12.0

192>>5 = 6

192>>>5 = 6

192\*Math.pow(0.5,5) = 6.0

192>>6 = 3

192>>>6 = 3

192\*Math.pow(0.5,6) = 3.0

-192>>1 = -96

-192>>>1 = 2147483552

-192\*Math.pow(0.5,1) = -96.0

-192>>2 = -48

-192>>>2 = 1073741776

-192\*Math.pow(0.5,2) = -48.0

-192>>3 = -24

-192>>>3 = 536870888

-192\*Math.pow(0.5,3) = -24.0

-192>>4 = -12

-192>>>4 = 268435444

-192\*Math.pow(0.5,4) = -12.0

-192>>5 = -6

-192>>>5 = 134217722

-192\*Math.pow(0.5,5) = -6.0

-192>>6 = -3

-192>>>6 = 67108861

-192\*Math.pow(0.5,6) = -3.0

왼쪽의 숫자가 int형이면 오른쪽 숫자의 lowest order의 5개의 비트만(이것은 0에서 31까지의 숫자이다)만 의미가 있다. 마찬가지로 왼쪽의 숫자가 long형이면 오른쪽 숫자의 lowest order의 6개의 비트(이것은 0에서 63까지의 숫자이다)만 있다. 이것의 의미는 다음 예제를 보면 분명해질 것이다.

public class Example {

public static void main(String[] args) {

int left = 1234567890;

int right = 1131;

System.out.println(left + " = " + toFullBinaryString(left));

System.out.println(right + " = " + toFullBinaryString(right));

System.out.println(left + ">>" + right + " = " + toFullBinaryString(left >> right) + " = " + (left>>right));

System.out.println(right + "&0x1f = " + toFullBinaryString(right & 0x1f) + " = " + (right&0x1f) );

System.out.println(left + ">>" + (right&0x1f) + " = " + toFullBinaryString(left >> (right&0x1f)) + " = " + (left>>(right&0x1f)));

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

1234567890 = 01001001100101100000001011010010

1131 = 00000000000000000000010001101011

1234567890>>1131 = 00000000000010010011001011000000 = 602816

1131&0x1f = 00000000000000000000000000001011 = 11

1234567890>>11 = 00000000000010010011001011000000 = 602816

* 1. 조건(conditional) 연산자
     1. ? :

이 조건 연산자는 if문의 어떤 경우를 줄여서 썼다고 생각하면 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i;

boolean b;

b = true;

if (b)

i = 1234;

else

i = -4312;

System.out.println(i);

i = (b ? 1234 : -4312);

System.out.println(i);

b = false;

if (b)

i = 1234;

else

i = -4312;

System.out.println(i);

i = (b ? 1234 : -4312);

System.out.println(i);

}

}

결과는 다음과 같을 것이다.

1234

1234

-4312

-4312

* 1. 기타
     1. 문자열 연견(string concatenation) 연산자 +

문자열을 다루는 부분을 참조하라.

* + 1. instanceof

리플렉션을 다루는 부분을 참조하라.

* 1. 연산자의 우선순위
     1. 연산자의 우선 순위

연산자에는 우선 순위가 있다. 그 순서는 다음 표와 같다. 제일 위에 있는 것이 가장 먼저 연산된다.

|  |
| --- |
| [] . 메쏘드의() x++ x—  ++x --x +x -x ~ !  new 캐스트의()  \* / %  더하기+ 빼기-  << >> >>>  < > >= <= instanceof  == !=  &  ^  |  &&  ||  ?:  = += -= \*= /= %= >>= <<= >>>= &= ^= |= |

다음 예제는 연산자의 우선 순위를 보여준다.

public class Example {

public static void main(String[] args) {

System.out.println(1 + 2 \* 3);

}

}

결과는 다음과 같을 것이다.

7

* + 1. 괄호

괄호 ()를 사용하면 연산자의 우선 순위를 무시할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println((1 + 2) \* 3);

}

}

결과는 다음과 같을 것이다.

9

괄호의 사용은 꼭 필요하지 않은 곳에서도 하는 것이 좋을 때가 많다. 왜냐하면 모든 사람이 연산자의 우선 순위의 모든 것을 다 외우는 것은 아니기 때문이다. 그리고 괄호를 사용하지 않고 연산자의 우선 순위에만 의지하면 실수를 할 가능성이 높아진다. 그러나 괄호를 남용하는 것도 좋지 않다. 괄호를 많이 사용할수록 코드의 가독성이 떨어지기 때문이다. 또는 당신의 코드를 LISP 언어의 코드로 오해할 수도 있다. 수많은 괄호의 짝을 찾는 것만큼 짜증나는 일도 없다. 그러므로 괄호를 적당한 수준에서 사용해야 한다. 만약 괄호가 너무 많아지거나 식이 너무 길어지면 식을 쪼개서 중간에 다른 변수에 저장하는 방식이 좋다. 그리고 그 변수에 의미 있는 이름을 부여한다면 코드를 읽는 시간을 훨씬 줄일 수 있다.

* 1. promotion과 캐스팅(casting)
     1. promotion

연산을 할 때에는 예를 들어 두 수를 더하려면 두 수의 타입이 같아야 한다. 두 피연산자(operand)의 타입을 같게 하기 위해 하는 것이 promotion이다. promotion은 넓히는 기본형 형변환(widening primitive conversion)과 관계가 있다.

* + 1. binary promotion

binary promotion은 이항 연산자(binary operator)가 연산될 때 일어난다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b1 = 3;

byte b2 = 4;

byte b3 = b1 + b2; // Error.

short sh = b1 + b2; // Error.

char ch = b1 + b2; // Error.

int i = b1 + b2; // OK.

long l = b1 + b2; // OK.

float f = b1 + b2; // OK.

double d = b1 + b2; // OK.

}

}

컴파일타임 오류가 난다. 이것은 초보자에게는 놀라운 일일 것이다. 왜 이런 일이 발생할까? b1 도 byte 형이고 b2도 byte 형이지만 두 값은 먼저 int형으로 promotion된 다음에 + 연산이 되는 것이다. 그러므로 결과값은 int형이다.

promotion에는 승진, 승격, 승급, 진급이라는 뜻이 있다. promotion은 정확히 어떻게 일어날까?

* 만약 두 피연산자(위의 예제에서는 b1, b2)중 하나가 double이면 둘은 모두 double이어야 한다. 아닐 경우 double로 promotion된다.
* 아니면, 만약 두 피연산자 중 하나가 float이면 둘은 모두 float이어야 한다. 아닐 경우 float로 promotion된다.
* 아니면, 만약 두 피연산자 중 하나가 long이면 둘은 모두 long이어야 한다. 아닐 경우 long으로 promotion된다.
* 아니면, 둘은 모두 int이어야 한다. 아닐 경우 int으로 promotion된다.

promotion이 일어나는 연산자에는 첫번째로 사칙연산이 있다. +, -, \*, /, % 등이 그것이다.

비교를 하는 연산자에도 promotion이 필요하다. <, <=, >, >=, ==, != 등이 그것이다.

비트 연산자에도 promotion이 필요하다. &, ^, |등이 그것이다.

어떤 경우에는 ? : 연산자에도 promotion이 필요하다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte byteValue = 1;

int intValue = 1;

boolean b = true;

byte byteVariable = b ? byteValue : byteValue; // OK.

int intVariable = b ? byteValue : byteValue; // OK.

byteVariable = b ? byteValue : intValue; // Error.

intVariable = b ? byteValue : intValue; // OK.

}

}

한군데서 컴파일타임 오류가 난다.

* + 1. unary promotion

어떤 경우에는 단항 연산자(unary operator)에도 promotion이 일어난다. 단항 연산자에서 일어나는 promotion은 좀더 단순하다. byte, short, char가 int형으로 변환되는 것이다.

배열을 생성하거나 배열에 접근하기 위한 인덱스 값에 promotion이 일어난다. 배열 인덱스 값은 int형이기 때문이다. 그러나 이 사실을 몰라도 상관은 없다.

부호를 나타내는 단항 연산자 +, -에도 promotion이 일어난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 23;

b = +23; // OK.

b = -23; // OK.

b = -b; // Error.

b = +b; // Error.

int i = -b; // OK.

i = +b; // OK.

}

}

초보자들에게는 놀랍게도 컴파일 에러가 난다. 왜냐하면 –b가 연산되면서 promotion이 일어나서 int 형으로 바뀌었기 때문이다.

비트연산자인 ~의 사용에도 promotion이 일어난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = -1;

b = ~b; // Error.

}

}

위의 예제와 마찬가지로 컴파일타임 오류가 난다.

다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

byte b = -1;

int i = ~b;

System.out.println(i);

}

}

결과는 다음과 같을 것이다.

0

그럼 결과가 왜 0이 되었는지 한번 추적해 보자. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

byte b = -1;

int promotedValue = b;

System.out.println(promotedValue);

System.out.println(toFullBinaryString(promotedValue));

int i = ~promotedValue;

System.out.println(i);

System.out.println(toFullBinaryString(i));

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

-1

11111111111111111111111111111111

0

00000000000000000000000000000000

먼저 –1이 int형 –1로 변환된다. –1을 이진수로 나타내면 11111111111111111111111111111111가 된다. 이것은 ~연산하면 00000000000000000000000000000000이 된다. 이것은 0이다.

shift 연산자 >>, >>>, <<에 적용되는 promotion은 binary promotion이 아니라 unary promotion이다. 먼저 binary promotion이 아니라는 증거를 보자. 예제를 보자.

public class Example {

public static void main(String[] args) {

long l = 1;

int i = 2;

int intVariable = i << l; // 에러가 나지 않는다.

}

}

컴파일이 에러 없이 잘 된다. 만약 << 연산을 하기 위해 binary promotion이 일어난다면 결과 값은 long형이 될 것이고 intVariable이 int형이기 때문에 컴파일타임 오류가 날 것이다.

이번에는 unary promotion이 일어난다는 증거를 보자.

public class Example {

public static void main(String[] args) {

byte b1 = 1;

byte b2 = 2;

byte b = b1 << b2; // Errror.

int i = b1 << b2; // OK.

}

}

컴파일타임 오류가 날 것이다. 그렇다면 결과값의 타입은 어떻게 결정될까?

public class Example {

public static void main(String[] args) {

int intValue = 1;

long longValue = 2;

int intVariable = intValue << longValue; // OK.

intVariable = longValue << intValue; // Error.

}

}

컴파일타임 오류가 난다. 왼쪽에 있는 값의 타입이 결과 값의 타입이 된다.

* + 1. 캐스팅(casting)

타입(type)이 다른데다가 변환(conversion)하는 것이 값을 심각하게 손상될 가능성이 있을 때는 즉 좁히는 기본형 변환(narrowing primitive conversion)일 때에는 명시적으로 캐스팅을 해야 한다. 그렇지 않으면 컴파일타임 오류가 난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int intValue = 32;

byte b = intValue; // Error.

short sh = intValue; // Error.

char ch = intValue; // Error.

int i = intValue; // OK.

long l = intValue; // OK.

float f = intValue; // OK.

double d = intValue; // OK.

}

}

이럴 때는 명시적으로 캐스팅을 해야 한다. 캐스팅을 하는 방법은 다음과 같다.

public class Example {

public static void main(String[] args) {

int intValue = 32;

byte b = (byte)intValue; // OK.

short sh = (short)intValue; // OK.

char ch = (char)intValue; // OK.

}

}

캐스팅은 컴파일러에게 다음과 같이 말해주는 것이다. “나도 이런 식으로 타입을 변환하면 위험한 줄 알고 있다. 그럼에도 필요해서 하는 것이니 그 책임은 내가 지겠다”.

캐스팅을 다음과 같이 사용할 수도 있으나 대부분의 경우 오히려 헷갈리게 한다.

public class Example {

public static void main(String[] args) {

int intValue = 322;

short sh = (byte)intValue; // OK. 그러나 바보같은 짓이다.

System.out.println(sh);

int i = (int)intValue; // OK. 그러나 의미가 없다.

}

}

결과는 다음과 같을 것이다.

66

캐스팅이 언제나 가능한 것은 아니다. 다음 예제는 금지된 캐스팅을 다룬다.

public class Example {

public static void main(String[] args) {

int intValue = 32;

String str = (String)intValue; // Error.

boolean booleanValue = (boolean)32; // Error.

Object obj = (Object)intValue; // Error.

Integer integer = (Integer)intValue; // Error.

}

}

컴파일이 되지 않을 것이다.

* + 1. 캐스팅에서 주의할 점

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

double d1 = 3.5;

double d2 = 5.5;

int i = (int)d1 + (int)d2;

System.out.println(i);

i = (int)(d1 + d2);

System.out.println(i);

}

}

결과는 다음과 같을 것이다.

8

9

캐스팅을 어떻게 하느냐에 따라 좀더 정확한 결과가 나올수도 아닐 수도 있다.

1. 배열의 기초
   1. 기본형의 배열
      1. 기본형의 1차원 배열

평균을 구하는 프로그램을 만들어 보자. 5개의 실수의 평균을 구하는 프로그램이다.

public class Example {

public static void main(String[] args) {

double[] doubleArray; // 배열 변수의 선언

doubleArray = new double[5]; // 배열 객체의 생성

doubleArray[0] = 0.5;

doubleArray[1] = 0;

doubleArray[2] = 1;

doubleArray[3] = 1.5;

doubleArray[4] = 2;

double sum = 0;

for (int i = 0; i < 5; i++) {

sum += doubleArray[i];

}

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

System.out.println("average = " + (sum / 5));

}

}

결과는 다음과 같을 것이다.

sum = 5.0

average = 1.0

배열 변수는 다음과 같이 선언한다.

double[] doubleArray; // 배열 변수의 선언

여기서 double[]는 double형 일차원 배열을 뜻한다. 그러므로 double형 일차원 배열 변수인 doubleArray가 선언된 것이다. 다음과 같이 해도 된다.

double doubleArray[]; // 배열 변수의 선언

이렇게 해도 double형 일차원 배열 변수 doubleArray가 선언된다. 그러나 이렇게 하면 형(type)에 대한 정보가 분산된다. double[]에서처럼 하나로 묶어서 하는 방법이 더 알아보기 쉽다.

배열 변수만 선언해서는 배열을 사용할 수 없다. 배열 객체를 생성해야만 한다. 배열 객체는 다음과 같이 생성한다.

doubleArray = new double[5]; // 배열 객체의 생성

여기서 new double[5]는 double형 일차원 배열 객체를 생성함을 뜻한다. 배열 크기는 5이다. 이것은 double형 변수가 5개 만들어지는 것과 같다. 이렇게 만들어진 배열 객체의 참조가 doubleArray라는 변수에 대입된다. 쉬운 말로 하면 doubleArray이라는 변수가 만들어진 배열 객체를 가리키는 것이다.

그 다음에 배열을 어떻게 사용하는지 알아보자. 배열은 다음과 같이 사용한다.

doubleArray[0] = 0.5;

여기서 doubleArray[0]은 이 배열의 0번째(첫번째)에 0.5를 대입하라는 뜻이다. 0과 같이 []안에 들어가는 숫자를 배열 인덱스라고 한다. 자바에서 배열 인덱스는 0부터 시작하는 정수다. 크기가 5이니까 배열 인덱스는 0부터 4까지 있다.

sum += doubleArray[i];

위에서는 배열에 어떤 값을 대입하는 것을 보여주었다면 여기서는 배열의 값을 알아내는 방법을 보여주고 있다. doubleArray[i]는 인덱스가 i인 배열 요소의 값을 sum에 더하고 있다. for문과 배열이 결합해서 편리하게 사용되고 있다.

* + 1. 기본형의 다차원 배열

다시 평균을 구하는 프로그램을 만들어 보자. 이번에도 5개의 실수의 평균을 구한다. 그러나 이번에는 5개의 실수의 묶음이 3개 있다. 각각의 묶음의 평균을 구하는 것이다.

public class Example {

public static void main(String[] args) {

double[][] doubleArray;

doubleArray = new double[3][5];

doubleArray[0][0] = 0;

doubleArray[0][1] = 0.5;

doubleArray[0][2] = 1;

doubleArray[0][3] = 1.5;

doubleArray[0][4] = 2;

doubleArray[1][0] = 1;

doubleArray[1][1] = 1.5;

doubleArray[1][2] = 2;

doubleArray[1][3] = 3.5;

doubleArray[1][4] = 3;

doubleArray[2][0] = 2;

doubleArray[2][1] = 2.5;

doubleArray[2][2] = 3;

doubleArray[2][3] = 3.5;

doubleArray[2][4] = 4;

double[] sum = new double[3];

for (int i = 0; i < 3; i++) {

for (int j = 0; j < 5; j++) {

sum[i] += doubleArray[i][j];

}

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

System.out.println("average = " + (sum[i] / 5));

System.out.println();

}

}

}

결과는 다음과 같을 것이다.

sum[0] = 5.0

average = 1.0

sum[1] = 11.0

average = 2.2

sum[2] = 15.0

average = 3.0

위의 프로그램의 값들을 정리하면 다음과 같다.

0, 0.5, 1, 1.5, 2 : 평균 1.0

1, 1.5, 2, 3.5, 3 : 평균 2.2

2, 2.5, 3, 3.5, 4 : 평균 3.0

이런식으로 2차원 뿐 아니라 더 높은 차원의 배열을 만들어서 사용할 수 있다. 예를 들어 3차원, 4차원, 5차원 배열 변수를 정의하고 배열 객체를 만들어서 그 참조를 그 변수에 대입하려면 다음과 같이 하면 된다.

int[][][] three = new int[10][10][10];

int[][][][] four = new int[10][10][10][10];

int[][][][][] five = new int[10][10][10][10][10];

다음과 같이해도 똑같은 일을 한다.

int three[][][] = new int[10][10][10];

int four[][][][] = new int[10][10][10][10];

int five[][][][][] = new int[10][10][10][10][10];

이렇게 할 수도 있다.

int[][][] three = new int[10][10][10];

int[][][] four[] = new int[10][10][10][10];

int[][][] five[][] = new int[10][10][10][10][10];

더 나아가 다음과 같이 할 수도 있다.

int[][][] three, four[], five[][];

three = new int[10][10][10];

four = new int[10][10][10][10];

five = new int[10][10][10][10][10];

그러나 이런식으로 하면 보는 사람이 헷갈릴 것이다.

엄격히 말하면 자바의 배열은 다차원 배열이 아니라 배열의 배열이다. 다차원 배열과 배열의 배열의 차이는 뒤에서 배우게 될 것이다.

* 1. 참조형의 배열
     1. 클래스의 배열

배열에 객체를 저장할 수 있다. 즉 참조형의 배열을 만들 수도 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

A[] arrayOfA = new A[4];

arrayOfA[0] = new A(3);

arrayOfA[1] = new A(5);

arrayOfA[2] = new A(6);

arrayOfA[3] = new A(7);

for (int i = 0; i < 4; i++)

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

}

}

class A {

int intValue;

A(int i) {

intValue = i;

}

public String toString() {

return "[Class A : intValue = " + intValue + "]";

}

}

실행 결과는 다음과 같다.

arrayOfA[0] = [Class A : intValue = 3]

arrayOfA[1] = [Class A : intValue = 5]

arrayOfA[2] = [Class A : intValue = 6]

arrayOfA[3] = [Class A : intValue = 7]

프로그램의 중간에 보면 다음과 같은 부분이 있을 것이다.

arrayOfA[i].toString();

여기서 arrayOfA는 A형의 일차원 배열이다. 그러므로 arrayOfA[i]는 A형의 참조 변수가 된다. 그래서 위와 같이 A형에 정의된 toString()메쏘드를 사용할 수 있는 것이다.

* + 1. 수퍼 클래스의 배열

다음 예를 보자.

public class Example {

public static void main(String[] args) {

SuperClass[] arraySuperClass = new SuperClass[3];

arraySuperClass[0] = new SuperClass();

arraySuperClass[1] = new SubClassA();

arraySuperClass[2] = new SubClassB();

for (int i = 0; i < 3; i++)

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

}

}

class SuperClass {

public String toString() {

return "SuperClass";

}

}

class SubClassA extends SuperClass {

public String toString() {

return "SubClass A";

}

}

class SubClassB extends SubClassA {

public String toString() {

return "SubClass B";

}

}

결과는 다음과 같다.

arraySuperClass[0] = SuperClass

arraySuperClass[1] = SubClass A

arraySuperClass[2] = SubClass B

이 예에서 SubClassA는 SuperClass를 상속받았고 SubClassB는 SubClassA를 상속받았다. 세 개의 클래스의 객체를 모두 SuperClass의 배열에 넣고 for문에서 한꺼번에 다루는 것을 볼 수 있을 것이다. 클래스형의 배열에는 그것의 서브 클래스의 객체를 넣을 수 있다. 이것은 자바 배열의 유연성을 보여주는 예이다.

만약 Object형의 배열을 만든다면 그 배열에 어떤 객체도 넣을 수 있을 것이다. 왜냐하면 모든 객체는 Object클래스로부터 상속받기 때문이다. 다음 예를 보자.

import java.awt.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object[] arrayObject = new Object[5];

arrayObject[0] = new Object();

arrayObject[1] = new Integer(23);

arrayObject[2] = new Panel();

arrayObject[3] = new ArrayList();

arrayObject[4] = new int[10];

for (int i = 0; i < 5; i++)

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

}

}

결과는 다음과 비슷할 것이다.

arrayObject[0] = java.lang.Object@7943a4

arrayObject[1] = 23

arrayObject[2] = java.awt.Panel[panel0,0,0,0x0,invalid,layout=java.awt.FlowLayout]

arrayObject[3] = []

arrayObject[4] = [I@14fe5c

Object형의 배열에는 여러가지 객체, 심지어 배열까지도 넣을 수 있다는 것을 알 수 있다. 왜냐하면 배열도 객체이기 때문이다.

toString 메쏘드는 Object클래스에 정의되어 있기 때문에 모든 객체에는 이 메쏘드가 있다.

* + 1. 인터페이스의 배열

인터페이스의 배열을 만들 수도 있다. 인터페이스 자체의 객체를 생성할 수는 없지만 그 인터페이스를 구현(implements)하는 객체를 저장할 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

SuperInterface[] arrayInterface = new SuperInterface[2];

arrayInterface[0] = new SubClassA();

arrayInterface[1] = new SubClassB();

for (int i = 0; i < 2; i++)

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

}

}

interface SuperInterface {

String print();

}

class SubClassA implements SuperInterface {

public String print() {

return "SubClass A";

}

}

class SubClassB implements SuperInterface {

public String print() {

return "SubClass B";

}

}

결과는 다음과 같다.

arrayInterface[0] = SubClass A

arrayInterface[1] = SubClass B

SubClassA와 SubClassB는 SupreInterface라는 인터페이스를 구현했기 때문에 그 인터페이스의 배열에 값을 저장할 수 있다.

* 1. 배열의 초기화
     1. 기본형 배열의 디폴트 초기값

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

boolean[] booleanArray = new boolean[4];

int[] intArray = new int[4];

double[] doubleArray = new double[4];

char[] charArray = new char[4];

System.out.println("booleanArray[0] = " + booleanArray[0]);

System.out.println("intArray[0] = " + intArray[0]);

System.out.println("doubleArray[0] = " + doubleArray[0]);

System.out.println("charArray[0] = " + charArray[0]);

System.out.println("(int)(charArray[0]) = " + (int)(charArray[0]));

}

}

결과는 다음과 같다.

booleanArray[0] = false

intArray[0] = 0

doubleArray[0] = 0.0

charArray[0] =

(int)(charArray[0]) = 0

배열의 디폴트 값으로는 어떤 값이 들어가는지 알 수 있을 것이다. boolean형에는 false, 정수와 실수형에는 0, char형에는 \u0000이 들어간다.

charArray[0] =

와 같이 아무것도 나오지 않는 것은 유니코드 0번에 해당하는 문자(null문자)가 원래 그렇게 생겼기 때문이다. int형으로 형변환하자 0이 표시된다.

* + 1. 참조형 배열의 디폴트 초기값

다음 예를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList[] arrayOfArrayList = new ArrayList[4];

for (int i = 0; i < 4; i++)

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

}

}

결과는 다음과 같다.

arrayOfArrayList[0] = null

arrayOfArrayList[1] = null

arrayOfArrayList[2] = null

arrayOfArrayList[3] = null

처음에 배열 객체가 만들어졌을 때는 모든 배열 요소(array component)가 null을 가리키고 있음을 알 수 있다.

* + 1. 배열의 초기화

배열을 만들고 그 값을 설정하자.

public class Example {

public static void main(String[] args) {

int[] intArray = new int[4];

intArray[0] = 10;

intArray[1] = 20;

intArray[2] = 30;

intArray[3] = 40;

for (int i = 0; i < 4; i++)

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

}

}

배열 변수를 선언하면서 배열을 초기화 할 수 있다. 다음 프로그램은 위의 예와 같은 기능을 한다.

public class Example {

public static void main(String[] args) {

int[] intArray = {10, 20, 30, 40};

for (int i = 0; i < 4; i++)

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

}

}

{와 } 사이에 콤마로 구분해서 값을 넣어주면 된다.

다차원 배열은 다음과 같이하면 된다.

public class Example {

public static void main(String[] args) {

int[][] intArray = { {10, 20, 30}, {40, 50, 60}, {70, 80, 90}};

for (int i = 0; i < 3; i++)

for (int j = 0; j < 3; j++)

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

}

}

결과는 다음과 같다.

intArray[0][0] = 10

intArray[0][1] = 20

intArray[0][2] = 30

intArray[1][0] = 40

intArray[1][1] = 50

intArray[1][2] = 60

intArray[2][0] = 70

intArray[2][1] = 80

intArray[2][2] = 90

{}를 이용해 초기화하는 방법은 배열을 선언함과 동시에 이루어져야 한다. 즉 다음과 같이 할 수는 없다.

int[] array;

array = {2, 3, 4}; // Errror

* 1. final 배열

final변수는 한번 지정해 주면 변할 수 없는 변수이다. 즉 참조값을 바꿀 수 없다. 배열 변수도 final로 지정해 줄수 있다. 다음 예제를 컴파일해 보자.

public class Example {

public static void main(String[] args) {

final int[] finalArray = {1, 2, 3, 4};

int[] anotherArray = {10, 20};

finalArray = anotherArray;

}

}

컴파일이 안되고 다음과 비슷한 오류 메시지(error message)가 나올 것이다.

"Example.java": Error #: 554 : final variable finalArray might already have been assigned to at line 7, column 3

finalArray변수가 final로 지정되어 있기 때문에 anotherArray를 대입할 수 없었다.

그러나 final이 배열 객체를 변하지 못하게 할 수 있다고 생각하면 안된다. final이 할 수 있는 것은 배열 변수가 바뀌지 못하게 하는 것이 전부다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

final int[] finalArray = {1, 2, 3, 4};

for (int i = 0; i < finalArray.length; i++)

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

for (int i = 0; i < finalArray.length; i++)

finalArray[i] = i \* 10;

System.out.println();

for (int i = 0; i < finalArray.length; i++)

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

}

}

결과는 다음과 같다.

finalArray[0] =1

finalArray[1] =2

finalArray[2] =3

finalArray[3] =4

finalArray[0] =0

finalArray[1] =10

finalArray[2] =20

finalArray[3] =30

아무 문제 없이 배열의 내용이 바뀌었다. 일차원 배열일 때는 배열의 크기가 변하지 않을 뿐이다. 다차원 배열일 때는 최상위 배열의 크기가 변하지 않는다. 하위 배열은 마음대로 바꿀 수 있다.

* 1. 크기가 0인 배열

자바의 배열은 크기가 0일 수도 있다. 즉 다음과 같이 배열 객체를 생성할 수 있다.

int[] zeroArray = new int[0];

이런 배열을 어디에 써먹을까? 다음 예를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(new Object());

al.add(new Object());

Object[] ao = al.toArray();

printArray(ao);

al.clear(); // 모두 제거한다.

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

ao = al.toArray();

printArray(ao);

}

private static void printArray(Object[] array) {

System.out.println("array.length = " + array.length);

for (int i = 0; i < array.length; i++)

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

}

}

결과는 다음과 비슷할 것이다.

array.length = 2

array[0] = java.lang.Object@5d87b2

array[1] = java.lang.Object@77d134

al cleared -------------------

array.length = 0

ArrayList.toArray메쏘드는 Object형 배열을 반환(return) 한다. 위의 예에서도 알 수 있듯이 ArrayList가 비어있을 때는 배열 크기가 0인 객체를 반환한다.

Example.printArray메쏘드는 Object형 배열을 매개변수로 받는다. for문에서 i < array.length로 테스트함으로써 크기가 0인 배열도 잘 처리함을 알 수 있다.

내용이 비어있을 때는 크기가 0인 배열을 넘겨주는 것이 null을 넘기는 것보다 더 나은 방법이다.

* 1. 배열에서의 예외상황
     1. NullPointerException

다음 예를 실행해 보자.

public class Example {

static int[] intArray;

public static void main(String[] args) {

intArray[0] = 1;

}

}

NullPointerException이 나는 것을 알 수 있다.

배열 변수 intArray가 선언되기는 했지만 intArray에 배열 객체가 지정되지 않았기 때문이다. 자바에서는 배열도 객체다. 실제로 intArray가 무엇을 가리키고 있는지 알아보자.

public class Example {

static int[] intArray;

public static void main(String[] args) {

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

intArray[0] = 1;

}

}

결과는 다음과 비슷할 것이다.

intArray = null

java.lang.NullPointerException

intArray는 null을 가리키고 있다(아무것도 가리키지 않고 있다). 그렇기 때문에 NullPointerException이 난 것이다.

* + 1. ArrayIndexOutOfBoundsException과 배열의 크기

다음 예를 실행해 보자.

public class Example {

public static void main(String[] args) {

int[] intArray = new int[4];

intArray[4] = 3;

}

}

ArrayIndexOutOfBoundsException이 날 것이다. 이 배열의 크기는 4다. 그러므로 이 배열에서 사용할 수 있는 인덱스는 0, 1, 2, 3이렇게 네 개이다. 배열 인덱스로 이것과 다른 것이 쓰인다면 ArrayIndexOutOfBoundsException이 난다. 이 예외상황은 자바 언어의 안정성의 한 측면을 보여준다. 엉뚱한 메모리를 건드릴 것 같으면 그것을 실행하지 않고 예외상황을 발생시키는 것이다. 이것은 C와 같은 언어에서와 다르다. C에서는 이런 상황에서는 엉뚱한 메모리를 읽거나 써서 예측할 수 없는 상황이 발생하고 때로는 운영체제를 다운시키기도 한다. 배열에 접근하기 전에 배열 인덱스를 일일이 검사함으로써 자바는 실행속도를 약간 희생했지만 안정성을 향상시켰다.

배열의 크기는 배열 객체가 생성될 때 배열 객체에 저장된다. 정확히 말하면 배열객체의 어떤 필드에 저장되고 그 필드의 이름은 length이다. for문 등을 이용해서 배열을 다룰 때 배열 크기를 하드 코딩(hardcoding, 숫자 등를 직접쓰는 것, 한마디로 리터럴이 필요 이상으로 많은 코드)하지 말고 length필드를 이용하자. 그러면 배열 크기가 바뀌더라도 그 부분을 고칠 필요가 없을 것이다. 처음의 예제를 그런식으로 고쳐 보자.

public class Example {

public static void main(String[] args) {

double[] doubleArray;

doubleArray = new double[5];

doubleArray[0] = 0.5;

doubleArray[1] = 0;

doubleArray[2] = 1;

doubleArray[3] = 1.5;

doubleArray[4] = 2;

double sum = 0;

for (int i = 0; i < doubleArray.length; i++) { // 이 부분을 주목하라.

sum += doubleArray[i];

}

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

System.out.println("average = " + (sum / 5));

}

}

for문에서 5가 doubleArray.length로 바뀐 것을 볼 수 있다.

1. 배열의 해부
   1. 배열은 객체다
      1. 자바 배열은 객체다

자바에서 배열은 객체이다. 그 증거를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

int[] array = new int[4];

Object obj = array; // 배열 객체를 Object형 변수에 저장할 수 있다.

Cloneable c = array; // 배열 객체를 Cloneable형 변수에 저장할 수 있다.

Serializable s = array; // 배열 객체를 Serializable형 변수에 저장할 수 있다.

System.out.println("array.toString() = " + array.toString());

System.out.println("array.hashCode() = " + array.hashCode());

System.out.println("array.getClass() = " + array.getClass());

}

}

결과는 다음과 비슷할 것이다.

array.toString() = [I@310d42

array.hashCode() = 3214658

array.getClass() = class [I

배열 객체(에 대한 참조)를 Object형 변수 obj에 대입할 수 있음을 알 수 있다. 또한 Object에 정의되어 있는 메쏘드인 toString, hashCode, getClass등을 사용할 수 있음을 알 수 있다.

배열은 Cloneable과 java.io.Serializable인터페이스를 구현하기 때문에 변수 c와 s에 배열 객체를 대입할 수 있었다. 이 말은 배열을 복제할 수 있고 Serialization이 가능하다는 말이다.

배열 객체에는 length라는 public final필드가 있어서 그 배열의 크기를 알 수 있다. 이 필드는 배열 객체가 생성될 때 설정된다.

배열 객체를 생성하기 위해 new라는 키워드를 사용해 왔는데 이것도 다른 클래스의 객체를 생성하는 것과 비슷하다. 배열은 다른 객체와 마찬가지로 동적으로 힙(heap)에 생성된다. 그리고 마찬가지로 쓰레기 수거기 (garbage collector)에 의해 수거된다.

배열의 바로 위 수퍼클래스(direct super class)는 Object클래스이다. 이 말은 Object가 배열의 유일한 수퍼 클래스라는 뜻이다.

* + 1. 배열의 멤버

배열은 객체이기 때문에 멤버를 가진다. 배열의 멤버에는 다음과 같은 것들이 있다.

* public final 필드인 length : 당연히 0또는 양의 정수이다.
* clone() 메쏘드 : Object클래스의 clone()메쏘드를 재정의(overriding) 했다.
* clone()을 제외한 다른 Object클래스의 메쏘드 : Object클래스의 메쏘드를 그대로 상속해서 쓴다.
  + 1. 배열 클래스의 이름

다른 객체가 항상 어떤 클래스의 객체이듯이 배열 객체도 항상 어떤 클래스의 객체이다. 배열 객체의 클래스를 배열 클래스라고 한다. 그 클래스도 이름을 가지고 있다. 다음 예를 보자. 이름을 알아내기 위해 리플렉션(reflection) 메커니즘을 이용했다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object object = new Object();

byte[] byteArray = new byte[19];

short[] shortArray = new short[19];

int[] intArray = new int[19];

int[][] intintArray = new int[19][20];

int[][][] intintintArray = new int[19][20][3];

long[] longArray = new long[19];

char[] charArray = new char[19];

boolean[] booleanArray = new boolean[19];

float[] floatArray = new float[19];

double[] doubleArray = new double[19];

Object[] objectArray = new Object[19];

Object[][] objectobjectArray = new Object[19][20];

ArrayList[] arrayListArray = new ArrayList[19];

Example[] exampleArray = new Example[10];

Example.InnerClass[] exampleInnerClassArray = new Example.InnerClass[10];

System.out.println("Class Name of object : " + object.getClass().getName());

System.out.println("Class Name of byteArray : " + byteArray.getClass().getName());

System.out.println("Class Name of shortArray : " + shortArray.getClass().getName());

System.out.println("Class Name of intArray : " + intArray.getClass().getName());

System.out.println("Class Name of intintArray : " + intintArray.getClass().getName());

System.out.println("Class Name of intintintArray : " + intintintArray.getClass().getName());

System.out.println("Class Name of longArray : " + longArray.getClass().getName());

System.out.println("Class Name of charArray : " + charArray.getClass().getName());

System.out.println("Class Name of booleanArray : " + booleanArray.getClass().getName());

System.out.println("Class Name of floatArray : " + floatArray.getClass().getName());

System.out.println("Class Name of doubleArray : " + doubleArray.getClass().getName());

System.out.println("Class Name of objectArray : " + objectArray.getClass().getName());

System.out.println("Class Name of objectobjectArray : " + objectobjectArray.getClass().getName());

System.out.println("Class Name of arrayListArray : " + arrayListArray.getClass().getName());

System.out.println("Class Name of exampleArray : " + exampleArray.getClass().getName());

System.out.println("Class Name of exampleInnerClassArray : " + exampleInnerClassArray.getClass().getName());

}

class InnerClass {

}

}

결과는 다음과 같다.

Class Name of object : java.lang.Object

Class Name of byteArray : [B

Class Name of shortArray : [S

Class Name of intArray : [I

Class Name of intintArray : [[I

Class Name of intintintArray : [[[I

Class Name of longArray : [J

Class Name of charArray : [C

Class Name of booleanArray : [Z

Class Name of floatArray : [F

Class Name of doubleArray : [D

Class Name of objectArray : [Ljava.lang.Object;

Class Name of objectobjectArray : [[Ljava.lang.Object;

Class Name of arrayListArray : [Ljava.util.ArrayList;

Class Name of exampleArray : [LExample;

Class Name of exampleInnerClassArray : [LExample$InnerClass;

배열 클래스의 이름은 [로 시작하고 차원이 높아질때마다 [가 하나씩 더붙는 것을 알 수 있다. 기본형의 배열일 때는 형에 따라 알파벳 문자가 하나 붙는다. 참조형의 배열일 때는 L다음에 그 참조형의 전체 이름(full name)이 붙는다. 단 중첩 타입일 때는 .대신에 $ 표시가 쓰인다.

* 1. 배열의 구조
     1. 배열 원소(element of the array)와 배열 요소 (component of the array)

다음 배열을 보자.

int[][][] intArray = new int[5][5][5];

intArray는 int형의 3차원 배열이다(엄밀히 말하면 배열의 배열의 배열).

배열은 배열 요소가 배열 크기(length 필드)만큼 모인 것을 말한다. 예를 들어 intArray[0]은 intArray(가 가리키고 있는) 배열의 배열 요소이다. 그리고 그 타입을 배열 요소 타입(component type)이라고 한다. intArray[0]의 타입은 int[][]형이다.

한단계 더 나아가서 intArray[0][0]은 intArray[0]의 요소이다. 마찬가지로 intArray[0][0][0]은 intArray[0][0]의 요소이다. 이때 intArray[0][0][0]의 타입은 배열형이 아닌 int형이다. 이럴 때 배열 원소라는 말을 쓴다. 즉 intArray[0][0][0]은 배열 intArray의 배열 원소이다. 배열 원소의 타입은 보통 기본형이거나 참조형이며 배열형이 아니다.

그러나 배열 원소가 배열을 가리키는 경우가 있을 수는 있다. 다음 예를 보자.

Object[] objectArray = new Object[1];

objectArray[0] = new int[10];

여기서 objectArray[0]은 objectArray의 배열 원소이며 Object형이다. 배열도 객체이기 때문에 int형 배열은 objectArray[0]에 대입될 수 있다.

* + 1. 배열의 배열

엄밀하게 따지면 자바의 배열은 다차원 배열이 아니라 배열의 배열이다. 이것의 의미가 무엇인지 알아보자.

public class Example {

public static void main(String[] args) {

int[][] intIntArray = {

{1, 2, 3, 4},

{5, 6, 7, 8},

{9, 10, 11, 11},

};

int[] intArray;

intArray = intIntArray[1];

for (int i = 0; i < intArray.length; i++) {

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

}

}

}

결과는 다음과 같다.

intArray[0] = 5

intArray[1] = 6

intArray[2] = 7

intArray[3] = 8

다음 코드에 주목하자.

intArray = intIntArray[1];

intIntArray[1]자체가 또한 배열을 가리킴을 알 수 있다. intIntArray는 크기가 4인 int[]형 배열 3개를 요소를 가지고 있다. 그림 배열의 해부 – 1은 intIntArray의 구조를 나타내고 있다.

-------------------------- 그림 배열의 해부 – 1 -------------------------------

intIntArray변수는 int[]의 배열을 가리키고 있다. intIntArray[0]은 int의 배열을 가리키고 있다. 마찬가지로 intIntArray[1]과 intIntArray[2]도 int의 배열을 가리키고 있다. 실제로 존재하는 배열 객체는 하나가 아니라 네개가 존재하는 것이다. int[]의 배열이 하나이고 int의 배열이 세 개이다.

이제까지는 배열을 생성할 때 모든 차원의 크기를 정해주었다. 예를 들어 다음처럼.

new array[3][5][6][2];

그러나 그럴 필요가 없다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int[][] arrayOfArray = new int[3][];

for (int i = 0; i < arrayOfArray.length; i++)

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

try {

for (int i = 0; i < arrayOfArray.length; i++)

for (int j = 0; j < arrayOfArray[i].length; j++)

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

} catch (Exception e) {

System.out.println(e);

}

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

arrayOfArray[0] = new int[3];

arrayOfArray[1] = new int[3];

arrayOfArray[2] = new int[3];

for (int i = 0; i < arrayOfArray.length; i++)

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

for (int i = 0; i < arrayOfArray.length; i++)

for (int j = 0; j < arrayOfArray[i].length; j++)

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

}

}

결과는 다음과 비슷할 것이다.

arrayOfArray[0] = null

arrayOfArray[1] = null

arrayOfArray[2] = null

java.lang.NullPointerException

---------------------------------

arrayOfArray[0] = [I@310d42

arrayOfArray[1] = [I@5d87b2

arrayOfArray[2] = [I@77d134

arrayOfArray[0][0] = 0

arrayOfArray[0][1] = 0

arrayOfArray[0][2] = 0

arrayOfArray[1][0] = 0

arrayOfArray[1][1] = 0

arrayOfArray[1][2] = 0

arrayOfArray[2][0] = 0

arrayOfArray[2][1] = 0

arrayOfArray[2][2] = 0

먼저 첫번째 인덱스가 들어가는 부분의 크기만 정해진 배열 객체가 만들어진다.

int[][] arrayOfArray = new int[3][];

이 때에는 arrayOfArray[i]는 null을 가리킨다. 즉 하위 배열(sub array)가 만들어지지 않은 상태이다. 그렇기 때문에 arrayOfArray[i].length를 실행할 때 NullPointerException이 나는 것이다.

다음의 코드가 하위 배열을 만든다.

arrayOfArray[0] = new int[3];

다음 코드를 보자.

int[][] twoDimArray = new int[3][3];

위의 코드로 만들어지는 배열은 하나가 아니다. 위의 코드와 같은 일을 하는 것은 다음의 코드이다.

int[][] twoDimArray = new int[3][0];

twoDimArray[0] = new int[3];

twoDimArray[1] = new int[3];

twoDimArray[2] = new int[3];

즉 하나의 int[][]형 배열과 세개의 int[]형 하위 배열이 만들어지는 것이다. 다음과 같은 배열을 생성한다면

int[][][] threeDimArray = new int[3][4][5];

int[][][]형 배열과 세개의 int[][]형 하위 배열과 열 두개의 int[]형 하위 배열이 만들어지는 것이다. 동등한 것을 하는 코드는 다음과 같다.

int[][][] threeDimArray = new int[3][][];

threeDimArray[0] = new int[4][5];

threeDimArray[1] = new int[4][5];

threeDimArray[2] = new int[4][5];

또는 다음과 같다.

int[][][] threeDimArray = new int[3][4][];

threeDimArray[0][0] = new int[5];

threeDimArray[0][1] = new int[5];

threeDimArray[0][2] = new int[5];

threeDimArray[0][3] = new int[5];

threeDimArray[1][0] = new int[5];

threeDimArray[1][1] = new int[5];

threeDimArray[1][2] = new int[5];

threeDimArray[1][3] = new int[5];

threeDimArray[2][0] = new int[5];

threeDimArray[2][1] = new int[5];

threeDimArray[2][2] = new int[5];

threeDimArray[2][3] = new int[5];

또는 완전히 풀어서(?) 본다면 다음과 같다.

int[][][] threeDimArray = new int[3][][];

threeDimArray[0] = new int[4][];

threeDimArray[1] = new int[4][];

threeDimArray[2] = new int[4][];

threeDimArray[0][0] = new int[5];

threeDimArray[0][1] = new int[5];

threeDimArray[0][2] = new int[5];

threeDimArray[0][3] = new int[5];

threeDimArray[1][0] = new int[5];

threeDimArray[1][1] = new int[5];

threeDimArray[1][2] = new int[5];

threeDimArray[1][3] = new int[5];

threeDimArray[2][0] = new int[5];

threeDimArray[2][1] = new int[5];

threeDimArray[2][2] = new int[5];

threeDimArray[2][3] = new int[5];

배열 크기는 다음과 같이 여러가지 방법으로 지정할 수 있다.

new int[2][3][4][5];

new int[2][3][4][];

new int[2][3][][];

new int[2][][][];

앞에서부터 임의의 갯수를 채울 수 있다.

new int[][][][]; // Error

위에서처럼 아무것도 지정해주지 않으면 안된다. 왜냐하면 최소한 최상위 배열의 크기는 알아야 하기 때문이다.

마찬가지로 다음과 같이 배열을 생성할 수는 없다. 배열 객체가 어떻게 만들어지는지를 생각해보면 왜 이런식으로 만들 수 없는지 알수 있을 것이다.

new int[][3][4][5];

new int[][][4][5];

new int[][][][5];

new int[2][][][5];

new int[2][][4][];

new int[][][4][];

* + 1. 들쭉날쭉한 배열

지금까지는 하위 배열의 크기가 일정한 배열만 만들어왔다. 자바의 배열은 배열의 배열이기 때문에 하위 배열의 크기는 다를 수 있다. 심지어 배열 크기가 0인 배열일 수도 있고 null을 가리킬 수도 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int[][] zigzagArray = new int[5][];

zigzagArray[0] = null;

for (int i = 1; i < zigzagArray.length; i++) {

zigzagArray[i] = new int[i - 1];

for (int j = 0; j < zigzagArray[i].length; j++)

zigzagArray[i][j] = (j + 1) \* i;

}

for (int i = 0; i < zigzagArray.length; i++) {

if (zigzagArray[i] == null) {

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

continue;

}

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

for (int j = 0; j < zigzagArray[i].length; j++)

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

}

}

}

결과는 다음과 같다.

zigzagArray[0] = null

zigzagArray[1].length = 0

zigzagArray[2].length = 1

zigzagArray[2][0] = 2

zigzagArray[3].length = 2

zigzagArray[3][0] = 3

zigzagArray[3][1] = 6

zigzagArray[4].length = 3

zigzagArray[4][0] = 4

zigzagArray[4][1] = 8

zigzagArray[4][2] = 12

그림 배열의 해부 – 2는 위 프로그램에서 만들어진 배열의 구조를 보여주고 있다.

------------------------------------ 그림 배열의 해부 – 2 ---------------------------------------

* + 1. 배열의 배열과 다차원 배열의 장단점

배열의 배열은 다차원 배열보다 유연하다. 이것은 들쭉날쭉한 배열을 보면 알 수 있다.

다차원 배열은 배열의 배열보다 빠르다. 다음 예를 보자.

int[][][][] array = new int[10][10][10][10];

array[4][3][2][1] = 10;

자바의 배열은 배열의 배열이기 때문에 array[4][3][2][1]에 10을 대입하기 위해서는 다음과 같은 단계를 거친다.

배열 array에서 배열 인덱스 4를 이용해 배열 요소 array[4]에 접근한다. array[4] 역시 배열 객체이다

배열 array[4]에서 배열 인덱스 3을 이용해 배열 요소 array[4][3]에 접근한다. array[4][3] 역시 배열 객체이다

배열 array[4][3]에서 배열 인덱스 2를 이용해 배열 요소 array[4][3][2]에 접근한다. array[4][3][2] 역시 배열 객체이다

배열 array[4][3][2]에서 배열 인덱스 1을 이용해 배열 요소(이것은 배열 원소이기도 하다) array[4][3][2][1]에 접근한다(여기에 10을 대입한다).

다차원 배열에서는 배열 인덱스 4, 3, 2, 1을 이용해 한번에 배열 요소 array[4][3][2][1]의 (메소리 상의) 위치를 계산해 낼 수 있다. 예를 들어 int가 4바이트라면 포인터 array가 가리키는 위치에서

(4 \* 10 \* 10 \* 10 + 3 \* 10 \* 10 + 2 \* 10 + 1) \* 4

바이트만큼 더 가면 된다(맞나? 하여튼 대충 이런식으로 계산된다).

이것은 C와 같은 언어에서 사용하는 방법이다.

* + 1. 배열의 복제

배열은 Cloneable인터페이스를 구현한다. 즉 복제가 가능하다는 말이다. 다음 예를 보자.

복제에 대해 자세한 것은 Object 클래스를 다룬 장의 clone 메쏘드를 다룬 부분을 참조하라.

배열의 복제를 해보자. 다음 예를 실행해보자.

public class Example {

public static void main(String[] args) {

int[] ai = {1, 3, 5, 6, 2};

int[] aiCloned = (int[])ai.clone();

for (int i = 0; i < aiCloned.length; i++)

System.out.println("ai[" + i + "] = " + ai[i] + ", aiCloned[" + i + "] = " + aiCloned[i]);

ai[0] = 10;

aiCloned[1] = 20;

System.out.println();

for (int i = 0; i < aiCloned.length; i++)

System.out.println("ai[" + i + "] = " + ai[i] + ", aiCloned[" + i + "] = " + aiCloned[i]);

}

}

다음과 같은 결과가 나올 것이다.

ai[0] = 1, aiCloned[0] = 1

ai[1] = 3, aiCloned[1] = 3

ai[2] = 5, aiCloned[2] = 5

ai[3] = 6, aiCloned[3] = 6

ai[4] = 2, aiCloned[4] = 2

ai[0] = 10, aiCloned[0] = 1

ai[1] = 3, aiCloned[1] = 20

ai[2] = 5, aiCloned[2] = 5

ai[3] = 6, aiCloned[3] = 6

ai[4] = 2, aiCloned[4] = 2

제대로 복제된 것을 알 수 있다. 두개의 배열 객체가 독립적이라는 것을 알 수 있다. 다시말해 ai[0]에 10을 대입해도 aiCloned[0]은 영향받지 않는다.

그림 배열의 해부 – 3은 배열이 복제된 직후의 상태를 보여준다.

------------------------------- 그림 배열의 해부 – 3 --------------------------

그림 배열의 해부 – 4는 10과 20이 각각 대입된 후의 상태를 보여준다.

------------------------------- 그림 배열의 해부 – 4 --------------------------

이제 2차원배열(배열의 배열)을 복제해 보자. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int[][] ai = { {2, 3}, {5, 6} };

int[][] aiCloned = (int[][])ai.clone();

for (int i = 0; i < ai.length; i++)

for (int j = 0; j < ai[i].length; j++)

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

for (int i = 0; i < aiCloned.length; i++)

for (int j = 0; j < aiCloned[i].length; j++)

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

System.out.println();

ai[0][0] = 10;

int[] array = {123, 312};

ai[1] = array;

for (int i = 0; i < ai.length; i++)

for (int j = 0; j < ai[i].length; j++)

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

for (int i = 0; i < aiCloned.length; i++)

for (int j = 0; j < aiCloned[i].length; j++)

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

}

}

다음과 같은 결과가 나올 것이다.

ai[0][0] = 2

ai[0][1] = 3

ai[1][0] = 5

ai[1][1] = 6

aiCloned[0][0] = 2

aiCloned[0][1] = 3

aiCloned[1][0] = 5

aiCloned[1][1] = 6

ai[0][0] = 10

ai[0][1] = 3

ai[1][0] = 123

ai[1][1] = 312

aiCloned[0][0] = 10

aiCloned[0][1] = 3

aiCloned[1][0] = 5

aiCloned[1][1] = 6

얕은 복제(shallow copy)가 된 것을 알 수 있다. ai[0][0]에 10을 대입하자 aiCloned[0][0]도 영향을 받았다. 그러나 ai[1]에 array를 대입한것은 aiCloned에 영향을 끼치지 않았다. 그림 X는 복제된 후의 배열과 ai[0][0]와 ai[1]를 변화시킨후의 배열이다. clone 메쏘드를 이용한 복제는 최상위 배열만 복제하고 하위 배열은 공유함을 알 수 있다.

그림 배열의 해부 – 5는 복제가 일어난 직후의 상태를 보여준다.

----------------------------------- 그림 배열의 해부 – 5 ------------------------------

그림 배열의 해부 – 6은 10과 {123, 312}가 각각 대입된 이후의 상태를 보여준다.

----------------------------------- 그림 배열의 해부 – 6 ------------------------------

다음 예제는 그것을 확인해 준다.

public class Example {

public static void main(String[] args) {

int[][][] ai = new int[3][3][3];

int[][][] aiCloned = (int[][][])ai.clone();

System.out.println("(ai==aiCloned) : " + (ai == aiCloned));

for (int i = 0; i < ai.length; i++)

System.out.println("(ai[" + i + "]==aiCloned[" + i + "]) : " + (ai[i] == aiCloned[i]));

}

}

결과는 다음과 같다.

(ai==aiCloned) : false

(ai[0]==aiCloned[0]) : true

(ai[1]==aiCloned[1]) : true

(ai[2]==aiCloned[2]) : true

(ai[i] == aiCloned[i])에서 ==는 두 변수가 하나의 객체를 참조하는지(identity가 같은지)를 알아보는 것이다.

이번에는 참조형 배열을 복제해 보자. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

StringBuffer[] asb = new StringBuffer[3];

asb[0] = new StringBuffer("zero");

asb[1] = new StringBuffer("one");

asb[2] = new StringBuffer("two");

StringBuffer[] asbCloned = (StringBuffer[])asb.clone();

asb[2].append(":2");

for (int i = 0; i < asb.length; i++)

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

for (int i = 0; i < asbCloned.length; i++)

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

}

}

결과는 다음과 같다.

asb[0] = zero

asb[1] = one

asb[2] = two:2

asbCloned[0] = zero

asbCloned[1] = one

asbCloned[2] = two:2

asb를 바꾸자 asbCloned도 바뀌었다. 그림 배열의 해부 – 7은 복제가 일어난 직후의 상태를 보여준다.

-------------------------- 그림 배열의 해부 – 7 -------------------------------

그림 배열의 해부 – 8은 “:2”가 덧붙여진 다음의 상태를 보여준다.

-------------------------- 그림 배열의 해부 – 8 -------------------------------

정리하면 기본형의 1차원 배열을 제외하면 기본형의 다차원 배열과 참조형의 배열은 얕은 복제(shallow copy)가 되어서 문제가 생기는 것을 알 수 있다.

다차원 배열 또는 참조형의 1차원 배열을 깊이 복제(deep copy)하기 위해서는 다음과 같이 해야 한다. 다음 예제를 보자. 이 예제는 int[][][]을 복제한다.

public class Example {

public static void main(String[] args) {

int[][][] array = {

{ { 1, 2, 3}, { 4, 5, 6}, { 7, 8, 9}},

{ {11, 12, 13}, {14, 15, 16}, {17, 18, 19}},

{ {21, 22, 23}, {24, 25, 26}, {27, 28, 29}},

};

int[][][] arrayCloned = deepCopy(array); // 복사

// 화면에 표시

for (int i = 0; i < array.length; i++)

for (int j = 0; j < array[i].length; j++)

for (int k = 0; k < array[i][j].length; k++) {

System.out.print("array["+i+"]["+j+"]["+k+"] = " + array[i][j][k]);

System.out.println(" arrayCloned["+i+"]["+j+"]["+k+"] = " + arrayCloned[i][j][k]);

}

// array를 모두 0으로.

for (int i = 0; i < array.length; i++)

for (int j = 0; j < array[i].length; j++)

for (int k = 0; k < array[i][j].length; k++)

array[i][j][k] = 0;

System.out.println();

// 화면에 표시.

for (int i = 0; i < array.length; i++)

for (int j = 0; j < array[i].length; j++)

for (int k = 0; k < array[i][j].length; k++) {

System.out.print("array["+i+"]["+j+"]["+k+"] = " + array[i][j][k]);

System.out.println(" arrayCloned["+i+"]["+j+"]["+k+"] = " + arrayCloned[i][j][k]);

}

}

public static int[][][] deepCopy(int[][][] a) {

int[][][] aCloned;

if (a == null) // a가 null이면 null을 리턴.

return null;

aCloned = (int[][][])a.clone(); // 일단 최상위 배열을 복제한다.

for (int i = 0; i < aCloned.length; i++) {

if (aCloned[i] == null) // null은 그대로 둔다.

continue;

aCloned[i] = (int[][])a[i].clone(); // 하위 배열을 복제한다.

for (int j = 0; j < aCloned[i].length; j++) {

if (aCloned[i][j] == null) // null은 그대로 둔다.

continue;

aCloned[i][j] = (int[])a[i][j].clone(); // 하위 배열을 복제한다.

}

}

return aCloned;

}

}

결과는 다음과 같다.

array[0][0][0] = 1 arrayCloned[0][0][0] = 1

array[0][0][1] = 2 arrayCloned[0][0][1] = 2

array[0][0][2] = 3 arrayCloned[0][0][2] = 3

array[0][1][0] = 4 arrayCloned[0][1][0] = 4

array[0][1][1] = 5 arrayCloned[0][1][1] = 5

array[0][1][2] = 6 arrayCloned[0][1][2] = 6

array[0][2][0] = 7 arrayCloned[0][2][0] = 7

array[0][2][1] = 8 arrayCloned[0][2][1] = 8

array[0][2][2] = 9 arrayCloned[0][2][2] = 9

array[1][0][0] = 11 arrayCloned[1][0][0] = 11

array[1][0][1] = 12 arrayCloned[1][0][1] = 12

array[1][0][2] = 13 arrayCloned[1][0][2] = 13

array[1][1][0] = 14 arrayCloned[1][1][0] = 14

array[1][1][1] = 15 arrayCloned[1][1][1] = 15

array[1][1][2] = 16 arrayCloned[1][1][2] = 16

array[1][2][0] = 17 arrayCloned[1][2][0] = 17

array[1][2][1] = 18 arrayCloned[1][2][1] = 18

array[1][2][2] = 19 arrayCloned[1][2][2] = 19

array[2][0][0] = 21 arrayCloned[2][0][0] = 21

array[2][0][1] = 22 arrayCloned[2][0][1] = 22

array[2][0][2] = 23 arrayCloned[2][0][2] = 23

array[2][1][0] = 24 arrayCloned[2][1][0] = 24

array[2][1][1] = 25 arrayCloned[2][1][1] = 25

array[2][1][2] = 26 arrayCloned[2][1][2] = 26

array[2][2][0] = 27 arrayCloned[2][2][0] = 27

array[2][2][1] = 28 arrayCloned[2][2][1] = 28

array[2][2][2] = 29 arrayCloned[2][2][2] = 29

array[0][0][0] = 0 arrayCloned[0][0][0] = 1

array[0][0][1] = 0 arrayCloned[0][0][1] = 2

array[0][0][2] = 0 arrayCloned[0][0][2] = 3

array[0][1][0] = 0 arrayCloned[0][1][0] = 4

array[0][1][1] = 0 arrayCloned[0][1][1] = 5

array[0][1][2] = 0 arrayCloned[0][1][2] = 6

array[0][2][0] = 0 arrayCloned[0][2][0] = 7

array[0][2][1] = 0 arrayCloned[0][2][1] = 8

array[0][2][2] = 0 arrayCloned[0][2][2] = 9

array[1][0][0] = 0 arrayCloned[1][0][0] = 11

array[1][0][1] = 0 arrayCloned[1][0][1] = 12

array[1][0][2] = 0 arrayCloned[1][0][2] = 13

array[1][1][0] = 0 arrayCloned[1][1][0] = 14

array[1][1][1] = 0 arrayCloned[1][1][1] = 15

array[1][1][2] = 0 arrayCloned[1][1][2] = 16

array[1][2][0] = 0 arrayCloned[1][2][0] = 17

array[1][2][1] = 0 arrayCloned[1][2][1] = 18

array[1][2][2] = 0 arrayCloned[1][2][2] = 19

array[2][0][0] = 0 arrayCloned[2][0][0] = 21

array[2][0][1] = 0 arrayCloned[2][0][1] = 22

array[2][0][2] = 0 arrayCloned[2][0][2] = 23

array[2][1][0] = 0 arrayCloned[2][1][0] = 24

array[2][1][1] = 0 arrayCloned[2][1][1] = 25

array[2][1][2] = 0 arrayCloned[2][1][2] = 26

array[2][2][0] = 0 arrayCloned[2][2][0] = 27

array[2][2][1] = 0 arrayCloned[2][2][1] = 28

array[2][2][2] = 0 arrayCloned[2][2][2] = 29

array를 모두 0으로 바꾸어도 arrayCloned는 변함이 없다.

* 1. 배열의 형변환(Casting)
     1. 기본형 배열의 형변환

예를 들어 int[]형 배열 intArray를 만들었다면 intArray[0]는 int형과 같이 취급된다. 그러므로

int[] intArray = new int[1];

byte byteValue = 1;

short shortValue = 1;

intArray[0] = byteValue;

intArray[0] = shortValue;

는 아무런 문제가 없으며

long longValue = 1;

intArray[0] = longValue;

은 컴파일타임 오류가 날 것이다. 이것을 해결하기 위해서는

intArray[0] = (int)longValue;

와 같이 해 주어야 한다. 특별할 것이 없는 당연한 것이다.

여기서 더 나아가 다음과 같은 것이 가능할 것이라고 생각하는 사람이 있을 것이다.

int[] intArray = new int[1];

long[] longArray = intArray; // 에러

그러나 이것은 금지되어 있다.

long[] longArray = (long[])intArray;

이렇게 명시적으로(long[]) 형변환을 한다고 해도 사정이 달라지지 않는다. 심지어 intArray가 null일 때도 안된다.

int[] intArray = null;

long[] longArray = intArray; // 에러

long[] longArray = (long[])intArray; // 에러

기본형 배열은 같은 타입에만 대입할 수 있다. 왜 금지 시켰을까? 만약 그것이 금지되지 않았다고 가정하고 다음 코드를 보자.

byte[] byteArray = new byte[1];

int[] intArray = byteArray; // 안되지만 된다고 가정하고...

intArray[0] = 1000000; // 이런일이 발생한다.

세번째 줄은 그 자체로 아무런 문제가 없다. int형 배열의 배열요소에 int형 값을 대입하는 것이다. 그러나 그 배열은 원래 byte형 배열이었다. 결국 byte형에 int형을 대입하는 꼴이 되었다. 참조형 배열에도 비슷한 문제가 발생하는데 여기서는 규칙이 좀 다르다.

* + 1. 참조형 배열의 형변환

기본형과 마찬가지로 참조형의 배열 요소는 그 참조형과 같이 취급된다. 즉 Object[]형의 열 objectArray를 만들었다면 objectArray[0]은 Object형과 같이 취급된다. 그러므로

Object[] objectArray = new Object[4];

objectArray[0] = new Object();

objectArray[1] = new ArrayList();

objectArray[2] = new int[4][5];

objectArray[3] = objectArray;

는 모두 아무런 문제가 없다.

기본형의 배열과 달리 참조형의 배열은 배열 자체로 형변환이 가능하다. 예를 들어

ArrayList[] arrayListArray = new ArrayList[10];

Object[] objectArray = arrayListAray;

ArrayList[] anotherArrayListArray = (ArrayList[])objectArray;

와 같이 쓸 수 있다.

* + 1. ArrayStoreException

참조형 배열의 형변환이 가능하기 때문에 문제가 발생할 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

SubClass[] subArray = new SubClass[2];

SuperClass[] superArray = subArray;

superArray[0] = new SuperClass();

}

}

class SuperClass {}

class SubClass extends SuperClass {}

컴파일은 무사히 잘 된다. 컴파일러의 입장에서 볼 때는superArray에 subArray를 대입했으므로 문제가 없다. 그리고 superArray[0]의 컴파일타임 형(compile-time type)은 SuperClass형이므로 SuperClass의 객체를 저장하는 것이 이상할 것이 없다.

그러나 ArrayStoreException이 나는 것을 알 수 있다. 왜냐 하면 superArray가 가리키고 있는 배열 객체는 SubClass형의 배열이며 superArray[0]은 사실상 SubClass형이기 때문이다. SubClass형 변수에 SuperClass형 객체를 넣으려고 하니까 에러가 난 것이다.

* 1. 배열에 관련된 클래스들
     1. System 클래스의 배열 복사

System클래스에는 다음과 같은 static 메쏘드가 있다.

public static void arraycopy(Object src, int src\_position, Object dst, int dst\_position, int length)

이 메쏘드는 dst배열에 있는 것을 scr배열에 복사한다. 매개변수가 배열형이 아니라 Object형인 이유는 기본형의 배열과 참조형의 배열 모두를 다룰 수 있게 하기 위해서이다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int[] arrayDestination = {1, 2, 3, 4, 5, 6, 7, 8, 9};

int[] arraySource = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110};

System.arraycopy(arraySource, 2, arrayDestination, 4, 3);

for (int i = 0; i < arraySource.length; i++)

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

for (int i = 0; i < arrayDestination.length; i++)

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

}

}

결과는 다음과 같다.

arraySource[0] = 10

arraySource[1] = 20

arraySource[2] = 30

arraySource[3] = 40

arraySource[4] = 50

arraySource[5] = 60

arraySource[6] = 70

arraySource[7] = 80

arraySource[8] = 90

arraySource[9] = 100

arraySource[10] = 110

arrayDestination[0] = 1

arrayDestination[1] = 2

arrayDestination[2] = 3

arrayDestination[3] = 4

arrayDestination[4] = 30

arrayDestination[5] = 40

arrayDestination[6] = 50

arrayDestination[7] = 8

arrayDestination[8] = 9

하나의 배열을 사용할 수도 있다. 이 때는 배열의 일부가 지정된 곳에 복사된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9};

System.arraycopy(array, 2, array, 4, 3);

for (int i = 0; i < array.length; i++)

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

}

}

결과는 다음과 같다.

array[0] = 1

array[1] = 2

array[2] = 3

array[3] = 4

array[4] = 3

array[5] = 4

array[6] = 5

array[7] = 8

array[8] = 9

참조형 배열을 복사해 보자.

public class Example {

public static void main(String[] args) {

StringBuffer[] arraySource = new StringBuffer[3];

StringBuffer[] arrayDestination = new StringBuffer[3];

arraySource[0] = new StringBuffer("zero");

arraySource[1] = new StringBuffer("one");

arraySource[2] = new StringBuffer("two");

System.arraycopy(arraySource, 0, arrayDestination, 0, 3); // 복사

for (int i = 0; i < arraySource.length; i++)

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

for (int i = 0; i < arrayDestination.length; i++)

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

arraySource[0].append("(nill)"); //arraySource에 변화를 준다.

System.out.println();

for (int i = 0; i < arraySource.length; i++)

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

for (int i = 0; i < arrayDestination.length; i++)

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

}

}

결과는 다음과 같다.

arraySource[0] = zero

arraySource[1] = one

arraySource[2] = two

arrayDestination[0] = zero

arrayDestination[1] = one

arrayDestination[2] = two

arraySource[0] = zero(nill)

arraySource[1] = one

arraySource[2] = two

arrayDestination[0] = zero(nill)

arrayDestination[1] = one

arrayDestination[2] = two

clone()을 이용해 복제하는 것과 비슷한 문제가 생기는 것을 알 수 있다. 여기서도 얕은 복제(shallow copy)가 이루어지는 것이다. 그림 배열의 해부 – 9는 복제한 직후의 상태를 보여주고 있고 그림 배열의 해부 – 10은 “(nill)”을 덧붙힌 후의 두 배열의 상태를 보여준다.

---------------------------------- 그림 배열의 해부 – 9 ------------------------------------------

---------------------------------- 그림 배열의 해부 – 10 ------------------------------------------

* + 1. Arrays 클래스

이 클래스는 찾기(search)와 정렬(sort)등을 할 수 있는 여러개의 메쏘드들이 있다. 하나씩 살펴 보기로 하자.

public static List asList(Object[] a)

이 메쏘드는 객체(Object)의 배열을 List로 바꾸어준다. List에 대해서는 컬렉션 클래스(collection classes)를 다룬 책을 참조하자. 이와 반대되는 일을 하는 메쏘드는 Collection.toArray()와 Collection.toArray(Object[])가 있다. 이것도 컬렉션 클래스를 다룬 책을 참조하라.

public static boolean equals(int[] a, int[] a2)

이 메쏘드는 두 배열이 같으면 true를 리턴한다. 두 배열이 같다는 것은 둘다 null이거나 두 배열의 크기가 같고 각 위치의 값이 모두 같을 때를 말한다. 여기에서 int형만 보여주었지만 모든 기본형의 배열과 참조형의 배열도 쓸 수 있다. 참조형에서는 같은지를 알아내기 위해 equals(Object) 메쏘드를 사용한다. 서로 다른 객체의 배열일때도 같을 수가 있다. 약간 억지가 있는 예제이지만 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object[] arrayString = new Object[5];

Object[] arrayInteger = new Object[5];

arrayString[0] = new StringClass("");

arrayString[1] = new StringClass("12");

arrayString[2] = new StringClass("1234");

arrayString[3] = new StringClass("123456");

arrayString[4] = new StringClass("12345678");

arrayInteger[0] = new IntegerClass(0);

arrayInteger[1] = new IntegerClass(2);

arrayInteger[2] = new IntegerClass(4);

arrayInteger[3] = new IntegerClass(6);

arrayInteger[4] = new IntegerClass(8);

boolean bEquals = Arrays.equals(arrayString, arrayInteger);

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

}

}

class StringClass {

String strValue;

StringClass(String str) {

strValue = str;

}

public boolean equals(Object objOther) {

if (objOther == null)

return false;

if (objOther instanceof StringClass) {

StringClass scOther = (StringClass)objOther;

if (scOther.strValue.equals(this.strValue))

return true;

else

return false;

}

if (objOther instanceof IntegerClass) {

IntegerClass icOther = (IntegerClass)objOther;

if (this.strValue.length() == icOther.intValue)

return true;

else

return false;

}

return false;

}

}

class IntegerClass {

int intValue;

IntegerClass(int i) {

intValue = i;

}

public boolean equals(Object objOther) {

if (objOther == null)

return false;

if (objOther instanceof StringClass) {

StringClass scOther = (StringClass)objOther;

if (scOther.strValue.length() == this.intValue)

return true;

else

return false;

}

if (objOther instanceof IntegerClass) {

IntegerClass icOther = (IntegerClass)objOther;

if (this.intValue == icOther.intValue)

return true;

else

return false;

}

return false;

}

}

결과는 다음과 같다.

bEquals = true

equals(Object) 메쏘드는 재정의(overriding)하기 나름이라는 것을 알 수 있을 것이다. 이 예제에서는 문자열(String)의 길이와 정수(int)의 크기가 같으면 같은 것으로 정의했다.

public static void fill(int[] a, int val)

public static void fill(int[] a, int fromIndex, int toIndex, int val)

위의 첫번째 메쏘드는 배열을 특정값으로 채운다. 두번째 메쏘드는 배열의 일부를 특정값으로 채운다. 이것도 모든 기본형의 배열과 참조형의 배열에 쓸 수 있다. 다음 예제는 참조형 배열을 사용한 것이다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

StringBuffer[] array = new StringBuffer[3];

Arrays.fill(array, new StringBuffer("fill")); // 복사

for (int i = 0; i < array.length; i++)

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

array[0].append("--changed"); //array[0] 에 변화를 준다.

System.out.println();

for (int i = 0; i < array.length; i++)

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

}

}

결과는 다음과 같다.

array[0] = fill

array[1] = fill

array[2] = fill

array[0] = fill--changed

array[1] = fill--changed

array[2] = fill--changed

그림 배열의 해부 – 11은 array[0]의 append메쏘드를 부르기 전의 상태를 배열의 해부 – 12는 후의 상태를 보여준다.

----------------------------------- 그림 배열의 해부 – 11 -----------------------------------

----------------------------------- 그림 배열의 해부 – 12 -----------------------------------

public static void sort(int[] a)

public static void sort(int[] a, int fromIndex, int toIndex)

이 메쏘드는 퀵정렬(quick sort)를 이용해서 정렬을 한다. 퀵정렬에 대해서는 자료구조론 책을 참조하라. 다른 기본형에 대해서도 마찬가지 메쏘드가 있다. 참조형에 대해서는 설명이 필요하다. 참조형에는 네개의 메쏘드가 있다.

public static void sort(Object[] a)

public static void sort(Object[] a, int fromIndex, int toIndex)

public static void sort(Object[] a, Comparator c)

public static void sort(Object[] a, int fromIndex, int toIndex, Comparator c)

기본형과 마찬가지로 범위를 정해줄 수도 안 정해줄 수도(이 때는 배열 전체) 있다. 그리고 Comparator를 취하는 메쏘드와 취하지 않는 메쏘드로 나뉜다. Comparator를 취하는 메쏘드는 Comparator.compare(Object, Object)메쏘드를 이용해서 어떤 것이 큰(?) 객체인지를 알아내고 Comparator를 취하지 않는 메쏘드는 Comparable.compareTo(Object) 메쏘드를 이용한다. Comparable과 Comparator인터페이스에 대해서는 컬렉션 클래스를 다룬 책을 참조하라.

public static int binarySearch(int[] a, int key)

이 메쏘드는 이진탐색(binary search)를 이용해서 key값을 찾아낸다. 이 메쏘드가 제대로 작동하기 위해서는 배열이 정렬되어 있어야 한다. 만약 배열이 정렬되지 않은 상태라면 sort()메쏘드를 이용해서 정렬하고 이 메쏘드를 사용하면 된다. 다른 기본형에 대해서도 마찬가지의 메쏘드가 있다. 참조형에는 두개의 메쏘드가 있는데 binarySearch 메쏘드와 마찬가지로 Comparable을 사용하는 것과 Comparator를 사용하는 것이 있다.

위의 equals(Object[], Object[])의 예에서와 마찬가지로 참조형에 해당하는 binarySearch와 sort 메쏘드도 반드시 배열 요소의 값이 같은 타입의 객체일 필요는 없다. sort메쏘드를 사용하는 다음 예를 보자. 위의 equals(Object[], Object[])의 예와 비슷하다. 즉 문자열의 길이와 정수의 크기를 비교하는 것이다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object[] array = new Object[10];

array[0] = new StringClass("s");

array[1] = new StringClass("str");

array[2] = new StringClass("strin");

array[3] = new StringClass("string7");

array[4] = new StringClass("string789");

array[5] = new IntegerClass(0);

array[6] = new IntegerClass(2);

array[7] = new IntegerClass(4);

array[8] = new IntegerClass(6);

array[9] = new IntegerClass(8);

for (int i = 0; i < array.length; i++)

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

Arrays.sort(array, new StringIntegerComparator());

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

for (int i = 0; i < array.length; i++)

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

}

}

class StringClass {

String strValue;

StringClass(String str) {

strValue = str;

}

public String toString() {

return strValue;

}

}

class IntegerClass {

int intValue;

IntegerClass(int i) {

intValue = i;

}

public String toString() {

return "" + intValue;

}

}

class StringIntegerComparator implements Comparator {

public int compare(Object objOne, Object objAnother) {

int one, another;

if ( (objOne==null) || (objAnother==null) )

throw new IllegalArgumentException();

if (objOne instanceof StringClass) {

one = ((StringClass)objOne).strValue.length();

} else if (objOne instanceof IntegerClass) {

one = ((IntegerClass)objOne).intValue;

} else {

throw new IllegalArgumentException();

}

if (objAnother instanceof StringClass) {

another = ((StringClass)objAnother).strValue.length();

} else if (objAnother instanceof IntegerClass) {

another = ((IntegerClass)objAnother).intValue;

} else {

throw new IllegalArgumentException();

}

return one - another;

}

}

결과는 다음과 같다.

array[0] = s

array[1] = str

array[2] = strin

array[3] = string7

array[4] = string789

array[5] = 0

array[6] = 2

array[7] = 4

array[8] = 6

array[9] = 8

----------------------------

array[0] = 0

array[1] = s

array[2] = 2

array[3] = str

array[4] = 4

array[5] = strin

array[6] = 6

array[7] = string7

array[8] = 8

array[9] = string789

우스꽝스럽기는 하지만 나름대로 정렬되어 있다. 이제 좀 그럴듯한 예를 만들어 보자. 포장 클래스(wrapper class)와 비슷한 클래스를 만든다. 하나는 정수(long)고 다른 하나는 실수(double)이다. 정수와 실수는 당연히 비교 가능하다. 표준 API에 존재하는 포장 클래스(java.lang.Long과 java.lang.Double)는 이런식으로 만들어지지 않았다. 궁금한 사람은 표준 API의 소스 코드를 참조하기 바란다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

AnotherNumber[] array = new AnotherNumber[10];

array[0] = new AnotherLong(1);

array[1] = new AnotherLong(2);

array[2] = new AnotherLong(3);

array[3] = new AnotherLong(4);

array[4] = new AnotherLong(5);

array[5] = new AnotherDouble(1.5);

array[6] = new AnotherDouble(2.5);

array[7] = new AnotherDouble(3.5);

array[8] = new AnotherDouble(4.5);

array[9] = new AnotherDouble(5.5);

for (int i = 0; i < array.length; i++)

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

Arrays.sort(array);

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

for (int i = 0; i < array.length; i++)

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

}

}

abstract class AnotherNumber implements Comparable {

public abstract double getDoubleValue();

public int compareTo(Object obj) {

if (obj == null)

throw new IllegalArgumentException();

if (obj instanceof AnotherNumber) {

if (this.getDoubleValue() > ((AnotherNumber)obj).getDoubleValue()) {

return 1;

} else if (this.getDoubleValue() < ((AnotherNumber)obj).getDoubleValue()){

return -1;

} else {

return 0;

}

} else {

throw new IllegalArgumentException();

}

}

}

class AnotherLong extends AnotherNumber {

private long longValue;

public AnotherLong(long value) {

longValue = value;

}

public double getDoubleValue() {

return longValue;

}

public String toString() {

return "" + longValue;

}

}

class AnotherDouble extends AnotherNumber{

private double doubleValue;

public AnotherDouble(double value) {

doubleValue = value;

}

public double getDoubleValue() {

return doubleValue;

}

public String toString() {

return "" + doubleValue;

}

}

결과는 다음과 같다.

array[0] = 1

array[1] = 2

array[2] = 3

array[3] = 4

array[4] = 5

array[5] = 1.5

array[6] = 2.5

array[7] = 3.5

array[8] = 4.5

array[9] = 5.5

----------------------------

array[0] = 1

array[1] = 1.5

array[2] = 2

array[3] = 2.5

array[4] = 3

array[5] = 3.5

array[6] = 4

array[7] = 4.5

array[8] = 5

array[9] = 5.5

잘 정렬되는 것을 볼 수 있다. 추상 클래스인 AnotherNumber 클래스가 무슨 역할을 하는지 잘 살펴보길 바란다.

물론 이 예도 약간의 문제가 생길 여지는 있다. long형의 유효숫자의 크기가 double형보다 크기 때문이다. 위의 예에서 main메쏘드만 다음과 같이 바꾸어서 실행해 보자.

public static void main(String[] args) {

AnotherNumber[] array = new AnotherNumber[6];

array[0] = new AnotherLong(Long.MAX\_VALUE );

array[1] = new AnotherLong(Long.MAX\_VALUE - 1);

array[2] = new AnotherLong(Long.MAX\_VALUE - 2);

array[3] = new AnotherDouble(Long.MAX\_VALUE - 4);

array[4] = new AnotherDouble(Long.MAX\_VALUE - 5);

array[5] = new AnotherDouble(Long.MAX\_VALUE - 6);

for (int i = 0; i < array.length; i++)

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

Arrays.sort(array);

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

for (int i = 0; i < array.length; i++)

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

}

(Arrays.sort메쏘드의 알고리즘이 바뀌지 않았다면) 결과는 다음과 같을 것이다.

array[0] = 9223372036854775807

array[1] = 9223372036854775806

array[2] = 9223372036854775805

array[3] = 9.223372036854776E18

array[4] = 9.223372036854776E18

array[5] = 9.223372036854776E18

----------------------------

array[0] = 9223372036854775807

array[1] = 9223372036854775806

array[2] = 9223372036854775805

array[3] = 9.223372036854776E18

array[4] = 9.223372036854776E18

array[5] = 9.223372036854776E18

제대로 정렬되지 않았음을 알 수 있다. 위의 모든 값이 double로 형변환되면서 9.223372036854776E18로 인식되기 때문이다. 정수에서 실수로의 형변환이 어떤 경우에는 위험할 수도 있다.

* + 1. Array 클래스와 리플렉션(reflection mechanism)

리플렉션 클래스 중에는 java.lang.reflect.Array클래스가 있다. 자세한 것은 리플렉션을 다룬 장을 참조하라.

다음 예제를 보자. 굳이 설명이 필요 없는 쉬운 예제이다.

public class Example {

public static void main(String[] args) {

int[] intArray = new int[4];

intArray[0] = 3;

intArray[1] = 6;

intArray[2] = 9;

intArray[3] = 12;

System.out.println("intArray[0] =" + intArray[0]);

System.out.println("intArray[1] =" + intArray[1]);

System.out.println("intArray[2] =" + intArray[2]);

System.out.println("intArray[3] =" + intArray[3]);

}

}

결과는 다음과 같다.

intArray[0] =3

intArray[1] =6

intArray[2] =9

intArray[3] =12

똑같은 일을 하는 프로그램을 Array클래스를 이용해서 만들어보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Object intArray = Array.newInstance(int.class, 4);

Array.set(intArray, 0, new Integer(3));

Array.set(intArray, 1, new Integer(6));

Array.setInt(intArray, 2, 9);

Array.setInt(intArray, 3, 12);

System.out.println("intArray[0] =" + Array.get(intArray, 0));

System.out.println("intArray[1] =" + Array.get(intArray, 1));

System.out.println("intArray[2] =" + Array.getInt(intArray, 2));

System.out.println("intArray[3] =" + Array.getInt(intArray, 3));

}

}

결과는 위와 마찬가지이다.

이번에는 다차원 배열을 다루어 보자.

public class Example {

public static void main(String[] args) {

long[][] longArray = new long[2][3];

for (int i = 0; i < longArray.length; i++)

for (int j = 0; j < longArray[i].length; j++)

longArray[i][j] = i \* 10 + j;

for (int i = 0; i < longArray.length; i++)

for (int j = 0; j < longArray[i].length; j++)

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

for (int i = 0; i < longArray.length; i++)

for (int j = 0; j < longArray[i].length; j++)

if (longArray[i][j] % 2 == 0) {

longArray[i][j] = longArray[i][j] \* 10;

} else {

longArray[i][j] = longArray[i][j] \* 1000;

}

System.out.println();

for (int i = 0; i < longArray.length; i++)

for (int j = 0; j < longArray[i].length; j++)

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

}

}

결과는 다음과 같다.

longArray[0][0] = 0

longArray[0][1] = 1

longArray[0][2] = 2

longArray[1][0] = 10

longArray[1][1] = 11

longArray[1][2] = 12

longArray[0][0] = 0

longArray[0][1] = 1000

longArray[0][2] = 20

longArray[1][0] = 100

longArray[1][1] = 11000

longArray[1][2] = 120

똑같은 일을 하는 클래스를 Array클래스를 이용해서 만들어 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Object dim = Array.newInstance(int.class, 2);

Array.setInt(dim, 0, 2);

Array.setInt(dim, 1, 3);

Object longArray = Array.newInstance(long.class, (int[])dim);

for (int i = 0; i < Array.getLength(longArray); i++) {

Object subArray = Array.get(longArray, i);

for (int j = 0; j < Array.getLength(subArray); j++)

Array.setLong(subArray, j, i\*10+j);

}

for (int i = 0; i < Array.getLength(longArray); i++) {

Object subArray = Array.get(longArray, i);

for (int j = 0; j < Array.getLength(subArray); j++)

System.out.println("longArray["+i+"]["+j+"] = " + Array.get(subArray, j));

}

for (int i = 0; i < Array.getLength(longArray); i++) {

Object subArray = Array.get(longArray, i);

for (int j = 0; j < Array.getLength(subArray); j++) {

long value = ((Long)Array.get(subArray, j)).longValue();

if (value % 2 == 0) {

Array.set(subArray, j, new Long(value\*10));

} else {

Array.setLong(subArray, j, value\*1000);

}

}

}

System.out.println();

for (int i = 0; i < Array.getLength(longArray); i++) {

Object subArray = Array.get(longArray, i);

for (int j = 0; j < Array.getLength(subArray); j++)

System.out.println("longArray["+i+"]["+j+"] = " + Array.get(subArray, j));

}

}

}

결과는 위와 똑같다.

사서 고생을 한다는 생각이 들것이다. 대부분의 경우에는 맞는 말이다. 만약 Array클래스를 사용해도 되고 사용안해도 할 수 있는 일이라면 사용하지 않고 쉬운 길로 가는 것이 좋다. 그러나 Array클래스를 사용하지 않고는 해결할 수 없는 일도 있다. 그것은 리플렉션 클래스 일반에도 해당하는 말이다.

* + 1. 범용 깊은 복제 클래스 만들기

앞에서 본 Array클래스의 사용 예는 쓸데 없이 고생하는 것이었다면 이번에는 제대로 된 사용법을 보게 될 것이다. 리플렉션을 이용해 범용 깊은 복제(deep copy) 메쏘드를 만들어 보자. 다음 예제는 어떤 기본형 배열이라도 깊은 복제를 해준다. 코드가 잘 이해가 안 가는 사람은 리플렉션을 다루는 장을 참조하라. 그리고 재귀 호출(recursive call)에 대해서는 자료구조론 책을 참조하라.

프로그램은 두개의 파일로 되어 있다. Example.java와 PrimitiveArrayDeepCopier.java가 그것이다.

Example.java파일

public class Example {

public static void main(String[] args) {

int[][][] intArray = {

{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} },

{ {10, 11}, {12, 13}, null },

null,

{ {14}, {15, 16}, {} },

};

print(intArray);

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

int[][][] intArrayCloned = (int[][][])PrimitiveArrayDeepCopier.deepCopy(intArray);

print(intArrayCloned);

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

intArray[0][0][0] = 1000;

print(intArrayCloned);

}

public static void print(int[][][] array) {

for (int i = 0; i < array.length; i++) {

int[][] subComponent = array[i];

if (subComponent == null)

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

else if (subComponent.length == 0)

System.out.println("array["+i+"].length = 0");

else

for (int j = 0; j < subComponent.length; j++) {

int[] subSubComponent = subComponent[j];

if (subSubComponent == null)

System.out.println("array["+i+"]["+j+"] = null");

else if (subSubComponent.length == 0)

System.out.println("array["+i+"]["+j+"].length = 0");

else

for (int k = 0; k < subSubComponent.length; k++) {

System.out.println("array["+i+"]["+j+"]["+k+"] = " + subSubComponent[k]);

}

}

}

}

}

PrimitiveArrayDeepCopier.java 파일

import java.lang.reflect.\*;

public class PrimitiveArrayDeepCopier {

public static final Class BOOLEAN\_ARRAY = (new boolean[0]).getClass();

public static final Class BYTE\_ARRAY = (new byte[0]).getClass();

public static final Class SHORT\_ARRAY = (new short[0]).getClass();

public static final Class CHAR\_ARRAY = (new char[0]).getClass();

public static final Class INT\_ARRAY = (new int[0]).getClass();

public static final Class LONG\_ARRAY = (new long[0]).getClass();

public static final Class FLOAT\_ARRAY = (new float[0]).getClass();

public static final Class DOUBLE\_ARRAY = (new double[0]).getClass();

// 파라미터 objOriginal은 항상 기본형의 배열이다. null일 수 없다.

public static Object deepCopy(Object objOriginal) {

int iLengthOfOriginal = Array.getLength(objOriginal);

Class clsOriginal = objOriginal.getClass();

Object objCloned = null;

// 1차원 배열일 경우.

if ( clsOriginal.equals(BOOLEAN\_ARRAY) ||

clsOriginal.equals(BYTE\_ARRAY) ||

clsOriginal.equals(SHORT\_ARRAY) ||

clsOriginal.equals(CHAR\_ARRAY) ||

clsOriginal.equals(INT\_ARRAY) ||

clsOriginal.equals(LONG\_ARRAY) ||

clsOriginal.equals(FLOAT\_ARRAY) ||

clsOriginal.equals(DOUBLE\_ARRAY) ) {

if (clsOriginal.equals(BOOLEAN\_ARRAY))

objCloned = Array.newInstance(boolean.class, iLengthOfOriginal);

else if (clsOriginal.equals(BYTE\_ARRAY))

objCloned = Array.newInstance(byte.class, iLengthOfOriginal);

else if (clsOriginal.equals(SHORT\_ARRAY))

objCloned = Array.newInstance(short.class, iLengthOfOriginal);

else if (clsOriginal.equals(CHAR\_ARRAY))

objCloned = Array.newInstance(char.class, iLengthOfOriginal);

else if (clsOriginal.equals(INT\_ARRAY))

objCloned = Array.newInstance(int.class, iLengthOfOriginal);

else if (clsOriginal.equals(LONG\_ARRAY))

objCloned = Array.newInstance(long.class, iLengthOfOriginal);

else if (clsOriginal.equals(FLOAT\_ARRAY))

objCloned = Array.newInstance(float.class, iLengthOfOriginal);

else if (clsOriginal.equals(DOUBLE\_ARRAY))

objCloned = Array.newInstance(double.class, iLengthOfOriginal);

else

throw new IllegalArgumentException();

for (int i = 0; i < Array.getLength(objCloned); i++) {

Array.set( objCloned, i, Array.get(objOriginal,i) );

}

return objCloned;

}

// 다차원 배열일 경우.

if (Array.getLength(objOriginal) > 0) {

objCloned = Array.newInstance( Array.get(objOriginal,0).getClass(), iLengthOfOriginal );

for (int i = 0; i < Array.getLength(objCloned); i++) {

Object objOriginalComponent = Array.get(objOriginal, i);

Object objClonedComponent;

if (objOriginalComponent == null)

objClonedComponent = null;

else

objClonedComponent = deepCopy(objOriginalComponent); // 재귀 호출.

Array.set(objCloned, i, objClonedComponent);

}

return objCloned;

} else {

// boolean

Object objBooleanArray = Array.newInstance(boolean.class, 0);

Class clsBooleanArray = objBooleanArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsBooleanArray)) {

objCloned = objBooleanArray;

return objCloned;

}

}

// byte

Object objByteArray = Array.newInstance(byte.class, 0);

Class clsByteArray = objByteArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsByteArray)) {

objCloned = objByteArray;

return objCloned;

}

}

// short

Object objShortArray = Array.newInstance(short.class, 0);

Class clsShortArray = objShortArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsShortArray)) {

objCloned = objShortArray;

return objCloned;

}

}

// char

Object objCharArray = Array.newInstance(char.class, 0);

Class clsCharArray = objCharArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsCharArray)) {

objCloned = objCharArray;

return objCloned;

}

}

// int

Object objIntArray = Array.newInstance(int.class, 0);

Class clsIntArray = objIntArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsIntArray)) {

objCloned = objIntArray;

return objCloned;

}

}

// long

Object objLongArray = Array.newInstance(long.class, 0);

Class clsLongArray = objLongArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsLongArray)) {

objCloned = objLongArray;

return objCloned;

}

}

// float

Object objFloatArray = Array.newInstance(float.class, 0);

Class clsFloatArray = objFloatArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsFloatArray)) {

objCloned = objFloatArray;

return objCloned;

}

}

// double

Object objDoubleArray = Array.newInstance(double.class, 0);

Class clsDoubleArray = objDoubleArray.getClass();

for (int i = 0; i < 200; i++) {

if (clsOriginal.equals(clsDoubleArray)) {

objCloned = objDoubleArray;

return objCloned;

}

}

throw new IllegalArgumentException();

}

}

}

결과는 다음과 같다.

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

---------------------

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

---------------------

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

처음에 원래 배열(intArray)의 상태를 보여주고 두번째로 복제된 배열(intArrayCloned)의 상태를 보여준다. 세번째에는 intArray[0][0][0]에 1000을 대입한 후에 복제된 배열의 상태를 보여준다. 원본에 변화를 주어도 복제본은 영향을 받지 않는다. 즉 깊은 복제가 된 것이다. 이것은 직렬화를 이용해서 좀더 간단히 해결할 수 있다. 직렬화를 다루는 부분을 참조하라.

1. 클래스의 기초
   1. 클래스와 객체
      1. 클래스(class)와 객체(object)

일단 기본형에 대한 비교에서 시작해 보자. 다음 코드를 보자.

int number = 5;

int는 형(type)이다. number는 int형 변수이다. 5는 int형 값이다. 그 값이 number에 대입되었다. 이번에는 참조형 코드를 보자.

Integer intValue = new Integer(3);

Integer는 형이다. intValue는 Integer형 변수이다. new Integer(3)으로 Integer형 객체가 만들어진다. 이렇게 해서 만들어진 객체(엄밀하게 말하면 객체에 대한 참조reference)가 intValue에 대입되었다. 여기서 Integer와 같은 형을 클래스라고 하고 new Integer(3)으로 만들어진 것을 객체라고 한다. int형의 값이 여러개 있듯이 Integer형의 객체도 여러개 있을 수 있다.

다른 식으로 비유할 수도 있다. 사람은 여러가지 특징을 가지고 있다. 팔이 두 개, 다리가 두개, 커다란 뇌, 언어의 사용, 사회성 등등. 사람을 클래스라고 할 수 있다. 사람 클래스에 속하는 객체에는 빌 게이츠, 빌 클린턴 등등이 있다.

클래스에는 필드와 메쏘드가 있다. 필드는 클래스의 상태를 나타내고 메쏘드는 클래스가 하는 행위를 나타낸다. 필드는 변수이고, 메쏘드는 함수이다. 문법적으로 보면 클래스는 변수와 함수가 합쳐진 것이다. 그러나 그것의 파급 효과는 엄청나다.

* + 1. 객체(인스턴스instance)의 생성과 사용

이제 실제로 어떤 클래스의 객체를 만들어서 사용해 보자. 다음 예제를 보자.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

Point p; // Point형 변수 p 만들기

p = new Point(); // Point형 객체 만들기

p.x = 10; // p가 가리키는 객체의 필드 x 사용하기

double valueX = p.getX(); // p가 가리키는 객체의 메쏘드 getX 사용하기

System.out.println(valueX);

}

}

결과는 다음과 같을 것이다.

10.0

import java.awt.\*;가 필요한 이유는 Point클래스가 java.awt 패키지에 들어 있기 때문이다. 자세한 것은 패키지를 다룬 장을 참조하라.

객체를 만들기 위해서는 new 연산자가 필요하다. 이렇게 만들어진 객체를 인스턴스라고도 한다. 객체를 만들 때 클래스 이름 다음에 ()는 항상 있어야 한다. 괄호안에는 그 생성자에 해당하는 인자를 넘겨줄 수 있다. 자세한 것은 생성자를 다루는 부분을 참조하라.

필드 사용하기 위해서는 객체를 가리키는 변수와 필드 이름 사이에 .(period)를 쓰면 된다.

메쏘드를 사용하기 위해서는 객체를 가리키는 변수와 메쏘드 이름 사이에 마찬가지로 .을 쓰면 된다. 메쏘드 다름에 오는 ()는 생략될 수 없다. 괄호 안에는 그 메쏘드에 해당하는 인자를 넘겨줄 수 있다.

객체는 명시적으로 생성되어야 한다. 변수를 만든다고 객체가 생성되는 것은 아니다. 이것은 지역 변수든 필드이든 마찬가지이다. 다음 예제를 보자.

import java.awt.\*;

public class Example {

static Point field;

public static void main(String[] args) {

field.x = 10;

}

}

결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

at Example.main(Example.java:7)

Exception in thread "main"

field 변수가 만들어졌지만 객체는 만들어지지 않았기 때문에 NullPointerException이 발생한다.

* + 1. 클래스의 정의

지금까지 우리자 본 예제에서는 항상 하나 이상의 클래스를 정의해 왔다. Example 클래스를 정의해 온 것이다. 그럼 가장 간단한 클래스를 정의해 보자.

class A {

}

위에서 클래스 A가 정의되었다.

여기에 필드와 메쏘드를 추가해 보자.

class A {

int fieldA;

int getFieldA() {

return fieldA;

}

}

클래스 A에는 int형 필드 fieldA와 반환형(return type)이 int형인 메쏘드 getFieldA가 정의되었다.

이제 이 클래스를 사용해 보자. 다음 예제는 우리가 만든 클래스를 사용하는 것을 보여준다.

public class Example {

public static void main(String[] args) {

A a; // A형 변수 a 만들기

a = new A(); // A형 객체 만들기

a.fieldA = 111; // a가 가리키는 객체의 필드 fieldA 사용하기

int result = a.getFieldA(); // a가 가리키는 객체의 메쏘드 getFieldA 사용하기

System.out.println(result);

}

}

class A {

int fieldA;

int getFieldA() {

return fieldA;

}

}

결과는 다음과 같을 것이다.

111

* 1. 메쏘드
     1. 메쏘드(method)의 정의

자바에서는 프로시져(procedure), 함수(fuction), 멤버 함수(member function)라는 말을 사용하지 않고 메쏘드란 단어를 사용한다. 메쏘드는 객체지향 프로그래밍(object-oriented programming)을 강조한 단어이다.

메쏘드는 클래스 안에서 정의된다. 자바의 메쏘드는 항상 어떤 클래스(또는 인터페이스interface)의 메쏘드이다. 독립적으로 존재하는 메쏘드는 없다.

메쏘드는 다음과 같이 정의된다.

int methodA(int parameterA, double parameterB) {

...

}

처음에 쓰인 int는 이 메쏘드의 반환형(return type)이 int형임을 나타낸다. 만약 반환값(return value)이 없는 메쏘드라면 void라고 쓰면 된다. methodA는 메쏘드의 이름이다. 위에 정의된 메쏘드는 int형 매개변수 parameterA와 double형 매개변수 parameterB에 각각 인자값을 받아들인다. 매개변수가 없어라도 괄호는 생략될 수 없다. 그 다음에 오는 {} 안에 메쏘드가 실제로 실행하는 문장들이 온다. 이 메쏘드 블럭도 생략될 수 없다(나중에 배울 추상 메쏘드에서는 그렇지 않지만).

메쏘드 이름과 매개변수 리스트를 합하여 그 메쏘드의 서명(signature)이라고 한다. 서명에서는 매개변수의 형이 중요하지 매개변수의 이름이 중요하지 않다. 그리고 반환형은 서명에 포함되지 않는다. 위의 예에서 메쏘드의 서명을 나타내면

methodA(int, double)

과 같다.

* + 1. 값에 의한 호출(call by value, copy by value, pass by value)

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int num = 10;

methodA(num);

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

}

static void methodA(int param) {

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

param = -20;

}

}

결과는 다음과 같다.

param = 10

num = 10

위에서 param을 형식 매개변수(formal parameter) 또는 매개변수(parameter)라고 한다. 다음 코드에서

methodA(num);

num을 실질 매개변수(actual parameter) 또는 인자(argument)라고 한다. 인자의 값이 매개변수에 대입된다. 그래서

param = 10

라는 결과가 나오는 것이다. 인수로 넘어간 변수 값은 메쏘드 내에서 매개변수의 값을 변화시켜도 변하지 않는다. 그래서 param에 -20을 대입했음에도 num의 값은 계속 10으로 남는다.

이것을 값의 복사(copy by value)라고 한다. 자바 메쏘드에서는 항상 매개변수에 값이 복사된다. 매개변수에 객체를 넘길 때는 copy by reference방식을 사용한다고 생각하는 사람들이 있는데 그것은 잘못된 생각이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("before method invocation");

System.out.println(sb);

method(sb);

System.out.println(sb);

}

static void method(StringBuffer sb) {

sb.append("-- after method");

}

}

결과는 다음과 같다.

before method invocation

before method invocation-- after method

메쏘드를 부를 다음에 sb의 상태가 변했다. 그래서 초보자들은 자바에서는 기본형은 copy by value를 사용하고 참조형에서는 copy by reference를 사용한다는 생각을 하는 것이다. 그러나 정확히 말하면 변한 것은 변수 sb가 가리키는 객체이지 변수 sb자체의 값은(이 값은 변수가 가리키는 객체에 대한 참조이다) 아니다. 즉 다음 예제에서 보이듯이 변수 자체의 값은 변하지 않는다.

public class Example {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("before method invocation");

System.out.println(sb);

method(sb);

System.out.println(sb);

}

static void method(StringBuffer sb) {

sb = null;

}

}

결과는 다음과 같다.

before method invocation

before method invocation

정확히 말하면 객체에 대한 참조값 즉 변수의 값이 매개변수에 넘어간다. 자바에서는 참조형에서도 기본형과 마찬가지로 copy by value방식을 사용하는 것이다. 자바는 매개변수 전달 방식을 하나로 통일하였다. 이것으로 표현력을 희생해서 실수할 가능성을 줄이고 학습량도 줄였다. 즉 문제를 단순화시켰다.

* + 1. final 매개변수(final parameter)

매개변수를 final로 선언할 수 있다. 다른 곳에서 쓰이는 final과 마찬가지로 여기서도 final로 선언된 매개변수는 값을 바꿀 수 없다. 다음 예제를 보자.

public class Example {

public void method(final int finalInt, final StringBuffer finalStringBuffer) {

finalInt = 5; // Error.

finalInt = finalInt; // Error.

finalStringBuffer = new StringBuffer(); // Error.

finalStringBuffer = finalStringBuffer; // Error.

}

}

컴파일이 되지 않을 것이다.

메쏘드 내에서 변수의 값이 바뀌지 않을 것이라는 것을 아는 것은 메쏘드의 코드를 이해하는데 도움이 된다. 만약 메쏘드의 알고리즘이 복잡하고 수백 라인이 넘을 때는 이런 작은 배려가 당신의 코드를 읽는 사람에게 커다란 도움이 된다. 뿐만 아니라 코드를 작성할 때도 실수할 가능성을 줄여준다. 이제부터는 복잡한 메쏘드에서는 값이 변하지 않는 파라미터는 모두 final로 선언해서 사용해 보라. 그러나 아주 짧은 메쏘드라면 도움이 되지 않기 때문에 final을 안쓰는 것이 낫다.

여기서도 객체와 변수의 관계를 정확히 하지 않으면 오해가 생길 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Original");

method(sb);

System.out.println(sb);

}

public static void method(final StringBuffer finalStringBuffer) {

finalStringBuffer.append("-Changed");

}

}

결과는 다음과 같을 것이다.

Original-Changed

변하지 않는 것은 finalStringBuffer가 어떤 객체를 가리킨다는 사실이지 그 객체 자체가 불변인 것은 아니다. 쉽게 말하면 finalStringBuffer와 그것이 가리키는 객체 사이의 연결이 끊어지지 않는다는 것이다. 그러므로 참조형 변수를 final로 선언하는 것은 기본형 변수처럼 그렇게 큰 의미는 없다.

* + 1. return문과 반환값

return문은 메쏘드의 실행을 즉시 끝내는 역할을 한다. 만약 반환형(return type)이 void가 아니라면 return문에 그 형에 해당하는 값을 넣어 주어야 한다. void형 메쏘드에는 return문이 없어도 된다. 그 때에는 메쏘드의 끝까지 실행된 다음에 리턴된다.

void형이 아닌 메쏘드에는 반드시 return문이 있어야 한다. 게다가 return 다음에 나오는 값은 그 메쏘드의 반환형에 대입 가능해야 한다. 그리고 void형이 아닌 메쏘드에서는 반드시 값을 반환하는 return문으로 메쏘드가 끝나야 한다. 여기서 return문으로 메쏘드가 끝나야 한다는 의미는 맨 마지막 줄이 return문이어야 한다는 의미는 아니다. 제어 흐름상에서 끝이라는 의미이다. 그러므로 다음 예제는 컴파일이 되지 않을 것이다.

public class Example {

int methodA() { // 리턴문이 없다.

}

int methodB() {

return 1.0; // double이 int로 변환될 수 없다.

}

int methodC(int a) {

if (a > 0)

return 1;

// int를 리턴하지 않을 수도 있다.

}

}

return키워드 뒤에는 여러가지가 들어갈 수 있다. 물론 void형에서처럼 아무것도 없을 때도 있다. 반환형과 호환되는 값을 가진다면 어떤 것도 올 수 있다. 다음 예제를 보자.

public class Example {

void methodA() { // void형에서는 return문이 없을 수도 있다.

}

void methodB() {

return;

}

int methodC() {

return 10;

}

int methodD(boolean b) { // 계산식이 쓰일 수도 있다.

if (b)

return 10 \* 20 + 30; // 괄호를 사용할 수도 안 할 수도 있다.

else

return (10 \* 20 + 30); // 복잡할 때는 괄호를 사용하는 것이 알아보기 쉬울 것이다.

}

int methodE() {

return Math.abs(-20); // 메쏘드를 부를 수도 있다.

}

int methodF() {

return methodA(); // Error. methodA는 int를 리턴하지 않는다.

}

int methodG() {

return Math.sqrt(1); // Error. double은 int에 대입될 수 있다.

}

int methodH() {

return (int)Math.sqrt(1); // OK. 명시적으로 형변환을 해 주었다.

}

String methodI() {

return new String("methodI"); // 객체를 생성해서 바로 리턴할 수도 있다.

}

Object methodJ() {

return new String("methodJ"); // OK. String형은 Object형에 호환된다.

}

}

두 군데에서 컴파일타임 오류가 날 것이다. 그러나 return문에 너무 복잡한 식을 쓰는 것은 좋지 않다.

return문은 정확이 무엇을 반환(return)할까? 이것을 정확히 알아야 한다. 일단 기본형은 간단하다. 그 기본형의 값을 반환하는 것이다. 문제는 참조형(객체형)인데 다음 예제를 보자. 잘 이해가 안되는 사람은 이 장을 다 읽고 다시 보기 바란다. 이 예제에서는 점의 움직임을 추적한다. 현재의 점의 위치(currentPoint)와 현재까지 움직인 거리(distanceMoved)를 속성으로 갖는다. 둘은 private으로 되어 있다. 즉 다른 클래스에서 직접 접근할 수 없는 것이다. 그리고 그 필드를 바꾸어주는 것은 move메쏘드 뿐이다. 그러므로 이 클래스의 contract는 깨질 수 없어 보인다. 즉 움직인 거리는 현재까지 move 메쏘드를 통해서 움직인 거리와 일치해야 하고 현재의 점의 위치는 현재까지 움직임으로 결정된 최종적인 점의 위치가 되어야 한다. 시작점은 원점이고 당연히 움직인 거리의 초기값은 0.0이다. 실제로 테스트 결과를 보더라도 모든 것이 잘 작동하는 것으로 보인다.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

MovingPoint mp = new MovingPoint();

mp.move(10, 0);

mp.showInformation();

mp.move(0, 10);

mp.showInformation();

mp.move(-10, 0);

mp.showInformation();

mp.move(0, -10);

mp.showInformation();

}

}

class MovingPoint {

private Point currentPoint = new Point(0, 0);

private double distanceMoved = 0.0;

// 이렇게 속성을 바꿀 수 있는 메쏘드를 Mutator라고 한다.

// Mutator는 보통 setXXX의 형식을 취한다.

public void move(int dx, int dy) {

distanceMoved += Math.sqrt(dx \* dx + dy \* dy);

currentPoint.translate(dx, dy);

}

// 이렇게 속성을 알아내는 메쏘드를 Accessor라고 한다.

// Accessor는 보통 getXXX의 형식을 취한다.

public Point getCurrentPoint() {

return currentPoint;

}

// 또하나의 Accessor

public double getDistanceMoved() {

return distanceMoved;

}

public void showInformation() {

System.out.println("currentPoint = (" + currentPoint.x + "," + currentPoint.y

+ ") distanceMoved = " + distanceMoved);

}

}

결과는 다음과 같다.

currentPoint = (10,0) distanceMoved = 10.0

currentPoint = (10,10) distanceMoved = 20.0

currentPoint = (0,10) distanceMoved = 30.0

currentPoint = (0,0) distanceMoved = 40.0

그러나 여기에는 함정이 있다. 바로 getCurrentPoint메쏘드이다. 이 메쏘드의 return 문을 무엇을 반환하는 것일까? 다음 예제를 보면 알 수 있을 것이다.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

MovingPoint mp = new MovingPoint();

Point p = mp.getCurrentPoint();

p.setLocation(100, 100);

mp.showInformation();

}

}

class MovingPoint {

private Point currentPoint = new Point(0, 0);

private double distanceMoved = 0.0;

public void move(int dx, int dy) {

distanceMoved += Math.sqrt(dx \* dx + dy \* dy);

currentPoint.translate(dx, dy);

}

public Point getCurrentPoint() {

return currentPoint;

}

public double getDistanceMoved() {

return distanceMoved;

}

public void showInformation() {

System.out.println("currentPoint = (" + currentPoint.x + "," + currentPoint.y

+ ") distanceMoved = " + distanceMoved);

}

}

결과는 (어떤 사람들에게는) 놀랍게도 다음과 같다.

currentPoint = (100,100) distanceMoved = 0.0

return문에서 반환하는 것은 참조형일 때에는 그 객체에 대한 참조이다. 객체를 복사해서 리턴하는 것이 아니다. currentPoint에 대한 참조를 이용해서 그 객체의 상태를 조작할 수 있었다. 만약 이것을 막으려면 다음과 같이 해야 한다.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

MovingPoint mp = new MovingPoint();

Point p = mp.getCurrentPoint();

p.setLocation(100, 100);

mp.showInformation();

}

}

class MovingPoint {

private Point currentPoint = new Point(0, 0);

private double distanceMoved = 0.0;

public void move(int dx, int dy) {

distanceMoved += Math.sqrt(dx \* dx + dy \* dy);

currentPoint.translate(dx, dy);

}

public Point getCurrentPoint() {

return new Point(currentPoint);

}

public double getDistanceMoved() {

return distanceMoved;

}

public void showInformation() {

System.out.println("currentPoint = (" + currentPoint.x + "," + currentPoint.y

+ ") distanceMoved = " + distanceMoved);

}

}

결과는 다음과 같다.

currentPoint = (0,0) distanceMoved = 0.0

이제는 객체를 조작할 수 없게 되었다. 이젠 정말 move메쏘드를 통해서만 객체의 상태를 바꿀 수 있다. 이것은 Point객체를 새로 만들어서 반환했기 때문이다. getXXX메쏘드에서 참조형 필드가 가리키는 객체의 참조를 그대로 넘겨줄 것인가 아니면 복사해서 넘겨줄 것인가는 독자가 선택해야 한다. 복사해서 넘겨주면 여기서 보았듯이 안정성이 높아진다. 그러나 여기서의 예에서는 getCurrentPoint 메쏘드를 부를 때마다 Point객체가 하나씩 만들어져야 한다. 이것은 실행 시간과 메모리 공간에 부담을 준다. 게다가 만약 복사되어야 하는 객체가 Point객체처럼 작은 객체가 아니라 커다란 객체라면 어쩔 것인가? 여기에 대한 답은 상황에 맞게 하라는 것 뿐이다. 안정성이 중요시되는 상황인지 효율성(속도)이 중요시되는 상황인지를 그리고 객체의 복사가 얼마나 부담을 주는지를 따져야 한다.

* + 1. 메쏘드의 중복정의 (overloading)

클래스가 다르다면 메쏘드의 이름이 같은 것은 문제가 안된다. 이것이 클래스를 사용하는 이점 중의 하나이다. 다음 예제는 무사히 컴파일된다.

public class Example {

public static void main(String[] args) {

A.main(args);

}

}

class A {

public static void main(String[] args) {

for (int i = 0; i < args.length; i++)

System.out.println(args[i]);

}

}

결과는 다음과 같다.

java Example Same method name does not matter.라고 도스창 등에서 입력

Same

method

name

does

not

matter.

다음 예제를 보자. 하나의 클래스에서 같은 이름의 두개의 메쏘드가 있기 때문에 에러가 난다. 컴파일러는 어떤 메쏘드를 사용할지 알 수가 없는 것이다.

public class Example {

void method() {

}

void method() {

}

}

그러나 같은 이름의 메쏘드를 하나의 클래스에 여러 개를 선언할 수 있다. 그러려면 컴파일러가 각 메쏘드를 구분할 수 있어야 한다. 컴파일러는 메쏘드 이름과 매개변수의 매수와 각 매개변수의 형(type)으로 각 메쏘드를 구분한다. 이 때 반환형(return type)은 고려되지 않는다. 위에서 보았듯이 메쏘드의 이름, 매개변수의 갯수, 각 매개변수의 형을 모두 합친 것은 그 메쏘드의 서명(signiture)라고 한다. 같은 이름의 여러개의 메쏘드를 선언하는 것을 메쏘드 중복정의 (overloading)라고 한다. 다음은 java.lang.Math클래스 중 일부를 인용한 것이다.

package java.lang;

import java.util.Random;

public final strictfp class Math {

// 많은 부분이 생략됨.

public static int abs(int a) {

return (a < 0) ? -a : a;

}

public static long abs(long a) {

return (a < 0) ? -a : a;

}

public static float abs(float a) {

return (a <= 0.0F) ? 0.0F - a : a;

}

public static double abs(double a) {

return (a <= 0.0D) ? 0.0D - a : a;

}

}

메쏘드의 이름은 abs로 모두 같지만 매개변수가 모두 다르기 때문에 컴파일러가 잘 구분할 수 있는 것이다. 중복정의된 메쏘드의 사용법은 대부분 명백하다. 여러 메쏘드 중 자신에게 필요한 매개변수를 제공하는 것을 사용하면 된다. 위의 예제에서 중복정의 장점을 볼 수 있을 것이다. 만약 중복정의를 사용할 수 없다면 각각의 형마다 다른 메쏘드 이름을 붙여야 할것이다. 위의 코드는 다음과 같이 만들어야 할 것이다.

package java.lang;

import java.util.Random;

public final strictfp class Math {

// 많은 부분이 생략됨.

public static int absInt(int a) {

return (a < 0) ? -a : a;

}

public static long absLong(long a) {

return (a < 0) ? -a : a;

}

public static float absFloat(float a) {

return (a <= 0.0F) ? 0.0F - a : a;

}

public static double absDouble(double a) {

return (a <= 0.0D) ? 0.0D - a : a;

}

}

메쏘드 이름이 더 많아진다는 것은 클래스를 만드는 사람에게나 사용하는 사람에게나 더 부담을 준다.

메쏘드 중복정의는 위의 예에서처럼 의미상 같은 일을 하는 것에 한정해서 사용해야 한다. 위의 예에서는 모두 절대값을 구하는 메쏘드였다. 다음 예제를 보자.

public class Example {

void overLoadingAbuse(String str) {

System.out.println(str);

}

String overLoadingAbuse(int i) {

return "" + i;

}

int overLoadingAbuse(int a, int b) {

return a + b;

}

}

컴파일은 잘 되지만 중복정의를 사용해서 더 헷갈리는 코드를 만들었다.

* 1. 필드
     1. 필드(field)와 지역 변수(local variable)

필드는 클래스의 상태를 나타내는 변수이다. 그렇다면 지역 변수와 필드는 어떻게 구별할까? 만약 그 변수가 메쏘드나 생성자등의 블록 내에서 정의된다면 그것은 지역 변수(local variable)이다. 변수가 클래스 블록 내에서(그리고 그 클래스의 메쏘드나 생성자 등의 블록의 밖에서) 정의된다면 그것은 필드(field)이다. 필드를 멤버 변수(member variable)이라고도 한다. 필드를 사용하려면 객체를 가리키는 변수나 클래스 이름에 .을 붙이고 필드 명을 쓰면 된다. 그러나 그 필드를 정의한 클래스에서는 필드 이름만 써도 된다.

다음은 필드를 정의하고 사용하는 예제이다.

public class Example {

public static void main(String[] args) {

A a = new A(); // a는 메쏘드 내에서 정의되었기 때문에 필드가 아니라 지역변수이다.

a.intField = 10; // 필드의 사용.

System.out.println(a.intField);

a.setIntField(20);

a.printIntField();

}

}

class A {

int intField;

void setIntField(int i) {

intField = i; // 필드가 정의된 클래스 내에서는 필드 이름(여기서는 intField)만 써도 된다.

}

void printIntField() {

System.out.println(intField); // 필드가 정의된 클래스 내에서는 필드 이름(여기서는 intField)만 써도 된다.

}

}

결과는 다음과 같을 것이다.

10

20

* + 1. 필드의 초기화와 디폴트(default) 초기값

필드를 선언하면서 초기화 할 수 있다. 만약 초기화하지 않았다면 디폴트 값으로 초기화된다. 이것은 지역변수와 다르다. 지역변수에서는 초기화되지 않은 변수는 사용할 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println(a.intField); // OK. 초기화하지 않아도 사용할 수 있다.

int intValue;

System.out.println(intValue); // Error. 변수에 값을 명시적으로 대입하지 않으면 사용할 수 없다.

}

}

class A {

int intField;

}

지역변수를 사용하는 데서 컴파일타임 오류가 난다.

필드는 다음과 같이 초기화 한다.

class A {

int intField = 10;

String strField = new String("init value");

}

다음 예제는 필드의 디폴트 초기값을 보여준다.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println("a.byteField = " + a.byteField);

System.out.println("a.shortField = " + a.shortField);

System.out.println("a.(int)charField = " + (int)a.charField);

System.out.println("a.intField = " + a.intField);

System.out.println("a.longField = " + a.longField);

System.out.println("a.floatField = " + a.floatField);

System.out.println("a.doubleField = " + a.doubleField);

System.out.println("a.booleanField = " + a.booleanField);

System.out.println("a.objectField = " + a.objectField);

System.out.println("a.stringField = " + a.stringField);

System.out.println("a.arrayField = " + a.arrayField);

}

}

class A {

byte byteField;

short shortField;

char charField;

int intField;

long longField;

float floatField;

double doubleField;

boolean booleanField;

Object objectField;

String stringField;

int[] arrayField;

}

결과는 다음과 같다.

a.byteField = 0

a.shortField = 0

a.(int)charField = 0

a.intField = 0

a.longField = 0

a.floatField = 0.0

a.doubleField = 0.0

a.booleanField = false

a.objectField = null

a.stringField = null

a.arrayField = null

숫자일 때는 0, 참조형일 때는 null, 논리형일 때는 false이다. 이것을 표로 나타내면 다음과 같다.

|  |  |
| --- | --- |
| boolean | false |
| char | ‘\u0000’ |
| byte, short, int, long | 0 |
| float | +0.0f |
| double | +0.0 |
| 참조형 | null |

실수형에서는 –0.0이 아니라 +0.0으로 초기화되는데 그렇게 중요한 것은 아니지만 다음 예제를 보면 그 차이를 알 수 있다.

public class Example {

static float defaultValue;

public static void main(String[] args) {

System.out.println(1 / defaultValue);

System.out.println(1 / +0.0);

System.out.println(1 / -0.0);

}

}

결과는 다음과 같을 것이다.

Infinity

Infinity

-Infinity

* + 1. final필드

지역 변수를 final를 선언할 수 있듯이 필드도 final로 선언할 수 있다. final로 선언된 필드는 한번 값이 정해지면 그 값을 바꿀 수 없다. final필드는 상수와 비슷하다. 그리고 관례에 따라 모두 대문자로 쓴다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.INT\_FIELD = 20; // Error. final필드의 값을 바꿀 수 없다.

}

}

class A {

final int INT\_FIELD = 10;

}

컴파일타임 오류가 날 것이다.

참조형 필드의 값도 마찬가지로 바꿀 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.FINAL\_FIELD = new StringBuffer("change"); // Error. final필드의 값을 바꿀 수 없다.

}

}

class A {

final StringBuffer FINAL\_FIELD = new StringBuffer("final");

}

컴파일타임 오류가 난다.

그러나 참조형 final 필드(지역 변수도 마찬가지)를 사용할 때는 조심할 것이 있다. 필드의 값을 바꿀 수 없다고 해서 필드가 가리키는 객체의 상태를 변화시킬 수 없는 것은 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println("a.FINAL\_FIELD = " + a.FINAL\_FIELD);

a.FINAL\_FIELD.append("-changed"); // final필드의 상태를 바꿀 수 있다.

System.out.println("a.FINAL\_FIELD = " + a.FINAL\_FIELD);

}

}

class A {

final StringBuffer FINAL\_FIELD = new StringBuffer("final");

}

결과는 다음과 같다.

a.FINAL\_FIELD = final

FINAL\_FIELD = final-changed

* 1. 생성자
     1. 생성자(constructor) 정의와 사용

생성자는 많은 부분에서 메쏘드와 비슷하다. 다음 예제를 보자. 생성자를 정의하고 사용하는 예제이다.

public class Example {

public static void main(String[] args) {

A a = new A(10); // 생성자의 사용법.

System.out.println("a.intValue = " + a.intValue);

}

}

class A {

A(int value) { // 생성자.

intValue = value;

}

int intValue;

}

결과는 다음과 같다.

a.intValue = 10

먼저 생성자에는 반환형을 지정하지 않는다. 생성자를 사용하는 new문에서는 항상 생성되는 객체를 반환하기 때문에 이것은 당연하다. 생성자에서 return문을 사용할 수는 있지만 값을 지정할 수는 없다. 다음 예제를 보자. 컴파일이 되지 않을 것이다.

public class Example {

Example(boolean b) {

if (b)

return; // OK.

else

return this; // Error.

}

}

생성자 이름(?)은 항상 클래스 이름과 같다. 이 예에서도 클래스 이름인 A를 사용했다. 생성자는 메쏘드와 마찬가지로 매개변수를 가질 수 있다. 물론 없을 수도 있다.

생성자를 메쏘드처럼 직접 부를 수는 없다. 그러므로 다음 예제는 컴파일이 되지 않을 것이다.

public class Example {

public static void main(String[] args) {

A.A(); // Error.

A a = new A();

a.A(); // Error.

}

}

class A {

A() {}

}

* + 1. 디폴트 생성자(default consturctor)와 no-argument 생성자(no-arg constructor)

만약 생성자를 명시적으로 정의해 주지 않으면 디폴트 생성자가 쓰인다. 디폴트 생성자는 매개 변수가 없으며 (거의) 아무 일도 하지 않는 생성자이다. 다음 예제를 생성자를 명시적으로 정의하지 않았다.

class A {

}

자바에서는 위의 예제를 다음과 같이 바꾸어 준다.

class A {

A() { // 디폴트 생성자.

super(); // 이것에 대해서는 상속을 다루는 부분을 참조하라.

}

}

생성자 중에 인자가 없는 생성자를 no-argument(no-arg) 생성자라고 한다. 디폴트 생성자도 인자가 없으므로 no-argument 생성자이다. 디폴트 생성자가 아니라도 즉 아래의 예도 인자가 없으므로 no-argemnt 생성자이다.

class A {

A() {

int i = 0;

}

}

* + 1. 초기화는 생성자에서 하자.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Circle a = new Circle();

a.centerX = 10;

a.centerY = 21;

a.radius = 10.0;

a.area = Math.PI \* a.radius \* a.radius;

Circle b = new Circle();

b.centerX = -10;

b.centerY = 231;

b.radius = 120.3;

b.area = Math.PI \* a.radius \* b.radius; // 실수. a.radius로 잘못 씀.

Circle c = new Circle();

c.centerX = -123;

// 실수. centorY 설정하는 것을 빼먹음.

c.radius = .3;

c.area = Math.PI \* c.radius \* c.radius;

}

}

class Circle {

int centerX;

int centerY;

double radius;

double area;

}

이것을 다음과 같이 바꿔보자.

public class Example {

public static void main(String[] args) {

Circle a = new Circle(10, 21, 10.0);

Circle b = new Circle(-10, 231, 120.3);

Circle c = new Circle(-123, 24, .3);

}

}

class Circle {

int centerX;

int centerY;

double radius;

double area;

Circle(int x, int y, double r) {

centerX = x;

centerY = y;

radius = r;

area = Math.PI \* r \* r;

}

}

필드의 값을 생성자에서 설정하면 여러가지 이익이 있다. 가장 먼저 눈에 띄는 것은 코드량이 줄었고 훨씬 깔끔해졌다는 것이다. 더 중요한 것은 실수할 가능성 중 많은 것이 원천봉쇄되었다는 것이다.

* 1. this
     1. this의 필요성

어떤 메쏘드에서 객체 자신을 반환(return)하려면 어떻게 할까? 또는 인자로 객체 자신을 넘겨주어야 할 때는 어떻게 할까? 그 때 사용하는 것이 this이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A(10);

A b = a.printAndGetMe();

System.out.println("Example.main() b.intValue = " + b.intValue);

}

}

class A {

int intValue;

A(int i) {

intValue = i;

}

A printAndGetMe() {

B.printA(this); // 자기 자신을 파라미터로 넘긴다.

return this; // 자기 자신을 리턴한다.

}

}

class B {

static void printA(A a) {

System.out.println("B.printA() intValue = " + a.intValue);

}

}

결과는 다음과 같다.

B.printA() intValue = 10

Example.main() b.intValue = 10

* + 1. 메쏘드 또는 생성자에서의 this

다음 예제는 메쏘드와 생성자에서 this를 사용하는 것을 보여준다.

class A {

int intValue;

A(int i) {

this.intValue = i; // intValue = i; 와 같다.

}

void setIntValue(int i) {

this.intValue = i; // intValue = i; 와 같다.

}

}

위에서는 this를 쓰지 않아도 된다. 사실 이럴 때는 안쓰는 것이 더 낫다. 그러나 this를 안 쓰면 안되는 경우가 있는데 지역 변수나 매개변수가 필드와 이름이 같은 경우이다. 다음을 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.setIntValueA(111);

System.out.println("a.intValue = " + a.intValue);

A b = new A();

b.setIntValueB(2222);

System.out.println("b.intValue = " + b.intValue);

}

}

class A {

int intValue = 10;

void setIntValueA(int i) {

int intValue;

intValue = i; // 여기서 intValue는 지역 변수

}

void setIntValueB(int i) {

int intValue;

this.intValue = i; // 여기서 intValue는 필드

}

}

결과는 다음과 같다.

a.intValue = 10

b.intValue = 2222

결과를 보면 this의 사용법을 알 수 있을 것이다. 위의 예는 this의 사용법을 설명하기 위한 예이다. 필드 이름과 지역 변수 이름을 같게 만드는 것은 아주 나쁜 스타일이다. 그러나 그것이 허용되는 경우가 있다. 다음 예제가 그것을 보여준다.

class A {

int intValue;

A(int intValue) {

this.intValue = intValue;

}

void setIntValue(int intValue) {

this.intValue = intValue;

}

}

위의 예제에서 생성자와 메쏘드의 매개변수의 이름을 필드와 같게 했다. 바로 그 필드를 설정하는 코드이기 때문이다. 이 때에는 예외적으로 이름 충돌이 권장된다.

* + 1. 생성자를 부르는 this

생성자도 메쏘드와 비슷하게 중복정의(overloading)될 수 있다. 생성자에서 중복정의된 다른 생성자를 부르는데 사용할 수 있다. this 키워드를 사용할 수 있다.

생성자를 부르는 this에 대해서 알아보자. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

new A();

System.out.println();

new A(2, 3);

System.out.println();

new A(4, 5, 6);

}

}

class A {

String strX;

String strY;

String strZ;

A(int x, int y, int z) {

System.out.println("A(int x, int y, int z)");

strX = "" + x;

strY = "" + y;

strZ = "" + z;

}

A(int x, int y) {

this(x, y, 0);

System.out.println("A(int x, int y)");

}

A() {

this(0, 0);

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

}

}

결과는 다음과 같다.

A(int x, int y, int z)

A(int x, int y)

A()

A(int x, int y, int z)

A(int x, int y)

A(int x, int y, int z)

실행되는 순서는 다음과 같다. new A();만 예로 들때.

new A();가 실행되면 생성자 A()가 실행된다.

this(0, 0);는 생성자 A(int, int)를 부른다.

this(x, y, 0);는 생성자 A(int, int, int)를 부른다.

생성자 A(int, int, int)의 실행이 끝나고 제어는 생성자 A(int, int)로 넘겨진다.

생성자 A(int, int)의 실행이 끝나고 제어는 생성자 A()로 넘겨진다.

생성자 A()가 끝나고 생성된 객체가 반환된다.

위의 예제에서 거의 모든 일이 생성자 A(int, int, int)에서 일어난다. 이런식으로 하면 중복되지 않는다. 이것은 다음과 같이 고쳐보면 알 수 있다. 코드의 중복이 일어나기 때문에 코드량이 많아지고 실수할 가능성도 커진다.

class A {

String strX;

String strY;

String strZ;

A(int x, int y, int z) {

strX = "" + x;

strY = "" + y;

strZ = "" + z;

}

A(int x, int y) {

strX = "" + x;

strY = "" + y;

strZ = "" + 0;

}

A() {

strX = "" + 0;

strY = "" + 0;

strZ = "" + 0;

}

}

생성자를 부르는 this앞에는 어떤 것도 올 수 없다. 예를 들어 다음 코드는 컴파일되지 않는다.

class A {

A(int x, int y) {

}

A() {

System.out.println("A()"); // this()앞에는 어떤 것도 올 수 없다.

this(0, 0);

}

}

* 1. 정적(static)과 비정적(non-static)
     1. 정적(static) 필드와 인스턴스(instance) 필드

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

A b = new A();

A.staticField = 10;

System.out.println("A.staticField = " + A.staticField);

System.out.println("a.staticField = " + a.staticField);

System.out.println("b.staticField = " + b.staticField);

System.out.println();

a.staticField = 50;

System.out.println("A.staticField = " + A.staticField);

System.out.println("a.staticField = " + a.staticField);

System.out.println("b.staticField = " + b.staticField);

System.out.println();

a.instanceField = 20;

b.instanceField = 30;

System.out.println("a.instanceField = " + a.instanceField);

System.out.println("b.instanceField = " + b.instanceField);

System.out.println();

a.instanceField = 222;

System.out.println("a.instanceField = " + a.instanceField);

System.out.println("b.instanceField = " + b.instanceField);

}

}

class A {

static int staticField; // 정적 필드

int instanceField; // 인스턴스 필드

}

결과는 다음과 같다.

A.staticField = 10

a.staticField = 10

b.staticField = 10

A.staticField = 50

a.staticField = 50

b.staticField = 50

a.instanceField = 20

b.instanceField = 30

a.instanceField = 222

b.instanceField = 30

필드에 static키워드를 붙이면 정적 필드, 안 붙이면 인스턴스 필드가 된다. 정적 필드(static field)를 정적 변수(static variable), 클래스 변수(class variable)라고도 한다. 인스턴스 필드(instance field)를 비정적 필드(non-static field), 인스턴스 변수(instance variable), 비정적 변수(non-static variable)라고도 한다. 수식어 없이 그냥 필드라고 하면 인스턴스 필드를 뜻하기도 하고 인스턴스 필드와 정적 필드 모두를 뜻하기도 한다.

정적 필드는 그 클래스에 하나밖에 없는 변수이고 인스턴스 필드는 객체(인스턴스) 마다 따로따로 있는 변수이다. 위의 예제를 잘 보면 둘 사이의 차이를 알 수 있을 것이다.

인스턴스 필드를 사용할 때에는

a.staticField = 50;

와 같이 사용한다.

정적 필드는

A.staticField = 10;

a.staticField = 50; // 이렇게 하면 사람들이 인스턴스 필드라고 착각하기 쉽다.

와 같이 사용할 수 있다. 그러나 클래스 이름을 사용하는 것이 좋다. 예를 들어 다음 예제처럼 사용하면 사람들을 골탕먹이기에나 좋다.

public class Example {

public static void main(String[] args) {

System println = null;

println.out.println("println.out.println"); // out은 정적 필드이다.

}

}

결과는 다음과 같다.

println.out.println

다음은 정적 필드의 사용법을 보여준다. Product클래스는 제품을 나타내고 serialNumber는 제품 번호를 나타낸다.

public class Example {

public static void main(String[] args) {

Product a = new Product();

System.out.println("a.serialNumber = " + a.serialNumber);

System.out.println("Product.nextSerialNumber = " + Product.nextSerialNumber);

Product b = new Product();

System.out.println("b.serialNumber = " + b.serialNumber);

System.out.println("Product.nextSerialNumber = " + Product.nextSerialNumber);

Product c = new Product();

System.out.println("c.serialNumber = " + c.serialNumber);

System.out.println("Product.nextSerialNumber = " + Product.nextSerialNumber);

}

}

class Product {

static int nextSerialNumber = 1;

int serialNumber;

Product() {

serialNumber = nextSerialNumber;

nextSerialNumber++;

}

}

결과는 다음과 같다.

a.serialNumber = 1

Product.nextSerialNumber = 2

b.serialNumber = 2

Product.nextSerialNumber = 3

c.serialNumber = 3

Product.nextSerialNumber = 4

* + 1. 인스턴스 메쏘드

인스턴스 메쏘드를 객체 메쏘드(object method) 또는 비정적 메쏘드(non-static method)라고도 한다. 인스턴스 메쏘드는 항상 어떤 객체와 연결되어 사용한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A(123);

a.print();

A b = new A(999);

b.print();

}

}

class A {

int field;

A(int i) {

field = i;

}

void print() {

System.out.println(field);

}

}

결과는 다음과 같을 것이다.

123

999

같은 print 메쏘드를 호출했는데도 어떤 객체와 연결되어 호출했는가에 따라 결과가 다르다. 인스턴스 메쏘드를 부르는 전형적인 모습은

a.print();

에서 볼 수 있다. 여기서 a는 print 메쏘드와 연결될 객체를 가리키고 있다. 인스턴스 메쏘드는 항상 어떤 객체(인스턴스)의 메쏘드라는 것을 기억하라. 그래서 이름도 객체(인스턴스) 메쏘드인 것이다. 그러나 다음 예제는 언뜻 보기에는 연결된 객체가 없는 듯이 보인다.

public class Example {

public static void main(String[] args) {

A a = new A(123);

a.print();

A b = new A(999);

b.print();

}

}

class A {

int field;

A(int i) {

field = i;

}

void print() {

printBinary(); // 여기에 주목하라.

printHex(); // 여기에 주목하라.

}

void printBinary() {

System.out.println(field + " : " + Integer.toBinaryString(field));

}

void printHex() {

System.out.println(field + " : " + Integer.toHexString(field));

}

}

결과는 다음과 같을 것이다.

123 : 1111011

123 : 7b

999 : 1111100111

999 : 3e7

위의 예제에서 printBinary 메쏘드와 printHex 메쏘드를 부를 때에는 명시적으로 연결된 객체를 알려주지 않았다. 그럴때도 연결되는 객체는 분명히 있다. 그리고 그 객체는 바로 this 객체이다. 그러므로 위의 예제는 다음과 동일하다.

public class Example {

public static void main(String[] args) {

A a = new A(123);

a.print();

A b = new A(999);

b.print();

}

}

class A {

int field;

A(int i) {

field = i;

}

void print() {

this.printBinary(); // 여기에 주목하라.

this.printHex(); // 여기에 주목하라.

}

void printBinary() {

System.out.println(field + " : " + Integer.toBinaryString(field));

}

void printHex() {

System.out.println(field + " : " + Integer.toHexString(field));

}

}

어떤 객체와 연결되어 있다는 사실은 실행 결과를 보아도 알 수 있다. 어떤 객체와 연결되어 있기 때문에 그 객체의 인스턴스 필드인 field를 사용할 수 있고 그 field 값에 따라 결과가 나오는 것이다. 이와 같은 예에서는 보통 this를 생략한다. 그 자신이기 때문에 지정해 줄 필요가 없기 때문이다.

다음 예제를 보면 인스턴스 메쏘드가 반드시 연결된 객체가 있어야 한다는 것을 잘 보여준다.

public class Example {

public static void main(String[] args) {

A a = null;

a.instanceMethod();

}

}

class A {

void instanceMethod() {

}

}

컴파일은 되지만 실행 결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

at Example.main(Example.java:4)

Exception in thread "main"

변수 a가 아무 객체도 가리키지 않으므로 즉 null이므로 인스턴스 메쏘드가 실행될 수 없는 것이다. 이와 같은 상황에서 발생하는 NullPointerException은 앞으로 프로그래밍을 하다보면 귀신같이 자신을 따라다니는 것을 알게될 것이다.

* + 1. 정적 메쏘드(static method)

정적 메쏘드는 클래스 메쏘드(class method)라고도 한다. 정적 필드와 마찬가지로 메쏘드를 선언할 때 static 키워드를 사용한다. 스태틱 메쏘드는 인스턴스 메쏘드와 달리 절대로 객체와 연결되지 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A.staticMethod();

}

}

class A {

static void staticMethod() {

System.out.println("I'm static method");

}

}

결과는 다음과 같을 것이다.

I'm static method

객체와 연결되지 않기 때문에

A.staticMethod();

와 같은 식으로 사용한다. 그리고 또한 객체와 연결되지 않기 때문에 인스턴스 필드를 사용할 수 없다. 필드는 정적 필드만 사용할 수 있는 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A.staticMethod();

}

}

class A {

static int staticField;

int instanceField;

static void staticMethod() {

staticField = 312; // OK.

instanceField = 888; // Error.

}

}

컴파일이 되지 않을 것이다. 마찬가지로 정적 메쏘드 안에서는 인스턴스 메쏘드도 사용할 수 없다. 인스턴스 메쏘드도 어떤 객체와 연결되어 있기 때문이다. 정적 메쏘드에서 인스턴스 메쏘드를 허용한다면 결국 인스턴스 필드를 사용할 수 있게 된다. 왜냐하면 인스턴스 메쏘드에서 인스턴스 필드를 사용하기 때문이다. 그러나 인스턴스 메쏘드는 정적 메쏘드를 사용할 수 있다. 이런 면에서 정적 메쏘드는 주기만 하고 받을 수는 없는 혈액형 O형과 비슷하다. 연결된 객체가 없으므로 정적 메쏘드는 당연히 this도 사용할 수 없다.

정적 메쏘드를 다음의 예제와 같이 사용할 수는 있다. 이것을 보고 정적 메쏘드가 객체와 연결된다고 착각할 수도 있다.

public class Example {

public static void main(String[] args) {

A a = new A();

a.staticMethod();

}

}

class A {

static void staticMethod() {

System.out.println("이렇게 사용하지 말자!!!");

}

}

결과는 다음과 같다.

이렇게 사용하지 말자!!!

그러나 a가 가리키는 객체와 연결되는 것이 절대 아니다. 여기서는 단지 a의 타입이 A라는 것을 알아내고 인스턴스 a가 아닌 클래스 A의 메쏘드 staticMethod를 부르는 것일 뿐이다. 다음과 같이 예제를 고쳐 보면 이해가 갈 것이다.

public class Example {

public static void main(String[] args) {

A a = null; // 여기에 주목하라.

a.staticMethod();

}

}

class A {

static void staticMethod() {

System.out.println("이렇게 사용하지 말자!!!");

}

}

객체와 연결되는 것이 아니므로 a가 null인데도 잘 실행된다. 위와 같은 사용법은 절대로 피해야 한다. 문법적으로 허용이 되긴 하지만 인스턴스 메쏘드라는 잘못된 인상을 주기 때문이다.

Math 클래스를 보면 모두 다 정적 메쏘드로 되어 있다. 정적 메쏘드의 쓰임을 잘 보여주는 클래스이다.

* 1. public과 private
     1. 인터페이스와 구현(interface and implimentation)

먼저 이 책에서 인터페이스라는 말은 세가지 의미로 쓰인다는 것에 주의해야 한다. 첫째로는 넓은 의미의 인터페이스로 다음 장에 나오는 contract와 거의 비슷하다고 보면 된다. 둘째로는 좁은 의미의 인터페이스로 contract중에 semantics(의미)를 제외한 문법적인 측면만을 뜻한다. 세번째로는 interface 키워드로 선언된 인터페이스 타입을 뜻한다. 여기서는 첫째의 의미로 쓰인다.

인터페이스와 구현을 나누는 것은 객체지향 프로그래밍의 핵심 중의 하나이다. 당신이 현재 사용하고 있는 컴퓨터의 인터페이스는 키보드와 모니터등으로 이루어져 있다. 컴퓨터의 구현은 CPU, 메인 보드, ... 등등으로 이루어져 있다. 인터페이스와 구현이 잘 나누어져 있으면 사용자가 구현에 대해 알 필요가 없다.

가게에 비유를 해보자. 가게에 가서 물건을 사는 방법은 여러가지가 있다. 여기서는 두 가지만 살펴보자. 하나의 방법은 가게 주인(점원)에게 “XXX 주세요”라고 요청하는 것이다. 또 다른 방법은 자신이 직접 XXX를 찾아내는 것이다. 물건을 직접 찾으려면 가게의 내부구조에 대해 잘 알아야 한다. 즉 가게의 구현에 대해 알아야 한다는 뜻이다. 가게에 물건이 배치된 구조가 그 가게의 구현이라고 할 수 있다. 여기서는 가게 주인이 인터페이스라고 할 수 있다. 만약 당신이 가게 내부 구조를 잘 안다면 직접 물건을 찾는 것이 더 빠를 것이다. 클래스도 비슷하다. 필드를 사용하는 것이 속도는 더 빠르다. 가게 주인(메쏘드)을 통해서 물건을 찾으면 더 느리다. 그러나 인터페이스를 통하면 가게의 내부구조에 대해 몰라도 된다는 장점이 있다. 이것은 물건을 사야할 가게의 수가 많고 그 가게가 클수록 점점 커다란 장점이 된다. 다른 말로 하면 객체 지향 프로그래밍은 커다란 프로그램을 작성할 때 커다란 힘을 발휘한다는 말이다. 예를 들어 1000라인 이하라면 어떻게 설계해도 별 문제가 안된다. 그러나 만약 10만 라인의 코드량이 된다면 인터페이스와 구현을 잘 구별해서 설계하지 않으면 재앙이 닥친다는 말이다. 그리고 인터페이스를 통하면 가게의 구조(클래스의 구현)가 바뀌어도 된다. 당신은 단지 카운터로 가서 주인에게 “XXX 주세요”라고만 할 줄 알면 된다. 인터페이스를 통하면 public 메쏘드의 서명(signature)와 반환형(return type) 그리고 그 메쏘드들 등에 대한 문서 정도만 보면 그 클래스를 사용할 수 있다. 즉 그 클래스의 소스 코드를 직접 볼 필요가 없다는 것이다.

클래스를 인터페이스와 구현을 나누는 데 커다란 도움이 되는 것이 접근 변경자(access modifier)이다. 접근 변경자에는 네개가 있다. public, protected, private, default(package)가 그것이다. 이 장에서는 public과 private에 대해서만 다룬다. 여기서는 인터페이스에 해당하는 것은 public변경자를 사용하고 구현에 해당하는 것은 private 변경자를 사용한다는 정도로 알아 두자. 사실 문제는 이렇게 간단하지 않다.

클래스의 사용자가 구현에 대해서 모른다는 것은 얼핏 보면 단점인것 같지만 커다란 장점이다. 클래스가 바뀐다고 해보자. 예를 들어 속도를 더빠르게 하기 위해. 클래스의 내부 구조가 바뀔 때 만약 인터페이스만 바뀌지 않는다면 클래스를 사용하는 사람이 만든 코드는 바뀌지 않아도 된다. 클래스의 세계에서는 서로를 모르면 모를수록 좋다. 그러나 그 클래스를 이용할 수 있을 정도는 알아야 한다. 그것이 바로 인터페이스이다.

* + 1. 필드는 private으로, 메쏘드는 public으로

지금까지는 예제에 지정자(modifier)를 거의 사용하지 않았다. 즉 다음과 같은 식이었다.

class Product {

static int nextSerialNumber = 1;

int serialNumber;

Product() {

serialNumber = nextSerialNumber;

nextSerialNumber++;

}

int getSerialNumber() {

return serialNumber;

}

}

만약 당신이 이렇게 public도 private도 protected도 안 붙인 코드를 본다면 그것은 둘 중에 하나를 뜻한다. 이 코드를 만든 사람은 초보이거나 고수이다. 즉 public과 private의 구분도 잘 모르는 초보이거나 default 변경자까지 알고 활용할 줄 아는 고수인 것이다. 표준 API를 보면 default접근을 사용하는 것을 많이 볼 수 있다. 아무것도 안 붙이면 default(package) 접근 수준이 된다. default 변경자에 대해서는 패키지를 다루는 부분을 참조하라.

이제는 필드 즉 데이터는 private으로 메쏘드 즉 그 클래스의 행동은 public으로 하자. 이것을 데이터 감추기(data hiding)이라고 한다.

먼저 문법적인 측면을 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.publicField = 10; // OK.

a.privateField = 20; // Error.

a.publicMethod(); // OK.

a.privateMethod(); // Error.

}

}

class A {

public int publicField;

private int privateField;

public void publicMethod() {

privateMethod(); // OK.

publicField = 10; // OK.

}

private void privateMethod() {

}

}

두 군데서 컴파일타임 오류가 날 것이다. public으로 지정되면 그 필드 또는 메쏘드를 어디에서도 쓸 수 있다. private으로 지정되면 그 필드 또는 메쏘드를 그것이 정의된 클래스 내에서만 쓸 수 있다.

이번에는 자동차를 예로 들어 public과 private 즉 인터페이스와 구현을 나누어 보자. 다음 예제를 보자. 자동차는 단순화되어 단 하나의 속성 즉 현재 속도 밖에 없다.

public class Example {

public static void main(String[] args) {

Car c = new Car();

c.setSpeed(103.5);

System.out.println("speed = " + c.getSpeed() + "km");

}

}

class Car {

private double speed = 0;

public void setSpeed(double d) {

speed = d;

}

public double getSpeed() {

return speed;

}

}

결과는 다음과 같다.

speed = 103.5km

speed필드를 private으로 만들어서 감추었다. 그러나 이것은 좋은 설계가 못된다. 사람들이 자동자를 운전할 때에는 속도를 지정해 주지 않는다. 사람들이 하는 것은 가속기를 밟거나 브레이크를 밟는 것이다. 그리고 계기판을 본다. 좀더 그럴 듯한 설계는 다음과 같을 것이다.

public class Example {

public static void main(String[] args) {

Car c = new Car();

c.accelerate(10.5);

c.showSpeed();

c.accelerate(23.2);

c.showSpeed();

c.putOnBrake(30.6);

c.showSpeed();

}

}

class Car {

private double speed = 0;

public void accelerate(double d) { // 가속

speed += d;

}

public void putOnBrake(double d) { // 감속

speed -= d;

if (speed < 0)

speed = 0;

}

public void showSpeed() { // 계기판

System.out.println("speed = " + speed + "km");

}

}

결과는 다음과 비슷할 것이다.

speed = 10.5km

speed = 33.7km

speed = 3.1000000000000014km

* + 1. 데이터를 감추어야 하는 이유

다음 예제는 필드를 public으로 선언했다. 여기서 Example 클래스는 당신이 아닌 다른 사람이 만든다고 생각해 보자. 정확히 하자면 패키지도 달라야 하고 PhoneBook, NameAndNumber 클래스가 public으로 선언되어야 하겠지만 여기서는 무시하기로 한다.

public class Example { // 다른 사람이 만든 클래스

public static void main(String[] args) {

PhoneBook pb = new PhoneBook();

pb.add("화재신고", "119");

pb.add("우리집", "000-0000");

pb.add("욕쟁이", "18-1818");

System.out.print(pb.list[0].name + " : ");

System.out.println(pb.list[0].number);

pb.list[0].number = "114"; // 객체의 상태를 잘못된 상태로 만들 수 있다.

System.out.println("화재신고 : " + pb.getNumber("화재신고"));

}

}

class PhoneBook { // 당신이 만든 클래스

public NameAndNumber[] list;

public int nextIndex;

public PhoneBook() {

list = new NameAndNumber[1000];

nextIndex = 0;

}

public void add(String name, String number) {

list[nextIndex] = new NameAndNumber(name, number);

nextIndex++;

}

public String getNumber(String name) {

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

NameAndNumber nanInList = list[i];

String nameInList = nanInList.name;

if (nameInList.equals(name)) {

return nanInList.number;

}

}

return "찾는 이름이 없습니다.";

}

}

class NameAndNumber { // 당신이 만든 클래스

public String name; // 너무 단순하기 때문에 그냥 public으로 했다.

public String number; // 너무 단순하기 때문에 그냥 public으로 했다.

public NameAndNumber(String name, String number) {

this.name = name;

this.number = number;

}

}

결과는 다음과 같을 것이다.

화재신고 : 119

화재신고 : 114

필드를 마음대로 건드릴 수 있게 하니까 객체의 상태가 깨지게 할 수도 있다. 이것이 필드를 private으로 선언해야 하는 하나의 이유이다. 만약 private으로 선언하게 하고 public 메쏘드만 이용하게 했다면 객체는 계속 정상적인 상태를 유지할 것이다. 다른 사람이 객체의 내부를 함부로 건드리지 못하기 때문이다.

여기서는 배열을 이용했는데 배열은 한번 만들어지면 크기가 바뀔 수 없다. 그것을 좀더 유연성이 있는 ArrayList 클래스로 바꾸고 싶다고 하자. 다음과 같이 바꿔보자.

import java.util.\*;

public class Example { // 다른 사람이 만든 클래스

public static void main(String[] args) {

PhoneBook pb = new PhoneBook();

pb.add("화재신고", "119");

pb.add("우리집", "000-0000");

pb.add("욕쟁이", "18-1818");

NameAndNumber nan = (NameAndNumber)pb.list.get(0); // 여기가 바뀌어야 했다.

System.out.print(nan.name + " : "); // 여기가 바뀌어야 했다.

System.out.println(nan.number); // 여기가 바뀌어야 했다.

nan.number = "114"; // 여기가 바뀌어야 했다.

System.out.println("화재신고 : " + pb.getNumber("화재신고"));

}

}

class PhoneBook { // 당신이 만든 클래스

public ArrayList list;

public int nextIndex;

public PhoneBook() {

list = new ArrayList();

}

public void add(String name, String number) {

list.add( new NameAndNumber(name, number));

}

public String getNumber(String name) {

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

NameAndNumber nanInList = (NameAndNumber)list.get(i);

String nameInList = nanInList.name;

if (nameInList.equals(name)) {

return nanInList.number;

}

}

return "찾는 이름이 없습니다.";

}

}

class NameAndNumber { // 당신이 만든 클래스

public String name; // 너무 단순하기 때문에 그냥 public으로 했다.

public String number; // 너무 단순하기 때문에 그냥 public으로 했다.

public NameAndNumber(String name, String number) {

this.name = name;

this.number = number;

}

}

결과는 다음과 같을 것이다.

화재신고 : 119

화재신고 : 114

당신의 클래스의 구현을 배열에서 ArrayList로 바꾸자 다른 사람이 만든 Example 클래스도 코드가 바뀌어야 했다. 만약 필드가 private였다면 어떠했을까? 다음을 보자. 먼저 배열을 이용한 것이다.

public class Example { // 다른 사람이 만든 클래스

public static void main(String[] args) {

PhoneBook pb = new PhoneBook();

pb.add("화재신고", "119");

pb.add("우리집", "000-0000");

pb.add("욕쟁이", "18-1818");

System.out.println("화재신고 : " + pb.getNumber("화재신고"));

}

}

class PhoneBook { // 당신이 만든 클래스

private NameAndNumber[] list;

private int nextIndex;

public PhoneBook() {

list = new NameAndNumber[1000];

nextIndex = 0;

}

public void add(String name, String number) {

list[nextIndex] = new NameAndNumber(name, number);

nextIndex++;

}

public String getNumber(String name) {

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

NameAndNumber nanInList = list[i];

String nameInList = nanInList.name;

if (nameInList.equals(name)) {

return nanInList.number;

}

}

return "찾는 이름이 없습니다.";

}

}

class NameAndNumber { // 당신이 만든 클래스

public String name; // 너무 단순하기 때문에 그냥 public으로 했다.

public String number; // 너무 단순하기 때문에 그냥 public으로 했다.

public NameAndNumber(String name, String number) {

this.name = name;

this.number = number;

}

}

결과는 다음과 같을 것이다.

화재신고 : 119

이번에는 ArrayList를 이용한 것으로 바꾸어보자.

import java.util.\*;

public class Example { // 다른 사람이 만든 클래스

public static void main(String[] args) {

PhoneBook pb = new PhoneBook();

pb.add("화재신고", "119");

pb.add("우리집", "000-0000");

pb.add("욕쟁이", "18-1818");

System.out.println("화재신고 : " + pb.getNumber("화재신고"));

}

}

class PhoneBook { // 당신이 만든 클래스

public ArrayList list;

public int nextIndex;

public PhoneBook() {

list = new ArrayList();

}

public void add(String name, String number) {

list.add( new NameAndNumber(name, number));

}

public String getNumber(String name) {

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

NameAndNumber nanInList = (NameAndNumber)list.get(i);

String nameInList = nanInList.name;

if (nameInList.equals(name)) {

return nanInList.number;

}

}

return "찾는 이름이 없습니다.";

}

}

class NameAndNumber { // 당신이 만든 클래스

public String name; // 너무 단순하기 때문에 그냥 public으로 했다.

public String number; // 너무 단순하기 때문에 그냥 public으로 했다.

public NameAndNumber(String name, String number) {

this.name = name;

this.number = number;

}

}

결과는 다음과 같을 것이다.

화재신고 : 119

여기서 중요한 것은 클래스의 내부구조를 배열에서 ArrayList로 바꾸었음에도 Example 클래스는 전혀 바뀌지 않고도 잘 작동한다는 것이다. 즉 당신이 만들 클래스의 구현이 바뀌어도 다른 사람이 만든 클래스를 그대로 재사용할 수 있는 것이다.

* + 1. 필드를 public으로 해야 할 때가 있다.

필드는 private으로 하는 것이 거의 원칙이지만(protected와 package접근에 대해서는 나중에 생각하자) 상수일 때는 대부분은 public으로 하는 것이 좋다. 필드를 private으로 하는 가장 큰 이유는 필드를 잘못 건드려서 엉뚱한 값이 되는 것을 방지하기 위해서이다. 상수는 변화될 수 없기 때문에 엉뚱한 값이 될 수도 없다. 필드를 private으로 하면 메쏘드를 거쳐서 사용해야 하기 때문에 불편하고 실행 속도도 느리다. 상수가 아닌 필드는 이 불편을 감수하고라도 데이터를 숨기는 것이 좋지만 상수까지 그럴 필요는 없다.

java.awt.Point 클래스를 보면 필드인 x, y가 public으로 선언되어 있다. 이런 클래스는 C의 구조체(structure)와 비슷하다. 데이터를 복잡하게 처리하는 클래스가 아니라 단지 데이터를 모아 놓은 것에 불과하다는 얘기다. 이럴 때는 데이터를 감추는 것이 별로 도움이 안되기 때문에 필드를 감추지 않는다.

* + 1. 메쏘드도 private으로 해야 할 때가 있다.

필드는 대부분 private으로 하는 것이 원칙이다. 그러나 대부분의 경우에 메쏘드를 public으로 해야 한다고 생각하면 그건 잘못된 생각이다. 이것은 초보자에게나 잠깐 적용될 수 있는 얘기다. 메쏘드도 인터페이스(객체 지향 용어)를 구성하는 메쏘드인지 아니면 단지 구현에 불과한지를 가려서 구현에 불과하면 private으로 해야 한다(필드와 마찬가지로 protected와 package에 대해서는 나중에 생각하자). 초보자에게는 약간 헷갈리는 얘기이지만 필드나 메쏘드나 되도록이면 숨기는 것이 좋다. 가능하면 private으로 하는 것이 좋다. 둘의 차이가 있다면 필드를 더더욱 숨겨야 한다는 것이다.

* + 1. 접근 변경자는 왜 있어야만 하나?

혹시 어떤 사람들이 이런 의문을 가질지도 모르겠다.

“인터페이스와 구현을 나누는 것은 좋다고 생각한다. 그런데 왜 접근 변경자가 필요한가? 그냥 구현에 해당하는 메쏘드나 필드를 다른 클래스에서 안쓰면 될 것이 아닌가?”

어느 정도는 맞는 말이다. 만약 자바에서 public, private등의 키워드를 제공하지 않았더라 하더라도 객체 지향 프로그래밍을 하지 못하는 것은 아니다. 클래스를 만드는 프로그래머는 인터페이스(객체지향 용어)에 해당하는 멤버를 클래스의 사용자를 위해서 문서화하고, 구현에 해당하는 멤버를 클래스의 유지보수를 하는 사람들 위해 문서화하면 된다. 그리고 클래스의 사용자는 구현에 해당하는 멤버에 대해 알고 있다 하더라도 안 쓰면 된다.

이것에 대해서는 난간에 비유하고 싶다. 고층 아파트의 베렌다에는 난간이 있다. 만약 난간이 없다고 생각해 보자. 그래도 베렌다에서 밖으로 벗어나지만 않는다면 안 떨어진다. 하지만 난간이 있는 것이 더 안전하다. 사람들은 실수를 하게 마련이다. private등의 키워드는 그런 실수를 컴파일 타임에 수정할 수 있게 해 준다. 난간이 있으면 안심하고 난간에 기댈 수도 있다.

* 1. 클래스의 멤버

필드, 메쏘드, 멤버 클래스, 멤버 인터페이스를 클래스의 멤버라고 한다. 멤버 클래스와 멤버 인터페이스에 대해서는 중첩 타입을 다루는 장을 참조하라. 생성자는 멤버가 아니다.

멤버는 상속이 된다. 생성자는 멤버가 아니기 때문에 상속이 되지 않는다. 상속에 대해서는 클래스의 상속을 다루는 장을 참조하라. 다음 예제를 보자.

class A {

A() { // 생성자는 멤버가 아니다.

}

int intValue; // 필드

int method() { // 메쏘드

}

class MemberClass { // 멤버 클래스

}

interface MemberInterface { // 멤버 인터페이스

}

}

1. 클래스의 상속
   1. 클래스 상속하기
      1. 수퍼 클래스(superclass)와 서브 클래스(subclass)

어떤 클래스를 상속하고 싶으면 다음과 같이 하면 된다.

class SubClass extends SuperClass {

}

위의 예에서 SubClass가 SuperClass를 상속(inherit, extend)하게 된다. 이 때 SuperClass는 SubClass의 바로 위의 수퍼 클래스(direct superclass)이다. 마찬가지로 SubClass는 SuperClass의 바로 아래 서브 클래스(direct subclass)이다.

어떤 클래스의 수퍼 클래스의 수퍼 클래스도 그 클래스의 수퍼 클래스이다. 마찬가지로 서브 클래스의 서브 클래스는 서브 클래스이다. 이 말은 상속된 클래스가 계속 상속된다는 것을 뜻한다. 자손의 자손도 자손이듯이.

실제로 상속에 상속을 계속하는 것은 많이 쓰인다. 다음 예제를 보자.

class MyFrame extends JFrame {

}

MyFrame 클래스가 javax.swing.JFrame클래스를 상속받았다. 그러나 이것이 끝이 아니다.

javax.swing.JFrame 클래스는 java.awt.Frame 클래스를 상속받았다.

java.awt.Frame 클래스는 java.awt.Window 클래스를 상속받았다.

java.awt.Window 클래스는 java.awt.Container 클래스를 상속받았다.

java.awt.Container 클래스는 java.awt.Component 클래스를 상속받았다.

java.awt.Component 클래스는 java.lang.Object 클래스를 상속받았다.

하나의 클래스를 여러 개의 클래스가 상속받을 수 있다. 예를 들어 javax.swing.JComponent를 상속받은 클래스에는 AbstractButton, BasicInternalFrameTitlePane, Box, Box.Filler, JColorChooser, JComboBox, JFileChooser, JInternalFrame, JInternalFrame.JDesktopIcon, JLabel, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollBar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, JViewport 등 이 있다. 그리고 모든 클래스는 Object 클래스를 상속받는다.

* + 1. 단일 상속(single inheritance)

하나의 클래스는 여러 개의 클래스를 상속받을 수 없다. 즉 다음 예제는 컴파일되지 않는다.

public class Example extends SuperClassA, SuperClassB {}

class SuperClassA {}

class SuperClassB {}

자바에서는 클래스의 다중상속을 지원하지 않는 것이다. 다중상속을 지원하느냐 안하느냐는 논란이 될 수 있다. 다중 상속은 위험한 도구이다. 그렇기 때문에 잘 사용하면 유용할 수 있지만 잘못 사용하면 재앙이 될 수 있다. 자바의 철학이 단순성이기 때문에 이것을 없애기로 한 것이다. 그러나 자바가 다중 상속 자체를 금지한 것은 아니다. 좀 어려운 말로 하면 구현(implementation)의 다중상속은 금지했지만 인터페이스(interface, 여기서는 자바 키워드가 아니라 객체 지향 용어)의 다중 상속은 허용한 것이다. 쉬운 말로 하면 클래스(class)의 다중 상속은 금지했지만 인터페이스(interface, 여기서는 자바 키워드)의 다중 상속은 허용했다는 말이다. 이것의 의미는 인터페이스를 익히고 나서야 감이 잡힐 것이다.

결론적으로 클래스는 항상 단 하나의 클래스를 상속해야 한다. 여기에는 단 하나의 예외가 있는데 그것은 Object 클래스이다. Object 클래스는 모든 클래스의 조상이며 자신은 어떤 클래스도 상속하지 않는다.

겉으로서는 어떤 클래스도 상속받지 않는 듯한 클래스가 있으나 사실 Object 클래스를 상속 받는 것이다. 다음 두 선언은 동일하다.

class A {

}

class A extends Object {

}

* + 1. final 클래스

상속은 객체지향 프로그래밍의 강력한 도구이다. 그러나 때에 따라서는 이것을 사용하지 않는 것이 더 나을 때가 있다. 그럴 때 사용하는 것이 final 클래스이다. final로 선언된 클래스는 상속될 수 없다. java.lang.Math 클래스가 바로 그런 클래스이다. 다음은 Math.java에서 인용한 것이다.

public final strictfp class Math {

...

}

final로 선언되어있다. 한번 Math 클래스를 상속받아보자.

public class Example extends Math {

}

컴파일 에러가 난다. Math 클래스에 정의되어 있는 사인 함수를 구하는 sin 메쏘드나 절대값을 구하는 abs 메쏘드를 재정의(overriding)하는 것은 별로 바람직한 일이 아니다. 그래서 아예 Math 클래스를 상속하는 것을 금지한 것이다.

* 1. 멤버의 상속
     1. 필드와 메쏘드의 상속

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass sup = new SuperClass();

sup.i = 111;

System.out.println("sup.getI() = " + sup.getI());

SubClass sub = new SubClass();

sub.i = 222;

System.out.println("sub.getI() = " + sub.getI());

}

}

class SuperClass {

public int i;

public int getI() {

return i;

}

}

class SubClass extends SuperClass {

}

결과는 다음과 같다.

sup.getI() = 111

sub.getI() = 222

여기서 주목할 곳은 다음 두 곳이다.

sub.i = 222;

System.out.println("sub.getI() = " + sub.getI());

SubClass에는 i와 getI()가 선언되지 않았음에도 사용할 수 있는 것이다. 이것은 SuperClass.i와 SuperClass.getI()를 상속받았기 때문이다. 필드와 메쏘드는 클래스의 멤버이며 상속된다. 다음 예도 상속의 예를 보여준다.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

sub.setValue(222);

System.out.println("sub.getValue() = " + sub.getValue());

}

}

class SuperClass {

public int value;

}

class SubClass extends SuperClass {

public void setValue(int i) {

this.value = i; // value를 상속 받았기 때문에 여기서 쓸 수 있다.

}

public int getValue() {

return value; // value를 상속 받았기 때문에 여기서 쓸 수 있다.

}

}

결과는 다음과 같다.

sub.getValue() = 222

* + 1. public 멤버

위에서 보았듯이 public으로 선언된 멤버는 당연히 상속이 된다.

* + 1. private 멤버

그러나 위의 예에서 필드를 public으로 선언했다. 그것은 좋은 방법이 아니다. 필드를 private으로 바꾸어 보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

sub.setValue(222);

System.out.println("sub.getValue() = " + sub.getValue());

}

}

class SuperClass {

private int value; // private으로.

}

class SubClass extends SuperClass {

public void setValue(int i) {

this.value = i; // Error. private 멤버는 상속되지 않는다.

}

public int getValue() {

return value; // Error. private 멤버는 상속되지 않는다.

}

}

컴파일이 되지 않을 것이다. private 멤버는 그 클래스에서만 사용할 수 있다. 서브 클래스일지라도 사용할 수 없다. 즉 private 멤버는 상속되지 않는다.

* + 1. protected 멤버

public으로 하자니 정보 감추기(information hiding, encapsulation)가 깨지고 private으로 하자니 상속이 안된다. 이것을 해결해 주는 것이 protected 키워드이다. protected로 선언된 멤버는 다른 클래스에서는 접근이 안되지만 서브 클래스에서는 접근할 수 있다(사실 protected의 정확한 의미는 좀 복잡하다. 자세한 것은 package를 다룬 장과 클래스와 인터페이스 – 못다한 이야기를 다룬 장을 참조하라). 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

sub.setValue(222);

System.out.println("sub.getValue() = " + sub.getValue());

}

}

class SuperClass {

protected int value; // protected로.

}

class SubClass extends SuperClass {

public void setValue(int i) {

this.value = i; // protected 멤버는 상속이 된다.

}

public int getValue() {

return value; // protected 멤버는 상속이 된다.

}

}

결과는 다음과 같다.

sub.getValue() = 222

다음 예에서 볼 수 있듯이 protected 메쏘드를 서브 클래스가 아닌 다른 클래스에서 사용할 수 없다. 예제가 좀 복잡해졌다. 예제는 세 개의 파일로 이루어졌다. Example.java, SuperClass.java, SubClass.java가 그것이다. 그리고 명시적인 패키지를 사용했다. 디렉토리 구조를 패키지에 맞추어 주어야 한다. 실행이 안되거나 이해가 안되는 사람은 패키지를 다룬 장을 참조하라.

Example.java

package bababook.java\_language.example;

import bababook.java\_language.super\_sub.\*;

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

sub.value = 222; // Error. protected는 여기서 접근할 수 없다.

System.out.println("sub.getValue() = " + sub.getValue());

}

}

SuperClass.java

package bababook.java\_language.super\_sub;

public class SuperClass {

protected int value;

}

SubClass.java

package bababook.java\_language.super\_sub;

public class SubClass extends SuperClass {

public void setValue(int i) {

this.value = i;

}

public int getValue() {

return value;

}

}

컴파일이 안될 것이다.

* + 1. 생성자는 상속되지 않는다

생성자는 멤버가 아니므로 상속되지 않는다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass(3, 5); // Error.

}

}

class SuperClass {

protected int x, y;

public SuperClass() {

}

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

}

필드와 메쏘드와는 달리 생성자는 상속이 안된다는 것을 알 수 있을 것이다. 그러므로 당연히 재정의되지도 않는다.

* + 1. 생성자를 부르는 super

위의 예제를 실행가능하게 하려면 다음과 같이 하면 된다. 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass(3, 5);

}

}

class SuperClass {

protected int x, y;

public SuperClass() {

}

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

public SubClass(int x, int y) { // 생성자를 만들어 주었다.

this.x = x;

this.y = y;

}

}

컴파일이 잘 되는 것을 알 수있다. 그러나 다음과 같이 super 키워드를 이용하는 것이 더 좋다.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass(3, 5);

}

}

class SuperClass {

protected int x, y;

public SuperClass() {

}

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

public SubClass() {

}

public SubClass(int x, int y) {

super(x, y);

}

}

여기서 super(x, y)는 SuperClass(int, int) 생성자를 불러 준다. 생성자를 부르는 super는 생성자를 부르는 this와 마찬가지로 생성자의 맨 위에 와야 한다. 그러므로 생성자를 부르는 super와 생성자를 부르는 this는 공존할 수 없다. 생성자의 맨 위는 한 곳 뿐이기 때문이다. 그러므로 다음 예제는 컴파일되지 않는다.

class SuperClass {

protected int x, y;

public SuperClass() {

}

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

public SubClass() {

}

public SubClass(int x, int y) {

int a = 0;

super(x, y); // 오류. 맨 위에 와야 한다.

}

}

다음 예제를 컴파일해 보자.

class SuperClass {

protected int x, y;

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

}

컴파일타임 오류가 날 것이다. 이제부터 그 이유를 알아보자. 생성자가 없으면 디폴트 생성자(default constructor)를 만들어 준다. 그러므로 위의 예제는 아래와 같다.

class SuperClass {

protected int x, y;

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

SubClass() {

}

}

이 예제도 컴파일이 안될 것이다. 만약 생성자에 다른 생성자를 부르는 this도 없고 수퍼 클래스의 생성자를 부르는 super도 없다면 맨 위에 super();가 자동으로 붙는다. 일단 정말 그런지 알아보자. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

new SubClass();

System.out.println();

new SubClass(2, 5);

}

}

class SuperClass {

protected int x, y;

public SuperClass() {

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

}

}

class SubClass extends SuperClass {

public SubClass() {

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

}

public SubClass(int x, int y) {

System.out.println("SubClass.SubClass(int, int)");

this.x = x;

this.y = y;

}

}

결과는 다음과 같다.

SuperClass.SuperClass()

SubClass.SubClass()

SuperClass.SuperClass()

SubClass.SubClass(int, int)

이것은 즉 다음 예제와 같은 것이다.

public class Example {

public static void main(String[] args) {

new SubClass();

System.out.println();

new SubClass(2, 5);

}

}

class SuperClass {

protected int x, y;

public SuperClass() {

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

}

}

class SubClass extends SuperClass {

public SubClass() {

super();

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

}

public SubClass(int x, int y) {

super();

System.out.println("SubClass.SubClass(int, int)");

this.x = x;

this.y = y;

}

}

결과는 위와 같다.

그러므로 위에서 컴파일타임 오류가 난 예제는 결국 아래의 예제와 같은 것이다.

class SuperClass {

protected int x, y;

public SuperClass(int x, int y) {

this.x = x;

this.y = y;

}

}

class SubClass extends SuperClass {

SubClass() {

super();

}

}

마찬가지로 컴파일타임 오류가 날 것이다. 위의 예제에서 super()가 찾는 것은 SuperClass의 SuperClass() 생성자인데 그것이 없는 것이다. 그런데 위의 예제를 다음과 같이 바꾸면 컴파일이 잘 될 것이다.

class SuperClass {

}

class SubClass extends SuperClass {

SubClass() {

super();

}

}

왜냐하면 SuperClass에 생성자가 명시적으로 정의되지 않았기 때문에 디폴트 생성자가 자동으로 만들어지기 때문이다. 위의 예제는 다음과 같다.

class SuperClass {

SuperClass() {

}

}

class SubClass extends SuperClass {

SubClass() {

super();

}

}

생성자를 부르는 super를 애용하도록 하자. 그 말은 무슨 뜻인가 하면 수퍼 클래스의 필드는 수퍼 클래스가 초기화하도록 하자는 말이다. 다른 말로 하면 남의 일에 간섭하지 말고 제 일만 하는 것이 좋다는 것이다. 그것이 조상의 일일지라고. 수퍼 클래스의 필드는 수퍼 클래스에서 초기화하고 서브 클래스의 필드는 서브 클래스에서 초기화하는 것이 좋다. 다음 예제를 보자.

class Point2D {

protected int x;

protected int y;

public Point2D(int x, int y) {

this.x = x;

this.y = y;

}

public Point2D() {

x = 0;

y = 0;

}

}

class Point3D extends Point2D {

protected int z;

public Point3D(int x, int y, int z) {

this.x = x;

this.y = y;

this.z = z;

}

public Point3D() {

x = 0;

y = 0;

z = 0;

}

}

여기서는 서브 클래스가 수퍼 클래스의 필드까지 책임을 지고 있다. 그것보다는 다음과 같이 바꾸는 것이 더 낫다.

class Point2D {

protected int x;

protected int y;

public Point2D(int x, int y) {

this.x = x;

this.y = y;

}

public Point2D() {

x = 0;

y = 0;

}

}

class Point3D extends Point2D {

protected int z;

public Point3D(int x, int y, int z) {

super(x, y);

this.z = z;

}

public Point3D() {

super();

z = 0;

}

}

그리고 가능하다면 다음과 같이 바꾸는 것이 좋다.

class Point2D {

private int x;

private int y;

public Point2D(int x, int y) {

this.x = x;

this.y = y;

}

public Point2D() {

x = 0;

y = 0;

}

}

class Point3D extends Point2D {

private int z;

public Point3D(int x, int y, int z) {

super(x, y);

this.z = z;

}

public Point3D() {

super();

z = 0;

}

}

필드를 private으로 바꾸고 protected나 public 메쏘드로 접근하게 하는 것이다. 여기서 예로 든 것은 너무나 단순하기 때문에 사실 이렇게 하는 것보다 필드를 public으로 하는 것이 나을지도 모른다. 사실 java.awt.Point의 필드는 public으로 선언되어 있다. 그러나 실제 프로그래밍에서는 수퍼 클래스도 나름대로의 contract를 가진 복잡한 클래스인 경우가 많다.

* 1. 메쏘드의 재정의(overriding)
     1. 기초

이전에 중복정의(overloading) 에 대해 배웠다. 메쏘드의 이름이 같지만 매개변수의 개수나 타입을 다르게 정의해 준 것이었다. 이번에는 메쏘드의 이름도 같고 매개변수의 개수나 타입도 같은 것을 만들어 보자. 다음 예제를 보자.

public class Example {

public void method(int a) {

a = 0;

}

public void method(int b) {

b = 3;

}

}

컴파일타임 오류가 난다. 이것은 당연하다. 예제를 다음과 같이 고쳐보면 그 이유를 분명히 알 수 있을 것이다.

public class Example {

public void methodCaller() {

method(123); // 어떤 메쏘드를 부를 것인가?

}

public void method(int a) {

a = 0;

}

public void method(int b) {

b = 3;

}

}

어떤 메쏘드를 부를지 결정할 수가 없기 때문이다. 매개 변수의 이름이 다르다고(a, b)해도 전혀 도움이 안된다. 그러나 상속과 연결해서는 이런 방식이 통한다. 다음 예제를 보자.

class SuperClass {

public void method() {

}

}

class SubClass extends SuperClass {

public void method() {

}

}

컴파일이 잘 된다. 이런 것을 재정의(overriding) 이라고 한다. 그렇다면 두 개(여러개가 될 수도 있다)의 메쏘드 중 하나를 선택해야 하는데 어떻게 선택할까? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass sup = new SuperClass();

sup.method();

SubClass sub = new SubClass();

sub.method();

sup = sub;

sup.method(); // 여기에 주목하라.

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같다.

SuperClass.method()

SubClass.method()

SubClass.method()

그 메쏘드가 불리는 객체가 어떤 객체냐에 따라 결정된다.

sup.method(); // 여기에 주목하라.

에서 볼 수 있듯이 객체를 가리키는 변수(sub)가 SuperClass형이라도 실제 변수가 가리키는 객체가 SubClass라면 SubClass.method()가 불리는 것을 알 수 있다.

* + 1. 동적 다형성(dynamic polymorphism)

중복정의된 메쏘드 중에 어떤 메쏘드를 부를지는 컴파일러가 결정할 수 있다. 즉 컴파일타임에 결정되는 것이다. 그러나 재정의된 메쏘드 중 어떤 클래스의 메쏘드를 부를지는 컴파일 타임에 결정될 수 없다. 위의 예제는 영리한 컴파일러라면

sup.method(); // 여기에 주목하라.

도 SubClass.method()를 불러야 한다는 것을 알 수 있을 것이다. 그러나 다음 예제는 아무리 천재 컴파일러라도 어떤 메쏘드를 불러야 할지 결정할 수 없다. 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

SuperClass sup;

Random rdm = new Random();

for (int i = 0; i < 10; i++) {

if (rdm.nextInt() % 2 == 0)

sup = new SuperClass();

else

sup = new SubClass();

sup.method();

}

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 비슷할 것이다.

SuperClass.method()

SubClass.method()

SuperClass.method()

SuperClass.method()

SubClass.method()

SubClass.method()

SuperClass.method()

SuperClass.method()

SubClass.method()

SubClass.method()

여기서 java.util.Random의 정확한 사용법은 중요하지 않다. 중요한 것은 Random 클래스의 생성자의 실행이 현재 시각에 의존한다는 것이고 컴파일러는 이것이 실행될 시각을 결코 알 수 없다는 것이다. 그리고 안다고 쳐도 for문에서 반복하는 것은 어떻게 할 것인가? 위의 예제는 재정의된 메쏘드의 선택이 동적으로 즉 컴파일 타임이 아니라 런타임에 결정된다는 것을 잘 보여준다. 위의 예제는 문법을 설명하는데는 유용하지면 별로 쓸모 있는 예제는 아니다. 그러면 이번에는 동적 다형성을 이용한 좀더 쓸모 있는 예제를 보자.

public class Example {

public static void main(String[] args) {

군인 군인들[] = new 군인[7];

군인들[0] = new 군인();

군인들[1] = new 장교();

군인들[2] = new 사병();

군인들[3] = new 병장();

군인들[4] = new 상병();

군인들[5] = new 일병();

군인들[6] = new 이병();

열받는다(군인들);

}

public static void 열받는다(군인[] 군인들) {

for (int i = 0; i < 군인들.length; i++)

군인들[i].열받는다();

}

}

class 군인 {

public void 열받는다() {

System.out.println("군인은 열받아도 참는다.");

}

}

class 장교 extends 군인 {

public void 열받는다() {

System.out.println("사병을 갈군다.");

}

}

class 사병 extends 군인 {

}

class 병장 extends 사병 {

public void 열받는다() {

System.out.println("상병을 갈군다.");

}

}

class 상병 extends 사병 {

public void 열받는다() {

System.out.println("일병을 갈군다.");

}

}

class 일병 extends 사병 {

public void 열받는다() {

System.out.println("이병을 갈군다.");

}

}

class 이병 extends 사병 {

public void 열받는다() {

System.out.println("탈영한다.");

}

}

결과는 다음과 같다.

군인은 열받아도 참는다.

사병을 갈군다.

군인은 열받아도 참는다.

상병을 갈군다.

일병을 갈군다.

이병을 갈군다.

탈영한다.

Example.열받는다(군인[]) 메쏘드에 주목하자. 수퍼 클래스의 배열을 받지만 그 배열에 들어있는 실제 객체(런타임 객체)에 따라 재정의된 메쏘드를 알아서 불러준다. (식별자를 한글로 쓰는 것은 권할 만한 일은 아니다.)

* + 1. 메쏘드를 부르는 super

때에 따라 재정의된 메쏘드에서 수퍼 클래스의 메쏘드를 실행할 필요가 있을 때가 있다. 다음 예제는 그 사용법을 보여준다.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

sub.method();

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

super.method();

}

}

결과는 다음과 같다.

SubClass.method()

SuperClass.method()

* + 1. 필드는 재정의되지 않는다

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass sup;

SubClass sub;

sub = new SubClass();

sup = sub;

sub.value = 1234;

System.out.println("sup.value = " + sup.value);

System.out.println("sub.value = " + sub.value);

sup = new SubClass();

sub = (SubClass)sup;

sup.value = 1234;

System.out.println("sup.value = " + sup.value);

System.out.println("sub.value = " + sub.value);

}

}

class SuperClass {

public int value = 111;

}

class SubClass extends SuperClass {

public int value = 222;

}

결과는 다음과 같다.

sup.value = 111

sub.value = 1234

sup.value = 1234

sub.value = 222

위의 예제 중

sup.value = 1234;

를 실행하자 SuperClass의 필드가 바뀌었다. 같은 이름의 메쏘드 중 어떤 메쏘드를 부를 것인가는 런타임의 실제 객체의 형(즉 런타임 형, run-time type)이 결정하는 반면 같은 이름의 필드 중 어떤 필드에 접근할 것인가는 변수의 형(즉 컴파일타임 형, compile-time type)에 의해 결정되는 것을 알 수 있다. 메쏘드 재정의는 유용한 기법인 반면 필드에 같은 이름을 쓰는 것은 아주 안좋은 생각이다. 하나의 기능을 여러가지 방식으로 구현하는 것은 유용하다. 그러나 하나의 정보가 두개의 클래스에 나뉘어 저장되는 것은 안좋은 것이다. 또는 서로 다른 정보임에도 같은 이름을 붙이는 것도 마찬가지로 안좋다.

* + 1. 정적 메쏘드는 재정의되지 않는다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass.method();

SubClass.method();

SuperClass sup = new SuperClass();

sup.method();

sup = new SubClass();

sup.method(); // 여기에 주목하라.

}

}

class SuperClass {

public static void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public static void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같다.

SuperClass.method()

SubClass.method()

SuperClass.method()

SuperClass.method()

정적 메쏘드는 동적으로 바인딩(dynamic binding, runtime binding)되지 않는다. 즉 정적으로 바인딩(static binding, compile time binding)된다. 다른 말로 하면 어떤 클래스의 메쏘드를 사용할지가 컴파일타임에 결정된다는 말이다. 정적(static) 메쏘드는 정적(static)으로 바인딩된다. 여기서 sup변수를 이용해서 정적 메쏘드를 불렀는데 정적 메쏘드에서는 중요한 것이 sup가 가리키는 객체가 아니라 sup 자체의 타입이다. 즉 컴파일 타임에 알 수 있는 sup의 타입이다. 게다가

sup.method();

에서처럼 정적 메쏘드를 객체 변수를 이용해 부르는 것은 가능하긴 하지만 좋은 방법은 아니다. 이것은 인스턴스 메쏘드라는 인상을 준다.

* + 1. final 메쏘드

때에 따라서는 메쏘드의 재정의를 불가능하게 하는 것이 필요할 때도 있다. 그럴 때 final을 사용한다. final 메쏘드는 재정의될 수 없다. 그러므로 다음 예제는 컴파일되지 않는다.

class SuperClass {

public final void method() {

}

}

class SubClass extends SuperClass {

public void method() {

}

}

이 때 주의할 것이 있다. 만약 final 메쏘드에서 private이 아닌non-final 메쏘드를 부른다면 모든 것이 수포로 돌아갈 수 있다는 것이다. 그 non-final 메쏘드가 재정의됨으로써 결국은 final 메쏘드의 구현이 바뀌게 되는 것이다. 또한 protected 필드나 public 필드가 있다면 그것을 건드림으로써 final 메쏘드의 구현을 바꿀 수 있으므로 조심해야 한다.

또한 어떤 메쏘드를 final로 선언할 때는 신중해야 한다. 재정의를 할 수 없다는 것은 객체 지향 프로그래밍에 심각한 제한을 가하는 것이기 때문이다. 서브 클래스는 더 속도가 빠른 구현등으로 바뀔 수 없다. 그러므로 보안상의 문제등이 있을 경우등에만 사용하는 것이 좋다.

클래스를 final로 선언하는 것은 더욱 커다란 제한이다. 아예 상속을 할 수 없기 때문이다. 만약 모든 메쏘드가 보안상 바뀌면 안된다면 클래스를 final로 선언하지 말고 모든 메쏘드를 final로 선언하는 방법도 있다. 그러면 서브 클래스에서 기능을 추가할 수는 있지만 이미 있는 기능의 구현을 바꿀 수는 없다.

* + 1. 수퍼 클래스에서 서브 클래스의 메쏘드 부르기

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass sup = new SubClass();

SubClass sub = new SubClass();

sup.method();

sub.method();

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같다.

SubClass.method()

SubClass.method()

재정의에 대해 배웠기 때문에 이 문제는 상당히 쉬웠을 것이다. 이 예제에서는 상속과 관련이 있는 SuperClass, SubClass의 외부에서 즉 Example 클래스에서 재정의된 메쏘드를 불렀다. 예상했던 대로 재정의된 서브 클래스의 메쏘드가 불렸다.

다음 예제는 서브 클래스에서 불렀을 때의 예제이다.

public class Example {

public static void main(String[] args) {

new SubClass().callMethod();

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

}

public void callMethod() {

method(); // 이번에는 여기서 부른다.

}

}

결과는 다음과 같다.

SubClass.method()

예상했던대로일 것이다. 재정의된 서브 클래스의 메쏘드가 불렸다.

그렇다면 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new SubClass().callMethod();

new SuperClass().callMethod();

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

public void callMethod() {

method(); // 여기를 주목하라.

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같다.

SubClass.method()

SuperClass.method()

예상했던대로인가? SuperClass.callMethod 메쏘드가 SubClass.method 메쏘드를 부른다. 눈치가 빠른 사람은 뭔가 심상치가 않다는 것을 느낄 것이다.

서브 클래스의 입장에서 세상을 보자. 문제는 명료하다. 서브 클래스를 만드는 사람은 수퍼 클래스의 소스 코드를 볼 수 있거나 적어도 문서를 보고 수퍼 클래스에서 무엇을 하는지 안다. 그것을 고려해서 재정의를 할 수도 수퍼 클래스의 구현을 그대로 쓸 수도 있다.

이젠 수퍼 클래스의 입장에서 세상을 보자. 위의 예제를 잘 이해했다면 수퍼 클래스는 그렇게 명료한 입장이 아니라는 것을 알 수 있을 것이다. 자신이 부른 메쏘드가 어떻게 재정의되느냐는 수퍼 클래스로서는 알 수가 없다. 위의 예제에서 보면 callMethod 메쏘드가 부르는 method 메쏘드가 어떻게 재정의될지 알 수 없는 것이다. 재정의가 필요할 때 수퍼 클래스가 할 수 있는 것은 메쏘드들과의 관계와 각 메쏘드가 하는 일을 잘 문서화한 후 서브 클래스를 만드는 사람이 모든 것을 잘 하기를 기도하는 것 뿐이다.

그 기도가 부족할 때 다음과 같은 일이 일어난다. 예제를 보자.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(300, 300);

show();

}

public static void main(String[] args) {

new Example();

}

public void paint(Graphics g) {

repaint();

}

}

뭔가 이상한 일이 벌어지고 있음을 알 수 있을 것이다. 무슨 일이 일어나는지 보자.

import java.awt.\*;

public class Example extends Frame {

private long count = 0;

public Example() {

setSize(300, 300);

show();

}

public static void main(String[] args) {

new Example();

}

public void paint(Graphics g) {

System.out.println(count++);

repaint();

}

}

결과는 다음과 같을 것이다.

0

1

2

3

...

무한 루프 비슷한 일이 벌어진다. 자세한 것은 그래픽 책을 참고하도록 하고 대충 말하자면 repaint 메쏘드는 update 메쏘드를 부르고 update 메쏘드를 paint 메쏘드를 부르기 때문이다. paint 메쏘드에서 다시 repaint 메쏘드를 부르니까 계속 paint가 불려지는 것이다. 이 예에서는 무엇을 그릴 것인가는 서브 클래스만 알 수 있다. 그러므로 paint 메쏘드의 재정의는 불가피하다.

상황이 다를 수도 있다. 서브 클래스에서 괜히 엉뚱하게 재정의해서 일을 망치는 것을 바라지 않을 경우가 있다. 이럴 경우 쓸 수 있는 방법이 두가지 정도 있다. 첫째는 그 메쏘드를 private으로 선언하는 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new SubClass().callMethod();

}

}

class SuperClass {

private void method() {

System.out.println("일 잘하는 SuperClass.method()");

}

public void callMethod() {

method();

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("뭔가를 망치고 싶어하는 SubClass.method()");

}

}

결과는 다음과 같다.

일 잘하는 SuperClass.method()

그러나 이 방법은 만약 method 메쏘드가 그 클래스 밖에서도 쓰여야 한다면 안 통한다. 그 때 쓸 수 있는 것이 final이라는 키워드이다. 다음 예제를 보자.

class SuperClass {

public final void method() { // final로 선언.

System.out.println("일 잘하는 SuperClass.method()");

}

public void callMethod() {

method();

}

}

class SubClass extends SuperClass {

public void method() { // Error. 결국 일을 망칠 순 없다.

System.out.println("뭔가를 망치고 싶어하는 SubClass.method()");

}

}

* 1. 추상 메쏘드와 추상 클래스
     1. 추상 메쏘드(abstract method)

재정의에서 더 나아가 아예 수퍼 클래스에서 메쏘드의 몸체(body)를 빼먹을 수도 있다. 여기서 주의할 것은 아무 일도 하지 않더라도 구체 메쏘드(concrete method)일 수 있다는 것이다. 다음 예제를 보자.

public abstract class Example {

public abstract void abstractMethod();

public void concreteMethod() {}

}

concreteMethod는 실제로 하는 일은 없지만 추상 메쏘드가 아니라 구체 메쏘드이다.

다음 예제를 보자. 위에 나온 예제를 약간 바꾸었다.

public class Example {

public static void main(String[] args) {

군인 군인들[] = new 군인[5];

군인들[0] = new 소위();

군인들[1] = new 병장();

군인들[2] = new 상병();

군인들[3] = new 일병();

군인들[4] = new 이병();

열받는다(군인들);

}

public static void 열받는다(군인[] 군인들) {

for (int i = 0; i < 군인들.length; i++)

군인들[i].열받는다();

}

}

abstract class 군인 {

public abstract void 열받는다(); // abstract 메쏘드

}

abstract class 장교 extends 군인 {

}

abstract class 사병 extends 군인 {

}

class 소위 extends 장교 {

public void 열받는다() {

System.out.println("병장을 갈군다.");

}

}

class 병장 extends 사병 {

public void 열받는다() {

System.out.println("상병을 갈군다.");

}

}

class 상병 extends 사병 {

public void 열받는다() {

System.out.println("일병을 갈군다.");

}

}

class 일병 extends 사병 {

public void 열받는다() {

System.out.println("이병을 갈군다.");

}

}

class 이병 extends 사병 {

public void 열받는다() {

System.out.println("탈영한다.");

}

}

결과는 다음과 같다.

병장을 갈군다.

상병을 갈군다.

일병을 갈군다.

이병을 갈군다.

탈영한다.

추상 메쏘드는 메쏘드의 몸체를 감싸는 {}대신에 끝에 ;만 있다. 그리고 abstract라는 지정자(modifier)를 사용한다. 이것은 메쏘드가 무슨 일을 할지를 아직 정의하지 않았다는 것을 나타낸다. 서브 클래스에서 추상 메쏘드의 몸체를 정의하는 것을 구현(implement)한다고 말한다.

정적 메쏘드는 동적 바인딩이 되지 않기 때문에 추상 메쏘드가 될 수 없다. 이것은 정적 메쏘드가 재정의되지 않는다는 사실과도 연결된다. 정적 메쏘드는 추상 메쏘드가 될 수 없기 때문에 인터페이스 안에서도 쓰일 수 없다.

* + 1. 추상 클래스(abstract class)

위의 예제에서 군인 클래스는 추상 클래스이다. 추상 클래스는 abstract 변경자가 쓰인 클래스를 말한다. 추상 클래스에는 위의 예와 같이 보통 추상 메쏘드가 있다. 이것은 이 클래스가 아직 미완성이며 서브 클래스가 추상 메쏘드를 구현하여 완성된 클래스를 만들어 줄 것을 바라는 것이다. 위에서 예를 들어 병장 클래스가 열받는다 메쏘드를 구현하여 클래스를 완성했다. 추상 클래스와 반대되는 병장 클래스와 같은 클래스를 구체 클래스(concrete class)라고 한다.

추상 클래스는 미완성 클래스이기 때문에 인스턴스를 만들 수 없다. 그러므로 다음 예제는 컴파일되지 않는다.

public class Example {

public static void main(String[] args) {

new AbstractClass(); // Error.

}

}

abstract class AbstarctClass {

}

위의 예제와 같이 추상 메쏘드가 없는 클래스도 추상 클래스로 정의될 수 있다. 그러나 흔히 쓰는 방식은 아니다. 많은 경우 이것은 설계상에 문제가 있다는 것을 나타낸다.

추상 클래스를 정의하는 abstract 변경자는 없어도 된다고 생각하는 사람도 있을 것이다. 메쏘드 중에 추상 메쏘드가 있는지를 알아보면 그 클래스가 추상 클래스인지 알 수 있기 때문이다. 그러나 메쏘드의 개수가 많고 상속에 상속을 거듭한다면 어떤 클래스가 추상 클래스인지 알아내기 위해서 수 천 라인의 코드를 뒤져 봐야 할지도 모른다. 그것은 불편할 뿐 아니라 실수를 유발하게 한다. 그래서 자바에서는 추상 클래스에는 abstract 변경자를 반드시 붙이게 했다.

구체 클래스의 서브 클래스가 추상 클래스로 변신할 수도 있다. 다음 예제를 보자.

class ConcreteClass {

private int value;

public void setValue(int value) {

this.value = value;

}

}

abstract class ReAbstractClass extends ConcreteClass {

public abstract void manipulateValue();

}

비추상(non abstract, 구체 concrete) 메쏘드를 추상 메쏘드로 바꿀 수도 있다. 일반적으로 적용되는 구현을 만들었는데 어떤 특수한 경우에는 그것이 적용될 수 없어서 구체 메쏘드를 다시 추상 메쏘드로 바꾸는 경우도 있다. 다음 예제를 보자.

class ConcreteClass {

private int value;

public void setValue(int value) {

this.value = value;

}

}

abstract class ReAbstractClass extends ConcreteClass {

public abstract void setValue(int value);

}

추상 메쏘드를 정의한 클래스는 반드시 추상 클래스여야 한다. 즉 구체 클래스에서는 추상 메쏘드를 정의할 수 없다. 수퍼 클래스의 추상 메쏘드를 모두 다 구현하지 않았다면 역시 추상 클래스여야 한다. 즉 구체 클래스에는 수퍼 타입(수퍼 클래스, 수퍼 인터페이스)에 선언된 구현되지 않은 추상 메쏘드가 있어서는 안된다. 그러므로 다음 예는 컴파일되지 않았다.

abstract class SuperClass {

public abstract void methodA();

public abstract void methodB();

}

class SubClass extends SuperClass {

public void methodA() {}

// methodB()가 구현되지 않았다. 그러므로 abstract 클래스로 선언되어야 한다.

}

컴파일이 되게 하고 싶으면 다음과 같이 고쳐야 한다.

abstract class SuperClass {

public abstract void methodA();

public abstract void methodB();

}

abstract class SubClass extends SuperClass { // OK. abstract 로 선언

public void methodA() {}

}

* + 1. 추상 클래스를 남용하지 말자

추상 클래스는 인스턴스를 만들 수 없다. 이것을 이용해 어떤 클래스를 추상 클래스로 만듦으로써 그 클래스의 인스턴스를 만들지 못하게 할 수 있다. 그러나 단지 이유가 이것 때문이라면 다른 방법이 더 낫다. 생성자를 private으로 바꾸는 것이다. 예를 들어 Math 클래스는 인스턴스가 필요 없다. 아래의 코드는 Math.java에서 인용한 것이다.

public final strictfp class Math {

private Math() {}

...

}

위와 같이 하면 인스턴스를 만들 수 없다. abstract는 실제로 구현이 덜 되어서 서브 클래스에서 구현을 완성해 주길 바랄 때만 사용하는 것이 좋다.

* + 1. abstract 키워드의 제한

추상 메쏘드는 private일 수 없다. 왜냐하면 private메쏘드는 서브 클래스에서도 접근할 수 없고 따라서 추상 메쏘드를 구현 할 수도 없기 때문이다.

추상 메쏘드는 정적(static)일 수 없다. 정적(static) 메쏘드는 동적으로 바인딩되지 않는다. 그러므로 정적 메쏘드는 재정의되지 않는다. 그리고 추상 메쏘드가 될 수도 없다.

추상 메쏘드는 final이 될 수 없다. final은 상속을 금지하기 때문이다. 상속되지 않는 것을 구현하거나 재정의할 수는 없다.

추상 메쏘드는 native 메쏘드일 수 없다. native 메쏘드는 native 코드에 의해 구현된 것이기 때문이다. 추상 메쏘드는 구현이 되지 않은 메쏘드여야 한다.

추상 메쏘드는 strictfp일 수 없다. strictfp는 구현에 관계된 키워드이기 때문이다.

마찬가지로 추상 메쏘드는 synchronized 메쏘드가 될 수 없다. synchronized도 구현에 관계된 키워드이기 때문이다.

* 1. 클래스의 contract와 구현
     1. contract와 구현

클래스에는 두가지 측면이 있다.

하나는 그 클래스가 어떤 일을 하는가(what is to be done)이다. 이것을 contract라고 한다. 영어 사전에 보면 contract는 협정, 계약 등의 뜻이 있다. 클래스의 contract는 그 클래스가 다른 클래스와 하는 계약이라고 볼 수 있다. 또는 그 클래스가 어떤 기능을 제공하겠다는 약속이다.

또 하나의 측면은 그 클래스가 그 일을 어떻게 하는가(how it is done)이다. 이것을 구현(implementation)이라고 한다. 예를 들어 데이터를 배열에 저장할 것인가 해쉬 테이블(hash table)에 저장할 것인가 아니면 데이터 베이스에 저장할 것인가 등이 구현에 해당한다. 또는 정렬 알고리즘을 거품 정렬(bubble sort)을 사용할 것인가 quick 정렬을 사용할 것인가 등등도 구현에 해당한다.

객체 지향 프로그래밍은 이 둘을 어떻게 잘 구분하는가에 달려있다. 다시말하면 contract를 얼마나 잘 정의하느냐에 달려있다. contract가 잘 정의되어 있다면 그것은 변하지 않을 것이다. 그렇다면 구현이 아무리 바뀌어도 그 클래스를 사용하는 사람은 아무런 불편을 느끼지 않을 것이다. 왜냐하면 클래스를 사용하는 사람은 구현이 어떻게 되는지는 신경쓰지 않아도 되기 때문이다.

객체는 태어나서 죽을 때까지 되도록 contract를 지키는 상태여야 한다. 어떤 순간에는 필드의 상태가 contract가 깨진 상태에 있다면 그 상태에 그 객체를 사용하면 버그가 생길 가능성이 크다.

* + 1. contract와 인터페이스

contract와 인터페이스(클래스의 기초를 다룬 장에 나오는 인터페이스와 구현에 대한 설명을 참조하라. 여기서는 좁은 의미의 인터페이스로 쓰인다)는 의미가 비슷하다. 그리고 구분을 안하고 사용할 수도 있다. 여기서는 그것이 구분된다면 어떻게 구분되는지 알아보자. 인터페이스는 contract의 문법적인 측면이다. 그러므로 contract는 인터페이스와 다른 어떤 것이 합쳐진 것이라고 할 수 있다. 다른 어떤 것은 바로 의미(semantics)이다. Object 클래스에 정의된 equals 메쏘드를 예로 들어 보자.

public boolean equals(Object obj) {

// 구현

}

여기서 equals 메쏘드가 Object형 객체를 인수를 받아들여서 boolean 형을 리턴한다고 규정한 것이 바로 인터페이스에 해당한다. 인터페이스는 코드상에 문법적으로 표현된다. 즉 메쏘드 같은 경우에는 메쏘드 이름, 매개변수 리스트, 반환형(return type), throws절에서의 예외 상황 규정등이 그것들이다. 그리고 equals가 같을 때는 true를 리턴하고 다를 때는 false를 리턴한다는 것 즉 equals의 의미를 규정하는 것이 그 메쏘드의 semantics이다. semantics는 문법적으로 표현되지 않는다. Semantics는 의미이기 때문에 인터페이스로 추측할 수는 있지만 정확한 것은 주로 문서화(documentation) 또는 주석(comment)으로 표현된다(javadoc으로 주석과 문서화를 통합할 수도 있다). equals 예에서는 API 문서를 보면 semantics가 정확히 표현된 것을 볼 수 있다. 인터페이스와 semantics가 합쳐지면 equals 메쏘드가 하는일 즉 equals 메쏘드의 contract가 되는 것이다. 그리고 이 메쏘드 블록 안에서 이 일을 어떻게 하는가를 규정하는 것이 구현이라고 할 수 있다. 구현은 소스 코드를 자세히 보아야 알 수 있다. 대부분의 메쏘드에서 인터페이스와 semantics는 금방 파악할 수 있지만 소스 코드 전체를 이해하는데는 많은 시간이 걸린다. 그렇기 때문에 contract를 잘 규정하는 것이 중요한 것이다. contract만 알면 그 클래스를 사용하는데 문제가 없다면 굳이 그 클래스의 소스 코드를 다 이해할 필요가 없다. 그리고 클래스의 사용자는 그 클래스가 구현되는 것에 신경을 쓰지않기 때문에 그 클래스의 구현에 간섭하지 않는다. 이것은 그 클래스의 구현이 나중에 바뀌어도 사용자의 코드가 바뀔 필요가 없다는 것을 뜻한다.

다음 예제는 인터페이스가 같아도 semantics는 상당히 다를 수 있을 수 있다는 것을 보여준다.

public class 연필 {

...

/\*\*

\* 글씨를 쓴다. 그리고 쓴 글자수를 리턴한다.

\*/

public int 쓰다() {

...

}

}

public class 묘 {

...

/\*\*

\* 시신은 묻는다. 정상적으로 묻었을 때는 묻은 시신수를 리턴한다.

\* 만약 살아나면 DeadAliveError를 던진다(throw).

\*/

public int 쓰다() {

...

}

}

public class 누명 {

...

/\*\*

\* 모함한 사람 수를 리턴한다.

\*/

public int 쓰다() {

...

}

}

contract는 누군가와의 약속이다. 이 약속은 크게 두 부류의 대상과 할 수 있다. 하나는 그 클래스의 사용자와의 약속이고 또 하나는 그 클래스의 구현자와의 약속이다.

* + 1. 사용자(user)에 대한 contract

여기서 사용자란 서브 클래스가 아닌 관계 즉 그 클래스와 친족관계가 아닌 클래스에서 그 클래스를 사용하는 것을 말한다(물론 서브 클래스도 사용자가 될 수 있다). 사용자와의 contract는 public 멤버들로 이루어진다. 주로 메쏘드가 public으로 선언된다. 여기서 모든 메쏘드가 아닌 꼭 공개해야만 하는 메쏘드만 public으로 선언해야 한다. 중첩 타입이 public으로 선언될 수도 있다. 가끔 필드가 public으로 선언되기도 하지만 상수가 아니라면 아주 예외적인 경우에만 사용해야 한다.

클래스를 만든 사람은 그 의미에 맞게 만들어야 하고 사용자도 그 의미대로 사용해야 한다. 예를 들어 다음과 같은 클래스를 보자.

class Circle {

private double radius; // 반지름

private double area; // 면적

...

}

이 클래스는 원을 표현한다. 원의 넓이는 반지름과 밀접한 관계가 있다. 즉

area == Math.PI \* radius \* radius;

인 것이다. 이 클래스의 객체의 상태는 항상 이런 관계를 반영해야 한다. 만약 이 클래스의 객체의 상태가 어떤 때는

radius == 100

area = 10

일 수 있다면 이것을 클래스를 만든 사람이 뭔가를 잘못 만들어서 계약(contract)를 어긴 것이다.

* + 1. 구현자(implementer)에 대한 contract

클래스는 사용자와 계약 관계를 맺을 뿐 아니라 그 구현자 즉 서브 클래스와도 계약 관계를 맺는다. 구현자와의 contract는 public 멤버와 protected 멤버로 구성된다. 구현자는 자손이기 때문에 보통의 사용자보다는 그 클래스에 대해 더 많이 아는 것이다. public 멤버의 사용을 자제해야 하듯이 protected 멤버의 사용도 자제하는 것이 좋다.

구현자 즉 서브 클래스에서도 수퍼 클래스의 contract를 존중해 주어야 할 것이다.

* + 1. contract의 상속과 implementation의 상속

객체지향 언어에서 상속은 두가지 측면이 있다. 그 하나는 contract가 상속된다는 것이다. contract만 상속되는 극단적인 경우가 인터페이스 타입을 이용하는 경우이다. 클래스를 상속한다면 그 클래스의 contract 뿐만 아니라 구현도 상속받는다. 예를 들어 수퍼 클래스에 있는 메쏘드가 어떤 알고리즘으로 구현되었는가 하는 것도 상속해서 그 메쏘드(알고리즘)을 그대로 사용할 수도 있는 것이다. 여기서 contract는 변하지 않아야 하는 것이다. 예를 들어 상속받은 메쏘드를 재정의한다면 다른 알고리즘을 사용할 수도 있지만 즉 구현을 바꿀 수도 있지만 그 메쏘드가 하는 일 즉 contract가 바뀌면 안된다.

* 1. 상속에 대한 조언
     1. 어떤 경우에 상속을 사용할까?

위의 예에서 군인은 아주 일반적인 무엇을 가리킨다. 장교와 사병은 모두 군인의 일종(?)이다. 또한 병장, 상병, 일병, 이병도 사병의 일종이다. 다시 말하면 ‘병장은 사병이다’라는 명제가 성립한다. 상속은 이럴 때 사용한다.

그러므로 다음과 같은 상속은 문법적으로는 오류가 아니지만 아주 안좋은 설계이다.

public class Example {

public static void main(String[] args) {

일병 열받은일병 = new 일병();

열받은일병.열받는다();

}

}

class 수류탄 {

public void 터진다() {

System.out.println("꽝!!!!!");

}

}

class 일병 extends 수류탄 {

public void 열받는다() {

수류탄을던진다();

}

public void 수류탄을던진다() {

터진다();

}

}

결과는 다음과 같다.

꽝!!!!!

더 나은 설계는 다음과 같을 것이다.

public class Example {

public static void main(String[] args) {

일병 열받은일병 = new 일병();

열받은일병.열받는다();

}

}

class 수류탄 {

public void 터진다() {

System.out.println("꽝!!!!!");

}

}

class 일병 {

수류탄 화풀이용수류탄 = new 수류탄();

public void 열받는다() {

수류탄을던진다();

}

public void 수류탄을던진다() {

화풀이용수류탄.터진다();

}

}

결과는 위와 같다. 일병이 수류탄을 가지고 있는(has a의 관계) 것이지 일병이 수류탄인(is a 관계) 것은 아니다.

* + 1. 상속보다는 composition을 이용하라

위의 수류탄 예제 중 두 번째 예제가 composition을 사용한 것이다. 위의 예제는 명백하게 상속이 아닌 composition을 사용해야 하는 예이다. 어떤 경우에는 상속도 그럴듯하고 composition도 그럴듯한 경우도 있다. 그럴때는 되도록이면 composition을 사용하는 것이 좋다. 상속관계는 유연성이 적다.

어떤 사람이 정신병원 관리 프로그램을 만든다고 생각해 보자. 이 사람이 방금 상속을 배웠다면 상속을 써먹고 싶어 안달일 것이다. 그래서 다음과 같은 클래스들을 만들었다.

public class Patient {} // 환자

public class MentalPatient extends Patient {} // 정신질환자

public class Schizophrenic extends MentalPatient {} // 정신분열증환자

public class Paranoiac extends MentalPatient {} // 편집증환자

그리고 자신이 드디어 상속을 멋지게 써먹었다고 기분좋아할 것이다. 그러나 위와 같은 클래스들을 사용하려고 할 때 어떤 환자가 정신분열증과 편집증을 동시에 앓고 있다면 어쩔 것인가? 자바에서는 클래스는 다중 상속되지 않는다. 만약 composition과 상속을 적절히 사용했다면 다음과 같이 더 유연하게 설계할 수 있을 것이다.

public class Patient {

private ArrayList mentalDiseases; // 그 환자가 앓고 있는 정신질환들의 컬렉션. composition의 사용

}

public class MentalDisease {} // 정신질환

public class Schizophrenia extends MentalDisease {} // 정신 분열증

public class Paranoia extends MentalDisease {} // 편집증

* + 1. 가능하면 멤버를 계층(hierarchy)의 위쪽으로

위의 예를 만약 다음과 같이 만들었다면

class 군인 {

}

class 장교 extends 군인 {

}

class 사병 extends 군인 {

}

class 소위 extends 장교 {

public void 열받는다() {

System.out.println("병장을 갈군다.");

}

}

class 병장 extends 사병 {

public void 열받는다() {

System.out.println("상병을 갈군다.");

}

}

class 상병 extends 사병 {

public void 열받는다() {

System.out.println("일병을 갈군다.");

}

}

class 일병 extends 사병 {

public void 열받는다() {

System.out.println("이병을 갈군다.");

}

}

class 이병 extends 사병 {

public void 열받는다() {

System.out.println("탈영한다.");

}

}

군인[i].열받는다();와 같은 사용법은 불가능해질 것이다. 열받는다 메쏘드는 모든 군인의 서브 클래스에 공통되는 것이다. 그것은 설사 군인 메쏘드에서 구현하는 것이 불가능하더라도 추상 메쏘드로 정의해 주는 것이 좋다.

* + 1. 고수는 protected와 private을 구별해서 사용한다

처음에는 모든 멤버를 public으로(또는 디폴트 접근으로) 정의했다. 그 다음에는 정보 감추기라는 것을 배워서 감추어야 할 것은 private로 정의했다. 이제 protected를 배웠다. 이제 감추어야 할 것은 모두 protected로 하고 싶은 유혹을 느낄 것이다. 처음에는 그렇게 하는 것도 좋다. 처음부터 모든 걸 알 수 있는 건 아니니까. 그러나 사람들은 private이라는 키워드를 없애지 않았다. 그것은 둘(private과 protected)을 구분하는 것이 유용하기 때문이다.

먼저 public과 private으로 나누었을 때 public으로 된 것은 다른 클래스에 대한 인터페이스였으며 private으로 된 것은 구현이었다. 다른 말로 하면 사용자에 대한 인터페이스와 사용자가 몰라야 되는 구현으로 나뉘었다. 이것은 사용자에 대한 불신을 나타내는 것이라고 말할 수도 있다. 클래스의 사용자가 내부 데이터를 잘못 건드려서 클래스의 상태가 깨지는 것을 방지하기 위해 private으로 선언한 것이다.

인터페이스(여기서 쓰이는 인터페이스는 자바 키워드가 아니라 객체 지향 용어이다)는 클래스의 사용자(user)에 대한 인터페이스만 있는 것이 아니다. 클래스의 인터페이스는 또 있는데 이것은 바로 클래스의 구현자(implementer)에 대한 인터페이스가 그것이다. 클래스의 구현자란 클래스를 상속하는 사람을 말한다. 프로그램의 사용자에 대한 인터페이스를 구현하는 것이 public이라면 프로그램의 구현자에 대한 인터페이스를 구성하는것은 public과 protected이다. 그리고 사용자를 불신해야 한다면 구현자도 불신해야 한다.

거꾸로 보자. 어떤 클래스를 만드는 사람이 사용자와 구현자를 불신하는 것이 좋다는 얘기를 했다. 거꾸로 사용자와 구현자는 그 클래스의 내부 구조를 모르는 것이 좋다. 만약 그 내부 구조를 안다면 클래스의 내부 구조가 바뀔 때마다 사용자와 구현자의 코드가 바뀌어야 한다. 그 내부 구조를 모르기 때문에 (인터페이스가 바뀌지 않는 한) 사용자와 구현자의 코드가 영향을 받지 않는 것이다.

이 이야기는 디폴트(패키지) 접근이 도입되면서 더 복잡해진다. 자세한 것은 패키지를 다루는 부분을 참조하라.

다음 예제를 보자. 좀 억지스럽더라도 참고 보기 바란다.

public class Example {

public static void main(String[] args) {

Circle c = new CircleWithCenter(1.0, 1.5, 2.3);

System.out.println(c.getArea());

c.setRadius(1.0);

System.out.println(c.getArea());

}

}

class Circle {

protected double radius; // 반지름

protected double area; // 면적

public Circle(double r) {

radius = r;

area = Math.PI \* r \* r;

}

public void setRadius(double r) {

radius = r;

area = Math.PI \* r \* r;

}

public double getRadius() {

return radius;

}

public double getArea() {

return area;

}

}

class CircleWithCenter extends Circle {

protected double centerX; // 중심의 X 좌표

protected double centerY; // 중심의 Y 좌표

public CircleWithCenter(double r, double x, double y) {

super(r);

centerX = x;

centerY = y;

}

public void setRadius(double r) {

radius = r;

area = 2 \* Math.PI \* r; // 실수

}

}

결과는 다음과 같다.

3.141592653589793

6.283185307179586

여기서 Circle 클래스는 후손들을 신뢰하여 radius와 area를 protected로 선언했다. 만약 이것들을 private으로 선언했다면 어떨까? 다음 예제를 보자.

class Circle {

private double radius; // 반지름

private double area; // 면적

public Circle(double r) {

radius = r;

area = Math.PI \* r \* r;

}

public void setRadius(double r) {

radius = r;

area = Math.PI \* r \* r;

}

public double getRadius() {

return radius;

}

public double getArea() {

return area;

}

}

class CircleWithCenter extends Circle {

protected double centerX; // 중심의 X 좌표

protected double centerY; // 중심의 Y 좌표

public CircleWithCenter(double r, double x, double y) {

super(r);

centerX = x;

centerY = y;

}

public void setRadius(double r) {

radius = r; // Error.

area = 2 \* Math.PI \* r; // Error.

}

}

컴파일타임 오류가 날 것이다. 실수할 수 있는 가능성을 하나 봉쇄한 것이다. Circle 클래스에 setRadius 메쏘드는 있지만 setArea 메쏘드는 없다는 것에도 주목하라. Circle 클래스의 area는 radius에 의해 결정되어야 한다. 여기서 클래스의 사용자들은 radius와 area에 직접 접근하지 못하고 setRadius, getRadius, getArea 메쏘드를 이용해 간접적으로 접근해야 한다는 것을 알 수 있다. 그렇게 함으로써 클래스의 contract가 지켜진다. 또한 클래스의 구현자(서브 클래스)도 또한 radius와 area에 직접 접근하지 못한다. 그렇게 함으로써 사용자의 실수를 막을 뿐 아니라 구현자의 실수도 막을 수 있다. 여기서의 예는 너무나 간단하기 때문에 이런 실수를 하지는 않겠지만 시스템이 커지면 즉 코드량이 많아지면 이런 실수를 막는 것이 중요한 문제가 된다.

접근 지정자(public, protected, default, private)을 사용할 때는 가능한 한 접근하지 못하게 하는 방향으로 사용해야 한다. 이것은 필드와 메쏘드 모두에 해당하는 얘기다. 가능한 한 최대한 private을 많이 사용하라. 정 안되면 protected를 사용하고 이것은 정말 외부에 공개할 수 밖에 없다고 판단될 때만 public을 사용하라. 이것은 타입의 계층(hierarchry)를 설계할 때와 비슷하다. 가능한 한 추상화 하는 것이 좋다. 즉 어떤 기능을 최대한 인터페이스에 넣고 정 안되면 추상 클래스에 넣고 그래도 안되면 구체 클래스에 넣어야 한다. 디폴트 접근(default, package, friendly access)에 대해서는 패키지를 다루는 장을 참조하라.

private과 public의 관계를 살펴 볼 때 private 멤버(필드, 메쏘등)를 사용하면 구현이 바뀌어도 다른 클래스의 코드가 바뀔 필요가 없다는 것을 알았다. 이것은 private과 protected의 관계에도 적용된다. private 멤버를 사용하면 구현이 바뀌어도 서브 클래스의 코드가 바뀔 필요가 없는 것이다.

1. 인터페이스
   1. 인터페이스란?
      1. 클래스가 할 수 있는 것과 인터페이스가 할 수 있는 것

클래스와 인터페이스는 아주 비슷하게 생겼다. 인터페이스는 어떻게 보면 클래스의 일종이라고 할 수도 있다.

그러나 인터페이스는 클래스가 할 수 있는 많은 것을 할 수 없다. 인터페이스의 메쏘드는 몸체(body)를 가질 수 없다. 즉 추상 메쏘드여야 한다. 그러므로 추상 클래스와 마찬가지로 인터페이스의 객체를 만들 수 없다. 인터페이스는 상수(static final)를 제외하고 필드를 가질 수 없다. 인터페이스는 객체를 만들 수 없기 때문에 생성자를 가질 수 없다. 인터페이스는 정적 메쏘드를 가질 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A(); // Error. 객체를 생성할 수 없다.

}

}

interface A {

A() {} // Error. 생성자를 가질 수 없다.

void methodA() {} // Error. 메쏘드의 몸체를 정의할 수 없다.

void methodB(); // OK.

static void methodC(); // Error. 정적 메쏘드는 안된다.

static final VALUE = 3; // OK.

static value = 5; // Error. 상수여야 한다.

final finalValue = 6; // Error. static final 이어야 한다.

}

컴파일이 안된다. 여기까지는 정말 쓸모 없는 것을 하나 만들어서 사람들을 귀찮게 하는 듯이 보인다. 그러나 인터페이스의 힘은 할 수 없는 것이 많다는 데서 나온다. 이것을 어렵게 말하면 추상화(abstraction)의 힘이라고 할 수 있다. 인터페이스가 할 수 없는 것은 무엇인가? 잘 보면 인터페이스가 할 수 없는 것들 중 대부분은 단 하나의 말로 설명할 수 있다. 그것은 바로 구현(implementation)이다. 메쏘드의 몸체는 바로 구현에 필요하다. 필드도 또한 구현에 필요하다. 즉 객체의 데이터를 저장하는데 필요하다. 생성자도 구현에 필요하다. 구현을 완성하지 않았으니 객체를 생성할 수도 없다. 인터페이스(자바 키워드)는 바로 구현이 제외된 것 즉 인터페이스(객체 지향 용어)를 표현하는 것이다. 인터페이스와 구현을 뚜렷이 구분하는 것은 객체 지향 설계에서 중요한 도구이다. 인터페이스 키워드는 바로 그것을 도와주는 역할을 한다.

인터페이스는 클래스가 할 수 없는 것을 할 수 있다. 그것은 바로 다중 상속과 관련된 것이다. 다음 예제를 보자.

interface SuperA {}

interface SuperB {}

interface C extends SuperA, SuperB {}

class D implements SuperA, SuperB {}

무사히 컴파일된다. 인터페이스가 여러 인터페이스를 다중 상속할 수 있을 뿐 아니라 클래스도 여러 인터페이스를 다중 상속(구현, implement)할 수 있다. 이것이 바로 인터페이스의 가장 커다란 존재의 이유중에 하나이다. 인터페이스는 C++의 순수 추상 클래스(pure abstract class)와 비슷하다. C++과 달리 자바에는 또하나의 키워드(interface)가 필요한데 그 이유는 C++에서 지원되는 클래스의 다중 상속이 자바에서는 지원되지 않기 때문이다. 자바는 다중 상속의 위험함을 피하기 위해 구현의 다중 상속 즉 클래스의 다중 상속을 금지했다. 그리고 다중 상속의 유용함을 이용하기 위해 인터페이스(객체 지향 용어)의 다중 상속 즉 인터페이스(자바 키워드)의 다중 상속을 가능하게 했다.

* + 1. 추상 클래스 – 인터페이스로 가는 길

클래스에는 구체 클래스(concrete class)와 추상 클래스(abstract class)가 있다. 그리고 추상 클래스 중에는 C++의 용어를 쓰면 순수 추상 클래스(pure abstract class)가 있다. 다음은 구체 클래스, 추상 클래스, 순수 추상 클래스의 예이다.

class ConcreteClass {

private int value;

public void setValue(int value) {

this.value = value;

}

public int getValue() {

return value;

}

public void showValue() {

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

}

}

abstract class AbstarctClass {

private int value;

public void setValue(int value) {

this.value = value;

}

public int getValue() {

return value;

}

public abstract void showValue();

}

abstract class PureAbstarctClass {

public abstract void setValue(int value);

public abstract int getValue();

public abstract void showValue();

}

여기서 순수 추상 클래스와 거의 비슷한 것이 자바의 인터페이스이다. 위의 예를 인터페이스로 고쳐 보자.

interface Interface {

public abstract void setValue(int value);

public abstract int getValue();

public abstract void showValue();

}

사실 순수 추상 클래스를 쓰는 것보다는 인터페이스를 쓰는 것이 낫다. 인터페이스는 다중 상속이 가능하기 때문이다. 그리고 인터페이스 키워드를 사용함으로써 그냥 추상 클래스가 아니라 순수하게 추상적인 형(type)임을 나타낼 수 있다. 구체 클래스가 형의 구체성의 극단이라면 인터페이스는 형의 추상성의 극단이다.

* 1. 인터페이스의 구현
     1. 구현하기

인터페이스의 구현은 추상 클래스를 상속하는 것과 거의 비슷하다. 다음 예를 보자.

interface Interface {

public abstract void method();

}

class SubClass implements Interface {

public void method() {}

}

SubClass가 Interface 인터페이스를 구현했다. extends 대신 implements라는 키워드를 사용한다.

* + 1. 다중 상속(multiple inheritance)

인터페이스는 여러 인터페이스를 상속할 수 있다. 즉 다중 상속이 가능하다. 인터페이스의 다중 상속으로 몇가지 문제가 발생할 수 있는데 그것은 클래스와 인터페이스 – 못다한 이야기를 다루는 장을 참조하라.

클래스는 여러 인터페이스를 상속(구현)할 수 있다. 즉 다중 상속이 가능하다. 이것은 하나의 클래스가 여러개의 인터페이스를 가질 수 있다는 것을 뜻한다. 다음 예제를 보자.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame {

private static Shape[] m\_ash;

private static Colored[] m\_aclrd;

public Example() {

setSize(400, 400);

addWindowListener(

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

}

public static void main(String[] args) {

m\_ash = new Shape[5];

Circle circleA = new Circle(new Point(200, 200), 100, Color.red);

Circle circleB = new Circle(new Point(200, 200), 60, Color.blue);

Circle circleC = new Circle(new Point(100, 200), 120, Color.black);

m\_ash[0] = circleA;

m\_ash[1] = circleB;

m\_ash[2] = circleC;

m\_ash[3] = new BlueSquare(new Point(200, 200), 130);

m\_ash[4] = new BlueSquare(new Point(100, 100), 200);

m\_aclrd = new Colored[5];

m\_aclrd[0] = circleA;

m\_aclrd[1] = circleB;

m\_aclrd[2] = circleC;

m\_aclrd[3] = new Dog("Kinky", Color.yellow);

m\_aclrd[4] = new Dog("Anarchist", Color.red);

new Example().show();

for (int i = 0; i < m\_aclrd.length; i++)

System.out.println(m\_aclrd[i].getColor()); // 여기에 주목하라.

}

public void paint(Graphics g) {

if (m\_ash != null)

for (int i = 0; i < m\_ash.length; i++)

m\_ash[i].draw(g); // 여기에 주목하라.

}

}

interface Shape {

void draw(Graphics g);

}

interface Colored {

Color getColor();

void setColor(Color c);

}

abstract class AbstractCircle implements Shape {

protected Point m\_pntCenter;

protected int m\_iRadius;

public AbstractCircle(Point center, int radius) {

m\_pntCenter = center;

m\_iRadius = radius;

}

}

class Circle extends AbstractCircle implements Colored {

protected Color m\_clr;

public Circle(Point center, int radius, Color clr) {

super(center, radius);

m\_clr = clr;

}

public void draw(Graphics g) {

g.setColor(m\_clr);

g.drawOval(m\_pntCenter.x - m\_iRadius, m\_pntCenter.y - m\_iRadius,

m\_iRadius \* 2, m\_iRadius \* 2);

}

public void setColor(Color clr) {

m\_clr = clr;

}

public Color getColor() {

return m\_clr;

}

}

class BlueSquare implements Shape {

private Point m\_pntLeftTop;

private int m\_iWidth;

public BlueSquare(Point leftTop, int width) {

m\_pntLeftTop = leftTop;

m\_iWidth = width;

}

public void draw(Graphics g) {

g.setColor(Color.blue);

g.drawRect(m\_pntLeftTop.x, m\_pntLeftTop.y, m\_iWidth, m\_iWidth);

}

}

class Dog implements Colored {

private Color m\_clr;

private String m\_strName;

public Dog(String name, Color clr) {

m\_strName = name;

m\_clr = clr;

}

public void setColor(Color clr) {

m\_clr = clr;

}

public Color getColor() {

return m\_clr;

}

public void bark() {

System.out.println("My Name Is " + m\_strName);

}

}

결과는 다음과 비슷할 것이다.

java.awt.Color[r=255,g=0,b=0]

java.awt.Color[r=0,g=0,b=255]

java.awt.Color[r=0,g=0,b=0]

java.awt.Color[r=255,g=255,b=0]

java.awt.Color[r=255,g=0,b=0]

Circle 클래스가 Shape와 Colored인터페이스 둘을 모두 구현하기 때문에 Shape 형 변수와 Colored 형 변수 모두에 들어갈 수 있었다. 즉 다음과 같은 코드가 가능했다.

m\_ash[0] = circleA; // Shape 형 변수에 대입

m\_aclrd[0] = circleA; // Colored 형 변수에 대입

Circle과 BlueSquare 모두 Colored를 구현했기 때문에 다음과 같은 코드가 가능했다.

m\_ash[i].draw(g); // 여기에 주목하라.

* + 1. 추상 클래스 – 인터페이스의 미완성 구현

어떤 클래스가 만약 추상 메쏘드가 있다면 그것은 추상 클래스여야 한다. 마찬가지로 수퍼 클래스의 추상 메쏘드 중 하나라도 구현하지 않은 것이 있다면 추상 클래스여야 한다. 마찬가지로 수퍼 인터페이스의 추상 메쏘드중 하나라도 구현하지 않은 것이 있다면 추상 클래스여야 한다. 그러므로 다음 예제는 컴파일되지 않는다.

interface Interface {

public abstract void methodA();

public abstract void methodB();

}

class SubClass implements Interface {

public void methodA() {}

// methodB()가 구현되지 않았다.

}

무사히 컴파일되게 하려면 다음과 같이 바꾸어야 한다.

interface Interface {

public abstract void methodA();

public abstract void methodB();

}

abstract class SubClass implements Interface { // abstract 로 선언했다.

public void methodA() {}

}

* 1. 인터페이스의 사용
     1. 인터페이스의 구현

다음 예제는 그래픽 프로그램이다. 여기서 중요한 것은 그래픽을 어떻게 다루는가가 아니라 인터페이스와 클래스가 어떻게 맞물려 있느냐이다. 그래픽에 대한 것은 그것은 다룬 책을 참조하라. 고전적인 예인 Shape과 그것을 구현하는 Circle, Triangle, Rectangle이다. java.awt.Shape과 헷갈리지 않도록 하자. 여기서 쓰인 Shape은 여기에 정의된 Shape이다.

public class Example extends Frame {

Shape[] shapes;

public Example() {

shapes = new Shape[3];

shapes[0] = new Circle(34, new Point(100, 200));

Point[] trianglePoints = {new Point(200, 300), new Point(100, 450), new Point(300, 400)};

shapes[1] = new Triangle(trianglePoints);

shapes[2] = new Rectangle(new Point(100, 100), new Point(400, 400));

}

public static void main(String[] args) {

Frame f = new Example();

f.addWindowListener(

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

f.setSize(500, 500);

f.show();

}

public void paint(Graphics g) {

if (shapes == null)

return;

for (int i = 0; i < shapes.length; i++)

shapes[i].draw(g);

}

}

interface Shape {

void draw(Graphics g);

}

class Circle implements Shape {

private int radius;

private Point center;

public Circle(int radius, Point center){

this.radius = radius;

this.center = center;

}

public void draw(Graphics g) {

g.drawOval(center.x - radius, center.y - radius, radius \* 2, radius \* 2);

}

}

class Triangle implements Shape {

private Point[] vertices;

public Triangle(Point[] vertices) {

if (vertices.length != 3)

throw new IllegalArgumentException();

this.vertices = vertices;

}

public void draw(Graphics g) {

g.drawLine(vertices[0].x, vertices[0].y, vertices[1].x, vertices[1].y);

g.drawLine(vertices[1].x, vertices[1].y, vertices[2].x, vertices[2].y);

g.drawLine(vertices[2].x, vertices[2].y, vertices[0].x, vertices[0].y);

}

}

class Rectangle implements Shape {

private Point leftTop;

private Point rightBottom;

public Rectangle(Point leftTop, Point rightBottom) {

this.leftTop = leftTop;

this.rightBottom = rightBottom;

}

public void draw(Graphics g) {

g.drawRect(leftTop.x, leftTop.y, rightBottom.x - leftTop.x, rightBottom.y - leftTop.y);

}

}

실행해 보면 윈도우에 삼각형, 원, 직사각형이 그려질 것이다.

shapes[i].draw(g);

에서 볼 수 있듯이 추상 클래스와 마찬가지이다.

인터페이스를 구현할 때는 implements라는 키워드를 사용한다. 클래스가 클래스를 상속할 때와 인터페이스가 인터페이스를 상속할 때는 extends라는 키워드를 사용한다. 당연한 일이지만 인터페이스가 클래스를 상속할 수는 없다. 넓은 의미로 보면 implements와 extends 모두 상속을 나타낸다.

* + 1. 컬렉션 클래스(collection classes)에서의 예

여기서 보여줄 클래스와 인터페이스는 Collection, List, AbstractCollection, AbstractList, ArrayList이다. List 인터페이스는 Collection 인터페이스를 상속한다. AbstractList 클래스는 AbstarctCollection 클래스를 상속하고 ArrayList 클래스는 AbstractList 클래스를 상속한다. Collection 인터페이스가 추상의 끝이라면 ArrayList는 구체의 끝이라고 할 수 있다. 인터페이스와 추상 클래스와 구체 클래스는 이렇게 추상-구체의 계열을 이룬다.

다음은 각각의 (주석을 제외한 그리고 부분적으로 생략된) 소스 코드이다. 어떤 형(type)이 얼마나 추상적인지 그 추상적인 것을 어떤 형이 구현해서 더 구체적으로 되는지를 살펴보자.

Collection.java

package java.util;

public interface Collection {

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator iterator();

Object[] toArray();

Object[] toArray(Object a[]);

boolean add(Object o);

boolean remove(Object o);

boolean containsAll(Collection c);

boolean addAll(Collection c);

boolean removeAll(Collection c);

boolean retainAll(Collection c);

void clear();

boolean equals(Object o);

int hashCode();

}

List.java

package java.util;

public interface List extends Collection {

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator iterator();

Object[] toArray();

Object[] toArray(Object a[]);

boolean add(Object o);

boolean remove(Object o);

boolean containsAll(Collection c);

boolean addAll(Collection c);

boolean addAll(int index, Collection c); // 새로 추상 메쏘드 정의

boolean removeAll(Collection c);

boolean retainAll(Collection c);

void clear();

boolean equals(Object o);

int hashCode();

Object get(int index); // 새로 추상 메쏘드 정의

Object set(int index, Object element); // 새로 추상 메쏘드 정의

void add(int index, Object element); // 새로 추상 메쏘드 정의

Object remove(int index); // 새로 추상 메쏘드 정의

int indexOf(Object o); // 새로 추상 메쏘드 정의

int lastIndexOf(Object o); // 새로 추상 메쏘드 정의

ListIterator listIterator(); // 새로 추상 메쏘드 정의

ListIterator listIterator(int index); // 새로 추상 메쏘드 정의

List subList(int fromIndex, int toIndex); // 새로 추상 메쏘드 정의

}

AbstractCollection.java

package java.util;

public abstract class AbstractCollection implements Collection {

protected AbstractCollection() {} // 생성자

public abstract Iterator iterator();

public abstract int size();

public boolean isEmpty() {...} // Collection 인터페이스 구현

public boolean contains(Object o) {...} // Collection 인터페이스 구현

public Object[] toArray() {...} // Collection 인터페이스 구현

public Object[] toArray(Object a[]) {...} // Collection 인터페이스 구현

public boolean add(Object o) {...} // Collection 인터페이스 구현

public boolean remove(Object o) {...} // Collection 인터페이스 구현

public boolean containsAll(Collection c) {...} // Collection 인터페이스 구현

public boolean addAll(Collection c) {...} // Collection 인터페이스 구현

public boolean removeAll(Collection c) {...} // Collection 인터페이스 구현

public boolean retainAll(Collection c) {...} // Collection 인터페이스 구현

public void clear() {...} // Collection 인터페이스 구현

public String toString() {...} // Object 클래스 재정의

}

AbstractList.java

package java.util;

public abstract class AbstractList extends AbstractCollection implements List {

protected AbstractList() {} // 생성자

public boolean add(Object o) {...} // AbstractCollection 클래스 재정의

abstract public Object get(int index);

public Object set(int index, Object element) {...} // List 인터페이스 구현

public void add(int index, Object element) {...} // List 인터페이스 구현

public Object remove(int index) {...} // List 인터페이스 구현

public int indexOf(Object o) {...} // List 인터페이스 구현

public int lastIndexOf(Object o) {...} // List 인터페이스 구현

public void clear() {...} // AbstractCollection 클래스 재정의

public boolean addAll(int index, Collection c) {...} // List 인터페이스 구현

public Iterator iterator() {...} // Collection 인터페이스 구현

public ListIterator listIterator() {...} // List 인터페이스 구현

public ListIterator listIterator(final int index) {...} // List 인터페이스 구현

private class Itr implements Iterator {...} // 중첩 클래스는 신경쓰지 말자.

private class ListItr extends Itr implements ListIterator {...} // 중첩 클래스는 신경쓰지 말자.

public List subList(int fromIndex, int toIndex) {...} // List 인터페이스 구현

public boolean equals(Object o) {...} // Object 클래스 재정의

public int hashCode() {...} // Object 클래스 재정의

protected void removeRange(int fromIndex, int toIndex) {...}

protected transient int modCount = 0;

}

class SubList extends AbstractList {...}

class RandomAccessSubList extends SubList implements RandomAccess {...}

ArrayList.java

package java.util;

public class ArrayList extends AbstractList implements List, RandomAccess, Cloneable, java.io.Serializable {

private static final long serialVersionUID = 8683452581122892189L;

private transient Object elementData[];

private int size;

public ArrayList(int initialCapacity) {...} // 생성자

public ArrayList() {...} // 생성자

public ArrayList(Collection c) {...} // 생성자

public void trimToSize() {...}

public void ensureCapacity(int minCapacity) {...}

public int size() {...} // Collection 클래스 구현

public boolean isEmpty() {...} // AbstractCollection 재정의

public boolean contains(Object elem) {...} // AbstractCollection 재정의

public int indexOf(Object elem) {...} // AbstractList 재정의

public int lastIndexOf(Object elem) {...} // AbstractList 재정의

public Object clone() {...} // Object 클래스 재정의

public Object[] toArray() {...} // AstractCollection 재정의

public Object[] toArray(Object a[]) {...} // AbstractCollection 재정의

public Object get(int index) {...} // List 인터페이스 구현

public Object set(int index, Object element) {...} // AbstractList 재정의

public boolean add(Object o) {...} // AbstractList 재정의

public void add(int index, Object element) {...} // AbstractList 재정의

public Object remove(int index) {...} // AbstractList 재정의

public void clear() {...} // AbstractList 재정의

public boolean addAll(Collection c) {...} // AbstracCollection 재정의

public boolean addAll(int index, Collection c) {...} // AbstractList 재정의

protected void removeRange(int fromIndex, int toIndex) {...} // AbstractList 재정의

private void RangeCheck(int index) {...}

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

private synchronized void readObject(java.io.ObjectInputStream s) throws java.io.IOException, ClassNotFoundException {...}

}

* + 1. 상수 모음

인터페이스는 상수 모음으로도 쓰인다. 다음은 javax.swing.SwingConstants의 (주석을 제외한) 전체 소스이다.

package javax.swing;

public interface SwingConstants {

public static final int CENTER = 0;

public static final int TOP = 1;

public static final int LEFT = 2;

public static final int BOTTOM = 3;

public static final int RIGHT = 4;

public static final int NORTH = 1;

public static final int NORTH\_EAST = 2;

public static final int EAST = 3;

public static final int SOUTH\_EAST = 4;

public static final int SOUTH = 5;

public static final int SOUTH\_WEST = 6;

public static final int WEST = 7;

public static final int NORTH\_WEST = 8;

public static final int HORIZONTAL = 0;

public static final int VERTICAL = 1;

public static final int LEADING = 10;

public static final int TRAILING = 11;

public static final int NEXT = 12;

public static final int PREVIOUS = 13;

}

예를 들어 다음과 같이 사용할 수 있다.

public class Example implements javax.swing.SwingConstants {

public static void main(String[] args) {

if (WEST == SOUTH) {

System.out.println("많이 드셨군요. 면허 취소입니다.");

} else {

System.out.println("50000원이면 봐주겠습니다");

}

}

}

결과는 다음과 같다.

50000원이면 봐주겠습니다

* + 1. marker 인터페이스

어떤 인터페이스는 멤버가 하나도 없다. 예를 들어 다음은 java.lang.Cloneable인터페이스의 (주석을 제외한) 전체 소스이다.

package java.lang;

public interface Cloneable {

}

보다시피 멤버가 하나도 정의되지 않은 썰렁한 인터페이스이다. Cloneable의 쓰임을 알 수 있듯이(Object 클래스를 다룬 부분을 참조하라) 이런 인터페이스는 자바 컴파일러나 자바 가상 머신 등이 어떤 클래스의 어떤 특성(mark)을 알아내는데 쓰인다. 그리고 marker 인터페이스라고 부른다.

marker 인터페이스에는 java.lang.Cloneable, java.io.Serializable, java.rmi.Remote등이 있다.

* + 1. 콜백(call back)과 인터페이스

이번에는 자바의 인터페이스가 어떻게 콜백의 역할을 하는지를 알아보자. 콜백이란 어떤 메쏘드(함수)를 등록해 놓으면 어떤 이벤트(event)가 발생했을 때 그 등록된 메쏘드가 불리어지는 것을 말한다. 콜백 함수는 함수를 등록하지만 자바에서는 인터페이스를 구현한 객체를 등록한다. 먼저 간단한 타이머를 만들어 보자. 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

SimpleTimer st = new SimpleTimer(3000); // 3초 후에 콜백 메쏘드들을 부른다.

st.addTimerListener(new ListenerA());

st.addTimerListener(new ListenerB());

st.startTimer();

}

}

interface TimerListener {

void timeToKill(); // 콜백 메쏘드

}

class ListenerA implements TimerListener {

public void timeToKill() {

System.out.println("Attack of the Killer Tomatoes - 존 드벨로");

}

}

class ListenerB implements TimerListener {

public void timeToKill() {

System.out.println("Return of the Killer Tomatoes - 존 드벨로");

}

}

class SimpleTimer extends Thread {

private ArrayList listenerList = new ArrayList();

private int milliSec;

public SimpleTimer(int milliSec) {

this.milliSec = milliSec;

}

public void addTimerListener(TimerListener listener) { // 인터페이스(콜백)를 등록한다.

listenerList.add(listener);

}

public void startTimer() {

start();

}

public void run() {

try {

sleep(milliSec); // 3초 경과

for (Iterator i = listenerList.iterator(); i.hasNext(); ) {

TimerListener listener = (TimerListener)i.next();

listener.timeToKill(); // 콜백 메쏘드를 불러준다.

}

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

결과는 다음과 같다.

Attack of the Killer Tomatoes - 존 드벨로

Return of the Killer Tomatoes - 존 드벨로

SimpleTimer.addTimerListener메쏘드를 불러주면(call) SimpleTimer 클래스가 TimerListener.timeToKill 메쏘드를 불러준다(call back).

다음으로 간단한 버튼을 만들어 보자. 예제를 보자.

import java.awt.\*;

import java.awt.event.\*;

import java.util.\*;

public class Example extends Frame {

public Example() {

SimpleButton sb = new SimpleButton();

sb.addButtonListener(new Listener()); // 인터페이스를 구현한 객체를 등록한다.

addWindowListener(

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

setSize(300, 300);

sb.setBounds(50, 100, 40, 20);

setLayout(null);

add(sb);

show();

}

public static void main(String[] args) {

new Example();

}

}

interface ButtonListener {

void buttonClicked();

}

class Listener implements ButtonListener {

private int count = 0;

public void buttonClicked() {

count++;

System.out.println("Button Clicked " + count);

}

}

class SimpleButton extends Panel implements MouseListener {

private ArrayList listenerList = new ArrayList();

private boolean mouseInside;

private boolean mousePressed;

private static final int DEFAULT = 0;

private static final int DOWN = 1;

private int state = DEFAULT;

private static final Color bright = new Color(250, 250, 250); //200, 200);

private static final Color middle = new Color(150, 150, 150); //150, 150);

private static final Color dark = new Color(50, 50, 50);//100, 100);

public SimpleButton() {

addMouseListener(this);

}

public void addButtonListener(ButtonListener listener) { // 인터페이스(콜백)를 등록한다.

listenerList.add(listener);

}

public void paint(Graphics g) {

int width = getWidth() - 1;

int height = getHeight() - 1;

switch (state) {

case DEFAULT :

g.setColor(middle);

g.fillRect(0, 0, width, height);

g.setColor(bright);

g.drawLine(0, 0, 0, height);

g.drawLine(0, 0, width, 0);

g.setColor(dark);

g.drawLine(width, 0, width, height);

g.drawLine(0, height, width, height);

break;

case DOWN :

g.setColor(middle);

g.fillRect(0, 0, width, height);

g.setColor(dark);

g.drawLine(0, 0, 0, height);

g.drawLine(0, 0, width, 0);

g.setColor(bright);

g.drawLine(width, 0, width, height);

g.drawLine(0, height, width, height);

break;

}

}

public void mouseClicked(MouseEvent e) {}

public void mousePressed(MouseEvent e) {

state = DOWN;

repaint();

}

public void mouseReleased(MouseEvent e) {

for (Iterator i = listenerList.iterator(); i.hasNext(); ) {

ButtonListener bl = (ButtonListener)i.next();

bl.buttonClicked(); // 인터페이스의 메쏘드를 불러준다.

}

state = DEFAULT;

repaint();

}

public void mouseEntered(MouseEvent e) {}

public void mouseExited(MouseEvent e) {}

}

버튼을 세번 누르면 결과는 다음과 같을 것이다.

Button Clicked 1

Button Clicked 2

Button Clicked 3

제대로 된 타이머와 버튼을 보려면 ActionListener인터페이스와 javax.swing.Timer, java.awt.Button 클래스 등을 참조하라.

* 1. 인터페이스의 멤버
     1. 인터페이스의 필드

인터페이스의 필드는 public static final이어야 한다. 그러므로 다음 예제는 컴파일되지 않을 것이다.

interface Interface {

public static final int VALUE\_A = 23; // OK.

static final int VALUE\_D = 4; // OK. public은 생략될 수 있다.

public final int VALUE\_E = 53; // OK. static은 생략될 수 있다.

public static int VALUE\_F = 543; // OK. final은 생략될 수 있다.

int VALUE\_G = 533; // OK. public static final 모두 생략될 수 있다.

private static final int VALUE\_B = 34; // Error. private 일 수 없다.

protected static final int VALUE\_C = 5; // Error. protected 일 수 없다.

}

인터페이스안에서 필드는 디폴트로 public static final이기 때문에 생략이 가능하다. 그리고 사실 생략하는 것이 관례이다. 인터페이스는 구현에 관계되는 것이 없어야 하기 때문에 상수만 허용한 것이다. 필드를 선언할 때 초기화하지 않으면 컴파일타임 오류가 난다. 그러므로 다음 예제는 컴파일이 안된다.

interface Interface {

int VALUE\_A; // Error.

}

조금만 생각하면 왜 그런지 알 수 있을 것이다. 선언할 때 초기화하지 않으면 값을 대입할 방법이 없다. 어디에서 대입할 것인가?

그렇다고 필드가 완벽히 변하지 않는다고 생각하면 안된다. 참조형일 때는 문제가 달라질 수도 있다. 다음 예를 보자.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

System.out.println(Interface.immutablePoint);

Interface.immutablePoint.translate(100, -200);

System.out.println(Interface.immutablePoint);

}

}

interface Interface {

Point immutablePoint = new Point(12, 34);

}

결과는 다음과 같다.

java.awt.Point[x=12,y=34]

java.awt.Point[x=112,y=-166]

변수가 가리키고 있는 객체를 다른 객체로 바꿀 수 없는 것이지 객체 자체가 변할 수 없는 것은 아니다. 이것을 이용해서 인터페이스에 구현에 해당하는 데이터를 넣을 수도 있다. 그러나 아주 특별한 경우가 아니면 아주 안좋은 생각이다.

* + 1. 인터페이스의 메쏘드 - public abstract

인터페이스의 메쏘드는 public abstract이다. 그러므로 다음 예제는 컴파일되지 않는다.

interface Interface {

public abstract void methodA(); // OK.

abstract void methodB(); // OK. public 은 생략 가능하다.

public void methodC(); // OK. abstract 는 생략 가능하다.

void methodD(); // OK. public abstract 모두 생략 가능하다.

protected abstract void methodE(); // Error. protected 가 올 수 없다.

private abstract void methedF(); // Error. private 이 올 수 없다.

public static void methodG(); // Error. static 일 수 없다.

void methodH() {} // Error. 아무일도 안한다고 해서 허용되지 않는다.

}

인터페이스의 메쏘드는 디폴트로 public abstract이기 때문에 public과 abstract는 생략 가능하다. 인터페이스의 메쏘드는 정적(static) 메쏘드일 수 없다. 정적 메쏘드는 상속이 되지 않아서 구현되거나 재정의가 안되기 때문에 추상(abstract) 메쏘드일 수가 없기 때문이다.

* + 1. 멤버 클래스와 멤버 인터페이스

이것에 대한 내용은 중첩 타입을 다룬 부분을 참조하라.

1. 패키지
   1. 패키지의 필요성
      1. 이름 충돌

패키지(package)의 존재 이유 중에 하나가 이름 충돌을 방지하는 것이다. 만약 어떤 회사에서 버튼을 만든다면 그 클래스 이름을 Button이라고 붙일 가능성이 있다. 다른 회사에서도 버튼을 만든다면 Button이라고 이름을 붙일 것이다. 또 다른 회사에서 두 회사의 버튼 클래스를 모두 써야 한다면 두 버튼 클래스를 어떻게 구별할 수 있을까? 패키지가 없다면 불가능하거나 아주 까다롭다(예를 들어 클래스 로더를 여러 개 띄워서 둘을 구별할 수 있을 것이다). 패키지는 이 문제를 해결한다. A사에서 만든 것은 a.Button이라고 부르고 B사에서 만든 것은 b.Button이라고 이름 붙이는 식이다.

* + 1. grouping

패키지의 또 다른 존재 이유는 비슷한 기능을 구현하는 타입들이나 하나의 기능을 구현하는 여러 타입들을 하나로 묶을 수 있다는 것이다. 예를 들어 java.awt 패키지에는 자바의 AWT(abstract widow toolkit)에 관련된 것을 모아 두었다.

* 1. package 사용하기
     1. package문

package 문은 소스파일의 (주석은 예외로 하고) 제일 위에 와야 한다. 패키지문은 다음과 같다.

package user\_interface;

패키지는 계층 구조를 가질 수 있다. 즉 패키지 아래에 서브 패키지가 있을 수 있다. 예를 들어

package user\_interface.ugly;

로 선언했다면 이 소스 파일에 있는 타입들은 user\_interface의 서브 패키지에 속하게 된다.

하나의 파일에 여러개의 패키지를 선언할 수는 없다.

패키지 이름은 보통 모두 소문자로 쓴다.

* + 1. package와 디렉토리

패키지와 디렉토리(폴더) 구조를 일치시켜줘야 한다. 예를 들어 user\_interface 패키지에 Button 클래스가 있고 user\_interface.ugly 패키지에 UglyButton 클래스가 있다면

user\_interface 디렉토리에 Button.java(Button.class) 파일과 ugly 디렉토리가 있어야 하고 ugly 디렉토리 안에 UglyButton.java(UglyButton.class) 파일이 있어야 한다.

* + 1. 컴파일 단위(compilation unit)

컴파일 단위는 보통 파일이다. 만약 public 클래스를 선언하고 싶으면 그 클래스의 이름을 파일명과 같이해야 한다. 예를 들어 UglyButton 클래스를 public으로 하고 싶으면 파일명 UglyButton.java 안에 선언해야 한다. 그러므로 하나의 파일에 선언할 수 있는 public 클래스의 수는 최대 하나이다. 만약 public이 아니라면 파일명과 다르게 선언할 수 있다. 그러나 일부러 이름을 틀리게 해서 사람들을 골탕먹일 필요는 없다. 하나의 파일에는 보통 하나의 타입만 선언한다. 그러지 않아도 좋은 경우는 테스트를 할때나 학습용일 때이다. 그리고 두 개 이상의 타입이 서로 밀접한 관계를 이루고 있을 때도 하나의 파일에 여러개의 타입을 선언한다. 예를 들어 다음은 AbstractArray.java에서 인용한 것이다.

package java.util;

public abstract class AbstractList extends AbstractCollection implements List {

...

}

class SubList extends AbstractList {

...

}

class RandomAccessSubList extends SubList implements RandomAccess {

...

}

* + 1. 디폴트 패키지(unnamed package)

지금까지 package문이 없는 예제를 많이 보아왔다. package문이 없는 곳에 선언된 타입은 디폴트 패키지에 속한다.

디폴트 패키지는 학습용이나 테스트용에 사용하면 좋다. 그러나 만약 클래스의 수가 많아지거나 상용으로 쓸 타입들을 정의한다면 이 장에서 설명된 관습에 따라 패키지를 명시적으로 정의해 주는 것이 좋다.

* + 1. 패키지와 컴파일 단위의 저장

소스의 저장은 보통 파일 시스템에 한다. 그리고 그 때는 보통 하나의 컴파일 단위(파일)에서 public은 최대 하나여야 한다는 등의 제한을 가한다. 그 이유는 구현을 하기 쉽게 하기 위해서이다.

소스는 반드시 파일 시스템에 저장될 필요는 없다. 데이터 베이스에 저장될 수도 있다. 그리고 데이터 베이스를 이용한 구현에는 위의 제한이 없을 수도 있다(자바를 만든 사람들은 없을 것을 권하고 있다). 자세한 것은 해당 문서를 참조하라.

* + 1. 유일한 이름 붙이기

패키지가 있더라도 이름충돌은 있을 수 있다. 예를 들어 버튼을 만든다면 두 회사에서 다음과 같은 패키지와 클래스 이름을 쓸 가능성이 있다.

ui.component.Button

이것을 막기위해 자바를 만든 사람들이 제안한 방법은 지구상에서 유일한 인터넷 도메인 이름을 앞에 붙이는 방식이다. 예를 들어 sun사의 도메인 이름은 sun.com이다. 그러므로 sun사에서 만든 패키지는 앞에 com.sun을 붙인다. 순서를 바꾼 이유는 com이 더 일반적이기 때문이다. 예를 들어 자바 스터디 네트워크(javastudy.co.kr)에서 버튼을 만든다면 다음과 같을 것이다.

kr.co.javastudy.ui.component.Button

좀 길어져서 불편하긴 하지만 이름 충돌을 확실히 막을 수 있다.

* 1. import 사용하기
     1. import문의 의미

import는 사실 아무것도 import하지 않는다. 단지 full name을 써야 하는 부담을 덜어줄 뿐이다. java.awt.Button을 사용하려면 꼭 import문이 필요한 것이 아니다. 즉

import java.awt.Button;

이나

import java.awt.\*;

이 반드시 필요한 것이 아니다. 다음 예에서 볼 수 있듯이 import문 없어도 살아갈 수 있다.

public class Example extends java.awt.Frame {

public Example() {

setSize(500, 300);

java.awt.Button b = new java.awt.Button("Push me");

setLayout(new java.awt.FlowLayout());

add(b);

b.addActionListener(

new java.awt.event.ActionListener() {

public void actionPerformed(java.awt.event.ActionEvent e) {

System.out.println("You pushed me");

}

}

);

addWindowListener(

new java.awt.event.WindowAdapter() {

public void windowClosing(java.awt.event.WindowEvent e) {

System.exit(0);

}

}

);

}

public static void main(String[] args) {

new Example().show();

}

}

그러나 import를 사용하여 다음과 같이 하면 코드를 쓰기도 쉽고 특히 읽는 것이 더 편할 것이다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame {

public Example() {

setSize(500, 300);

Button b = new Button("Push me");

setLayout(new FlowLayout());

add(b);

b.addActionListener(

new ActionListener() {

public void actionPerformed(ActionEvent e) {

System.out.println("You pushed me");

}

}

);

addWindowListener(

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

}

}

);

}

public static void main(String[] args) {

new Example().show();

}

}

java.lang패키지에 있는 것은 import문이 없어도 된다. 너무 자주 쓰이기 때문에 생략해도 되는 것이다. 예를 들어 위의 예에서

System.exit(0);

의 System은 원래 java.lang.System이다.

* + 1. single-type-import

이것은 하나의 클래스를 import할 때 쓰인다. 예를 들어

import java.awt.Button;

에서는 Button 클래스 하나를 import했다.

* + 1. type-import-on-demand

이것은 하나의 패키지 또는 클래스에 있는 것을 모두 import하기 위해 쓰인다. \*는 일종의 wild card라고 할 수 있다.

import java.awt.\*;

라고 하면 java.awt.Button을 그냥 Button이라고 쓸 수 있는 것이다. 이 예제에서는 java.awt 패키지에 있는 모든 타입을 import했다. 그러나 위와 같이 import하면 java.awt의 서브 패키지인 java.awt.event 패키지는 import되지 않는다.

type-import-on-demand로 패키지 안에 있는 타입만 import할 수 있는 것은 아니다. 다음 예제처럼 하나의 타입안에 있는 타입들 즉 중첩 타입들을 import할 수도 있다. 다음 예는 두 개의 파일로 이루어져 있다.

A.java 파일

package a;

public class A {

public static class InnerA {

}

public static class InnerB {

}

}

B.java 파일

package b;

import a.A.\*;

public class B {

public void method() {

new InnerA();

new InnerB();

}

}

이렇게 하면

new InnerA();

에서 InnerA 클래스가 중첩타입이 아니라는 인상을 주기 때문에 별로 권할 만한 것은 아니다. 아래와 같이 하는 것이 보는 사람을 헷갈리지 않게 한다.

package b;

import a.\*;

public class B {

public void method() {

new A.InnerA();

new A.InnerB();

}

}

* + 1. import의 충돌

두 개의 import된 패키지 안에 같은 이름을 가진 클래스가 있을 수 있다. 그럴 땐 어떻게 될까? 다음 예제를 보자.

import java.util.\*;

import javax.swing.\*;

public class Example {

private Timer tmr; // Error.

}

컴파일타임 오류가 날 것이다. 왜냐하면 java.util.Timer와 javax.swing.Timer중에서 컴파일러가 어느 것을 고를지 모르기 때문이다. 이 문제를 해결하는 방법은 여러가지이다. 아예 긴 이름을 써주는 방법이 있다.

import java.util.\*;

import javax.swing.\*;

public class Example {

private java.util.Timer tmrA;

private javax.swing.Timer tmrB;

}

아니면 하나를 single-type-import를 사용해서 import하는 방법이 있다.

import java.util.\*;

import java.util.Timer;

import javax.swing.\*;

public class Example {

private Timer tmr; // java.util.Timer가 쓰인다.

}

아니면 하나를 import하지 않는 방법도 있다.

import javax.swing.\*;

public class Example {

private Timer tmr;

}

만약 java.util 패키지와 javax.swing 패키지에 있는 클래스들이 많이 쓰인다면 이 방법은 좋은 방법이 아니다. 수많은 긴 이름 때문에 코드가 지저분해질 것이다.

* 1. 패키지의 의미
     1. 패키지의 멤버

패키지의 멤버에는 패키지, 클래스, 인터페이스가 있다.

* + 1. 패키지와 접근

최상위 클래스와 최상위 인터페이스에 public을 붙일 수도 있고 안 붙일 수도 있다.

클래스나 인터페이스의 멤버에는 접근 변경자(access modifier)가 세 개가 있다는 것을 배웠다. public, protected, private이 그것이다. 여기에다 하나를 추가해야 되는데 그것은 접근 변경자 셋 중 하나도 쓰지 않는 것이다. 그것을 디폴트 접근 또는 패키지 접근 또는 freindly 접근이라고 부른다. 이렇게 지정된 멤버 또는 타입은 같은 패키지 안에서는 접근이 가능하고 다른 패키지에서는 불가능하다.

protected 접근은 서브 클래스에서 가능했다. 서브 클래스가 아니더라도 같은 패키지 안에 있으면 protected로 선언된 멤버에 접근이 가능하다.

그러므로 접근에는 가장 개방적인 public에서 가장 폐쇄적인 private에 이르기까지 하나의 계열을 이룬다. 정리해 보면 다음과 같다.

private : 같은 클래스 내에서

default : 같은 클래스 내에서 + 같은 패키지 내에서

protected : 같은 클래스 내에서 + 같은 패키지 내에서 + 서브 클래스에서

public : 같은 클래스 내에서 + 같은 패키지 내에서 + 서브 클래스에서 + 여기에 속하지 않는 다른 곳에서도

protected에 대해서는 이야기가 아직 끝나지 않았다. 자세한 것은 클래스와 인터페이스 – 못다한 이야기를 다루는 장을 참조하라.

이젠 다음과 같은 코드에서 컴파일타임 오류가 안나도 당황하지 않을 것이다.

public class Example {

public void method() {

SuperClass sup = new SuperClass();

// 여기서 에러가 안나는 것이 이상했을 것이다.

// Example 클래스는 SuperClass의 서브 클래스도 아닌데...

// 그러나 둘다 같은 패키지(여기서는 unnamed 패키지)에 속하기 때문에 가능한 것이다.

sup.protectedMember = 123; // OK.

}

}

class SuperClass {

protected int protectedMember;

}

class SubClass extends SuperClass {

public void method() {

protectedMember = 12; // 서브 클래스이니까 쓸 수 있다.

}

}

* + 1. 패키지 계층의 특권 – 없다

패키지는 접근제한과 관련이 있다. 즉 디폴트(default, package) 접근과 protected 접근은 패키지와 관련이 있다. 같은 패키지 안에 있는 클래스들은 더 친한 클래스로 여겨져서 default 접근 멤버와 protected 멤버를 접근할 수 있다. 그러나 그 이상으로 추론하면 안된다. 예를 들어 다음과 같은 클래스가 있다고 하자.

a.b.A;

a.B;

a.c.C;

b.D;

a.b.E;

클래스 A와 클래스 E는 같은 패키지 않에 있으므로 서로 접근(access)에서 특권을 누린다. D와 E는 같은 패키지 안에 있지 않으므로 특권을 누리지 못한다. 문제는 다음과 같이 추론할 수도 있다는 것이다.

C와 A는 모두 패키지 a에 속한다. 그러므로 C는 D에 비해서 A와 더 친하다. 그러므로 C는 A에 접근함에 있어서 D보다 특권을 누려야 한다.

그러나 전혀 그렇지 않다. 같은 패키지가 아니면 패키지 계층도에서 서로 가깝다고 해서 무슨 특권을 누릴 수 있는 것이 아니다. A와 C는 완전히 남남이다. 즉 디폴트 접근이나 protected접근에서 무슨 특권을 기대할 수 없다. 그렇다고 두 클래스가 a에 속한다는 것이 전혀 의미가 없는 것은 아니다. 그것은 문법적인 특권은 없지만 설계상의 의미가 있다. 패키지 계층에서 가까운 위치에 있다면 뭔가 그럴만한 이유가 있어야 한다. 예를 들어

java.awt

java.awt.event

두 패키지는 문법적으로는 남남이지만 둘다 AWT와 관련된 클래스나 인터페이스를 모아놓았다.

* + 1. package의 인터페이스와 구현

클래스에 대해서 배울 때 클래스의 인터페이스와 구현을 나누었다. 이 개념을 패키지까지 확장할 수 있다. 즉 클래스를 하나의 단위로 생각하고 그 단위안에만 알려져야 할 것들과 밖에도 알려져야 할것들로 나누었듯이 패키지도 하나의 단위로 생각하고 그 인터페이스와 구현으로 나누어야 한다.

최상위 타입에는 두 가지의 접근 중 하나를 지정할 수 있는데 public과 default가 그것이다. 이제부터는 몽땅 다 public으로 선언하지 말고 과연 이 클래스 또는 인터페이스가 이 패키지 밖에도 알려져야 하는가를 생각해 보기 바란다. 패키지 밖에 알려져서는 안되는 것이 바로 그 패키지의 구현에 해당하는 것이 될 것이고 그 타입에는 public이라는 키워드를 붙이지 말아야 한다. 이순신 장군의 말처럼 ‘우리 패키지의 죽음을 적에게 알리지 말라.’

위에 든 예 중에서 AbstractArray가 패키지의 인터페이스이고 SubList와 RandomAccessSubList가 패키지의 구현에 해당한다.

1. 중첩 타입
   1. 중첩 타입이란?
      1. 최상위 타입(top level tpye) 과 중첩 타입(nested type)

어떤 타입이 다른 타입 안에 정의된다면 그 안에 정의된 타입을 중첩 타입이라고 한다. 중첩 타입이 아닌 타입을 최상위 타입이라고 한다. 쉬운 말로 하면 우리가 흔히 쓰는 클래스나 인터페이스가 최상위 타입이고 클래스나 인터페이스를 다른 클래스나 인터페이스에 넣어서 세상을 복잡하게 만드는 것이 중첩 타입이다.

중첩 타입을 감싸는 타입을 외부 타입(enclosing type)이라고 한다. 외부 타입은 최상위 타입일 수도 있고 외부 타입도 중첩타입일 수도 있다. 이것은 중첩 타입이 또 중첩될 수 있기 때문이다.

* + 1. 중첩 타입의 종류

중첩타입에는 중첩 클래스와 중첩 인터페이스가 있다.

중첩타입은 클래스 안에 중첩될 수도 있고 인터페이스 안에 중첩될 수도 있다.

중첩타입은 정적(static)일 수도 있고 비정적(non static)일 수도 있다.

그러므로 위의 세가지를 이용하면 다음 여덟가지의 중첩타입을 추론할 수 있다.

클래스 안의 static 중첩 클래스(OK)

클래스 안의 static 중첩 인터페이스(OK)

클래스 안의 non static 중첩 클래스(OK, 이것을 이너 클래스라고 한다)

클래스 안의 non static 중첩 인터페이스(인터페이스는 non static일 수 없다)

인터페이스 안의 static 중첩 클래스(OK)

인터페이스 안의 static 중첩 인터페이스(OK)

인터페이스 안의 non static 중첩 클래스(인터페이스 안에는 non static 중첩 클래스가 정의될 수 없다)

인터페이스 안의 non static 중첩 인터페이스(인터페이스는 non static일 수 없다)

다음 예제를 보자.

class TopLevelClass {

static interface A {}

static class B {}

interface C {} // 디폴트로 static이다.

class D {} // non static nested class. 내부 클래스.

}

interface TopLevelInterface {

static interface E {}

static class F {}

interface G {} // 디폴트로 static이다.

class H {} // 디폴트로 static이다.

}

C, G, H는 static 키워드가 없지만 모두 정적(static)이다. 중첩 인터페이스는 디폴트로 static이며 인터페이스 안에 있는 중첩 타입이 디폴트로 static이기 때문이다. 그러므로 중첩 타입의 종류는 8가지가 아니라. 5가지이다. 즉 다음과 같다.

클래스 안의 static 중첩 클래스(OK)

클래스 안의 static 중첩 인터페이스(OK)

클래스 안의 non static 중첩 클래스(OK, 이것을 내부 클래스inner class라고 한다)

인터페이스 안의 static 중첩 클래스(OK)

인터페이스 안의 static 중첩 인터페이스(OK)

* + 1. 용어에 대해

여기서는 자바를 만든 James Gosling의 용법을 사용했다. 다른 사람은 다른 용법을 사용하기도 한다. 예를 들어 <Core Java>라는 책은 nested란 말 대신에 inner란 말을 쓴다. 다음에 왼쪽에 있는 것이 Gosling의 용법이고 오른쪽에 대응하는 말이 <Core Java>의 용법이다.

중첩 클래스 : 내부(inner) 클래스

중첩 인터페이스 : 내부 인터페이스

정적 중첩 클래스 : 정적 내부 클래스

비정적(non static) 중첩 클래스(내부 클래스) : 비정적(non static) 내부 클래스

* + 1. 중첩 타입의 접근 지정자(access modifier)

최상위 타입에는 접근 지정자가 둘만 쓰일 수 있다. public과 디폴트 접근(package access, friendly access, public키워드가 없는 경우)가 그것이다. 디폴트 접근 에 대해서는 패키지를 다루는 장을 참조하라.

이에 반해 중첩 타입에서는 public, protected, 디폴트 접근(default access), private 모두가 쓰일 수 있다. 중첩 타입은 필드나 메쏘드와 마찬가지로 멤버이다. 그리고 다른 멤버와 마찬가지 방식으로 접근 지정자의 영향을 받는다.

* + 1. 중첩 타입의 유용성

중첩 타입은 이름 충돌 방지에 유용하다. 그리고 비슷한 성질의 타입을 모아두는 역할도 한다. 이것은 패키지의 역할과 비슷하다.

내부 클래스는 여러가지에 쓰인다. AWT의 이벤트 작동에 사용하기도 하고 JavaBeans에도 사용된다. 또는 다른 여러가지 클래스 설계에 쓰일 수 있다.

* + 1. 중첩 타입은 멤버이다

아래에서 배우는 지역 클래스(익명 내부 클래스 포함)를 제외하면 중첩 타입은 멤버이다. 그러므로 필드나 메쏘드처럼 상속이 되며 .을 사용하여 접근할 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

SubOuterClass.NestedInterface ni;

ni = new SubOuterClass.NestedClass();

}

}

class SuperOuterClass {

interface NestedInterface {}

}

class SubOuterClass extends SuperOuterClass {

public static class NestedClass implements NestedInterface {}

}

NestedClass는 SubOuterClass의 멤버이기 때문에

ni = new SubOuterClass.NestedClass();

와 같이 SubOuterClass다음에 .을 찍고 NestedClass라고 쓸 수 있다.

public static class NestedClass implements NestedInterface {}

에서의 NestedInterface처럼 그 멤버가 속한 타입 안에서는 간단한 이름(simple name)을 사용할 수도 있다. 위의 코드를 아래와 같이 길게 쓸 수도 있었다. 이것은 필드나 메쏘드와 마찬가지이다.

public static class NestedClass implements SuperOuterClass.NestedInterface {}

또는

public static class NestedClass implements SubOuterClass.NestedInterface {}

그리고 NestedInterface는 SubOuterClass에 선언되지 않았음에도 아래와 같이 쓸 수 있는 것은 SuperOuterClass에 선언된 것을 상속받았기 때문이다.

SubOuterClass.NestedInterface ni;

그리고 중첩 타입은 멤버이므로 멤버에 적용될 수 있는 접근 지정자(access modifier)를 사용할 수 있다. 이것은 중첩 타입을 private으로 함으로써 숨길 수 있다는 것을 뜻한다. 아래에서 볼 수 있듯이 몇몇 경우에는 접근 변경자로 접근을 제한할 수 없다.

* 1. 인터페이스 안에서의 중첩 타입
     1. 중첩 인터페이스(nested interface)

인터페이스 안에 인터페이스를 정의할 수 있다. 다음 예에서 볼 수 있듯이 인터페이스안에 중첩된 인터페이스는 public이어야 한다.

interface TopLevelInterface {

interface NestedInterfaceA {} // default가 아니라 public 이다.

public interface NestedInterfaceB {}

protected interface NestedInterfaceC {} // Error. protected 일 수 없다.

private interface NestedInterfaceD {} // Error. private 일 수 없다.

}

* + 1. 중첩 클래스(nested class)

인터페이스 안에 클래스를 정의할 수 있다. 이것은 보통 반환값(return value)이나 매개변수가 그 인터페이스에만 쓰이는 특이한 것일 때 쓰인다. 다음 예제를 보자.

interface SomeInterface {

IntegerAndString getIntegerAndString();

class IntegerAndString {

public int intValue;

public String strValue;

}

}

인터페이스 안에 중첩된 인터페이스와 마찬가지로 인터페이스 안에 중첩된 클래스도 public이어야 한다. 다음 예제를 보자.

interface TopLevelInterface {

class NestedClassA {} // default가 아니라 public 이다.

public class NestedClassB {}

protected class NestedClassC {} // Error. protected 일 수 없다.

private class NestedClassD {} // Error. private 일 수 없다.

}

* 1. 클래스 안에서의 중첩 타입
     1. 중첩 인터페이스(nested interface)

클래스 안에 중첩된 인터페이스는 여러 가지 접근 지정자를 사용할 수 있다. 다음 예제는 잘 컴파일될 것이다.

class OuterClass {

public interface NestedInterfaceA {}

protected interface NestedInterfaceB {}

interface NestedInterfaceC {}

private interface NestedInterfaceD {}

}

중첩 인터페이스는 최상위 인터페이스와 거의 비슷하다.

* + 1. 정적 중첩 클래스(static nested class)

클래스 안에 중첩된 정적 클래스도 여러 가지 접근 지정자를 사용할 수 있다. 다음 예제는 잘 컴파일 될 것이다.

class OuterClass {

public class NestedClassA {}

protected class NestedClassB {}

class NestedClassC {}

private class NestedClassD {}

}

중첩 인터페이스와 마찬가지로 정적 중첩 클래스는 최상위 클래스와 거의 비슷하다. 정적 중첩 클래스는 패키지와 거의 비슷한 역할을 한다. 즉 이름 충돌을 방지하고 비슷한 기능을 가진 클래스를 모을 수 있는 것이다.

* + 1. 내부 클래스(inner class)

중첩 타입 중에 내부 클래스 즉 비정적(non static) 중첩 클래스가 가장 복잡하다. 이너 클래스에 대해서는 바로 뒤에 다룬다.

* 1. 내부 클래스
     1. 이너 클래스(inner class)와 외부 클래스(outer class, enclosing class)의 관계

이너 클래스는 클래스 안에 정의된 비정적(non static) 중첩 클래스이다. 이것은 다른 중첩 타입과 다른 점이 있는데 그것은 외부 클래스의 객체와 연결되어 있다는 것이다. 하나의 내부 클래스 객체는 어떤 하나의 외부 클래스 객체에 연결된다(예외가 있긴 한데 그것은 클래스와 인터페이스 – 못다한 이야기를 다룬 장을 참조하라). 그러나 외부 클래스 객체(외부 객체, enclosing object)는 0개 1개 또는 여러개의 내부 클래스 객체와 연결되어 있을 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

OuterClass ocZero = new OuterClass("Zero");

OuterClass ocOne = new OuterClass("One");

OuterClass.InnerClass icA = ocOne.new InnerClass("A");

OuterClass ocThree = new OuterClass("Three");

OuterClass.InnerClass icB = ocThree.new InnerClass("B");

OuterClass.InnerClass icC = ocThree.new InnerClass("C");

OuterClass.InnerClass icD = ocThree.new InnerClass("D");

System.out.println(icA.innerValue + " - " + icA.getOuterValue());

System.out.println(icB.innerValue + " - " + icB.getOuterValue());

System.out.println(icC.innerValue + " - " + icC.getOuterValue());

System.out.println(icD.innerValue + " - " + icD.getOuterValue());

}

}

class OuterClass {

public String outerValue;

public OuterClass(String str) {

outerValue = str;

}

public class InnerClass {

public String innerValue;

public InnerClass(String str) {

innerValue = str;

}

public String getOuterValue() {

return outerValue;

}

}

}

결과는 다음과 같다.

A - One

B - Three

C - Three

D - Three

이것을 그림으로 보면 그림 중첩 타입 – 1과 같다.

----------------------------- 그림 중첩 타입 – 1 ---------------------------------

이너 클래스의 객체는 다음과 같이 생성한다.

OuterClass.InnerClass icA = ocOne.new InnerClass("A");

먼저 ocOne과 같이 외부 클래스의 객체를 쓰고 .을 찍고 new를 사용한다. 외부 클래스를 명시적으로 지정하지 않을 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

OuterClass oc = new OuterClass("Outer");

System.out.println(oc.innerObjectA.innerValue + " - " + oc.innerObjectA.getOuterValue());

System.out.println(oc.innerObjectB.innerValue + " - " + oc.innerObjectB.getOuterValue());

}

}

class OuterClass {

public String outerValue;

public InnerClass innerObjectA;

public InnerClass innerObjectB;

public OuterClass(String str) {

outerValue = str;

innerObjectA = new InnerClass("InnerA");

innerObjectB = this.new InnerClass("InnerB");

}

public class InnerClass {

public String innerValue;

public InnerClass(String str) {

innerValue = str;

}

public String getOuterValue() {

return outerValue;

}

}

}

결과는 다음과 같다.

InnerA - Outer

InnerB – Outer

아래 두 라인을 보자.

innerObjectA = new InnerClass("InnerA");

innerObjectB = this.new InnerClass("InnerB");

둘은 동등한 일을 한다. 즉 현재 객체(this)와 연결된 내부 클래스의 객체를 생성한다. 외부 객체가 명시적으로 지정되지 않는다면 this를 지정하는 것과 같다. 다음 예에서 보듯이 this가 외부 객체로 적당하지 않을 때는 컴파일타임 오류가 난다.

public class Example {

public void method() {

OuterClass.InnerClass ic;

ic = new InnerClass(); // Error. 여기서 this는 Example이다. 즉 OuterClass가 아니다.

}

}

class OuterClass {

public class InnerClass {

}

}

* + 1. 내부 클래스의 구현

자바의 중첩 타입은 JDK 버젼 1.1에 추가되었다. 언어의 새로운 측면이 추가될 때 한가지 선택을 해야 할 것이 있었다. 그것은 JDK 1.0 버전의 가상머신에서 언어의 새로운 측면을 쓸 수 있게 하느냐 하는 것이다. 사람들은 호환성을 유지하기로 하였다. 그래서 중첩 타입을 쓴 코드는 소스 코드를 변환해서(source code transformation) 최상위 타입으로 바꾸어 버린 것이다. 따라서 내부 클래스도 최상위 클래스로 변환된다. 이것은 보안에 문제가 발생하게 한다(보안에 대한 것은 다음에 다루기로 한다). 이런식으로 구현하는 것이 미래에는 바뀔지도 모른다. 하여튼 자바 컴파일러가 무슨 일을 하는가를 아는 것은 내부 클래스의 이해에 도움이 된다. 다음 예제를 보자. 먼저 다음 예제를 컴파일해 보자.

public class Example {

public static void main(String[] args) {

OuterClass oc = new OuterClass();

OuterClass.InnerClass ic = oc.new InnerClass();

System.out.println(ic.getOuterClassPrivateValue());

ic.setOuterClassPrivateValue(999);

System.out.println(ic.getOuterClassPrivateValue());

}

}

class OuterClass {

private int privateValue = 123;

public class InnerClass {

public void setOuterClassPrivateValue(int value) {

privateValue = value;

}

public int getOuterClassPrivateValue() {

return privateValue;

}

}

}

그러면 Example.class, OuterClass.class, OuterClass$InnerClass.class가 생성될 것이다. 이 중에 OuterClass와 OuterClass$InnerClass를 디컴파일(decompile)해보자. 여기서는 윈도우즈 운영체제에서 Jad.exe를 이용했다. –noinner는 내부 클래스임을 고려하지 않고 있는 그래로 디컴파일하라는 뜻이다. 자세한 것은 자신이 가지고 있는 디컴파일러의 문서를 참조하면 될 것이다.

C:\ decompiler\Jad>jad -noinner OuterClass

C:\ decompiler\Jad>jad -noinner OuterClass$InnerClass

그러면 두 개의 파일이 생성될 것이다. 두 개의 파일을 합치면 다음과 비슷할 것이다.

// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.

class OuterClass

{

OuterClass()

{

privateValue = 123;

}

static int access$0(OuterClass x$0, int x$1)

{

return x$0.privateValue = x$1;

}

static int access$1(OuterClass x$0)

{

return x$0.privateValue;

}

private int privateValue;

}

public class OuterClass$InnerClass

{

public OuterClass$InnerClass(OuterClass this$0)

{

this.this$0 = this$0;

}

public void setOuterClassPrivateValue(int value)

{

OuterClass.access$0(this$0, value);

}

public int getOuterClassPrivateValue()

{

return OuterClass.access$1(this$0);

}

private final OuterClass this$0; /\* synthetic field \*/

}

자바 컴파일러가 하는 일을 묘사하면 다음과 같다. 먼저 우리가 쳐 넣은 소스 코드를 바로 위에 디컴파일된 소스 코드로 변환한다(source code transformation). 그 다음에 변환된 소스 코드를 컴파일한다.

OuterClass는 그대로이고 OuterClass의 내부 클래스인 InnerClass는 OuterClass$InnerClass로 클래스 이름이 바뀌었다. 이것을 OuterClass.InnerClass와 헷갈리지 말기 바란다. OuterClass.InnerClass는

OuterClass

.

InnerClass

이렇게 세 개의 토큰으로 이루어져 있다. 반면 OuterClass$InnerClass는 하나의 토큰 즉 하나의 식별자이다. 단지 식별자에 쓰인 문자중 하나인 $가 낯선 문자일 뿐이다. 예를들어 OuterClass\_InnerClass와 같이 할 수도 있었다. 그렇게 하지 않은 이유는 아래 어딘가에 설명된다.

이제 하나하나 분석해 보자. 먼저 두 개의 변환된 클래스는 모두 최상위 클래스이다. 왜냐하면 JDK 1.0에서는 중첩 타입이 없었으므로 1.0 버젼의 가상 머신이 이해하게 하기 위해서는 중첩타입은 최상위 타입으로 바꾸어주어야 하기 때문이다.

그렇게 되면 한가지 문제가 발생한다. 바로 InnerClass에서 OuterClass의 private 필드인 privateValue에 접근한다는 것이다. 최상위 클래스에서는 절대로 다른 최상위 클래스의 private 멤버에 접근할 수 없다. 그래서 편법을 쓰게 되었다. 그것이 무엇인가 하면 패키지 접근의 메쏘드를 private 필드당 두 개씩 만드는 것이다. 하나는 그 값을 알아내는 것(access$1 메쏘드)이고 다른 하나는 그 값을 설정하는 것(access$0 메쏘드)이다. 이너 클래스는 외부 클래스와 같은 소스 파일에 들어가기 때문에 당연히 같은 패키지 안에 있다. 그래서 이 편법이 통하는 것이다. 그리고

privateValue = value;

는

OuterClass.access$0(this$0, value);

로 변환되고

return privateValue;

는

OuterClass.access$1(this$0);

로 변환된다. 그리하여 OuterClass$InnerClass는 OuterClass의 private 멤버에 접근할 수가 있었다.

또 하나의 문제점은 내부 클래스는 외부 클래스와 연결되어 있다는 것이다. 정확히 말하면 내부 클래스의 객체는 외부 클래스의 객체와 연결되어 있어야 한다. 이것은 내부 클래스에 하나의 필드가 있음으로써 가능하다. 그것이 바로

private final OuterClass this$0; /\* synthetic field \*/

이다. 그리고 이것은 생성자에서 설정된다. 그러므로 생성자는

public OuterClass$InnerClass(OuterClass this$0)

{

this.this$0 = this$0;

}

와 같이 변환되었다. 실제로 생성자가 어떻게 쓰이는지는 Example.class를 디컴파일하면 알 수 있다. 디컴파일하면 다음과 비슷할 것이다.

// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.

// Jad home page: http://www.geocities.com/kpdus/jad.html

public class Example

{

public Example()

{

}

public static void main(String args[])

{

OuterClass oc = new OuterClass();

OuterClass$InnerClass ic = new OuterClass$InnerClass(oc);

System.out.println(ic.getOuterClassPrivateValue());

ic.setOuterClassPrivateValue(999);

System.out.println(ic.getOuterClassPrivateValue());

}

}

생성자를 쓰는 부분은 다음이다.

OuterClass$InnerClass ic = new OuterClass$InnerClass(oc);

다음 예제는 private 메쏘드를 접근하는 예제이다. 원리는 필드와 같다.

원래 코드

class OuterClass {

private int privateMethod(int value) {

return value \* value;

}

public class InnerClass {

public int callOuterClassPrivateMethod(int value) {

return privateMethod(value);

}

}

}

변환된 코드

// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.

// Jad home page: http://www.geocities.com/kpdus/jad.html

class OuterClass

{

OuterClass()

{

}

private int privateMethod(int value)

{

return value \* value;

}

static int access$0(OuterClass x$0, int x$1)

{

return x$0.privateMethod(x$1);

}

}

public class OuterClass$InnerClass

{

public OuterClass$InnerClass(OuterClass this$0)

{

this.this$0 = this$0;

}

public int callOuterClassPrivateMethod(int value)

{

return OuterClass.access$0(this$0, value);

}

private final OuterClass this$0; /\* synthetic field \*/

}

* + 1. 내부 클래스의 문제점 – 보안의 구멍과 복잡성

어떤 최상위 클래스가 어떤 최상위 클래스의 private 멤버를 접근할 수 있도록 뒷구멍을 만들어 놓음으로써 이너 클래스를 JDK 1.0의 가상머신에서도 돌도록 했다. 이 뒷구멍은 곧 보안의 구멍(security hole)이 될 수도 있다. 누군가가 나쁜 마음을 먹고 access$0과 같이 생긴 메쏘드에 접근할 수도 있는 것이다. 이것은 약간은 어렵다. 왜냐하면 $는 식별자에 쓰일 수 없는 문자여서 코드 중에

OuterClass.access$0(this$0, value);

와 같은 것을 넣으면 컴파일러가 싫어할 것이기 때문이다. 사실 $ 문자를 사용한 이유는 실수로 내부 클래스에서 만든 뒷구멍에 들어가지 못하게 하기 위해서이다. 그러나 자바의 class 파일의 형식(format)을 아는 사람이라면 헥스 에디터(hex editer) 같은 것으로 조작하기는 그리 고도의 기술을 요하는 것은 아니다. 아주 짜증나는 단순 노동을 해야겠지만. 이런 문제 때문에 언젠가는 내부 클래스를 (편법을 쓰지 않고) 언어 수준에서 제대로 지원할 때가 올지도 모른다.

내부 클래스 또는 넓게는 중첩 타입의 또 다른 문제점은 이 장을 읽는 사람은 누구나 느끼듯이 복잡하다는 것이다. 자바의 철학 중의 하나는 단순성이다. 그래서 C++의 많은 문법을 포기했다. 단순성은 커다란 장점이다. 내부 클래스는 여러가지 유용성이 있지만 자바에 복잡성을 추가했다.

자바에 내부 클래스를 추가함으로써 자바가 할 수 있는 일이 많아졌지만 보안과 복잡성의 문제도 생겼다. 어찌되었든 우리는 내부 클래스에서 벗어나지는 못할 것이다. 이미 많은 사람들이(여기에는 자바 표준 API를 만드는 사람들도 포함된다) 내부 클래스를 사용하기 때문이다.

* + 1. 지역 클래스(local class)

내부 클래스는 블럭(block, 클래스나 인터페이스를 정의하는 {}는 블럭이 아니다) 안에서도 쓰일 수 있다. 이것을 지역 클래스라 한다. 다음 예제를 보자. 내부 클래스를 이용한다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

SimpleArrayList sal = new SimpleArrayList(5);

for (int i = 0; i < sal.getLength(); i++)

sal.set(i, new String("" + i + (i+1) + (i+2) + (i+3) + (i+4)));

for (Iterator i = sal.getIterator(); i.hasNext(); ) {

System.out.println(i.next());

}

}

}

class SimpleArrayList {

Object[] array;

public SimpleArrayList(int size) {

array = new Object[size];

}

public void set(int index, Object o) {

array[index] = o;

}

public int getLength() {

return array.length;

}

public Iterator getIterator() {

return new SimpleArrayListIterator();

}

class SimpleArrayListIterator implements Iterator {

private int index = 0;

public boolean hasNext() {

if (index < array.length)

return true;

else

return false;

}

public Object next() {

Object oReturn = array[index];

index++;

return oReturn;

}

public void remove() {

throw new UnsupportedOperationException();

}

}

}

결과는 다음과 같다.

01234

12345

23456

34567

45678

이번에는 지역 클래스를 이용해 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

SimpleArrayList sal = new SimpleArrayList(5);

for (int i = 0; i < sal.getLength(); i++)

sal.set(i, new String("" + i + (i+1) + (i+2) + (i+3) + (i+4)));

for (Iterator i = sal.getIterator(); i.hasNext(); ) {

System.out.println(i.next());

}

}

}

class SimpleArrayList {

Object[] array;

public SimpleArrayList(int size) {

array = new Object[size];

}

public void set(int index, Object o) {

array[index] = o;

}

public int getLength() {

return array.length;

}

public Iterator getIterator() {

class SimpleArrayListIterator implements Iterator {

private int index = 0;

public boolean hasNext() {

if (index < array.length)

return true;

else

return false;

}

public Object next() {

Object oReturn = array[index];

index++;

return oReturn;

}

public void remove() {

throw new UnsupportedOperationException();

}

}

return new SimpleArrayListIterator();

}

}

SimpleArrayListIterator 클래스는 getIterator 메쏘드 이외에서는 쓸 수 없다. 즉 이름을 접근할 수 없는 것이다. 지역 클래스가 아닐 때는 외부에서도 SimpleArrayList.SimpleArrayListIterator라고 이름을 붙일 수가 있었지만 지역 클래스가 되면서 이런식으로 이름 붙이는 것이 불가능해졌다. 그러므로 지역 클래스는 어떤 블럭(메쏘드, 생성자 등)에서만 쓰일 때 쓰면 좋다. 다른 곳에서는 아예 이름을 쓸 수가 없으니 신경을 쓰지 않아도 되는 이점이 있다.

* + 1. 익명 내부 클래스(anonymous inner class)

지역 클래스에는 클래스의 이름을 붙이지 않아도 된다. 이렇게 이름이 없는 클래스를 익명 내부 클래스라 한다. 위의 예를 익명 내부 클래스로 바꾸어 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

SimpleArrayList sal = new SimpleArrayList(5);

for (int i = 0; i < sal.getLength(); i++)

sal.set(i, new String("" + i + (i+1) + (i+2) + (i+3) + (i+4)));

for (Iterator i = sal.getIterator(); i.hasNext(); ) {

System.out.println(i.next());

}

}

}

class SimpleArrayList {

Object[] array;

public SimpleArrayList(int size) {

array = new Object[size];

}

public void set(int index, Object o) {

array[index] = o;

}

public int getLength() {

return array.length;

}

public Iterator getIterator() {

return new Iterator() {

private int index = 0;

public boolean hasNext() {

if (index < array.length)

return true;

else

return false;

}

public Object next() {

Object oReturn = array[index];

index++;

return oReturn;

}

public void remove() {

throw new UnsupportedOperationException();

}

}; // 여기가 return문의 끝이다. 그래서 세미 콜론이 필요하다.

}

}

new 다음에 수퍼 타입의 이름 그리고 그 다음에 ()가 오고 그 다음에 클래스를 정의해 준다. 이전까지는 클래스를 정의하고 그 이름을 이용해서 new문으로 클래스의 객체를 생성했는데 여기서는 그것이 동시에 이루어진다.

* 1. 중첩 타입 사용에 대한 조언
     1. 익명 내부 클래스를 남용하지 말라

익명 내부 클래스는 쓰기에 편리하지만 알아보기가 힘들다. 그러므로 익명 내부 클래스는 정말 짧은 코드에서만 사용하자. 다음 예를 보자. 이것은 윈도우의 닫기 버튼을 누르면 프로그램이 종료되게 하는 코드이다. 들여쓰기도 아래와 같이 알아보기 쉽게 하는 것이 좋다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame {

public static void main(String[] args) {

Frame f = new Example();

f.addWindowListener(

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

f.show();

}

}

* + 1. 중첩을 많이 하지 말라

중첩 클래스 안에 중첩 클래스를 쓰는 것(이중 중첩)은 거의 안 쓰인다. 더구나 더 많은 중첩은 피해야 한다. 중첩에 중첩을 거듭하면 정말 알아보기 힘든 코드가 된다.

1. 클래스와 인터페이스 – 못다한 이야기
   1. main 메쏘드
      1. entry point

main 메쏘드를 entry point라고 한다. 프로그램은 항상 여기서부터 시작한다.

java.exe를 사용해서 프로그램을 띄울때 우리는 항상 main 메쏘드가 있는 클래스를 지정해준다.

하나의 프로그램은 여러 개의 클래스들로 구성된다. 보통 그 중에 하나의 클래스에 main 메쏘드가 있다. 프로그램이 실행되는 길은 보통 하나인 것이다. 그러나 main 메쏘드가 반드시 프로그램 중에 하나의 클래스에만 있을 필요는 없다. 심지어 모든 클래스에 main 메쏘드를 만들어서 프로그램으로 들어가는 길을 다양하게 만들 수도 있다.

이런 식의 방식은 완성된 프로그램에서는 쓰이지 않으나 프로그램을 개발하는 과정에 즉 테스트하는 과정에 사용하면 유용하다. 하나의 클래스가 제대로 작동하는지 테스트하기 위해 그 클래스에 main 메쏘드를 만들어서 그 클래스 자체에서 테스트하는 것이다. 여러 클래스들이 연결되어 복잡해지기 전에 하나의 클래스를 테스트하면 버그를 훨씬 쉽게 잡을 수 있다.

* + 1. public static void main(String[] args)

main 메쏘드가 public인 이유는 가상 머신에서 실행할 수 있어야 하기 때문이다. 즉 가상 머신에서 접근할 수 있어야 한다. main 메쏘드가 static인 이유는 main 메쏘드가 속해 있는 클래스의 인스턴스를 만들지 않아도 되게 하기 위해서이다. main 메쏘드가 void인 이유는 반환값이 없기 때문이다.

System.exit메쏘드를 프로그램을 강제로 종료하는 메쏘드이다. 인자를 이용해 프로그램의 종료 상태를 나타낼수 있다. 그러면 그 프로그램을 부른 운영체제 등이 그 정보를 이용할 수 있을 것이다.

이번에는 String[]형인 매개변수에 대해 알아보자. 이 매개변수에는 실행을 할 때 명령행 인자(command line argument)를 얻는데 쓰인다. 이 때 클래스 이름은 제외된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(args);

System.out.println(args.length);

for (int i = 0; i < args.length; i++)

System.out.println(args[i]);

}

}

위의 예제를

C:\Java>java Example first second third

와 같이 실행하면 결과는 다음과 비슷할 것이다.

[Ljava.lang.String;@5efa1a

3

first

second

third

위의 예제를

C:\Java>java Example

와 같이 실행하면 결과는 다음과 비슷할 것이다.

[Ljava.lang.String;@5efa1a

0

크기가 0인 String 형 배열로 설정된다.

* 1. 상속
     1. 상속의 순환(cyclic inheritance)

상속은 순환할 수 없다. 그러므로 다음 예제는 컴파일되지 않을 것이다.

class A extends C {

}

class B extends A {

}

class C extends B {

}

인터페이스도 마찬가지이다. 다음 예제도 컴파일 되지 않을 것이다.

interface A extends C {

}

interface B extends A {

}

interface C extends B {

}

만약 상속의 순환이 컴파일 타임에 발견되지 않고 런타임에 발견되면 ClassCircularityError가 발생한다. 네트워크로 서로 독립적으로 컴파일된 클래스들을 로딩하다 보면 그런 일이 생길 수도 있을 것이다.

* + 1. 여러 갈래로 상속받는 인터페이스

다음 예제를 보자.

interface GrandMother {}

interface Uncle extends GrandMother {}

interface Aunt extends GrandMother {}

interface Cousin extends Uncle, Aunt {}

GrandMother 클래스가 Cousin 클래스에 두 갈래로 상속된다. 이런 경우 문제가 되지 않는다. 컴파일이 잘 될 것이다.

* 1. 필드
     1. 지정자

필드 지정자 (field modifier)에는

public protected private static final transient volatile

가 있다. 여러개의 지정자가 한꺼번에 사용될 때에는 위에 나열된 순서로 쓰는 것이 좋다. 그렇게 하지 않아도 컴파일타임 오류는 나지 않지만 다음과 같이 쓰면 보는 사람이(그 사람이 정말 자바를 할 줄 안다면) 짜증을 낼 것이다.

final static public int 너\_죽을래 = 444;

public, protected, private과 default는 상호 배타적이다. 즉 public이면서 private일 수는 없다.

volatile에 대해서는 쓰레드를 다루는 부분을 참조하라.

transient에 대해서는 직력화(serialization)을 다루는 부분을 참조하라.

* + 1. 필드 초기화문

필드 초기화문(field initialization clause)에는 정적 필드 초기화문(static field initialization clause, initializer for class variable)과 인스턴스 필드 초기화문(instance field initialization clause, initializer for instance variable)이 있다.

정적 필드 초기화문에는 this나 super 키워드가 올 수 없다. 정적 필드는 객체와 연결되지 않기 때문이다. 반면 인스턴스 필드 초기화문에는 this나 super를 사용할 수 있다. 다음 예제를 보자.

class SuperClass {

static int superField = 123;

}

class SubClass extends SuperClass {

static int a = this.superField; // Error.

static int b = super.superField; // Error.

}

컴파일이 되지 않을 것이다. 반면에 다음 예제를 보자.

class SuperClass {

static int staticField = 123;

int instanceField = 987;

}

class SubClass extends SuperClass {

int a = this.staticField; // OK. but bad style.

int b = super.staticField; // OK. but bad style.

int c = SuperClass.staticField; // OK.

int d = this.instanceField; // OK. but bad style.

int e = super.instanceField; // OK.

int f = instanceField; // OK.

}

컴파일이 잘 될 것이다.

* + 1. 정적 초기화 블록(static initializer, static initialization block)

정적 필드는 보통 선언하면서 초기화한다. 즉 다음과 같이 한다.

class Example {

public static final int ONE = 1;

public static final int ONE\_PLUS\_TWO = 1 + 2;

private static int someHardlyCalculatedValue = calculateHardly();

private static int calculateHardly() {

int some = 1;

int hardly = 2;

int calculated = 3;

int value = 4;

return some+hardly+calculated+value;

}

}

그러나 이렇게 해결이 안되거나 다른 식으로 하는 것이 더 편할 경우가 있다. 다음 예를 보자.

class Example {

private static A a;

private static B b;

static { // 정적 초기화 블록

a = new A();

b = new B();

a.setB(b);

b.setA(a);

}

}

class A {

private B b;

public void setB(B b) {

this.b = b;

}

}

class B {

private A a;

public void setA(A a) {

this.a = a;

}

}

정적 초기화 블록에는 return문이 올 수 없다. 그리고 검사되는 예외상황(checked exception)을 던지는 것(메쏘드 호출, 생성자 호출)이 올 수는 있지만 반드시 try문으로 처리해 주어야 한다. 그러므로 다음 예제는 컴파일타임 오류가 날 것이다.

class Example {

static {

throwingMethod(); // Error.

try {

throwingMethod(); // OK.

} catch (Exception e) {

}

return; // Error.

}

static void throwingMethod() throws Exception {

}

}

정적 초기화 블록에 왜 return문이 올 수 없게 만들었는지는 잘 모르겠다. 검사되지 않는 예외상황(unchecked exception)을 반드시 try문으로 처리해야 하는 이유는 다음과 같다. 메쏘드에서는 그렇게 처리해 주지 않아도 된다. 처리하지 않으려면 throws를 이용해서 자신도 그 예외 상황을 던질(throw) 수 있음을 알리면 된다. 그러나 정적 초기화 블록에서는 throws를 사용할 수 없다.

정적 초기화 블록이 정적(static)이기 때문에 this나 super가 올 수도 없다. 다음 예제를 보자. 컴파일이 되지 않을 것이다.

class Super {

static int a = 123;

}

class Sub extends Super{

static int b = 999;

static {

System.out.println(this.b); // Error.

System.out.println(super.a); // Error.

return; // Error.

}

}

정적 초기화 블록은 멤버가 아니다. 그러므로 상속되지도 않는다.

* + 1. 인스턴스 초기화 블록(instance initializer, instance initialization block)

인스턴스 필드를 초기화 할 수 있는 곳은 세 군데다. 첫째로 선언하면서 동시에 초기화 할 수 있으며 둘째로 생성자에서 초기화 할 수 있고 셋째가 인스턴스 초기화 블록이다. 인스턴스 초기화 블록은 어떤 생성자를 사용하든 항상 생성자가 불리기 전에 먼저 불리게 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new A();

System.out.println();

new A(1);

System.out.println();

new A(1, 1);

}

}

class A {

public A() {

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

}

public A(int a) {

System.out.println("A(int)");

}

public A(int a, int b) {

System.out.println("A(int, int)");

}

{ // 인스턴스 초기화 블록

System.out.println("instance initializer");

}

}

결과는 다음과 같다.

instance initializer

A()

instance initializer

A(int)

instance initializer

A(int, int)

인스턴스 초기화 블록은 모든 생성자에 공통되는 부분이 있을 때 사용한다. 다음 예제를 보자.

class A {

private String[] strArray;

private int x;

private int y;

public A() {

strArray = new String[3];

strArray[0] = "E";

strArray[1] = "tHe";

strArray[2] = "Carr";

x = 0;

y = 0;

}

public A(int x) {

strArray = new String[3];

strArray[0] = "E";

strArray[1] = "tHe";

strArray[2] = "Carr";

this.x = x;

this.y = 0;

}

public A(int x, int y) {

strArray = new String[3];

strArray[0] = "E";

strArray[1] = "tHe";

strArray[2] = "Carr";

this.x = x;

this.y = y;

}

}

인스턴스 초기화 블록을 이용해서 바꾸었다.

class A {

private String[] strArray;

private int x;

private int y;

{

strArray = new String[3];

strArray[0] = "E";

strArray[1] = "tHe";

strArray[2] = "Carr";

}

public A() {

this(0, 0);

}

public A(int x) {

this(x, 0);

}

public A(int x, int y) {

this.x = x;

this.y = y;

}

}

거의 대부분의 경우에 인스턴스 초기화 블록의 사용은 생성자의 사용으로 대체할 수 있다. 위의 예제를 다음과 같이 바꾸었다.

class A {

private String[] strArray;

private int x;

private int y;

public A() {

this(0, 0);

}

public A(int x) {

this(x, 0);

}

public A(int x, int y) {

strArray = new String[3];

strArray[0] = "E";

strArray[1] = "tHe";

strArray[2] = "Carr";

this.x = x;

this.y = y;

}

}

결론적으로 인스턴스 초기화 블록을 사용할 것인가 말것인가는 대부분의 경우 스타일의 문제이다.

인스턴스 초기화 블록에서는 return문을 사용할 수 없다. 그러므로 다음 예제는 컴파일 되지 않을 것이다.

public class Example {

{

return; // Error.

}

}

그리고 this, super를 사용할 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Sub();

}

}

class Super {

int a = 123;

}

class Sub extends Super{

int b = 999;

{

System.out.println(this.b);

System.out.println(super.a);

}

}

결과는 다음과 같을 것이다.

999

123

인스턴스 초기화 블록은 멤버가 아니다. 그러므로 상속되지도 않는다. 이것은 생성자가 멤버가 아니라는 사실과 일맥상통한다.

* + 1. final필드를 사용해야 하는 경우

final 필드를 사용해야 하는 경우에 대해 알아보자. 먼저 그 값은 한번 설정하면 절대 변하지 않는 값이어야 한다. 아주 가끔 변하더라도 변하는 것은 변하는 것이기 때문에 final을 사용하면 안된다. 또 하나는 (인스턴스 필드일 경우에는) 적어도 생성자의 실행을 마칠 때까지 그 값을 정할 수 있어야 한다. 한번 설정하고 다시는 변하지 않는 값이더라도 생성자가 끝날때까지 값을 설정할 수 없다면 final을 사용할 수 없다.

위의 조건이 성립하더로도 final을 사용하지 않는 것이 좋을 때도 있다. 예를 들어 그 필드를 설정하는 것이 너무 많은 비용(실행시간, 메모리 등)이 들고 그 필드가 거의 사용되지 않는 경우이다. 그 때에는 그 필드가 필요할 때 설정하는 것이 performance를 위해서 좋다. 이렇게 하는 것을 lazy initialization이라고 한다. 이럴 때 물론 final을 사용할 때에 얻을 수 있는 안정성을 잃지만 performance에서 이익을 보는 것이다.

어떤 필드가 한번 설정하면 절대로 바뀌지 않음에도 생성자가 끝날때까지도 그 값을 설정할 수 없을 때는 final 키워드를 사용할 수 없다. 이 때에는 다음과 같이 하는 방법이 있다. 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

try {

System.out.println(a.getSemiFinal());

} catch (A.NotSetException e) {

System.out.println("아직 semiFinal이 설정되지 않았다.");

}

try {

a.setSemiFinal(123);

} catch (A.AlreadySetException e) {

System.out.println("semiFinal은 이미 설정된 상태다.");

}

try {

System.out.println(a.getSemiFinal());

} catch (A.NotSetException e) {

System.out.println("아직 semiFinal이 설정되지 않았다.");

}

try {

a.setSemiFinal(987);

} catch (A.AlreadySetException e) {

System.out.println("semiFinal은 이미 설정된 상태다.");

}

}

}

class A {

private int semiFinal;

private boolean semiFinalSet = false;

public static class AlreadySetException extends Exception {}

public static class NotSetException extends Exception {}

public void setSemiFinal(int i) throws AlreadySetException {

if (semiFinalSet) {

throw new AlreadySetException();

} else {

semiFinal = i;

semiFinalSet = true;

}

}

public int getSemiFinal() throws NotSetException {

if (semiFinalSet)

return semiFinal;

else

throw new NotSetException();

}

}

결과는 다음과 같을 것이다.

아직 semiFinal이 설정되지 않았다.

123

semiFinal은 이미 설정된 상태다.

* + 1. blank final

final 필드 중에서 필드를 선언할 때 초기화하지 않는 필드를 blank final 필드라고 한다. 때에 따라서 그렇게 밖에 할 수 없을 때가 있다. 다음 예를 보자. 생성자의 인수로 설정하기 때문에 생성자에서 설정해야만 한다.

public class Example {

private final int blankFinalValue;

public Example(int value) {

blankFinalValue = value;

}

}

static blank final 필드는 적어도 스태틱 초기화 블록에서는 초기화를 해 주어야 한다. 즉 다음과 같이 해 주어야 한다.

public class Example {

private static final int staticBlankFinalValue;

static {

staticBlankFinalValue = 12; // OK.

}

}

스태틱 필드를 생성자에서 초기화 해주는 것는 너무 늦는다. 스태틱 필드는 객체의 생성할 때가 아니라 클래스가 로딩될 때 초기화된다.

public class Example2 {

private static final int staticBlankFinalValue;

public Example2(int value) {

staticBlankFinalValue = value; // Error. 너무 늦었다.

}

}

인스턴스 필드는 적어도 생성자에서는 초기화해 주어야 한다. 다음 예를 보자.

class Example {

private final int instanceBlankFinalValue;

static {

instanceBlankFinalValue = 12; // Error. 여기엔 인스턴스 필드가 올 수 없다.

}

}

class Example2 {

private final int instanceBlankFinalValue;

{

instanceBlankFinalValue = 12; // OK.

}

}

class Example3 {

private final int instanceBlankFinalValue;

public Example3(int value) {

instanceBlankFinalValue = value; // OK.

}

}

class Example4 {

private final int instanceBlankFinalValue; // Error. 생성자가 끝나도 초기화되지 않는다.

}

인스턴스 blank final 필드는 어떤 생성자가 쓰이더라도 생성자가 끝날때까지는 초기화되어야 한다. 다음 예제에서는 Example() 생성자가 불린다면 초기화가 안될 것이기 때문에 컴파일타임 오류가 난다.

public class Example {

final int blankFinal;

Example() { // Error. 여기서는 blankFinal 이 초기화되지 않는다.

}

Example(int i) {

blankFinal = i;

}

}

생성자가 끝나기 전에 초기화 한다고 다음과 같이 하면 안된다. 컴파일이 안될 것이다.

public class Example {

final int blankFinal;

Example() {

setBlankFinal();

}

void setBlankFinal() {

blankFinal = 123; // Error.

}

}

왜냐하면 setBlankFinal 메쏘드는 생성자가 끝난 다음에도 불릴 수 있기 때문이다.

* 1. 필드의 초기화 순서
     1. 필드의 초기화에서는 코드상의 순서가 중요할 수 있다

다음 예제를 보자.

class A {

int a = b; // Error.

int b = 2;

}

컴파일 에러가 난다. 순서가 틀렸기 때문이다. b의 값을 모르는데 b의 값으로 a의 값을 설정할 수는 없다. 다음과 같이 해도 역시 컴파일타임 오류다.

class A {

int a;

{

a = b; // Error.

}

int b = 2;

}

그러므로 다음과 같이 고쳐야 한다.

class A {

int b = 2;

int a = b; // OK.

}

메쏘드를 사용하면 상황이 달라진다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println(a.a);

}

}

class A {

int a = getB();

int b = 2;

public int getB() {

return b;

}

}

결과는 다음과 같을 것이다.

0

에러는 안나지만 getB 메쏘드가 불릴때는 아직 b가 초기화되지 않았기 때문에 디폴트 값인 0을 리턴한다.

* + 1. 정적 필드의 교차 초기화

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("A.field = " + A.field);

System.out.println("B.field = " + B.field);

}

}

class A {

static int field;

static {

System.out.println("A static initializer start");

field = B.b();

System.out.println("A static initializer end");

}

public static int a() {

System.out.println("A.a()");

return field;

}

}

class B {

static int field;

static {

System.out.println("B static initializer start");

field = 33;

field = A.a();

System.out.println("B static initializer end");

}

public static int b() {

System.out.println("B.b()");

return field;

}

}

결과는 아마 다음과 같을 것이다. 가상 머신에 따라 결과가 다를지도 모른다.

A static initializer start

B static initializer start

A.a()

B static initializer end

B.b()

A static initializer end

A.field = 0

B.field = 0

여기서 주목할 것은 A의 필드를 B의 메쏘드에 의해 초기화하고 B의 필드를 A의 메쏘드에 의해 초기화한다는 것이다. 위의 결과로 볼 때 A가 먼저 로딩(load)되고 있다. A의 정적 초기화 블록이 실행된다. “A static initializer start”가 뿌려진다. 그 다음에는 B의 메쏘드를 부른다는 사실을 안 후 B를 로딩한다. B의 정적 초기화 블록이 실행된다. “B static initializer start”가 뿌려진다. B.field에 33을 대입한다. 다음에 A의 메쏘드를 부르는 코드가 있다. 이 때 다시 A로 가서 초기화하려 한다면 계속 순환할 뿐이다. 그래서 어쩔 수 없이 A의 메쏘드를 부른다. 이 때 A.field는 디폴트 초기값인 0이 설정되어 있다. 그러므로 A.a()는 0을 리턴하고 그것이 B.field에 대입된다. B.field는 0이 다시 0이 된 것이다. 이리하여 B의 정적 필드의 초기화가 끝난다. A로 다시 돌아와서 초기화를 계속하는데 B.b()가 0을 리턴하여 A.field도 0으로 초기화된다.

위의 예제를 아주 약간 바꾸었다.

public class Example {

public static void main(String[] args) {

System.out.println("B.field = " + B.field); // 순서가 바뀌었다.

System.out.println("A.field = " + A.field); // 순서가 바뀌었다.

}

}

class A {

static int field;

static {

System.out.println("A static initializer start");

field = B.b();

System.out.println("A static initializer end");

}

public static int a() {

System.out.println("A.a()");

return field;

}

}

class B {

static int field;

static {

System.out.println("B static initializer start");

field = 33;

field = A.a();

System.out.println("B static initializer end");

}

public static int b() {

System.out.println("B.b()");

return field;

}

}

결과는 아마 다음과 같을 것이다. 가상 머신에 따라 결과가 다를지도 모른다.

B static initializer start

A static initializer start

B.b()

A static initializer end

A.a()

B static initializer end

B.field = 33

A.field = 33

어떤 클래스가 먼저 로딩되는가에 따라 두 필드값이 다르게 설정되었다. 위의 예제처럼 잘 따려보면 왜 이렇게 되는지 이해가 갈 것이다. 결론적으로 말하면 이런 식으로 초기화하면 안된다는 것이다.

* + 1. 정적 필드와 인스턴스 필드

정적 필드는 인스턴스 필드를 이용해서 초기화 할 수 없다. 그러므로 다음 예제는 컴파일되지 않을 것이다.

class A {

int instanceField = 123;

static int staticField = instanceField; // Error.

}

반대로 할 경우 즉 인스턴스 필드를 정적 필드를 이용해서 초기화 할 수는 있다. 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println(a.instanceField);

}

}

class A {

int instanceField = staticField;

static int staticField = 123;

}

결과는 다음과 같을 것이다.

123

정적 필드는 인스턴스 필드보다 항상 먼저 초기화된다. 인스턴스 필드는 객체가 생성될 때 초기화 되며 정적 필드는 클래스를 로딩할 때 초기화된다. 그리고 당연히 클래스가 먼저 로딩되어야 객체를 생성할 수 있다.

* + 1. 중첩 클래스의 초기화 순서

아래의 예제들은 시스템마다 결과가 다를 수 있다. 여기 결과는 JDK1.4(java 1.4.0-beta-b65)에서 실행한 결과이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Outer();

}

}

class Outer {

static int outerField = initOuterField();

static int initOuterField() {

System.out.println("Outer.initOuterField()");

return 0;

}

static class Nested {

static int nestedField = initNestedField();

static int initNestedField() {

System.out.println("Nested.initNestedField()");

return 0;

}

}

}

결과는 다음과 비슷할 것이다.

Outer.initOuterField()

위의 예제에서 Example 클래스만 다음과 같이 바꾸어서 실행해 보자.

public class Example {

public static void main(String[] args) {

new Outer.Nested();

}

}

결과는 다음과 비슷할 것이다.

Nested.initNestedField()

이번에는 다음과 같이 바꾸어서 실행해 보자.

public class Example {

public static void main(String[] args) {

new Outer();

new Outer.Nested();

}

}

결과는 다음과 비슷할 것이다.

Outer.initOuterField()

Nested.initNestedField()

이번에는 다음과 같이 바꾸어 보자.

public class Example {

public static void main(String[] args) {

new Outer.Nested();

new Outer();

}

}

결과는 다음과 비슷할 것이다.

Nested.initNestedField()

Outer.initOuterField()

마치 서로 별개의 최상위 클래스인 것처럼 클래스가 로딩된다. 이것은 정적 필드의 초기화가 클래스가 로딩되면서 실행되기 때문에 위의 예제를 통해서 알 수 있는 것이다.

* 1. 이름 충돌
     1. 이름 충돌의 종류

같은 이름을 쓰는 것은 대부분의 경우 안좋은 일이다. 그러므로 되도록 피하도록 해야 한다. 그리고 자바에서는 이름 충돌을 막는데 도움이 되는 패키지와 중첩 타입이 있다. 그러나 이름 충돌을 완전히 막기는 어렵다. 그 이유 중에 가장 큰 이유는 하나의 프로그램이 돌아가기 위해서는 아주 많은 사람의 손을 거쳐야 하기 때문이다. 다음과 같은 아주 작은 프로그램을 보자.

public class Example {

public static void main(String[] args) {

new java.awt.Frame().show();

}

}

당신이 작성한 코드는 다섯줄에 불과하지만 이 프로그램이 실행되기 위해서는 적어도 java.lang.Object, java.awt.Component, java.awt.Container, java.awt.Window, java.awt.Frame 등의 클래스들이 필요하다. 주석을 포함한 것이긴 하지만 이중에 Component 클래스는 7000라인이 넘고 Container 클래스는 3000라인이 넘는다. 그리고 커다란 프로젝트의 경우에는 10만 라인이 넘는게 보통이다. 게다가 API를 만드는 사람과 애플리케이션 프로그래머와는 거의 의사 소통이 없다. 예를 들어 당신은 자바 API 만든 사람과 얘기해 본 적이 있나 기억을 더듬어 보라.

(패키지 이름을 포함한) 클래스 이름이 충돌하면 거의 해결책이 없다고 보아야 한다. 이것은 인터넷 도메인 네임을 이용해서 거의 막을 수 있다. 그 이외에도 여러가지로 이름이 충돌할 수 있다.

이름 충돌은 허용되는 이름 충돌과 허용되지 않는 이름 충돌이 있다. 허용되지 않는 이름 충돌은 컴파일타임 오류가 날 것이다. 허용된 이름 충돌에는 감추기(hiding)과 가리기(shadowing)가 있다. 각각에 대해서는 아래에서 설명한다. 감추기와 가리기라는 용어를 여기서처럼 구별하지 않는 사람들도 있다. 아주 드문 ambiguity라는 것도 있다는데 여기서는 다루지 않는다.

* + 1. 메쏘드의 서명(signature) 충돌

두 메쏘드의 서명이 충돌하면 안된다. 왜냐하면 어떤 메쏘드를 쓸 지 알 수 없기 때문이다. 이것은 하나는 정적 메쏘드고 하나는 인스턴스 메쏘드일 때도 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.method(); // 만약 컴파일이 잘 된다면 어떤 메쏘드를 불러야 할지 모를 것이다.

}

}

class A {

public void method() {

}

public static void method() { // Error.

}

}

컴파일이 되지 않을 것이다.

두 메쏘드가 서명은 같고 반환형(return type)이 달라도 안된다. 그러므로 다음 예제는 컴파일 되지 않는다.

public class Example {

public void method() {}

public int method() {} // Error.

}

두 메쏘드를 두개의 인터페이스에서 상속받아도 서명은 같고 반환형이 다르면 안된다. 그러므로 다음 예제는 컴파일되지 않는다.

public abstract class Example implements A, B {

}

interface A {

void method();

}

interface B {

int method();

}

인터페이스 A의 메쏘드와 인터페이스 B의 메쏘드 모두를 만족시킬 수 있도록 구현할 수 없기 때문이다. 이것은 하나는 클래스에서 하나는 인터페이스에서 상속받을 때도 마찬가지이다. 다음 예제도 마찬가지로 컴파일되지 않는다.

public abstract class Example extends A implements B {

}

class A {

public void method() {}

}

interface B {

int method();

}

* + 1. 다중 상속된 필드의 충돌

인터페이스는 다중 상속이 가능하므로 필드이름이 충돌할 수 있다. 다음 예제를 보자.

interface A {

int RED = 1;

int BLACK = 20;

}

interface B {

int RED = 2;

int BLACK = 20;

}

interface C extends A, B {

int MY\_COLOR = RED; // Error.

int YOUR\_COLOR = A.RED; // OK.

int HER\_COLOR = BLACK; // Error. 값이 같아도 에러.

}

두개의 인터페이스에서 같은 이름의 필드를 상속받았을 때는 A.RED와 같이 인터페이스의 이름과 함께 사용해야 한다. A.BLACK과 B.BLACK의 값이 같음에도 컴파일타임 오류가 남에 주의하라. 이것은 하나는 수퍼 클래스 하나는 수퍼 인터페이스에서 필드를 상속받을 때도 마찬가지이다.

이 때 하나의 인터페이스에 선언된 필드가 여러갈래도 상속받는다면 simple name을 사용해도 아무런 문제가 없다. 다음 예제를 보자.

public class Example implements SuperA, SuperB {

public static void main(String[] args) {

System.out.println(FIELD); // OK.

}

}

interface GrandSuper {

int FIELD = 123;

}

interface SuperA extends GrandSuper {

}

interface SuperB extends GrandSuper {

}

결과는 다음과 같을 것이다.

123

Example 클래스는 FIELD 필드를 SuperA와 SuperB로부터 상속받는다. 그러나 결국은 GrandSuper라는 하나의 인터페이스에서 상속받는 것이므로 그냥 FIELD라고 사용해도 문제가 되지 않는다. 하지만 GrandSuper.FIELD라고 사용하면 코드를 읽는 사람에게 도움이 될 것이다.

* + 1. 다중 상속된 중첩 타입

인터페이스는 다중 상속이 가능하므로 중첩 타입의 이름이 충돌할 수 있다. 이것은 필드와 비슷하다. 다음 예제를 보자.

public class Example implements A, B {

public static void main(String[] args) {

new Nested(); // Error.

new A.Nested(); // OK.

new B.Nested(); // OK.

}

}

interface A {

class Nested {}

}

interface B {

class Nested {}

}

필드와 마찬가지로 simple name을 사용하면 컴파일타임 오류가 난다.

아래 예제와 같이 헷갈리지 않는 경우에는 필드와 마찬가지로 simple name을 사용할 수 있다.

public class Example implements SuperA, SuperB {

public static void main(String[] args) {

new Nested(); // OK.

}

}

interface GrandSuper {

class Nested {}

}

interface SuperA extends GrandSuper {

}

interface SuperB extends GrandSuper {

}

* + 1. 금지된 이름 충돌

멤버 타입 이름은 그것을 포함하는 타입과 같을 수 없다. 그러므로 다음 예제는 컴파일되지 않을 것이다.

public class Example {

class Example {

}

}

* + 1. 지역 변수와 매개변수

지역 변수와 매개 변수는 이름이 충돌하면 안된다. 다음 예제를 보자.

public class Example {

public void method(int parameter) {

double parameter; // Error.

try {

} catch (RuntimeException parameter) { // Error.

}

}

}

그러나 중첩 타입 안에 있으면 이름 충돌이 허용되며 밖에 있는 이름이 가려진다(shadowed). 그리고 밖에 있는 이름에는 접근할 수 없다.

public class Example {

public void method(int parameter) {

double local;

class Nested {

public void nestedMethod(int parameter) { // OK.

double local; // OK.

}

}

}

}

밖에 있는 매개변수와 지역 변수에 접근할 수 없음은 큰 문제가 되지 않는다. 어짜피 중첩 타입과 외부 타입이 하나의 파일 안에 있으므로(이것은 한 사람 또는 밀접하게 협동하는 사람들에 의해 코드가 쓰여짐을 뜻한다) 충돌하는 이름을 고치면 되기 때문이다.

* + 1. 허용된 이름 충돌

메쏘드나 필드 이름은 타입 이름과 같을 수 있으나 좋은 생각은 아니다. 다음은 컴파일이 될 것이다.

public class Example {

public Example() { // 생성자

}

public void Example() { // 메쏘드

}

int Example; // 필드

}

필드와 메쏘드와 멤버 타입의 이름이 같을 수 있으나 좋은 생각은 아니다. 다음 예제는 컴파일이 될 것이다.

public class Example {

int WhoAmI; // 필드

void WhoAmI() {} // 메쏘드

class WhoAmI {} // 멤버 타입

}

* + 1. 필드를 가리는(shadow) 지역변수

필드는 매개변수나 지역 변수에 의해 가려질 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new A().shadow();

}

}

class A {

int value = 3;

static int staticValue = 4;

public void shadow() {

int value = 2345;

int staticValue = 4321;

System.out.println(value);

System.out.println(this.value);

System.out.println(staticValue);

System.out.println(A.staticValue);

}

}

결과는 다음과 같다.

2345

3

4321

4

이런 가리기 때문에 흔히 하는 실수가 있다. 다음 예제를 보자.

class A {

int field;

A(int i) {

int field = i; // 흔히 하는 실수. 원래는 필드에 대입할 생각이었다.

}

}

여기서 int 키워드를 잘못 하용해서 생성자에서 필드가 초기화되지 않는 일이 일어났다.

* + 1. 지역 변수를 가리는 필드

중첩타입의 필드는 바깥쪽의 지역 변수나 매개변수를 가린다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

o.method(1);

}

}

class Outer {

public void method(int parameter) {

int localVariable = 2;

class Inner {

int parameter = 111;

int localVariable = 222;

public void show() {

System.out.println(parameter);

System.out.println(localVariable);

}

}

new Inner().show();

}

}

결과는 다음과 같을 것이다.

111

222

가려진 매개변수와 지역변수에는 접근할 수 없다. 곰곰히 생각해도 접근할 방법이 떠오르지 않을 것이다. 다음과 같이 좀더 복잡한 경우도 있을 수 있다.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

o.method(1);

}

}

class SuperClass {

int parameter = 111;

int localVariable = 222;

}

class Outer {

public void method(int parameter) {

int localVariable = 2;

class Inner extends SuperClass {

public void show() {

//System.out.println(parameter); // Error.

//System.out.println(localVariable); // Error.

System.out.println(super.parameter);

System.out.println(super.localVariable);

System.out.println(this.parameter);

System.out.println(this.localVariable);

}

}

new Inner().show();

}

}

결과는 다음과 같을 것이다.

111

222

111

222

이 때도 앞의 예제와 비슷하긴 한데 반드시 super나 this 키워드를 사용해야 한다는 것이다. 안 그러면 지역변수나 매개변수라고 코드를 읽는 사람이 착각하기 쉽다. 이것은 코드를 쓰는 사람도 마찬가지이다. 코드를 쓰는 사람은 컴파일타임 오류를 보고 수퍼 클래스에 같은 이름의 필드가 있다는 것을 깨닫고 매개변수나 지역 변수의 이름을 바꾸어서 이름 충돌을 피할 수 있을 것이다.

private 멤버는 상속되지 않기 때문에 다음 예제에서 가리기는 일어나지 않는다.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

o.method(1);

}

}

class SuperClass {

private int parameter = 111;

private int localVariable = 222;

}

class Outer {

public void method(final int parameter) {

final int localVariable = 2;

class Inner extends SuperClass {

public void show() {

System.out.println(parameter);

System.out.println(localVariable);

}

}

new Inner().show();

}

}

결과는 다음과 같을 것이다.

1

2

* + 1. 필드를 가리는 중첩타입의 필드

중첩타입의 멤버는 같은 이름의 외부 타입(enclosing type)의 멤버를 가린다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

Outer.Inner i = o.getInnerObject();

i.show();

}

}

class Outer {

int field = 111;

public void method() {

System.out.println("Outer.method()");

}

class Inner {

int field = 222;

public void method() {

System.out.println("Inner.method()");

}

public void show() {

System.out.println(field);

System.out.println(Outer.this.field);

method();

Outer.this.method();

}

}

public Inner getInnerObject() {

return new Inner();

}

}

결과는 다음과 같을 것이다.

222

111

Inner.method()

Outer.method()

가려진 멤버에 접근하기 위해서는 Outer.this와 같이 해야 한다.

* + 1. 필드의 감추기(hiding)

수퍼 클래스에 있는 필드를 서브 클래스에서 선언하면 어떻게 될까? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public int value = 3;

}

class SubClass extends SuperClass {

public int value = 33;

public void method() {

System.out.println(value);

}

}

결과는 다음과 같을 것이다.

33

이런 상황을 필드의 감추기라고 한다. 여기서는 서브 클래스의 필드 value가 수퍼 클래스의 필드 value 감춘 것이다. 이것은 정적 필드도 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass.method();

}

}

class SuperClass {

public static int value = 3;

}

class SubClass extends SuperClass {

public static int value = 33;

public static void method() {

System.out.println(value);

}

}

결과는 다음과 같을 것이다.

33

이 때 수퍼 클래스의 필드는 감추어진 것이지 사라진 것이 아니다. 그러므로 다음 예제처럼 수퍼 클래스의 필드에 접근할 수 있다.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public int value = 3;

}

class SubClass extends SuperClass {

public int value = 33;

public void method() {

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

System.out.println("this.value = " + this.value);

System.out.println("super.value = " + super.value);

System.out.println("((SuperClass)this).value = " + ((SuperClass)this).value);

System.out.println("((SuperClass)super).value = " + ((SuperClass)super).value);

}

}

결과는 다음과 같을 것이다.

value = 33

this.value = 33

super.value = 3

((SuperClass)this).value = 3

((SuperClass)super).value = 3

다음은 정적 필드에 대한 예이다.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public static int value = 3;

}

class SubClass extends SuperClass {

public static int value = 33;

public void method() {

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

System.out.println("this.value = " + this.value);

System.out.println("super.value = " + super.value);

System.out.println("((SuperClass)this).value = " + ((SuperClass)this).value);

System.out.println("((SuperClass)super).value = " + ((SuperClass)super).value);

System.out.println("SuperClass.value = " + SuperClass.value);

}

}

결과는 다음과 같을 것이다.

value = 33

this.value = 33

super.value = 3

((SuperClass)this).value = 3

((SuperClass)super).value = 3

SuperClass.value = 3

정적 필드에 접근하기 위해 this나 super등을 사용하는 것은 좋은 방법이 아니다. SuperClass.value가 가장 좋은 방법일 것이다.

정적 필드를 인스턴스 필드가 가릴수 있으며 반대도 가능하다. 그리고 감추어진 필드와 감추는 필드의 타입이 같을 필요도 없다. 이런식으로 문법이 유연한 것은 재정의(overriding)와 다르게 감추기는 대부분 의도하지 않은 결과이기 때문이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public static int a = 3;

public String b = "String";

}

class SubClass extends SuperClass {

public int a = 33;

public java.awt.Point b = new java.awt.Point(20, 30);

public void method() {

System.out.println(a);

System.out.println(b);

}

}

결과는 다음과 비슷할 것이다.

33

java.awt.Point[x=20,y=30]

* + 1. 중첩 타입의 이름 충돌

중첩 타입은 필드와 마찬가지로 감추어질 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new SubOuter().method();

}

}

class SuperOuter {

static class Nested implements java.io.Serializable {

}

class Inner implements java.io.Serializable {

}

}

class SubOuter extends SuperOuter {

static class Nested implements Cloneable {

}

class Inner implements Cloneable {

}

public void method() {

Class[] interfaces = Nested.class.getInterfaces();

for (int i = 0; i < interfaces.length; i++)

System.out.println(interfaces[i]);

interfaces = SuperOuter.Nested.class.getInterfaces();

for (int i = 0; i < interfaces.length; i++)

System.out.println(interfaces[i]);

interfaces = Inner.class.getInterfaces();

for (int i = 0; i < interfaces.length; i++)

System.out.println(interfaces[i]);

interfaces = SuperOuter.Inner.class.getInterfaces();

for (int i = 0; i < interfaces.length; i++)

System.out.println(interfaces[i]);

}

}

결과는 다음과 비슷할 것이다.

interface java.lang.Cloneable

interface java.io.Serializable

interface java.lang.Cloneable

interface java.io.Serializable

* + 1. 외부 클래스의 감춰진(hidden) 필드에 접근하기

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

OuterSub o = new OuterSub();

OuterSub.Inner i = o.getInnerObject();

i.method();

}

}

class OuterSuper {

int field = 123;

}

class OuterSub extends OuterSuper {

int field = 9990;

class Inner {

public void method() {

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

System.out.println("OuterSub.this.field = " + OuterSub.this.field);

System.out.println("OuterSub.super.field = " + OuterSub.super.field);

System.out.println("((OuterSuper)OuterSub.this).field = " + ((OuterSuper)OuterSub.this).field);

}

}

public Inner getInnerObject() {

return new Inner();

}

}

결과는 다음과 같을 것이다.

field = 9990

OuterSub.this.field = 9990

OuterSub.super.field = 123

((OuterSuper)OuterSub.this).field = 123

* + 1. 감추기(hiding)와 재정의(overriding)의 차이

수퍼 클래스와 서브 클래스에 같은 이름(메쏘드의 경우는 서명signature)이 있으면 메쏘드는 재정의가 되며 필드는 감추기가 된다. 다른 말로 하면 필드는 재정의되지 않는다. 재정의 특징은 어떤 메쏘드를 사용할지가 동적으로 결정된다는 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass s = new SubClass();

s.method();

}

}

class SuperClass {

public int hide = 3;

public void method() {

System.out.println(hide);

override();

}

public void override() {

System.out.println("SuperClass.override()");

}

}

class SubClass extends SuperClass {

public int hide = 33;

public void override() {

System.out.println("SubClass.override()");

}

}

결과는 다음과 같을 것이다.

3

SubClass.override()

* + 1. 멤버 타입 감추기

멤버 타입도 필드와 마찬가지로 감추어질 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public static class Inner {

public static int field = 10;

}

}

class SubClass extends SuperClass {

public static class Inner {

public static int field = 987;

}

public void method() {

System.out.println(Inner.field);

}

}

결과는 다음과 같을 것이다.

987

수퍼 클래스의 타입에 접근하기 위해서는 다음과 같이 해야 한다.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public static class Inner {

public static int field = 10;

}

}

class SubClass extends SuperClass {

public static class Inner {

public static int field = 987;

}

public void method() {

System.out.println(SuperClass.Inner.field);

System.out.println(((SuperClass)this).Inner.field);

System.out.println(super.Inner.field);

}

}

결과는 다음과 같을 것이다.

10

10

10

Inner가 정적 멤버(static member)이기 때문에 this나 super를 사용하는 것보다는 SuperClass.Inner.field와 같이 하는 것이 더 낫다.

멤버 인터페이스가 멤버 클래스를 감출 수 있을 뿐 아니라 심지어 필드가 멤버 클래스를 감출수도 있다. 문법적으로 그렇다는 얘기다. 보통은 타입은 대문자로 시작하고 메쏘드나 필드는 소문자로 시작하기 때문에 이런 일이 일어나지는 않는다.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public static class A {

public static int field = 10;

}

public static class B {

public static int field = 20;

}

}

class SubClass extends SuperClass {

interface A {

public static int field = 10000;

}

int B = 20000;

public void method() {

System.out.println(A.field);

System.out.println(B);

}

}

결과는 다음과 같을 것이다.

10000

20000

* + 1. 정적 메쏘드의 감추기

인스턴스 메쏘드의 경우에는 재정의되는 반면 정적 메쏘드는 감추어진다. 다음 예제를 보면 재정의되지 않음을 알 수 있다.

public class Example {

public static void main(String[] args) {

SuperClass s = new SubClass();

s.callMethod();

}

}

class SuperClass {

public static void method() {

System.out.println("SuperClass.method()");

}

public static void callMethod() {

method();

}

}

class SubClass extends SuperClass {

public static void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같을 것이다.

SuperClass.method()

정적 메쏘드도 final로 선언할 수 있다. 인스턴스 메쏘드를 final로 선언하면 재정의가 금지되고 정적 메쏘드를 final로 선언하면 감추기가 금지된다. 다음 예제는 컴파일되지 않을 것이다.

class SuperClass {

public static final void method() { // final로 선언했다.

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public static void method() { // Error.

System.out.println("SubClass.method()");

}

}

정적 메쏘드를 인스턴스 메쏘드로 감추거나 반대로 할 수는 없다. 다음 예제는 컴파일되지 않을 것이다. 만약 허용된다면 재정의와의 혼돈이 일어날 것이다.

class SuperClass {

public static void method() {

System.out.println("SuperClass.method()");

}

}

class SubClass extends SuperClass {

public void method() { // Error.

System.out.println("SubClass.method()");

}

}

필드와 메쏘드는 서로 감추지 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

}

}

class SuperClass {

public int field = 999;

}

class SubClass extends SuperClass {

public void field() {

System.out.println("SubClass.method()");

}

public void method() {

System.out.println(field);

field();

}

}

결과는 다음과 같을 것이다.

999

SubClass.method()

메쏘드는 항상 메쏘드 이름 다음에 ()가 쓰이고 필드는 바로 다음에 ()가 쓰이는 일이 절대 없기 때문에 이름이 충돌해도 감추거나 하지 않는다. 그러나 컴파일러가 헷갈리지 않는다고 사람도 헷갈리지 않는 것은 아니다. 그러므로 되도록 피하는 것이 좋다.

* + 1. 정적 메쏘드 감추기의 제한

정적 메쏘드의 감추기에서는 반환형(return type)이 다르면 안된다. 필드의 감추기에서는 타입이 달라도 허용되는 것과는 다르다. 다음 예제는 컴파일되지 않을 것이다.

class SuperClass {

public static Object method() {

return null;

}

}

class SubClass extends SuperClass {

public static void method() { // Error.

}

}

메쏘드 감추기에서는 throws절과 접근에도 재정의와 같은 제한이 따른다. 다음 예제를 보자. 컴파일이 되지 않을 것이다.

class SuperClass {

public static void a() {}

protected static void b() {}

public static void c() throws java.io.IOException {}

public static void d() throws Exception {}

}

abstract class SubClass extends SuperClass {

protected static void a() {} // Error.

public static void b() {} // OK.

public static void c() throws Exception {} // Error.

public static void d() throws java.io.IOException {} // OK.

}

정적 메쏘드의 감추기에 왜 재정의와 같은 제한이 따라야 하는지는 잘 모르겠다.

* + 1. 감추기는 왜 허용되었나

이름 충돌은 사람들을 헷갈리게 한다. 그렇다면 아예 컴파일 타임에서 금지하는 것이 더 낫지 않을까? 즉 수퍼 클래스의 protected 필드나 public 필드와 같은 이름의 필드를 서브 클래스에서 아예 쓰지 못하게 하면 되는 것이 아닐까? 그런데 문제는 그렇게 간단하지가 않다. 만약 수퍼 클래스와 서브 클래스가 한 사람(또는 하나의 집단)에 의해 만들어진다면 문제가 간단하지만 그렇지 않은 경우가 많다. API와 같이 많이 쓰이는 클래스를 만든 사람들은 그것의 서브 클래스를 만드는 사람을 전혀 통제할 수 없다. 예를 들어 이런 시나리오가 있을 수 있다. 당신이 훌륭한 프로그래머라고 상상해 보자. 당신이 다음과 같은 훌륭한 클래스를 만들었다.

public class ExcellentClass {

}

이 클래스가 너무나 훌륭하기 때문에 많은 사람이 이 클래스의 서브 클래스를 만들어서 사용하고 있다고 생각하자. 이 때 당신이 이 클래스를 업그레이드하는 과정에서 protected 필드를 하나 추가했다.

public class ExcellentClass {

protected int upgrade;

}

그런데 이 업그레이드가 일어나기 전에 어떤 사람이 이미 이 이름의 필드를 사용하였다면 어떨까? 즉 당신이 모르는 어떤 사람이 다음과 같은 클래스를 만들어서 사용하고 있었다면.

public class SubClass extends ExcellentClass {

protected int upgrade;

}

그 사람은 자신이 만든 SubClass와 당신이 만든 업그레이드된 ExcellentClass를 동시에 사용할 수 없게 된다.

* 1. 메쏘드
     1. header와 몸체(body)

메쏘드는 header와 몸체로 이루어진다. 메쏘드 header는 메쏘드 지정자, 반환형, 메쏘드 이름, 괄호, 매개변수 리스트, throws 절로 이루어진다. 여기서 반환형, 메쏘드 이름, 괄호는 생략될 수 없다. 메쏘드 몸체는 메쏘드 블록 또는 세미콜론으로 이루어진다. abstract 메쏘드나 native 메쏘드에서는 세미콜론이 쓰이고 다른 메쏘드에서는 블록으로 이루어진다.

* + 1. 지정자

메쏘드 지정자(method modifier)에는 다음과 같은 것이 있다.

public protected private abstract static final synchronized native strictfp

필드와 마찬가지로 여러개가 쓰인다면 위에 나온 순서로 적어 주는 것이 좋다.

필드와 마찬가지로 public, protected, default, private은 상호 배타적이다. 즉 예를 들어 동시에 public이면서 private일 수는 없다.

abstract는 private과 동시에 올 수 없다. private은 상속이 되지 않기 때문이다. abstract는 static과 동시에 올 수 없다. 이것은 정적(static)메쏘드는 동적 바인딩이되지 않기 때문에 재정의도 되지 않는다는 사실과 연관되어 있다. abstract는 final과 동시에 올 수 없다. final은 서브 클래스에서 재정의될 수 없기 때문이다. abstract는 native, strictfp, synchronized와 동시에 올 수 없다. native, strictfp, synchronized는 모두 구현에 관계되는 것들이며 abstract는 구현을 서브 클래스로 미루는 것이기 때문이다.

native에 대해서는 native 메쏘드를 다루는 책을 참조하라.

strictfp에 대해서는 strict-FP를 다루는 부분을 참조하라.

synchronized에 대해서는 쓰레드를 다루는 부분을 참조하라.

* + 1. 메쏘드의 매개변수의 개수는 가변일 수 없다

자바 메쏘드에서는 매개변수의 개수가 가변일 수 없다. 그렇다고 너무 좌절할 필요는 없다. 이것은 대부분의 경우 Object형의 배열로 해결 가능하다. 다음 예를 보자. 여기서는 Number형 배열을 사용했다. 배열 뿐 아니라 여러가지 컬렉션 클래스의 객체를 사용할 수도 있다. 그리고 기본형일 때에는 여기에 있는 예처럼 포장(wrapper) 클래스를 사용하면 된다.

public class Example {

public static void main(String[] args) {

Number[] an = new Number[5];

an[0] = new Integer(10);

an[1] = new Float(12.3);

short sh = 321;

an[2] = new Short(sh);

an[3] = new Long(333);

an[4] = new Double(-0.7);

double sum = plus(an);

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

}

public static double plus(Number[] numbers) {

double s = 0;

for (int i = 0; i < numbers.length; i++) {

s += numbers[i].doubleValue();

}

return s;

}

}

결과는 다음과 비슷할 것이다.

sum = 675.6000001907348

* + 1. 메쏘드는 최대 하나만 반환(return)할 수 있다

자바 메쏘드는 반환값(return value)이 없거나(void) 하나일 수 있다. 즉 여러 개의 반환값을 가질 수 없다(예외 상황exception을 별도로 한다면). 이것이 문제가 될 경우가 있다. 즉 메쏘드의 결과값이 여러 개여서 그것은 모두 알아야 할 때가 있다. 이것은 여러가지 방법으로 해결할 수 있다.

먼저 배열이나 컬렉션 클래스를 반환하는 방법이 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String[] returned = returnArray();

for (int i = 0; i < returned.length; i++)

System.out.println(returned[i]);

}

public static String[] returnArray() {

String[] array = new String[3];

array[0] = "first";

array[1] = "second";

array[2] = "third";

return array;

}

}

결과는 다음과 같을 것이다.

first

second

third

여기서는 String형 배열을 사용했는데 Object형 배열을 사용하거나 컬렉션 클래스를 사용하면 어떤 객체들도 가능하다.

만약 반환될 객체들이 서로 동질적이라면 배열이나 컬렉션 클래스를 사용하는 것이 좋겠지만 서로 이질적이면 배열 등을 사용하면 헷갈릴 수 있다. 이 때는 클래스를 새로 정의해서 반환하는 방법이 더 좋다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Return returned = returnObject();

System.out.println(returned.first);

System.out.println(returned.second);

System.out.println(returned.third);

}

public static Return returnObject() {

Return r = new Return(1, "second", new StringBuffer("third"));

return r;

}

public static class Return {

public int first;

public String second;

public StringBuffer third;

public Return(int first, String second, StringBuffer third) {

this.first = first;

this.second = second;

this.third = third;

}

}

}

결과는 다음과 같을 것이다.

1

second

third

매개변수를 이용하는 방법도 있으나 그리 좋은 방법이 아니므로 특별한 경우에만 사용해야 한다. 매개변수는 메쏘드에서의 출력이 아닌 메쏘드로의 입력으로만 사용하는 것이 좋다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Return returned = new Return();

returnByParameter(returned);

System.out.println(returned.first);

System.out.println(returned.second);

System.out.println(returned.third);

}

public static void returnByParameter(Return r) {

r.first = 1;

r.second = "second";

r.third = new StringBuffer("third");

}

public static class Return {

public int first;

public String second;

public StringBuffer third;

}

}

결과는 다음과 같을 것이다.

1

second

third

* + 1. final 메쏘드

private 메쏘드는 암시적으로(implicitily) final 메쏘드이다. 다른 말로 하면 굳이 final 키워드를 사용하지 않아도 final이 된다는 말이다. 마찬가지로 final 클래스의 메쏘드도 final 메쏘드이다. final 클래스가 아예 서브 클래스를 만들 수 조차 없기 때문에 당연히 재정의가 될 리가 없다. 두 경우 명시적으로 메쏘드에 final 키워드를 사용해도 된다. 다음 예제는 조금 헷갈릴 것이다.

class Super {

private final void method() {

}

}

class Sub extends Super {

private void method() {

}

}

컴파일타임 오류가 나지 않는다. 왜냐하면 Sub 클래스의 메쏘드와 Super 클래스의 메쏘드는 서로 완전히 별개이기 때문이다. 서로 알지 못하기 때문에 재정의될 수도 없다. 그러므로 Super.method()가 final로 선언했음에도 똑같은 signature의 메쏘드가 Sub 클래스에도 만들어질 수 있는 것이다.

* + 1. final 메쏘드와 performance

메쏘드를 final로 하면 속도가 약간 빨라진다. 왜냐하면 재정의가 되지 않기 때문에 동적 바인딩(dynamic binding)하는 시간을 절약할 수 있기 때문이다. 그리고 inlining도 가능하다. private 메쏘드도 사실상 final이기 때문에 마찬가지 이득을 얻을 수 있다. 정적 메쏘드도 재정의가 되지 않기 때문에 마찬가지이다. 그러나 이런 측면은 그리 중요한 것은 아니다. 메쏘드를 private 또는 final로 선언하는 것은 속도면에서 이익을 얻으려는 것이 주목적이 아니라 클래스의 안전성과 재활용성을 높이기 위함이다. 속도는 작은 부수적 효과라고 생각하면 된다.

inline에 대해 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

int i = a.getField();

System.out.println(i);

}

}

class A {

private int field = 10;

public final int getField() {

return field;

}

}

결과는 다음과 같을 것이다.

10

이것을 컴파일러가 다음과 비슷하게 바꾸어 줄 수 있다.

public class Example {

public static void main(String[] args) {

A a = new A();

int i = a.field; // inlining

System.out.println(i);

}

}

class A {

private int field = 10;

public final int getField() {

return field;

}

}

메쏘드를 부르는 것보다 필드에 접근하는 것이 비용이 적게 들기 때문이다. 물론 위와 같이 코딩에서 컴파일을 하면 컴파일타임 오류가 날 것이다. field가 private이기 때문이다. 위의 코드는 실제 코드가 아니라 컴파일러가 바이트 코드를 만드는 과정에서 최적화 하는 방식을 보여주는 것이다. 이것은 getField 메쏘드가 final이기 때문에 가능했다. 구현이 바뀌지 않기 때문에 inline을 해도 안전한 것이다. 사실 inlining은 그렇게 단순하게 생각할 일은 아니다. 예를 들어 Example 클래스와 A 클래스가 다른 사람에 의해 다른 시간에 컴파일된다면 문제가 생길수 있다. 이런 문제들에 대해서는 이 책에서 다루지 않는다.

* + 1. final 매개변수(parameter)

변하지 않는 메쏘드의 매개변수를 final로 선언하면 코드를 쓰는데 실수할 가능성을 줄이고 코드를 이해하는데 도움이 될 뿐 아니라 컴파일러나 가상머신이 코드를 최적화하는데도 약간 도움이 된다.

메쏘드의 매개변수가 final인지 아닌지는 메쏘드의 서명(signature)에 포함되지 않는다. 그러므로 그것을 이용해서 중복정의할 수 없다. 다음 예제를 보자.

public class Example {

public void method(int a) {

}

public void method(final int a) {

}

}

컴파일이 되지 않을 것이다.

이번에는 재정의와 final 매개변수와의 관계를 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass s = new SuperClass();

s.callMethod();

s = new SubClass();

s.callMethod();

}

}

class SuperClass {

public void method(int a) {

System.out.println("SuperClass.method()");

}

public void callMethod() {

method(12);

}

}

class SubClass extends SuperClass {

public void method(final int a) {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같을 것이다.

SuperClass.method()

SubClass.method()

재정의가 되었다.

* + 1. 중복정의(overloading)와 상속

중복정의된 메쏘드 중 하나를 재정의해도 나머지 하나의 메쏘드를 상속받아 쓸 수 있다. 이것은 C++과 다르다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.method();

s.method(123);

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method()");

}

public void method(int a) {

System.out.println("then congress is the opposite of progress?");

}

}

class SubClass extends SuperClass {

public void method() {

System.out.println("If con is the opposite of pro,");

}

}

결과는 다음과 같을 것이다.

If con is the opposite of pro,

then congress is the opposite of progress?

* + 1. 중복정의와 서명 – 다른 제한(반환형, static 여부, throws절)이 없다

중복정의에서 중요한 것은 서명이다. 다른 것은 중요하지 않다. 적어도 문법적으로는 그렇다. 다음 예제를 보자.

public class Example {

public void method() {

}

public int method(int a) throws Exception {

return 1;

}

public static Object method(int a, int b) {

return null;

}

}

무사히 컴파일된다. 인스턴스 메쏘드와 정적 메쏘드를 혼합할 수도 있고, 반환형이 다를 수도 있고 throws 절이 다를 수도 있다. 그러나 이런 식으로 코딩한다면 중복정의의 장점을 살리는 것이 아니라 혼동만 초래될 것이다. 중복정의의 목적은 비슷한 기능을 하는 메쏘드들에 같은 이름을 붙임으로써 편리하게 하자는 것이다. 전혀 다른 것들에 같은 이름을 붙여서 사람들을 골탕먹이는 것이 중복정의의 목적이 아니다.

* + 1. 중복정의된 메쏘드의 선택

대부분의 경우 중복정의된 메쏘드 중 어떤 메쏘드를 부를 것인가를 선택하는데는 별로 어려움이 없다. 직관적으로 척 보면 알 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

OverLoading ol = new OverLoading();

int a = 1;

int b = 2;

String s = "12";

ol.method(a);

ol.method(a, b);

ol.method(s);

}

}

class OverLoading {

public void method(int i) {

System.out.println("OverLoading.method(int)");

}

public void method(int i, int j) {

System.out.println("OverLoading.method(int, int)");

}

public void method(String str) {

System.out.println("OverLoading.method(String)");

}

}

결과는 다음과 같다.

OverLoading.method(int)

OverLoading.method(int, int)

OverLoading.method(String)

이젠 조금 헷갈리는 예제를 보자.

public class Example {

public static void main(String[] args) {

OverLoading ol = new OverLoading();

String s = "12";

ol.method(s);

}

}

class OverLoading {

public void method(Object o) {

System.out.println("OverLoading.method(Object)");

}

public void method(String str) {

System.out.println("OverLoading.method(String)");

}

}

결과는 다음과 같다.

OverLoading.method(String)

두 메쏘드 다 사용할 수 있으면 더 잘 맞는 메쏘드를 사용함을 알 수 있다. 다음은 Float.java에서 인용한 것이다.

public final class Float extends Number implements Comparable {

public int compareTo(Object o) {

return compareTo((Float)o);

}

public int compareTo(Float anotherFloat) {

return Float.compare(value, anotherFloat.value);

}

...

}

Comparable.compareTo(Object)를 재정의하는 것도 모자라서 compareTo(Float)으로 중복정의했다. 왜 이런짓을 했냐면 (Float)o와 같이 형변환하는 시간이 아까워서이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Float fA = new Float(10.);

Float fB = new Float(20.);

Object oB = fB;

long lStartTime;

lStartTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

fA.compareTo(fB);

System.out.println(System.currentTimeMillis() - lStartTime);

lStartTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

fA.compareTo(oB);

System.out.println(System.currentTimeMillis() - lStartTime);

}

}

결과는 다음과 비슷할 것이다.

601

821

두번째 것 즉 Object 형을 넘겨주는 것이 시간이 더 많이 걸린다. 만약 둘이 같게 나온다면 컴퓨터가 너무 좋아서 그런 것이니 for문의 반복횟수를 늘려보면 될 것이다. 0의 갯수를 많이 늘렸는데도 두 숫자가 같거나 거의 차이가 안나면 혹시 자신이 쓰는 컴퓨터가 슈퍼 컴퓨터가 아닌지 의심해 보기 바란다.

이젠 또다는 예제를 보자.

public class Example {

public static void main(String[] args) {

Object o = new String();

String s = new String();

OverLoading.method(o, s);

}

}

class OverLoading {

public static void method(Object o, String s) {

System.out.println("method(Object, String)");

}

public static void method(String s, Object o) {

System.out.println("method(String, Object)");

}

public static void method(String s1, String s2) {

System.out.println("method(String, String)");

}

}

결과는 다음과 같다.

method(Object, String)

method(String, String)이 불리지 않았다는 것에 주목하라. 중복정의된 메쏘드 중에 선택하는 것은 컴파일 타임에 결정된다. 그러므로 실제 변수에 어떤 값이 저장되는지는 무시된다. 즉

OverLoading.method(o, s);

에서 o에 String 객체가 저장되었다는 사실이 무시되는 것이다. 즉 o의 런타임 객체인 String이 아니라 컴파일 타임 객체인 Object가 쓰이는 것이다.

위의 예제를 조금 고쳐 보자. 컴파일이 되지 않을 것이다.

public class Example {

public static void main(String[] args) {

String s1 = new String();

String s2 = new String();

OverLoading.method(s1, s2); // Error

}

}

class OverLoading {

public static void method(Object o, String s) {

System.out.println("method(Object, String)");

}

public static void method(String s, Object o) {

System.out.println("method(String, Object)");

}

}

당신이 컴파일러라도 어떤 것을 선택할지 결정할 수 없을 것이다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 1; // OK

OverLoading.method((byte)1); // OK

OverLoading.method(1); // Error

}

}

class OverLoading {

public static void method(byte b) {

}

}

컴파일이 되지 않는다.

byte b = 1; // OK

가 무사히 컴파일되는 것을 생각하면 이상한 일이다. 위의 코드(byte b = 1;)가 왜 컴파일타임 오류가 안나는지는 형변환을 다룬 부분을 참조하라. 만약

OverLoading.method(1); // Error

가 컴파일타임 오류가 안난다면 다음과 같은 코드에서 문제가 발생할 것이다.

public class Example {

public static void main(String[] args) {

OverLoading.method(1); // 둘 중에 어떤 메쏘드를 사용해야 하나?

}

}

class OverLoading {

public static void method(byte b) {

System.out.println("method(byte)");

}

public static void method(int i) {

System.out.println("method(int)");

}

}

매개변수를 넘겨 줄 때는 1이 byte로 자동 형변환되지 않기 때문에 위의 예제는 무사히 컴파일 되고 결과는 다음과 같다.

method(int)

* + 1. 재정의의 제한

재정의를 할 때에는 반환형이 같아야 한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass s;

if (args.length == 0)

s = new SuperClass();

else

s = new SubClass();

int i = s.method(); // 만약 컴파일타임 오류가 안 난나면 이 코드가 어떻게 될지 생각해 보자.

}

}

class SuperClass {

public int method() {

}

}

class SubClass extends SuperClass {

public void method() { // Error.

}

}

컴파일이 되지 않을 것이다. 그 이유는 위의 main 메쏘드를 잘 보면 알 수 있을 것이다. 다음과 같은 좀더 그럴 듯한 예도 컴파일이 되지 않는다. 재정의를 하려면 무조건 반환형이 같아야 한다.

class SuperClass {

public Object method() {

}

}

class SubClass extends SuperClass {

public String method() { // Error.

}

}

이것은 중복정의에서는 반환형이 다른 것이 허용되는 것과 다르다. 중복정의에서는 컴파일 타임에 어떤 메쏘드를 쓸지 결정하기 때문에 이것이 적어도 문법적으로는 문제가 되지 않는다. 재정의에서는 컴파일러가 어떤 메쏘드를 쓸지 알 수 없다. 런타임에 가서야 알 수 있다. 그러므로 반환형이 같아야 한다.

synchronized, native, strict-fp등의 변경자는 재정의할 때 마음대로 바꿀 수 있다. 그리고 재정의할 때 메쏘드를 final 메쏘드로 고정할 수도 있고 구체 메쏘드를 abstract로 만들 수도 있다. 구체 메쏘드를 다시 추상 메쏘드로 만드는 경우는 그 서브 클래스가 예외적인 경우라서 수퍼 클래스의 구현을 사용할 수 없을 때이다. 재정의할 때 매개변수를 final로 바꾸거나 반대로 할 수도 있다.

* + 1. 재정의와 접근 제한

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new SubClass();

}

}

class SuperClass {

public void publicMethodA() {}

public void publicMethodB() {}

public void publicMethodC() {}

public void publicMethodD() {}

protected void protectedMethodA() {}

protected void protectedMethodB() {}

protected void protectedMethodC() {}

protected void protectedMethodD() {}

void defaultMethodA() {}

void defaultMethodB() {}

void defaultMethodC() {}

void defaultMethodD() {}

private void privateMethodA() {}

private void privateMethodB() {}

private void privateMethodC() {}

private void privateMethodD() {}

}

class SubClass extends SuperClass {

public void publicMethodA() {}

protected void publicMethodB() {} // Error.

void publicMethodC() {} // Error.

private void publicMethodD() {} // Error.

public void protectedMethodA() {}

protected void protectedMethodB() {}

void protectedMethodC() {} // Error.

private void protectedMethodD() {} // Error.

public void defaultMethodA() {}

protected void defaultMethodB() {}

void defaultMethodC() {}

private void defaultMethodD() {} // Error.

public void privateMethodA() {}

protected void privateMethodB() {}

void privateMethodC() {}

private void privateMethodD() {}

}

컴파일타임 오류가 날 것이다. 컴파일타임 오류가 난 곳의 공통점은 수퍼 클래스보다 접근을 더 제한한다는 것이다. 예를 들어 public 메쏘드를 protected로 재정의할 때이다(사실 수퍼 클래스에 private으로 선언된 멤버는 상속이 되지 않기 때문에 재정의된 것이 아니다). 수퍼 클래스보다 접근을 더 많이 허용하는 것은 괜찮다. 그 이유는 무엇인가? 다음 예를 보면 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

User u = new User();

SuperClass sp = new SuperClass();

u.method(sp);

}

}

class User {

public void method(SuperClass sp) {

sp.publicMethod();

}

}

class SuperClass {

public void publicMethod() {

System.out.println("SuperClass.publicMethod()");

}

}

결과는 다음과 같다.

SuperClass.publicMethod()

뭐 특별할 것 없는 지겨운 예제이다. 그런데 어떤 사람이 SuperClass를 상속하는 SubClass를 만들고 메쏘드의 접근을 더 제한하는 방식으로 재정의했다면 어떻게 될까? 다음 예제가 그것을 보여준다.

public class Example {

public static void main(String[] args) {

User u = new User();

SuperClass sp = new SubClass();

u.method(sp);

}

}

class User { // 나 User 클래스는 아무짓도 안했다.

public void method(SuperClass sp) { // sp는 SubClass의 객체가 넘어온다.

sp.publicMethod(); // SubClass.publicMethod 는 private 이다.

}

}

class SuperClass {

public void publicMethod() {

System.out.println("SuperClass.publicMethod()");

}

}

class SubClass extends SuperClass {

private void publicMethod() { // 만약 이것이 무사히 컴파일된다면...

System.out.println("SubClass.publicMethod()");

}

}

잘 살펴보면 죄없는 User 클래스의 재활용성이 타격을 받는다는 것을 알 수 있다.

* + 1. 추상 메쏘드가 추상 메쏘드를 재정의할 수 있다

추상 메쏘드를 추상 메쏘드가 재정의할 수 있다. 이것은 보통 주석을 넣기 위해 사용한다. 다른 재정의와 마찬가지로 접근을 그대로 하거나 넓혀야 하며 수퍼 클래스의 throws 절에 있는 것만 throws에 선언할 수 있다.

abstract class SuperClass {

public abstract void a();

protected abstract void b();

public abstract void c() throws java.io.IOException;

public abstract void d() throws Exception;

}

abstract class SubClass extends SuperClass {

protected abstract void a(); // Error.

public abstract void b(); // OK.

public abstract void c() throws Exception; // Error.

public abstract void d() throws java.io.IOException; // OK.

}

위의 예제에서 메쏘는 b는 접근이 더 넓혀졌으며 메쏘드 d는 throws 절이 더 좁혀졌다. 추상 메쏘드의 추상 메쏘드에 의한 재정의는 이런 이유로도 쓰일 수 있다.

* + 1. 접근할 수 없는 조상의 메쏘드

조상의 필드나 스태틱 메쏘드는 가려지더라도 접근할 수 있다. 그러나 재정의된 메쏘드는 그렇지 않은 경우가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Me m = new Me();

m.method();

}

}

class Grandma {

int hiddenField = 65;

public static void hiddenMethod() {

System.out.println("Grandma.hiddenMethod()");

}

}

class Mom extends Grandma {

int hiddenField = 40;

public static void hiddenMethod() {

System.out.println("Mom.hiddenMethod()");

}

}

class Me extends Mom {

int hiddenField = 15;

public static void hiddenMethod() {

System.out.println("Me.hiddenMethod()");

}

public void method() {

System.out.println("this.hiddenField = " + this.hiddenField);

System.out.println("super.hiddenField = " + super.hiddenField);

System.out.println("((Grandma)super).hiddenField = " + ((Grandma)super).hiddenField);

Me.hiddenMethod();

Mom.hiddenMethod();

Grandma.hiddenMethod();

}

}

결과는 다음과 같을 것이다.

this.hiddenField = 15

super.hiddenField = 40

((Grandma)super).hiddenField = 65

Me.hiddenMethod()

Mom.hiddenMethod()

Grandma.hiddenMethod()

이번에는 재정의된 메쏘드를 살펴보자.

public class Example {

public static void main(String[] args) {

Me m = new Me();

m.method();

}

}

class Grandma {

public void overriddenMethod() {

System.out.println("Grandma.overriddenMethod()");

}

}

class Mom extends Grandma {

public void overriddenMethod() {

System.out.println("Mom.overriddenMethod()");

}

}

class Me extends Mom {

public void overriddenMethod() {

System.out.println("Me.overriddenMethod()");

}

public void method() {

overriddenMethod();

super.overriddenMethod();

((Grandma)super).overriddenMethod();

}

}

결과는 다음과 같을 것이다.

Me.overriddenMethod()

Mom.overriddenMethod()

Mom.overriddenMethod()

보통은 Mom.overriddenMethod()에서 super.overriddenMethod()를 불러줌으로써 Grandma.overriddenMethod()을 사용한다. 그러나 만약 Grandma.overriddenMethod()를 사용하고 싶지 않다면 Mom 클래스처럼 재정의함으로써 자신의 서브 클래스에서는 Grandma.overriddenMethod()를 절대로 접근하지 못하게 할 수도 있다.

* + 1. private 메쏘드와 디폴트 접근 메쏘드

private 메쏘드는 상속되지 않기 때문에 재정의되지도 가려지지도 않는다. 비록 상속 관계라도 private 메쏘드는 아무런 관계도 맺지 않는 것이다. 그러므로 그만큼 안전하다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.callMethod();

}

}

class SuperClass {

private void method() {

System.out.println("SuperClass.method()");

}

public void callMethod() {

method();

}

}

class SubClass extends SuperClass {

private void method() {

System.out.println("SubClass.method()");

}

}

결과는 다음과 같을 것이다.

SuperClass.method()

만약 재정의되었다면 SubClass.method()가 불렸을 것이다.

이번에는 디폴트 접근 메쏘드로 바꾸어 보자.

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.callMethod();

}

}

class SuperClass {

void method() { // 여기가 바뀌었다.

System.out.println("SuperClass.method()");

}

public void callMethod() {

method();

}

}

class SubClass extends SuperClass {

void method() { // 여기가 바뀌었다.

System.out.println("SubClass.method()");

}

}

결과는 다음과 같을 것이다.

SubClass.method()

즉 재정의가 된 것이다. 두 클래스 SuperClass, SubClass가 같은 패키지에 있기 때문에 상속이 되는 것이다. 그럼 다른 패키지에 넣어 보자. 다음 예제를 보자. 두 개의 파일로 이루어져 있다.

Example.java

package subclass;

import superclass.SuperClass;

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

s.callMethod();

}

}

class SubClass extends SuperClass {

void method() {

System.out.println("SubClass.method()");

}

}

SuperClass.java

package superclass;

public class SuperClass {

void method() {

System.out.println("SuperClass.method()");

}

public void callMethod() {

method();

}

}

결과는 다음과 같을 것이다.

SuperClass.method()

패키지를 다르게 하니까 재정의되지 않는다.

서로 무관한 메쏘드들이기 때문이 재정의나 감추기에 따르는 제한도 따르지 않는다. 다음 예제는 무사히 컴파일될 것이다.

class SuperClass {

private void a() {}

private static void b() {}

private void c() throws java.io.IOException {}

}

class SubClass extends SuperClass {

private int a() { // 반환형이 달라도 OK.

return 0;

}

private static int b() { // 반환형이 달라도 OK.

return 0;

}

private void c() throws Exception {} // throws 가 넓혀져도 OK.

}

이번에는 패키지 접근에 대해 알아보자. 다음 예제를 보자.

class SuperClass {

void a() {}

static void b() {}

void c() throws java.io.IOException {}

}

class SubClass extends SuperClass {

int a() { // Error.

return 0;

}

static int b() { // Error.

return 0;

}

void c() throws Exception {} // Error.

}

이번에는 패키지를 달리해 보자. 두 개의 파일로 이루어져 있다.

SuperClass.java

package superclass;

public class SuperClass {

void a() {}

static void b() {}

void c() throws java.io.IOException {}

}

SubClass.java

package subclass;

import superclass.SuperClass;

public class SubClass extends SuperClass {

int a() { // OK.

return 0;

}

static int b() { // OK.

return 0;

}

void c() throws Exception {} // OK.

}

protected 메쏘드를 쓸 것인가 private 메쏘드를 쓸 것인가를 심사숙고해야 한다. 그리고 가능하다면 더욱 안전한 private 메쏘드를 사용해야 한다. private 메쏘드는 또한 다른 클래스 심지어 서브 클래스에도 알려지지 않기 때문이 마음대로 없애거나 추가할 수 있다는 장점이 있다. 이것은 어느정도 디폴트 접근 메쏘드에도 적용된다. 왜냐하면 하나의 패키지에 있는 클래스는 한 사람 또는 아주 밀접하게 협동하는 그룹에 의해 만들어지기 때문이다.

* + 1. 정적 메쏘드를 부르는 이상한 방법(super 키워드)

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Sub().instanceMethod();

}

}

class Super {

public static void staticMethod() {

System.out.println("Super.staticMethod()");

}

}

class Sub extends Super {

public void instanceMethod() {

super.staticMethod(); // OK. but bad style.

Super.staticMethod(); // OK. and good style.

}

}

결과는 다음과 같을 것이다.

Super.staticMethod()

Super.staticMethod()

super 키워드를 이용해서 정적 메쏘드를 부르고 있는데 좋은 방법은 아니다.

* + 1. 구식 선언 방식

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println(getIntArray()[2]); // 이런식으로 코딩하지 말라.

System.out.println(getIntAnotherArray()[2]);

}

public static int[] getIntArray() {

return new int[]{1, 2, 3};

}

public static int getIntAnotherArray()[] { // []가 어디에 붙어 있는지 주목하라.

return new int[]{111, 222, 333};

}

}

결과는 다음과 같을 것이다.

3

333

이전에 이런 선언 방식을 썼다는 이유로 여전히 문법적으로는 배열의 []를 메쏘드 매개변수 리스트 다음에 붙이는 방식이 허용되고 있다. 그러나 사용하지 않는 것이 좋을 것이다.

매개변수 선언도 마찬가지 방식이 허용된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

printArray(new int[]{1, 2, 3});

printAnotherArray(new int[]{111, 222, 333});

}

public static void printArray(int[] array) {

for (int i = 0; i < array.length; i++) {

System.out.print(array[i] + " ");

}

System.out.println();

}

public static void printAnotherArray(int array[]) { // []가 어디에 붙어 있는지 주목하라.

printArray(array);

}

}

결과는 다음과 같을 것이다.

1 2 3

111 222 333

* 1. 접근(access)
     1. 타입의 접근과 멤버의 접근

어떤 타입에 먼저 접근해야 그 타입의 멤버에 접근할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A.PrivateClass.publicMember = 1; // Error

}

}

class A {

private static class PrivateClass {

public static int publicMember;

}

}

publicMember가 public으로 선언되어 있지만 그 멤버가 선언된 클래스인 PrivateClass에 접근할 수 없으면 publicMember에도 접근할 수 없다.

* + 1. 디폴트 생성자의 접근

디폴트 생성자의 접근은 클래스의 접근과 같다. 다음 예제를 보자.

public class Example {

}

class A {

}

두 클래스 모두 생성자가 정의되지 않았다. 그러므로 다음과 같이 자동으로 디폴트 생성자가 만들어진다.

public class Example {

public Example() {

super();

}

}

class A {

A() {

super();

}

}

클래스가 public이면 디폴트 생성자도 public이고 클래스가 패키지 접근이면 디폴트 생성자도 패키지 접근이 된다.

정말 그런지 테스트해 볼 수도 있다. 다음 예제를 보자.

public class Example {

private class Private {}

class Package {}

protected class Protected {}

public class Public {}

}

예제를 컴파일 한 후 다음과 비슷하게 디컴파일하면

C:\Decopile\jad>jad -noinner Example

C:\Decopile\jad>jad -noinner Example$Private

C:\Decopile\jad>jad -noinner Example$Package

C:\Decopile\jad>jad -noinner Example$Protected

C:\Decopile\jad>jad -noinner Example$Public

아래와 비슷한 결과가 나올 것이다.

// Decompiled by Jad v1.5.7f. Copyright 2000 Pavel Kouznetsov.

public class Example

{

public Example()

{

}

}

class Example$Private

{

private Example$Private(Example this$0)

{

}

}

class Example$Package

{

Example$Package(Example this$0)

{

}

}

public class Example$Protected

{

protected Example$Protected(Example this$0)

{

}

}

public class Example$Public

{

public Example$Public(Example this$0)

{

}

}

* + 1. private은 per-object가 아니라 per-class로 작동한다

private 멤버에의 접근은 같은 객체가 아니더라도 같은 클래스이기만 하면 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A one = new A(1);

A two = new A(2);

one.accessAnotherObjectsPrivateField(one);

one.accessAnotherObjectsPrivateField(two);

}

}

class A {

private int privateField;

public A(int i) {

privateField = i;

}

public void accessAnotherObjectsPrivateField(A a) {

System.out.println(a.privateField); // 여기를 주목하라.

}

}

결과는 다음과 같을 것이다.

1

2

하나의 객체가 다른 객체의 private 멤버에 접근할 수 있음을 알 수 있다. 같은 클래스라도 서로 다른 객체이면 private에 접근하지 못하게 언어를 만들 수도 있다. 그러나 하나의 클래스는 한 사람이 만들거나 밀접히 협력하는 사람들에 의해 만들어지기 때문에 자바처럼 다른 객체의 private에 접근할 수 있도록 만들어도 크게 문제될 것은 없다.

* + 1. protected의 진정한 의미는?

protected가 서브클래스에서 접근할 수 있다는 것은 누구나 알 것이다. 많은 사람이 모르는 것은 protected가 같은 패키지에 속하는 클래스에서도 접근할 수 있다는 사실이다. 그러므로 public, protected, default(package, friendly), private으로 갈수록 접근이 점점 제한되는 것이다. protected의 의미에 대한 이야기는 여기가 끝이 아니다. 다음 예제를 보자. 두 개의 파일로 되어 있다.

SuperClass.java

package super\_package;

public class SuperClass {

protected int protectedValue;

}

SubClass.java

package sub\_package;

import super\_package.\*;

public class SubClass extends SuperClass {

public void methodA(SubClass sub) {

System.out.println(sub.protectedValue);

}

public void methodB(SubClass sub) {

SuperClass sup = sub;

System.out.println(sup.protectedValue); // Error.

}

public void methodC(SuperClass sup) {

System.out.println(sup.protectedValue); // Error.

}

public void methodD(SuperClass sup) {

SubClass sub = (SubClass)sup;

System.out.println(sub.protectedValue);

}

}

두 군데에서 컴파일타임 오류가 난다. 분명히 서브 클래스에서 접근했는데도 수퍼 클래스의 protected 멤버를 접근할 수 없는 경우가 있는 것이다. 왜 이런 식으로 만들었는지 설명해 보겠다. 위의 예에서 볼 수 있듯이 SubClass에서는 SuperClass에서 정의된 protected 멤버가 속한 객체가 SubClass일 때만 접근이 허용된다. SuperClass의 서브 클래스는 SubClass만은 아닐 것이다. 예를 들어 AnotherSubClass도 SuperClass의 서브 클래스일 수 있다. AnotherSubClass도 SuperClass의 protected 멤버를 상속받는다. 그리고 AnotherSubClass는 그 protected 멤버의 contract에 어떤 것을 덧붙일 수 있다. 그 덧붙여진 contract는 SubClass의 contract와 충돌할 수도 있다. 그래서 SubClass에서는 그 객체가 SubClass또는 SubClass의 서브 클래스임이 보장될 때만 SuperClass에서 정의된 protected 멤버에 접근 할 수 있다.

* + 1. 디폴트 접근 클래스를 다른 패키지에서 사용하기

디폴트 접근 클래스를 다른 패키지에서는 보통 사용할 수 없다. 그러므로 다음 예제는 컴파일되지 않을 것이다. 다음 예제는 두개의 파일로 이루어져 있다.

Example.java

package example;

import another.\*;

public class Example {

public static void main(String[] args) {

new Another(); // Error.

}

}

Another.java

package another;

class Another { // 디폴트 접근이다.

}

그러나 디폴트 접근 클래스에 접근하는 방법이 있다. 다음 예제를 보자. 세개의 파일로 이루어져 있다.

Example.java

package example;

import another.\*;

public class Example {

public static void main(String[] args) {

Another a = Factory.getAnother();

a.method();

}

}

Another.java

package another;

public class Another {

public void method() {

System.out.println("Another.method()");

}

}

class PackageAccess extends Another {

public void method() {

System.out.println("PackageAccess.method()");

}

public void anotherMethod() {

System.out.println("PackageAccess.anotherMethod()");

}

}

Factory.java

package another;

public class Factory {

public static Another getAnother() {

return new PackageAccess();

}

}

결과는 다음과 같을 것이다.

PackageAccess.method()

여기서 주의할 것은 Example.java의 코드에 PackageAccess라는 문자열이 보이지 않는다는 것이다. PackageAccess에 접근하기 위해 그것의 수퍼 클래스를 이용했다. 그러므로 Example 클래스에서 PackageAccess.anotherMethod() 메쏘드는 여전히 사용할 수 없다. 여기서는 Another가 PackageAccess의 수퍼클래스였는데 수퍼 인터페이스였어도 마찬가지 결과가 나왔을 것이다.

* 1. 인터페이스
     1. 여러 인터페이스에서 상속된 메쏘드

다음 예제를 보자.

interface 군인 {

void 열받는다(int stress);

}

interface 보일러 {

void 열받는다(int fuel);

}

interface 군바리보일러 extends 군인, 보일러 {

}

두 개의 인터페이스에서 서명(signature)와 반환형이 같은 메쏘드를 상속받아도 된다. 그러나 여기서 문제는 인터페이스 군바리보일러가 인터페이스 군인과 인터페이스 보일러의 contract 즉 의미를 만족시킬 수 있느냐이다.

위의 예에서 서명이 틀리다면 그것은 두개의 독립된 메쏘드이기 때문에 즉 메쏘드가 중복정의된 것이기 때문에 아무런 문제가 없을 것이다. 만약 서명은 같은데 반환형 다르다면 컴파일타임 오류가 난다.

* + 1. 같은 서명(signature)의 메쏘드가 이미 있는 경우

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Interface i = new SubClass();

i.method();

}

}

class SuperClass {

public void method() {

System.out.println("SuperClass.method");

}

}

class SubClass extends SuperClass implements Interface {

}

interface Interface {

void method();

}

결과는 다음과 같을 것이다.

SuperClass.method

Interface.method()와 같은 서명과 같은 반환형의 메쏘드가 이미 SubClass의 수퍼 클래스인 SuperClass에 있다. 이 때에는 SuperClass.method()가 Interface.method()를 구현한 것으로 친다. 그러나 throws절에서 문제가 생길 수 있다. 다음 예제를 보자.

class SuperClass {

public void method() throws Exception {

System.out.println("SuperClass.method");

}

}

class SubClass extends SuperClass implements Interface {

}

interface Interface {

void method() throws java.io.IOException;

}

컴파일이 되지 않을 것이다. 예외 상황을 다룬 장에서 상속과 예외 상황을 참조하라.

* 1. 객체의 생성
     1. 생성자의 변경자 제한

생성자에 abstract변경자를 사용할 수 없다. 생성자는 멤버가 아니므로 상속도 되지 않는다. 그러므로 abstract 변경자는 말이 안된다.

생성자에 static변경자를 사용할 수 없다. 정적 생성자와 인스턴스 생성자를 나누는 것도 말이 안된다.

생성자에 final 변경자를 사용할 수 없다. 이것은 생성자가 상속되지 않는다는 것과 관련된다.

생성자에 native 변경자를 사용할 수 없다. 이것은 구현하기 쉽게 하기 위해서라고 한다.

생성자에 strictfp 변경자는 사용할 수 없다. 이것에 대해서는 strictfp를 다루는 부분을 참조하라.

생성자에 synchronized 변경자를 사용할 수 없다. 객체에 lock을 사용하기 위해서는 객체가 만들어져야 한다.

* + 1. 객체가 생성되는 방법

객체는 어떤 클래스의 인스턴스(instance)이거나 배열 객체이다. 객체는 여러가지 방식으로 생성될 수 있다.

먼저 new 연산자를 이용하는 방법이 있다. 클래스의 객체는 new 연산자를 이용하면 그 클래스의 생성자가 불린다. 배열도 new 연산자를 이용해서 생성할 수 있다. 배열을 초기화할 때에는 {}를 이용해서 배열객체를 생성할 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Object o = new Object();

int[] intArray = new int[10];

short[] shortArray = {10, 20, 30};

A a = new A();

}

}

class A {

public A() {

System.out.println("constructor of A");

}

}

결과는 다음과 같을 것이다.

constructor of A

String 객체는 new 연산자를 이용하는 방법 말고도 다른 방법으로 객체를 만들 수 있다. 문자열 리터럴에 의해 String 객체가 만들어질 수도 있고, 문자열 연결 연산자 +에 의해 String 객체가 만들어질 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String a = new String("a");

String b = "b";

String c = a + b;

System.out.println(a + " " + b + " " + c);

}

}

결과는 다음과 같을 것이다.

a b ab

리플렉션 메커니즘으로 객체를 만들수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

A a = (A)A.class.newInstance();

} catch (IllegalAccessException e) {

System.out.println(e);

} catch (InstantiationException e) {

System.out.println(e);

}

}

}

class A {

public A() {

System.out.println("constructor of A");

}

}

결과는 다음과 같을 것이다.

constructor of A

직렬화(serialization)을 이용해서 객체를 만들 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A(); // 여기서 생성자가 불린다.

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

try {

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(a);

} catch (IOException e) {

System.out.println(e);

}

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

try {

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject(); // 여기서는 생성자가 불리지 않는다.

} catch (ClassNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

class A implements Serializable {

public A() {

System.out.println("constructor of A");

}

}

결과는 다음과 같을 것이다.

constructor of A

clone 메쏘드를 이용할 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int[] array = {1, 2, 3};

int[] clonedArray = (int[])array.clone();

A a = new A(); // 여기서 생성자가 불린다.

A clonedA = (A)a.clone(); // 여기서는 생성자가 불리지 않는다.

}

}

class A implements Cloneable {

public A() {

System.out.println("constructor of A");

}

public Object clone() {

Object obj = null;

try {

obj = super.clone();

} catch (CloneNotSupportedException e) {

System.out.println(e);

} finally {

return obj;

}

}

}

결과는 다음과 같을 것이다.

constructor of A

직렬화와 clone 메쏘드를 이용할 때는 생성자가 불리지 않았다. 그것은 native 메쏘드를 이용했기 때문이다. 만약 객체가 메모리에 저장되는 방식을 잘 알고 있다면 자신이 native 메쏘드를 이용해서 객체를 만들 수도 있다. 그러면 생성자를 우회하기 때문에 깨진 객체(contract와 모순되는)를 만들 수도 있을 것이다.

결론적으로 말하면 특별한 경우가 아니면 new 문을 사용하는 것이 쓰기에도 편리하고 코드를 보는 사람도 이해하기 쉽다.

* + 1. 생성자를 부르는 this와 super에서의 제한

생성자를 부르는 this와 super에서는 인스턴스 필드나 메쏘드를 사용할 수 없다. 다음 예제를 보자.

class SuperClass {

protected int x;

protected int y;

public SuperClass(int a, int b) {

}

}

class SubClass extends SuperClass {

private int z;

public SubClass(int a, int b) {

this(a, b, z); // 인스턴스 필드 z가 쓰일 수 없다.

}

public SubClass(int a, int b, int c) {

super(x, y); // 인스턴스 필드 x, y가 쓰일 수 없다.

}

public SubClass() {

this(instanceMethod(), 1); // 인스턴스 메쏘드 instanceMethod가 쓰일 수 없다.

}

public int instanceMethod() {

return 1;

}

}

컴파일이 되지 않을 것이다.

물론 this도 쓰일 수 없다. 그러므로 다음 예제를 컴파일되지 않을 것이다.

class A {

public A(A a) {

}

public A() {

this(this); // Error

}

}

정적 필드나 정적 메쏘드는 사용할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new A();

}

}

class A {

static int staticField = 123;

public static int staticMethod() {

return 999;

}

public A(int a, int b) {

System.out.println(a);

System.out.println(b);

}

public A() {

this(staticField, staticMethod());

}

}

결과는 다음과 같을 것이다.

123

999

* + 1. 복제 생성자(copy constructor)

생성자 중에는 다른 객체를 그대로 복제하는 생성자가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String a = new String("a");

String b = new String(a);

System.out.println(a == b);

System.out.println(a.equals(b));

}

}

결과는 다음과 같을 것이다.

false

true

String 클래스에는 복제 생성자가 있다. 그러나 이 방식은 잘 쓰이지 않는다. 복제에 쓰이는 더 일반적인 방식은 clone 메쏘드이다. 복제 생성자를 만들때는 clone 메쏘드를 재정의할 때처럼 여러가지를 주의해야 한다. 예를 들어 shallow copy가 되는 것에 주의해야 한다. 자세한 것은 Object 클래스를 다루는 부분을 참조하라.

* + 1. 생성자를 부르는 this의 순환

다음 예제처럼 생성자를 부르는 this가 순환하면 컴파일타임 오류가 난다.

class A {

public A() {

this(0);

}

public A(int a) {

this(a, 0);

}

public A(int a, int b) {

this();

}

}

비슷한 것을 메쏘드에서 하면 어떻게 될까? 다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

method();

}

public static void method() {

method(0);

}

public static void method(int a) {

method(a, 0);

}

public static void method(int a, int b) {

method();

}

}

컴파일은 잘 된다. 결과는 다음과 비슷할 것이다.

java.lang.StackOverflowError

at Example.method(Example.java:11)

at Example.method(Example.java:7)

at Example.method(Example.java:15)

at Example.method(Example.java:11)

at Example.method(Example.java:7)

at Example.method(Example.java:15)

at Example.method(Example.java:11)

…

스택(stack)이 부족해질 때까지 계속 재귀적으로 호출(recursive call)한다.

* + 1. 객체 생성에서의 무한 루프

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new A();

}

}

class A {

A a = new A();

}

컴파일은 잘 되지만 결과는 다음과 비슷할 것이다.

java.lang.StackOverflowError

at A.<init>(Example.java:8)

at A.<init>(Example.java:8)

at A.<init>(Example.java:8)

at A.<init>(Example.java:8)

…

* + 1. 객체가 생성될 때의 초기화 순서

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new SubClass();

}

}

class SuperClass {

static int staticField = initializerForStaticField();

int instanceField = initializerForInstanceField();

static {

System.out.println("SuperClass static initializer");

}

{

System.out.println("SuperClass instance initializer");

}

public SuperClass() {

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

}

static int initializerForStaticField() {

System.out.println("SuperClass.initializerForStaticField()");

return 1;

}

static int initializerForInstanceField() {

System.out.println("SuperClass.initializerForInstanceField()");

return 1;

}

}

class SubClass extends SuperClass {

static {

System.out.println("SubClass static initializer");

}

static int staticField = initializerForStaticField();

{

System.out.println("SubClass instance initializer");

}

int instanceField = initializerForInstanceField();

public SubClass() {

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

}

static int initializerForStaticField() {

System.out.println("SubClass.initializerForStaticField()");

return 1;

}

static int initializerForInstanceField() {

System.out.println("SubClass.initializerForInstanceField()");

return 1;

}

}

결과는 다음과 같다.

SuperClass.initializerForStaticField()

SuperClass static initializer

SubClass static initializer

SubClass.initializerForStaticField()

SuperClass.initializerForInstanceField()

SuperClass instance initializer

SuperClass.SuperClass()

SubClass instance initializer

SubClass.initializerForInstanceField()

SubClass.SubClass()

전체적으로 볼 때 먼저 수퍼 클래스의 정적 초기화(정적 초기화 블록static inializer, 정적 필드 초기화문initializer for static field)가 먼저 일어난다. 그 다음에 서브 클래스의 정적 초기화가 일어난다. 그 다음에 수퍼 클래스의 인스턴스 초기화(인스턴스 초기화 블록instance initializer, 인스턴스 필드 초기화문initializer for instance field)가 일어난 후 수퍼 클래스의 생성자가 불린다. 그 다음에 서브 클래스의 인스턴스 초기화가 일어난 후에 서브 클래스의 생성자가 불린다. 이 전체적인 순서는 코드 상의 위치와 상관 없다. 정적 초기화는 인스턴스 초기화보다 항상 먼저 일어난다. 그 이유는 정적 초기화는 클래스가 로딩될 때 일어나며 인스턴스 초기화는 객체가 생성될 때 일어나는데 객체가 생성되기 위해서는 먼저 클래스가 로딩되어야 하기 때문이다. 그리고 인스턴스 초기화가 먼저 일어난 후에 생성자가 불린다. 또 하나의 원칙은 수퍼 클래스가 서브 클래스보다 먼저 초기화된다는 것이다. 위의 예제의 결과를 보면 코드 상의 위치가 분명히 영향을 끼쳤음을 볼 수 있다. 예를 들어 수퍼 클래스에서는 정적 필드 초기화문이 정적 초기화 블록보다 먼저 실행되었고 서브 클래스에서는 그 반대다. 왜냐하면 코드상의 위치가 반대이기 때문이다. 인스턴스 필드 초기화문과 인스턴스 초기화 블록의 관계도 마찬가지이다.

그러나 코드를 읽는 사람이 이런 것까지 신경쓰게 코딩하는 것은 정말 안 좋은 습관이다. 코드를 보는 사람은 정적 필드가 인스턴스 필드보다 먼저 초기화된다는 정도만 알아도 코드를 알아볼 수 있도록 해야한다. 아주 특별한 경우를 제외하면.

* + 1. 생성자에서 부르는 메쏘드

생성자에서 메쏘드를 부르는 경우는 흔하다. 생성자에서 복잡한 일을 한다면 메쏘드들로 쪼개는 것이 좋다. 여기서 문제가 되는 것은 그 메쏘드들이 재정의될 수 있다는 것이다. 다음 예제를 보자. 여기서는 굳이 메쏘드를 부르게 할 필요는 없지만 설명을 위한 것이다.

public class Example {

public static void main(String[] args) {

System.out.println(new Position(111, 999));

System.out.println(new WarpPosition(111, 999));

}

}

class Position {

private int x;

private int y;

public Position(int x, int y) {

setPosition(x, y);

}

public void setPosition(int x, int y) {

this.x = x;

this.y = y;

}

public String toString() {

return "[" + x + "," + y + "]";

}

}

class WarpPosition extends Position {

public WarpPosition(int x, int y) {

super(x, y); // 여기서 제대로 했지만.

}

public void setPosition(int x, int y) {

super.setPosition(y, x); // 이런 일이 벌어질 수 있다.

}

}

결과는 다음과 같을 것이다.

[111,999]

[999,111]

setPosition 메쏘드를 재정의해서 수퍼 클래스의 생성자가 하는 일을 망쳤다. 심지어 두 필드 x, y가 private임에도 그렀다. 이런 일이 있을 수 있다는 점에 조심해야 한다. 이것을 해결하는 방법은 여러가지이다. 한 가지 방법은 그 메쏘드를 private 메쏘드로 바꾸는 것이다. 그러나 private으로 할 수 없는 경우도 있다. 그럴 때는 final로 해도 된다면 final 메쏘드로 바꾸거나 정적(static)으로 해도 된다면 정적 메쏘드로 바꾸면 된다. private 메쏘드, final 메쏘드, 정적 메쏘드는 모두 재정의될 수 없다. 이것도 저것도 안되면 문서에다 적어 놓아서 주의를 주는 방법이 있다.

* + 1. new 문과 쓰레기 수거(garbage collection)

new 문으로 객체가 생성될 때 메모리가 부족하면 반드시 쓰레기 수거를 한다. 그 다음에도 메모리가 부족하면 어쩔 수 없이 OutOfMemeryError가 발생한다.

* 1. 중첩 타입
     1. simple name과 qualified name

중첩 타입을 simple name으로 쓸 수 있는 경우와 그렇지 못한 경우가 있다(simple name과 qualified name에 대해서는 언어의 기본 요소를 다루는 장을 참조하라). 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

//Nested a; // Error.

Outer.Nested n;

//Inner b; // Error.

Outer.Inner i;

//n = new Nested(); // Error.

n = new Outer.Nested();

Outer o = new Outer();

i = o.new Inner();

//i = o.new Outer.Inner(); // Error.

}

}

class Outer {

class Inner {

}

static class Nested {

}

public void makeObjects() {

new Inner();

new Outer.Inner();

new Nested();

new Outer.Nested();

}

}

class OuterSub extends Outer {

public void makeObjects() {

new Inner();

new Outer.Inner();

new Nested();

new Outer.Nested();

}

}

Inner와 Nested가 선언된 Outer 내에서는 simple name을 사용할 수 있다. 그리고 Outer의 서브 클래스인 OuterSub에서도 마찬가지로 simple name을 사용할 수 있다. 이것은 필드나 메쏘드 등 다른 멤버와 마찬가지다.

그러나 Example 클래스에서는 simple name을 사용할 수 없다. 그러나 예외적으로 외부 객체를 명시적으로 지정하면서 이너 클래스를 생성할 때는 다음과 같이 simple name을 사용해야 한다.

i = o.new Inner();

* + 1. 내부 클래스의 정적 멤버

non-static 중첩 클래스 즉 내부 클래스는 정적 멤버를 가질 수 없다. 예외는 컴파일 타임 상수이다. 다음 예제를 보자.

class Outer {

class Inner {

static int staticfield; // Error.

static void staticMethod() {} // Error.

static {} // Error.

static class StaticClass {} // Error.

interface Interface {} // Error.

}

}

컴파일이 되지 않을 것이다. 멤버 클래스가 정적 클래스라면 사정이 다르다. 다음 예제를 보자.

class Outer {

static class Inner {

static int staticfield; // OK.

static void staticMethod() {} // OK.

static {} // OK.

static class StaticClass {} // OK.

interface Interface {} // OK.

}

}

컴파일이 잘 된다. 내부 클래스 안에 인스턴스 멤버가 있다면 그것도 허용된다. 다음 예제를 보자.

class Outer {

class Inner {

int staticfield; // OK.

void staticMethod() {} // OK.

{} // OK.

class StaticClass {} // OK.

}

}

컴파일이 잘 될 것이다. 위에서 얘기했듯이 컴파일 타임 상수는 허용된다. 그러므로 다음 예제는 컴파일이 된다.

class Outer {

class Inner {

static final int staticField = 1; // OK.

}

}

이너 클래스가 정적 멤버를 선언할 수는 없지만 상속받을 수는 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

o.method();

}

}

class SuperClass {

static int staticField = 123;

}

class Outer {

class Inner extends SuperClass {

}

public void method() {

Inner i = new Inner();

System.out.println(i.staticField);

}

}

결과는 다음과 같을 것이다.

123

왜 내부 클래스에 정적 멤버가 금지되었는지는 글쓴이도 잘 모르겠다.

* + 1. blank final의 초기화

외부 클래스의 blank final 필드는 중첩 클래스에서 초기화될 수 없다. 다음 예제는 컴파일이 되지 않을 것이다.

class Outer {

static final int blankFinal;

static class Inner {

static {

blankFinal = 123; // Error.

}

}

}

인스턴스 필드도 마찬가지다.

class Outer {

final int blankFinal;

Outer() {

class Inner {

{

blankFinal = 123; // Error.

}

}

new Inner();

}

}

* + 1. 정적 멤버 클래스

정적 멤버 클래스에는 외부 객체가 없다. 그러므로 외부 클래스의 인스턴스 메쏘드나 인스턴스 필드에는 접근할 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Outer.Nested().method();

}

}

class Outer {

static int staticField = 1111;

int instanceField = 2222;

static void staticMethod() {

System.out.println("Outer.staticMethod()");

}

void instanceMethod() {

System.out.println("Outer.instanceMethod()");

}

static class Nested {

void method() {

System.out.println(staticField);

System.out.println(instanceField); // Error.

staticMethod();

instanceMethod(); // Error.

}

}

}

두 군데서 컴파일타임 오류가 날 것이다. 컴파일타임 오류가 나는 두 줄은 주석화하거나 없애고 실행하면 결과는 다음과 같을 것이다.

1111

Outer.staticMethod()

* + 1. 지역 클래스에서 지역 변수의 접근하기

지역 클래스는 심지어 그 지역(메쏘드, 생성자)의 지역변수나 매개변수에 접근할 수도 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

new OuterClass().method(123);

}

}

class OuterClass {

public void method(final int parameter) {

final double localVariable = 2134.432;

class LocalClass {

public void localMethod() {

System.out.println(parameter);

System.out.println(localVariable);

}

}

new LocalClass().localMethod();

}

}

결과는 다음과 같다.

123

2134.432

매개변수와 지역변수를 final로 선언했다는 것에 주의하라. 그렇게 하지 않으면 컴파일타임 오러가 날 것이다. 또한 지역 클래스에서 사용되는 지역 변수는 definitely assigned되어야 한다. 매개 변수는 당연히 인수로 넘어오는 값으로 설정되기 때문에 문제될 것이 없지만. 심지어 다음과 같이 실제로 객체가 생성되기 전에 대입되어도 컴파일타임 오류가 난다.

public class Example {

public static void main(String[] args) {

new OuterClass().method(123);

}

}

class OuterClass {

public void method(final int parameter) {

final double localVariable;

class LocalClass {

public void localMethod() {

System.out.println(parameter);

System.out.println(localVariable);

}

}

localVariable = 23.23;

new LocalClass().localMethod();

}

}

왜 그런지 알아보자. 먼저 다음 예제를 보자. 지역 클래스로 만들어진 객체는 그 지역 클래스가 선언된 메쏘드의 실행이 끝나도 살아 남을 수 있다.

public class Example {

public static void main(String[] args) {

OuterClass oc = new OuterClass();

Method m = oc.method(123);

m.method(); // m 이 가리키는 객체는 LocalClass의 객체이다. 아직도 살아 남았다.

}

}

class OuterClass {

public Method method(final int parameter) {

final double localVariable = parameter + 123.213;

class LocalClass implements Method {

public void method() {

System.out.println(parameter);

System.out.println(localVariable);

}

}

return new LocalClass();

}

}

interface Method {

void method();

}

결과는 다음과 같을 것이다.

123

246.213

무언가 이상하지 않은가? LocalClass.method()에서 쓰이는 parameter와 localMethod는 Outer의 method 메쏘드의 실행이 끝나면서 사라진다. 그렇다면 위와 같은 결과는 어떻게 가능한가? 다시 한번 디컴파일러의 힘을 빌어보자.

C:\ decompiler\Jad>jad -noinner OuterClass$1$LocalClass.class

와 비슷하게 하면 다음과 비슷한 결과가 나올 것이다.

// Decompiled by Jad v1.5.7f. Copyright 2000 Pavel Kouznetsov.

package bababook.bavalang;

import java.io.PrintStream;

class OuterClass$1$LocalClass

implements Method

{

OuterClass$1$LocalClass(int val$parameter, double val$localVariable)

{

this.val$parameter = val$parameter;

this.val$localVariable = val$localVariable;

}

public void method()

{

System.out.println(val$parameter);

System.out.println(val$localVariable);

}

private final double val$localVariable; /\* synthetic field \*/

private final int val$parameter; /\* synthetic field \*/

}

여기서 val$localVariable과 val$parameter라는 필드가 만들어졌다. 이 필드는 final 필드이다. 만약 접근되는 지역 변수가 final이 아니라면 그 지역 변수의 값이 바뀔 때마다 지역 변수에 컴파일러가 만든 필드도 같이 변해야 할 것이다. 이런 골치 아픈 상황을 만들지 않기 위해 final만 허용한 것이다.

new 문 이전에 final 필드를 설정해도 컴파일타임 오류가 나는 이유는 잘 모르겠다.

* + 1. static context에 선언된 내부 클래스

static context는 정적(static) 메쏘드 등처럼 인스턴스와 연결되지 않은 곳을 말한다.

내부 클래스의 객체에는 외부 객체가 연결되어 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer("I'm Outer Object");

o.makeInnerObject();

}

}

class Outer {

private String outerField;

public Outer(String str) {

outerField = str;

}

class Inner {

private String innerField;

public Inner(String str) {

innerField = str;

}

public void showOuterObject() {

System.out.println(this);

System.out.println(Outer.this);

}

public String toString() {

return innerField;

}

}

public void makeInnerObject() {

Inner i = new Inner("I'm Inner Object");

i.showOuterObject();

}

public String toString() {

return outerField;

}

}

결과는 다음과 같을 것이다.

I'm Inner Object

I'm Outer Object

이것은 지역 클래스일 때도 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer("I'm Outer Object");

o.makeInnerObject();

}

}

class Outer {

private String outerField;

public Outer(String str) {

outerField = str;

}

public void makeInnerObject() {

class Inner {

private String innerField;

public Inner(String str) {

innerField = str;

}

public void showOuterObject() {

System.out.println(this);

System.out.println(Outer.this);

}

public String toString() {

return innerField;

}

}

Inner i = new Inner("I'm Inner Object");

i.showOuterObject();

}

public String toString() {

return outerField;

}

}

결과는 다음과 같을 것이다.

I'm Inner Object

I'm Outer Object

지역 클래스를 정적 메쏘드에서 선언하면 어떻게 될까? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer("I'm Outer Object");

o.makeInnerObject();

}

}

class Outer {

private String outerField;

public Outer(String str) {

outerField = str;

}

public static void makeInnerObject() { // static 메쏘드로 바꾸었다.

class Inner {

private String innerField;

public Inner(String str) {

innerField = str;

}

public void showOuterObject() {

System.out.println(this);

System.out.println(Outer.this); // 컴파일 에러.

}

public String toString() {

return innerField;

}

}

Inner i = new Inner("I'm Inner Object");

i.showOuterObject();

}

public String toString() {

return outerField;

}

}

컴파일이 되지 않을 것이다. 컴파일타임 오류가 나는 줄을 주석화하고(아니면 빼버리고) 실행하면 결과는 다음과 같을 것이다.

I'm Inner Object

정적 메쏘드, 정적 초기화 블록 등에 선언된 내부 클래스를 static context에 선언된 내부 클래스라고 한다. 이들에게는 예외적으로 외부 객체가 연결되지 않는다. 그러므로 인스턴스 필드와 인스턴스 메쏘드에도 접근할 수 없다.

* + 1. 중첩타임의 상속

중첩 타입은 상속될 때 메쏘드보다는 필드와 비슷하다. 메쏘드는 재정의되지만 필드는 감추어진다. 중첩 타입도 필드와 마찬가지이다.

* + 1. 상속되진 않지만 접근할 수는 있는 경우

중접 타입에서는 상속되진 않지만 접근할 수는 있는 희한한 경우도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer();

Outer.SubClass s = o.getSubClassObject();

s.callMethodSub();

System.out.println();

s.accessField();

System.out.println();

s.callMethodSuper();

}

}

class Outer {

class SuperClass {

private int field = 111;

private void method() {

System.out.println("SuperClass.method()");

}

public void callMethodSuper() {

method();

}

}

class SubClass extends SuperClass {

private void method() {

System.out.println("SubClass.method()");

}

public void callMethodSub() {

super.method();

method();

}

public void accessField() {

System.out.println(super.field);

//System.out.println(field); // Error.

}

}

public SubClass getSubClassObject() {

return new SubClass();

}

}

결과는 다음과 같을 것이다.

SuperClass.method()

SubClass.method()

111

SuperClass.method()

SubClass에서 SuperClass의 private 필드와 private 메쏘드를 super 키워드를 이용해서 접근할 수 있음을 알 수 있다. 이것은 SuperClass와 SubClass가 모두 Outer 클래스의 중첩 클래스이기 때문에 가능하다. 쉽게 말하면 최상위 클래스인 Outer 클래스 내에서는 Outer 클래스 안에 있는 어떤 멤버에도 접근이 가능하다.

SubClass에서

System.out.println(field);

처럼 super 키워드를 사용하지 않으면 컴파일타임 오류가 나는데 이것은 field가 상속되지 않음을 보여준다. 그리고 SuperClass.callMethodSuper() 메쏘드에서 method 메쏘드를 불렀을 때 재정의되지 않기 때문에 SuperClass.method() 메쏘드가 불렸다.

다음 예제를 보자.

class Outer {

static class SuperClass {

private static int field = 111;

}

static class SubClass extends SuperClass {

int a = SuperClass.field; // OK.

}

}

컴파일이 잘 될 것이다. SubClass에서 SuperClass의 private 멤버에 접근했다. 여기까지는 위에서 알아본대로이다. 그렇다면 다음 예제는 어떨까?

class Outer {

static class SuperClass {

private static int field = 111;

}

static class SubClass extends SuperClass {

int a = SuperClass.field; // OK.

}

}

class GrandChild extends Outer.SubClass {

int b = Outer.SuperClass.field; // Error.

}

SubClass의 서브클래스인 GrandChild 클래스에서는 SuperClass의 private 필드에 접근할 수 없다. 다음과 같이 Outer 클래스를 상속받은 경우라도 마찬가지이다.

class Outer {

static class SuperClass {

private static int field = 111;

}

static class SubClass extends SuperClass {

int a = SuperClass.field; // OK.

}

}

class OuterSub extends Outer {

int b = Outer.SuperClass.field; // Error.

}

어떤 클래스를 상속받는 private에 접근할 수 있는 특권은 사라진다. 이것은 GrandChild 클래스와 OuterSub 클래스는 Outer 클래스를 만든 사람과 관계 없는 사람이 만들었을 가능성이 있다는 사실과 관계가 있다. 즉 믿을 수 없는 것이다.

* + 1. 자신의 외부 클래스를 상속받는 중첩 클래스

다음 예제는 놀랍게도 컴파일이 되었다.

public class Example {

public static void main(String[] args) {

Outer o = new Outer("outer outer");

Outer.Inner i = o.new Inner("inner", "super outer");

i.method();

}

}

class Outer {

String outerField;

Outer(String str) {

outerField = str;

}

class Inner extends Outer {

String innerField;

Inner(String a, String b) {

super(b);

innerField = a;

}

public void method() {

System.out.println(this.outerField);

System.out.println(super.outerField);

System.out.println(outerField);

System.out.println(innerField);

System.out.println(Outer.this.outerField);

}

}

}

결과는 다음과 같을 것이다.

super outer

super outer

super outer

inner

outer outer

위의 outerField가 두 개가 있다는 것에 주의하라 하나는 외부 객체의 outerField(outer outer)이고 하나는 내부 객체의 수퍼 클래스의 outerField(super outer)이다.

반대로 하면 안된다. 다음 예제는 컴파일타임 오류가 날 것이다.

class Outer extends Inner { // Error.

class Inner {

}

}

* + 1. 중첩 클래스와 신뢰 관계

위의 예제(상속되진 않지만 접근할 수 있는 경우)에서 보듯이 private 접근은 최상위 클래스 단위로 적용된다. 다시 말하면 최상위 클래스에 있는 어떤 코드도 그 클래스 안에 있는 어떤 코드에도 접근할 수 있다. 그것이 중첩 타입에 있든 아니든. 그러므로 내부 클래스가 외부 클래스의 private 멤버에 접근할 수 있을 뿐 아니라. 외부 클래스에서 내부 클래스의 private 멤버에 접근할 수 있다. 내부 클래스에서 다른 내부 클래스의 private 멤버에 접근하는 것은 위의 예제에서 살펴 보았다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer o = new Outer("Programmer Guffaws\n");

Outer.Inner i = o.new Inner("None, that's a hardware problem!");

i.innerMethod();

Outer another = new Outer("How many programmers does it take to screw in a lightbulb?");

another.outerMethod(i);

}

}

class Outer {

private String outerPrivate;

Outer(String str) {

outerPrivate = str;

}

class Inner {

private String innerPrivate;

Inner(String str) {

innerPrivate = str;

}

public void innerMethod() {

System.out.println(outerPrivate); // 외부 객체의 private 멤버에 접근

}

}

public void outerMethod(Inner i) {

System.out.println(outerPrivate);

System.out.println(i.innerPrivate); // 내부 객체의 private 멤버에 접근

}

}

결과는 다음과 같을 것이다.

Programmer Guffaws

How many programmers does it take to screw in a lightbulb?

None, that's a hardware problem!

정적 중첩 클래스일 때도 private 멤버를 상호 접근할 수 있는 것은 마찬가지이다. 단 인스턴스 필드나 인스턴스 메쏘드에는 접근할 수 없는 경우가 있을 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Outer.outerMethod();

Outer.callNestedMethod();

Outer.Nested.nestedMethod();

Outer.Nested.callOuterMethod();

}

}

class Outer {

static private int outerField = 108;

static void outerMethod() {

System.out.println(Nested.nestedField);

}

static void callNestedMethod() {

Nested.nestedMethod();

}

static class Nested {

static private int nestedField = 666;

static void nestedMethod() {

System.out.println(outerField);

}

static void callOuterMethod() {

outerMethod();

}

}

}

결과는 다음과 같을 것이다.

666

108

108

666

정적 중첩 클래스에서 외부 클래스의 인스턴스 필드에는 접근하지 못할 수도 있다. 다음 예제는 컴파일이 되지 않을 것이다.

class Outer {

private int outerField = 108;

static class Nested {

static void nestedMethod() {

System.out.println(outerField); // Error.

}

}

}

그러나 다음과 같이 사용할 수는 있다.

public class Example {

public static void main(String[] args) {

Outer.Nested.nestedMethod(new Outer());

}

}

class Outer {

private int outerField = 108;

static class Nested {

static void nestedMethod(Outer o) {

System.out.println(o.outerField); // OK.

}

}

}

결과는 다음과 같을 것이다.

108

* + 1. 내부 클래스와 상속

내부 클래스와 상속의 관계를 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubOuter so = new SubOuter();

SubOuter.SubInner si = so.getSubInnerObject();

si = so.new SubInner();

}

}

class SuperOuter {

class SuperInner {

}

}

class SubOuter extends SuperOuter {

class SubInner extends SuperInner {

}

public SubInner getSubInnerObject() {

return new SubInner();

}

}

이 예제를 컴파일한 후 디컴파일(decompile)하면 즉

C:\Decopile\jad>jad -noinner Example.class

C:\Decopile\jad>jad -noinner SuperOuter.class

C:\Decopile\jad>jad -noinner SuperOuter$SuperInner.class

C:\Decopile\jad>jad -noinner SubOuter.class

C:\Decopile\jad>jad -noinner SubOuter$SubInner.class

와 비슷하게 디컴파일하면 다음과 비슷하게 나올 것이다.

// Decompiled by Jad v1.5.7f. Copyright 2000 Pavel Kouznetsov.

public class Example

{

public Example()

{

}

public static void main(String args[])

{

SubOuter so = new SubOuter();

SubOuter$SubInner si = so.getSubInnerObject();

si = new SubOuter$SubInner(so);

}

}

class SuperOuter

{

SuperOuter()

{

}

}

class SuperOuter$SuperInner

{

SuperOuter$SuperInner(SuperOuter this$0)

{

}

}

class SubOuter extends SuperOuter

{

SubOuter()

{

}

public SubOuter$SubInner getSubInnerObject()

{

return new SubOuter$SubInner(this);

}

}

class SubOuter$SubInner extends SuperOuter$SuperInner

{

SubOuter$SubInner(SubOuter this$0)

{

super(this$0); // 여기에 주목하라.

}

}

SubInner 객체가 외부 객체와 연결할 필요가 있듯이 그 수퍼 클래스인 SuperInner도 외부 객체와 연결되어야 한다. 여기서는 SubOuter가 SuperOuter의 서브 클래스이므로 SubOuter의 객체가 SuperInner의 외부 객체 역할을 할 수 있었다.

public class Example {

public static void main(String[] args) {

SubOuter so = new SubOuter();

SubOuter.SubInner si = so.getSubInnerObject();

si.superInnerMethod();

si.superOuterMethod();

}

}

class SuperOuter {

class SuperInner {

public void superInnerMethod() {

System.out.println(this);

System.out.println(SuperOuter.this);

}

}

}

class SubOuter extends SuperOuter {

class SubInner extends SuperInner {

public void superOuterMethod() {

System.out.println(this);

System.out.println(SubOuter.this);

}

}

public SubInner getSubInnerObject() {

return new SubInner();

}

}

결과는 다음과 비슷할 것이다.

SubOuter$SubInner@7077e

SubOuter@7ced01

SubOuter$SubInner@7077e

SubOuter@7ced01

하나의 객체(7ced01)가 SuperInner와 SubInner 모두의 외부 객체로 역할함을 알 수 있다. 반드시 하나의 객체가 그럴 필요는 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubOuter subO = new SubOuter();

SuperOuter superO = new SuperOuter();

SubOuter.SubInner si = subO.getSubInnerObject(superO);

si.superInnerMethod();

si.superOuterMethod();

}

}

class SuperOuter {

class SuperInner {

public void superInnerMethod() {

System.out.println(this);

System.out.println(SuperOuter.this);

}

}

}

class SubOuter extends SuperOuter {

class SubInner extends SuperInner {

public SubInner(SuperOuter so) {

so.super(); // 여기에 주목하라.

}

public void superOuterMethod() {

System.out.println(this);

System.out.println(SubOuter.this);

}

}

public SubInner getSubInnerObject(SuperOuter so) {

return new SubInner(so);

}

}

결과는 다음과 비슷할 것이다.

SubOuter$SubInner@7077e

SuperOuter@7ced01

SubOuter$SubInner@7077e

SubOuter@2c04e8

이번에는 SuperInner의 외부 객체는 SuperOuter 클래스의 객체(7ced01)이고 SubInner의 외부 객체는 SubOuter 클래스의 객체(2c04e8)이다. 내부 클래스의 수퍼 클래스의 생성자를 부르는 것이 그냥 super()가 아니라

so.super();

라는 것에 주의하라. 왜냐하면 외부 객체까지 제공해야 하기 때문이다. 그리고 다음 예제와 같은 경우에는 외부 객체를 제공하는 것이 반드시 필요하다.

public class Example {

public static void main(String[] args) {

SuperOuter so = new SuperOuter();

SuperInnerExtended sie = new SuperInnerExtended(so);

sie.superInnerMethod();

sie.superInnerExtendedMethod();

}

}

class SuperOuter {

class SuperInner {

public void superInnerMethod() {

System.out.println(this);

System.out.println(SuperOuter.this);

}

}

}

class SuperInnerExtended extends SuperOuter.SuperInner {

public SuperInnerExtended(SuperOuter so) {

so.super(); // 반드시 필요하다.

}

public void superInnerExtendedMethod() {

System.out.println(this);

}

}

결과는 다음과 비슷할 것이다.

bababook.bavalang.SuperInnerExtended@1a698a

bababook.bavalang.SuperOuter@7077e

bababook.bavalang.SuperInnerExtended@1a698a

또는 다음과 같은 경우도 있을 수 있다.

public class Example {

public static void main(String[] args) {

SuperOuter superO = new SuperOuter();

SubOuter subO = new SubOuter();

SubOuter.SubInner si = subO.getSubInnerObject(superO);

si.superInnerMethod();

si.subInnerMethod();

}

}

class SuperOuter {

class SuperInner {

public void superInnerMethod() {

System.out.println(this);

System.out.println(SuperOuter.this);

}

}

}

class SubOuter { // SuperOuter를 상속하지 않는다.

class SubInner extends SuperOuter.SuperInner {

public SubInner(SuperOuter so) {

so.super(); // 반드시 필요하다.

}

public void subInnerMethod() {

System.out.println(this);

System.out.println(SubOuter.this);

}

}

public SubInner getSubInnerObject(SuperOuter so) {

return new SubInner(so);

}

}

결과는 다음과 비슷할 것이다.

bababook.bavalang.SubOuter$SubInner@7077e

bababook.bavalang.SuperOuter@7ced01

bababook.bavalang.SubOuter$SubInner@7077e

bababook.bavalang.SubOuter@2c04e8

* + 1. 중첩 타입의 중첩

중첩 타입 안에 또 다른 중첩 타입을 선언 할 수 있다. 정적 중첩 타입의 경우에는 좀더 덜 복잡하다. 내부 클래스 안에 내부 클래스를 선언할 때는 정말 복잡하다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

OuterMost om = new OuterMost();

OuterMost.Middle m = om.getMiddleObject();

OuterMost.Middle.InnerMost im = m.getInnerMostObject();

im.method();

}

}

class OuterMost {

String outerMostField = "Barnyard Poem\n\n";

String field = "I trembled and shook\nAnd felt her heart\nSlowly she spread\n";

public Middle getMiddleObject() {

return new Middle();

}

class Middle {

String middleField = "The sky was dark\nThe moon was high\nWe were alone\n";

String field = "I did my best\nAnd placed my hand\nUpon her breast\n";

public InnerMost getInnerMostObject() {

return new InnerMost();

}

class InnerMost {

String innerMostField = "Just she and I\nHer hair was brown\nHer eyes were too\n";

String field = "I knew just what\nShe wanted to do\nSo with my courage\n";

String anotherField = "Her legs apart\nI knew she was ready\nBut I didn't know how\n"

+ "It was my first try\nAt milking a cow";

public void method() {

System.out.print(outerMostField);

System.out.print(middleField);

System.out.print(innerMostField);

System.out.print(this.field);

System.out.print(Middle.this.field);

System.out.print(OuterMost.this.field);

System.out.println(anotherField);

}

}

}

}

결과는 다음과 같을 것이다.

Barnyard Poem

The sky was dark

The moon was high

We were alone

Just she and I

Her hair was brown

Her eyes were too

I knew just what

She wanted to do

So with my courage

I did my best

And placed my hand

Upon her breast

I trembled and shook

And felt her heart

Slowly she spread

Her legs apart

I knew she was ready

But I didn't know how

It was my first try

At milking a cow

이런식으로 중첩에 중첩을 거듭하는 설계는 되도록 피해야 한다. 코드를 이해하기가 너무 어려워진다. 다른 식의 해결책을 먼저 찾아보고 정말 안되면 그 때 사용하라.

1. 변수
   1. 변수(variable)의 종류
      1. 필드(field)

클래스에 정의된 필드도 변수다. 필드에는 정적 필드(static field, class variable)와 인스턴스 필드(instance field, non static field, instance variable)가 있다. 필드에 대한 자세한 내용을 클래스와 인터페이스를 다루는 장들을 참조하라.

정적 필드는 그 필드가 선언된 클래스나 인터페이스가 준비될(prepared) 때 만들어진다. 그리고 클래스나 인터페이스가 unload될 때 없어진다.

인스턴스 필드는 그 필드가 선언된 클래스나 그 클래스의 서브 클래스의 객체가 생성될 때 만들어진다. 그리고 그 객체에 대한 참조가 없어서 그 객체에 대한 finalization을 마쳤을 때 인스턴스 필드는 없어진다.

* + 1. 배열 요소 (array component)

배열 요소도 변수의 일종이다. 배열 요소에 대해서는 배열을 다룬 부분을 참조하라.

배열 요소는 배열 객체가 만들어질 때 만들어진다. 그리고 그 배열에 대한 참조가 하나도 없을 때 없어진다.

* + 1. 매개변수 (parameter)

매개변수도 변수다. 매개변수에는 메쏘드 매개변수, 생성자 매개변수, 예외상황 처리기(exception-handler) 매개변수가 있다. 메쏘드 매개변수와 생성자 매개변수에 대해서는 클래스를 다룬 부분을 참조하라. 예외 상황 처리 매개변수에 대해서는 예외 상황을 다루는 부분을 참조하라.

메쏘드 매개변수는 그 메쏘드가 불려질 때 만들어진다. 그리고 인자(argument)로 초기화된다. 그리고 그 메쏘드의 실행이 종료되면 매개변수도 없어진다.

생성자 매개변수도 메쏘드 매개변수와 비슷하다.

예외 상황 처리기 매개변수는 예외상황(exception)이 catch 절에 의해 잡힐(caught) 때 만들어진다. 그리고 발생한 예외 상황으로 초기화된다. 그리고 그 catch 절의 블록의 실행이 종료되면 없어진다.

* + 1. 지역변수

지역 변수도 변수다.

* 1. 변수의 선언과 초기화
     1. 변수의 선언

변수는 다음과 같이 선언한다.

public class Example {

public static void main(String[] args) {

int intValue;

String str;

int[] intArray;

long longArrary[]; // 이 방법은 권할 만한 방법이 아니다.

}

}

한 줄에 같은 타입의 변수를 여러 개 동시에 선언할 수도 있다. 그리고 배열 같은 경우에는 다른 타입의 변수도 여러 개 동시에 선언할 수 있다. 그러나 별로 권할 만한 것은 아니다. 한 줄에 하나씩만 선언하면 그 각각에 깔끔하게 주석을 붙일 수 있기 때문이다. 그리고 여러 타입의 배열을 동시에 선언하면 아무래도 읽는 사람을 헷갈리게 한다. 다음을 보자.

public class Example {

public static void main(String[] args) {

int x, y, z;

byte[] oneD, twoD[], threeD[];

}

}

위의 예제는 다음과 같이 한 줄에 하나씩 선언하는 것이 좋다.

public class Example {

public static void main(String[] args) {

int x; // x 에 대한 주석

int y; // y 에 대한 주석

int z; // z 에 대한 주석

byte[] oneD; // oneD 에 대한 주석

byte[][] twoD; // twoD 에 대한 주석

byte[][][] threeD[]; // threeD 에 대한 주석

}

}

참조형에서 변수를 선언했다고 객체가 생기는 것은 아니다. 객체는 따로 생성해 주어야 한다. 배열도 참조형이므로 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int[] intArray;

intArray[0] = 123; // Error.

String str;

str.length(); // Error.

}

}

다음과 같이 객체를 생성해서 그 변수에 대입해야 한다.

public class Example {

public static void main(String[] args) {

int[] intArray;

intArray = new int[10];

intArray[0] = 123; // OK.

String str;

str = "123";

str.length(); // OK.

}

}

* + 1. 변수의 초기화

정적 필드, 인스턴스 필드, 배열 요소는 변수가 선언되면서 특별히 초기화하지 않으면 다음과 같이 자동으로 초기화된다.

byte형 변수는 (byte)0 으로

short형 변수는 (short)0 으로

int형 변수는 0으로

long형 변수는 0L으로

char형 변수는 ‘\u0000’으로

float형 변수는 +0.0f으로

double형 변수는 +0.0d으로

boolean형 변수는 false로

참조형 변수는 null로

매개변수는 각각 지정된 값으로 초기화된다.

지역 변수는 자동으로 초기화되지 않는다. 값이 정해지지 않은 상태에서 지역변수를 사용하면 컴파일타임 오류가 난다.

변수를 선언하면서 초기화할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int intVariable = 132;

int[] intArray = new int[10];

String str = new String();

Object[] objArray = new Object[10];

}

}

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 4; i++) {

int value = method();

}

}

static int method() {

System.out.println("Example.method()");

return 0;

}

}

결과는 다음과 같을 것이다.

Example.method()

Example.method()

Example.method()

Example.method()

for 문 내에서 반복될때마다 초기화가 다시 된다.

* + 1. final

변수를 final로 선언하면 그 변수의 값은 변할 수 없다. 기본형에서는 그 변수의 값이 불변이다. 그러나 참조형에서는 문제가 좀 복잡해진다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

final StringBuffer sb = new StringBuffer("Terran");

sb = new StringBuffer("Zerg"); // Error.

}

}

sb를 다른 객체를 가리키게 하려하자 컴파일타임 오류가 난다. 즉 바꿀 수 없다. sb는 “Terran”을 저장하는 객체를 가리키고 있어야만 하는 것이다. 그러나 다음과 같은 사태를 막을 수는 없다.

public class Example {

public static void main(String[] args) {

final StringBuffer sb = new StringBuffer("Terran");

sb.append(" 중심의 Random");

System.out.println(sb);

}

}

결과는 다음과 같을 것이다.

Terran 중심의 Random

* + 1. blank final

final 변수를 선언하면서 그 줄에서 초기화하지 않을 수도 있다. 이런 final 변수를 blank final 변수라고 한다.

* + 1. definite assignment

필드나 배열 요소는 디폴트로 초기값이 정해진다. 그러므로 따로 초기화하지 않아도 사용할 수 있다. 그리고 매개변수도 넘겨주는 값으로 초기화되기 때문에 바로 사용할 수 있다. 그러나 지역 변수는 변수를 선언할 때 초기화 해 주거나 선언한 다음에 어떤 값을 대입해 주어야 그 변수를 사용할 수 있다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int intVariable;

System.out.println(intVariable); // Error.

}

}

컴파일타임 오류가 난다. 위 예는 너무나 명백하다. 그러나 그리 명백하지 않은 경우도 있다. 다음 예를 보자.

public class Example {

public void method(boolean b) {

int intVariable;

if (b) {

intVariable = 123;

}

System.out.println(intVariable); // Error.

}

}

b가 false인 경우도 있으므로 컴파일타임 오류가 난다. 다음 예는 어떨까?

public class Example {

public void method() {

int intVariable;

if (true) {

intVariable = 123;

}

System.out.println(intVariable); // OK.

}

}

이번에는 if의 조건문이 컴파일 타임에 알 수 있는 값인 true이기 때문에 intVariable이 초기화되었음은 컴파일러가 알 수 있다.

* + 1. definite unassignment

blank final은 단 한번만 값을 설정할 수 있다. 그것을 보장하기 위한 것이 definite unassignment다. 그것은 그 때까지 blank final변수가 절대로 값이 설정되지 않았으므로 값을 설정할 수 있다는 말이다. 자세한 것은 <The Java Language Specification>을 참조하라.

* 1. 타입(type)
     1. 타입

자바 언어는 타입이 엄격한(strongly typed) 언어다. 그래서 컴파일 타임에 어떠 식이나 변수의 타입을 알 수 있다. 이렇게 엄격하게 타입을 지키면 컴파일 타임에 많은 오류를 잡을 수 있다. 이런 면에서 자바는 타입이 엄격하지 않은 자바 스크립트와 극단적으로 다르다. 자바 스크립트는 편리한 점이 있지만 실수를 할 수 있는 곳이 무궁무진하다.

타입에는 기본형과 참조형이 있다.

* + 1. 기본형(primitive type)

기본형 변수는 반드시 그 타입의 값을 가진다. 예를 들어 boolean 변수는 true, false중 하나를 가진다. boolean 변수에 숫자가 들어갈 순 없다. 마찬가지로 int형 변수가 float형 값을 가질 수는 없다. 마찬가지로 float형 변수가 int형 값을 가질 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = 12345;

float f = 12345;

System.out.println("i = " + toFullBinaryString(i) + "(비트 패턴)");

System.out.println("f = " + toFullBinaryString(Float.floatToIntBits(f)) + "(비트 패턴)");

}

public static String toFullBinaryString(int number) {

String str = Integer.toBinaryString(number);

int length = str.length();

if (length < 32) {

for (int i = 0; i < 32 - length; i++)

str = "0" + str;

}

return str;

}

}

결과는 다음과 같을 것이다.

i = 00000000000000000011000000111001(비트 패턴)

f = 01000110010000001110010000000000(비트 패턴)

둘은 12345를 나타내지만 비트 패턴이 다른 엄격히 구분되는 값이다. 정수형끼리는 비트 수가 다르다. 예외적으로 short와 char의 일부는 비트수와 비트 패턴이 같을 수도 있다.

기본형 변수는 다른 변수와 상태를 공유하지 않는다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

int i = 12;

int j = i;

i = 23;

System.out.println(j);

}

}

결과는 다음과 같을 것이다.

12

i와 j가 전혀 다른 메모리 공간을 차지하기 때문에 i를 23으로 바꾸어 주었다고 j가 변하지는 않는다.

기본형에 대해 자세한 것은 기본형을 다루는 부분을 참조하라.

* + 1. 참조형(reference type)

참조형에는 클래스형, 인터페이스형, 배열형이 있다. 엄밀히 따지면 null 타입도 있지만 null을 타입으로 보느냐 아니냐는 실제로 그렇게 중요하지는 않다.

참조형의 값은 객체가 아니라 객체에 대한 참조(reference)이다. 이것의 의미는 아래에 있는 =에 대한 설명을 보면 확실해질 것이다.

참조형 변수에는 자신과 같은 타입이 아닌 객체의 참조가 저장될 수도 있다. 클래스형 변수나 인터페이스형 변수에는 서브 타입(subtype)의 객체의 참조도 저장될 수 있다. 이것은 동적 다형성(dynamic polymorphism, run-time polymorphism)과 밀접한 관계가 있다. 배열 변수도 자신과 같은 타입이 아닌 객체의 참조가 저장될 수 있다. 자세한 것은 배열을 다루는 장을 참조하라. 그리고 null은 어떤 참조형 변수에도 대입될 수 있다. null은 그 변수가 어떤 객체도 가리키지 않는다는 의미로 받아들이면 된다.

* + 1. 컴파일 타임 타입(compile time type)과 런 타임 타입(run time type)

기본형에서는 그 타입에 해당하는 값만 저장될 수 있기 때문에 컴파일 타임 타입이 곧 런 타임 타입이다. 그러나 참조형에서는 그렇지가 않다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass s = new SuperClass(); // 1

s = new SubClass(); // 2

}

}

class SuperClass {

}

class SubClass extends SuperClass {

}

1과 2 모두에서 s의 컴파일 타임 타입은 SuperClass이다. 그러나 1에서의 s의 런타임 타입은 SuperClass이고 2에서의 s의 런타임 타입은 SubClass이다.

런타임 타입은 instanceof 연산자나 리플렉션 등을 이용해서 알 수 있다. 이것에 대해서는 리플렉션을 다루는 장을 참조하라.

런타임 타입은 메쏘드 재정의와 밀접한 관계가 있다. 이것에 대해서는 클래스의 상속을 다룬 장을 참조하라.

* 1. 객체(object)와 참조(reference)
     1. 객체와 참조

기본형에서의 대입은 값을 대입하는데 비해 참조형에서는 객체 자체를 대입하는 것이 아니라 객체의 참조를 대입한다. 참조는 일종의 포인터라고 생각하면 된다. 즉 메모리 번지(memory address)라고 생각하면 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

String strOne = "One";

String strTwo = "Two";

str = strOne;

str = strTwo;

str = null;

}

}

먼저 세개의 변수 str, strOne, strTwo가 만들어지고 두 개의 객체가 만들어진다. 객체 “One”에 대한 참조가 strOne에 대입된다. 객체 “Two”에 대한 참조는 strTwo 변수게 대입된다. str 변수는 아직 초기화되지 않은 상태이다. 이 때의 그림을 보면 그림 변수 – 1과 같다.

-------------------------- 그림 변수 – 1 -----------------------------

이제

str = strOne;

이 실행된다. 이 때 이루어지는 것은 strOne이 가리키는 객체에 대한 참조가 str에 대입되는 것이다. 즉 str 변수도 strOne이 가리키는 변수를 가리키는 것이다. 이 때의 상태를 그림으로 나타내면 그림 변수 – 2와 같다.

-------------------------- 그림 변수 – 2 -----------------------------

이제

str = strTwo;

가 실행된다. 이것이 실행된 다음의 상태를 그림으로 나타내면 그림 변수 – 3과 같다.

-------------------------- 그림 변수 – 3 -----------------------------

이제

str = null;

가 실행된다. 이것이 실행된 다음에는 다시 그림 변수 – 1의 상태로 돌아간다.

str이 초기화되지 않은 상태와 null을 가리키는 상태 즉 아무것도 가리키지 않은 상태는 아무런 차이가 없어 보인다. 그러나 차이가 있다. 초기화되지 않은 상태에서는 그 변수가 (=의 왼쪽에 사용되는 방식을 제외하면)사용될 수 없다. 그래서 컴파일타임 오류가 난다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

str.toString(); // Error.

String another = str; // Error.

str = "One"; // OK. =의 왼쪽에 사용되었다.

}

}

이번에는 null로 설정된 예제이다.

public class Example {

public static void main(String[] args) {

String str = null;

String another = str; // OK.

str.toString(); // Compile OK. 그러나 실행하면 예외 상황이 발생한다.

}

}

결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

at Example.main(Example.java:5)

Exception in thread "main"

여기서

String another = str;

이 실행되면 another도 null을 가리키는 것으로 초기화된다.

기본형과 달리 하나의 객체를 여러 개의 변수가 가리키고 있을 수 있다. 즉 여러 변수가 상태를 공유하는 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

StringBuffer a = new StringBuffer("Zerg");

StringBuffer b = a;

a.append("-Hatchery");

System.out.println(b);

}

}

결과는 다음과 같을 것이다.

Zerg-Hatchery

* 1. 참조형의 형변환
     1. 넓히는 참조형 형변환(widening reference conversion)

넓히는 참조형 형변환은 안전한 형변환(safe cast)이다. 넓히는 형변환을 upcast라고도 한다. 보통 클래스 다이어그램을 그릴 때 수퍼 클래스를 위에 그리기 때문이다. 넓히는 참조형 형변환에는 다음과 같은 것들이 있다.

* 서브 클래스에서 수퍼 클래스로
* 클래스에서 수퍼 인터페이스로
* null에서 참조형(클래스형, 인터페이스형, 배열형)으로
* 서브인터페이스에서 수퍼 인터페이스로
* 인터페이스에서 Object형으로
* 배열형에서 Object형으로
* 배열형에서 Cloneable형으로
* 배열형에서 java.io.Serializable형으로
* A[]형에서 B[]형으로. 이 때 A,B는 참조형이며 A에서 B로의 형변환이 넓히는 참조형 형변환이어야 한다.

다음 예제는 위의 나열된 형변환에 대한 예를 보여준다.

public class Example {

public static void main(String[] args) {

SuperClass supC;

SubClass subC;

SuperInterface supI;

SubInterface subI;

Object o;

SuperClass[] supArray;

SubClass[] subArray;

Cloneable c;

java.io.Serializable s;

subC = new SubClass();

subArray = new SubClass[10];

supC = subC; // 서브 클래스에서 수퍼 클래스로

subI = subC; // 클래스에서 수퍼 인터페이스로

subC = null; // null 에서 클래스형으로

supI = subI; // 서브 인터페이스에서 수퍼 인터페이스로

o = supI; // 인터페이스에서 Object형으로

o = subArray; // 배열형에서 Object형으로

c = subArray; // 배열형에서 Cloneable형으로

s = subArray; // 배열형에서 Serializable형으로

supArray = subArray; // SubClass[]형에서 SuperClass[]형으로

}

}

interface SuperInterface {}

interface SubInterface extends SuperInterface {}

class SuperClass {}

class SubClass extends SuperClass implements SubInterface {}

컴파일이 잘 될 것이다. 결과로 나오는 건 없지만 오류 없이 실행도 잘 될 것이다.

* + 1. 좁히는 참조형 형변환(narrowing reference conversion)과 캐스팅(casting)

좁히는 참조형 형변환은 위험한 형변환(unsafe cast)이다. 넓히는 형변환을 downcast라고도 한다. 보통 클래스 다이어그램을 그릴 때 서브 클래스를 아래에 그리기 때문이다. 이 때는 명시적으로 캐스팅을 해주어야 한다. 참조형 캐스팅의 의미는 기본형 캐스팅의 의미와 비슷하다. 즉 “나도 위험한 걸 안다. 그러나 필요하니까 하는거다”라는 말을 컴파일러에게 해 주는 것이다. 기본형에서의 위험이 데이터를 잃어버릴 가능성이라면 참조형에서는 아예 예외상황이 발생한다. 예를 들어 있지도 않은 메쏘드나 필드를 사용하라고 하는 경우 등이다. 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

String str;

ArrayList al = new ArrayList();

al.add(new String("Show me the money"));

al.add(new String("Power overwhelming"));

al.add(new String("Operation CWAL"));

al.add(new String("Black Sheep Wall"));

for (Iterator i = al.iterator(); i.hasNext(); ) {

str = i.next(); // Error.

System.out.println(str);

}

}

}

컴파일타임 오류가 날 것이다. 왜냐하면 Iterator.next() 메쏘드가 Object형를 반환(return)하기 때문이다. 그러므로 String형 변수인 str에 대입하기 위해서는 명시적으로 캐스팅(casting)을 할 필요가 있다. 즉 다음과 같이 바꿔야 한다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

String str;

ArrayList al = new ArrayList();

al.add(new String("Show me the money"));

al.add(new String("Power overwhelming"));

al.add(new String("Operation CWAL"));

al.add(new String("Black Sheep Wall"));

for (Iterator i = al.iterator(); i.hasNext(); ) {

str = (String)i.next(); // OK.

System.out.println(str);

}

}

}

결과는 다음과 같을 것이다.

Show me the money

Power overwhelming

Operation CWAL

Black Sheep Wall

참조형 형변환이 위험하다는 것은 다음 예제를 보면 알 수 있을 것이다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

String str;

ArrayList al = new ArrayList();

al.add(new String("Show me the money"));

al.add(new String("Power overwhelming"));

al.add(new StringBuffer("Operation CWAL")); // 여기가 문제.

al.add(new String("Black Sheep Wall"));

for (Iterator i = al.iterator(); i.hasNext(); ) {

str = (String)i.next(); // Compile-time OK. Run-time Error

System.out.println(str);

}

}

}

결과는 다음과 비슷할 것이다.

Show me the money

Power overwhelming

java.lang.ClassCastException: java.lang.StringBuffer

at Example.main(Example.java:13)

Exception in thread "main"

ClassCastException이 난다.

좁히는 형변환에는 다음과 같은 것들이 있다.

* 수퍼 클래스에서 서브 클래스로
* 클래스형에서 인터페이스형으로. 이때 클래스는 인터페이스를 구현하지 않는다. 만약 인터페이스를 구현한다면 넓히는 형변환이다. 그리고 클래스는 final이면 안된다. final이면 금지된 형변환이다.
* Object형에서 인터페이스형으로. 이것은 클래스형에서 인터페이스형으로의 변환의 하나의 예이다. Object형은 어떤 인터페이스로 구현하지 않으면 final도 아니다.
* Object형에서 배열형으로
* 인터페이스에서 final이 아닌 클래스형으로
* 인터페이스에서 final인 클래스형으로. 이 때 클래스는 인터페이스를 구현해야 한다.
* 인터페이스에서 인터페이스로. 이때 두 인터페이스에는 서명(signature)는 같고 반환형(return type)은 다른 메쏘드가 있어서는 안된다.
* A[]형에서 B[]형으로. 이 때 A,B는 참조형이며 A에서 B로의 형변환이 좁히는 참조형 형변환이어야 한다.

좁히는 참조형 형변환이 되기 위해서는 두가지 조건이 있어야 한다. 하나는 형변환이 가능한 경우가 있어야 한다. 즉 런타임에 아무 문제가 없는 경우가 있어야 한다. 또 하나는 위험해야 한다. 즉 런타임에 문제가 발생할 수도 있어야 한다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SuperClass supC;

SubClass subC;

SuperInterface supI;

SubInterface subI;

AnInterface anI;

FinalClass finalC;

Object o;

SuperClass[] supArray;

SubClass[] subArray;

Cloneable c;

java.io.Serializable s;

supC = new SubClass();

o = supC;

subC = supC; // Error. 수퍼 클래스에서 서브 클래스로

supI = supC; // Error. 클래스에서 인터페이스로

supI = o; // Error. Object형에서 인터페이스로

o = new SubClass[12];

supArray = o; // Error. Object형에서 배열형으로

supC = supI; // Error. 인터페이스에서 final이 아닌 클래스로

supI = new FinalClass();

finalC = supI; // Error. 인터페이스에서 final 클래스로

anI = new SubClass();

supI = anI; // Error. 인터페이스에서 인터페이스로

subArray = supArray; // Error. SuperClass[]형에서 SubClass[]형으로

System.out.println("end of main()");

}

}

interface AnInterface {}

interface SuperInterface {}

interface SubInterface extends SuperInterface {}

class SuperClass {}

class SubClass extends SuperClass implements SubInterface, AnInterface {}

final class FinalClass implements SuperInterface {}

캐스팅을 안했기 때문에 컴파일타임 오류가 난다. 캐스팅을 해보자.

public class Example {

public static void main(String[] args) {

SuperClass supC;

SubClass subC;

SuperInterface supI;

SubInterface subI;

AnInterface anI;

FinalClass finalC;

Object o;

SuperClass[] supArray;

SubClass[] subArray;

Cloneable c;

java.io.Serializable s;

supC = new SubClass();

o = supC;

subC = (SubClass)supC; // 수퍼 클래스에서 서브 클래스로

supI = (SuperInterface)supC; // 클래스에서 인터페이스로

supI = (SuperInterface)o; // Object형에서 인터페이스로

o = new SubClass[12];

supArray = (SuperClass[])o; // Object형에서 배열형으로

supC = (SuperClass)supI; // 인터페이스에서 final이 아닌 클래스로

supI = new FinalClass();

finalC = (FinalClass)supI; // 인터페이스에서 final 클래스로

anI = new SubClass();

supI = (SuperInterface)anI; // 인터페이스에서 인터페이스로

subArray = (SubClass[])supArray; // SuperClass[]형에서 SubClass[]형으로

System.out.println("end of main()");

}

}

interface AnInterface {}

interface SuperInterface {}

interface SubInterface extends SuperInterface {}

class SuperClass {}

class SubClass extends SuperClass implements SubInterface, AnInterface {}

final class FinalClass implements SuperInterface {}

컴파일이 잘된다. 결과는 다음과 같을 것이다. 즉 무사히 실행된 것이다.

end of main()

다음 예제는 어떻게 위험할 수 있는지 보여주는 예제이다.

public class Example {

public static void main(String[] args) {

SuperClass supC;

SubClass subC;

SuperInterface supI;

SubInterface subI;

AnInterface anI;

FinalClass finalC;

Object o;

SuperClass[] supArray;

SubClass[] subArray;

Cloneable c;

java.io.Serializable s;

supC = new SuperClass();

try {

subC = (SubClass)supC; // 수퍼 클래스에서 서브 클래스로

} catch (Exception e) {

System.out.println("subC = (SubClass)supC; --> " + e);

}

try {

supI = (SuperInterface)supC; // 클래스에서 인터페이스로

} catch (Exception e) {

System.out.println("supI = (SuperInterface)supC; --> " + e);

}

o = new Object();

try {

supI = (SuperInterface)o; // Object형에서 인터페이스로

} catch (Exception e) {

System.out.println("supI = (SuperInterface)o; --> " + e);

}

try {

supArray = (SuperClass[])o; // Object형에서 배열형으로

} catch (Exception e) {

System.out.println("supArray = (SuperClass[])o; --> " + e);

}

supI = new FinalClass();

try {

supC = (SuperClass)supI; // 인터페이스에서 final이 아닌 클래스로

} catch (Exception e) {

System.out.println("supC = (SuperClass)supI; --> " + e);

}

supI = new SubClass();

try {

finalC = (FinalClass)supI; // 인터페이스에서 final 클래스로

} catch (Exception e) {

System.out.println("finalC = (FinalClass)supI; --> " + e);

}

supI = new FinalClass();

try {

anI = (AnInterface)supI; // 인터페이스에서 인터페이스로

} catch (Exception e) {

System.out.println("anI = (AnInterface)supI; --> " + e);

}

supArray = new SuperClass[312];

try {

subArray = (SubClass[])supArray; // SuperClass[]형에서 SubClass[]형으로

} catch (Exception e) {

System.out.println("subArray = (SubClass[])supArray; --> " + e);

}

System.out.println("end of main()");

}

}

interface AnInterface {}

interface SuperInterface {}

interface SubInterface extends SuperInterface {}

class SuperClass {}

class SubClass extends SuperClass implements SubInterface, AnInterface {}

final class FinalClass implements SuperInterface {}

결과는 다음과 비슷할 것이다.

subC = (SubClass)supC; --> java.lang.ClassCastException: bababook.bavalang.SuperClass

supI = (SuperInterface)supC; --> java.lang.ClassCastException: bababook.bavalang.SuperClass

supI = (SuperInterface)o; --> java.lang.ClassCastException: java.lang.Object

supArray = (SuperClass[])o; --> java.lang.ClassCastException: java.lang.Object

supC = (SuperClass)supI; --> java.lang.ClassCastException: bababook.bavalang.FinalClass

finalC = (FinalClass)supI; --> java.lang.ClassCastException: bababook.bavalang.SubClass

anI = (AnInterface)supI; --> java.lang.ClassCastException: bababook.bavalang.FinalClass

subArray = (SubClass[])supArray; --> java.lang.ClassCastException: [Lbababook.bavalang.SuperClass;

end of main()

* + 1. 넓히는 형변환과 캐스팅

넓히는 형변환은 굳이 캐스팅을 해주지 않아도 되고 사실 안하는 것이 관례다. 그러나 예외적으로 캐스팅을 하면 코드를 읽는데 도움이 되는 경우도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

SubClass sub = new SubClass();

System.out.println(sub.field);

SuperClass sup = (SuperClass)sub; // 여기에 주목하라.

System.out.println(sup.field);

}

}

class SuperClass {

int field = 1;

}

class SubClass extends SuperClass {

int field = 111;

}

결과는 다음과 같다.

111

1

여기서의 캐스팅은 감추어진(hidden) SuperClass의 필드를 사용할 것임을 강조하고 있다.

* + 1. 캐스팅에서 주의할 점

다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(new String("i'm string"));

String s = (String)al.get(0); // 여기에 주목하라.

System.out.println(s);

}

}

결과는 다음과 같을 것이다.

i'm string

위의 예제에서

(String)al.get(0);

과 같이 캐스팅하는 것을 흔히 볼 수 있을 것이다. 여기서 문제가 되는 것은 캐스팅의 () 연산자와 메쏘드나 필드를 사용하는데 쓰이는 .연산자 사이의 우선 순위이다. 여기서 .이 우선권을 갖는다. 그러므로 위의 식은 다음과 같다.

(String)(al.get(0));

다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object o = new ArrayList();

String s = new String("i'm string");

o.add(s); // Error

}

}

Object 클래스에는 add 메쏘드가 없으므로 ArrayList형으로 캐스팅 해주어야 한다. 다음과 같이 하면 어떨까?

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object o = new ArrayList();

String s = new String("i'm string");

(ArrayList)o.add(s); // Error

}

}

역시 컴파일타임 오류가 난다. 다음과 같이 해주어야 한다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

Object o = new ArrayList();

String s = new String("i'm string");

((ArrayList)o).add(s); // OK.

System.out.println(o);

}

}

컴파일이 잘 될 것이다. 결과는 다음과 비슷할 것이다.

[i'm string]

이것은 .이 아주 많이 연결되었을 때도 마찬가지이다. 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

String s = new String("i'm string");

ArrayList alA = new ArrayList();

alA.add(s);

ArrayList alB = new ArrayList();

alB.add(alA);

ArrayList alC = new ArrayList();

alC.add(alB);

s = (String)((ArrayList)((ArrayList)alC.get(0)).get(0)).get(0); // 여기를 잘 이해해 보자.

System.out.println(s);

}

}

결과는 다음과 같을 것이다.

i'm string

위의 예제의 괄호들을 잘 살펴보고 이해하는 건 좋은데 이런 식의 코딩을 따라하는 건 곤란하다.

이것은 필드에 접근하는 .도 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String s = (String)A.field;

System.out.println(s);

s = (String)A.a.a.a.a.a.a.a.a.a.a.a.a.a.a.a.a.a.a.field;

System.out.println(s);

}

}

class A {

static Object field = "String";

static A a;

}

많은 사람이 기묘하게 느끼겠지만 무사히 컴파일된다. 결과는 다음과 같다.

String

String

그냥 재미로 만들었다.

public class Example {

public static void main(String[] args) {

Object o = new Object().getClass().getClass().getClass().getClass().getClass().getClass().getClass().getClass();

System.out.println(o);

}

}

결과는 다음과 비슷할 것이다.

class java.lang.Class

오늘의 결론은 .과 ()의 사용을 자제하여 사람들을 헷갈림으로부터 구제하자는 것이다.

* + 1. 금지된 형변환

어떤 형변환은 절대로 불가능하다는 것이 컴파일타임에 알 수 있다. 이럴 때는 명시적으로 캐스팅을 하더라도 컴파일타임 오류가 난다.

금지된 형변환에는 다음과 같은 것들이 있다.

* 참조형에서 기본형으로
* 기본형에서 참조형으로. 이 때 String형은 예외다.
* null에서 기본형으로
* null형으로. 이건 실제적으로 의미는 없다. null형 변수가 없기 때문이다.
* boolean형으로
* boolean형에서 String형은 제외한 형으로
* 클래스에서 클래스로. 이 때 하나의 클래스가 다른 클래스의 서브 클래스가 아니어야 한다. 그리고 String형은 예외다.
* final 클래스에서 인터페이스로. 이때 클래스는 인터페이스를 구현하지 않아야 한다.
* Object형이 아닌 클래스형에서 인터페이스형으로.
* 인터페이스에서 final 클래스로. 이 때 클래스는 인터페이스를 구현하지 않아야 한다. String형은 예외다.
* 인터페이스에서 인터페이스로. 이 때 두 인터페이스에는 서명(signature)는 같고 반환형은 다른 메쏘드가 있다.
* 배열형에서 클래스형으로. 이때 Object, String 클래스는 예외다.
* 배열형에서 인터페이스형으로. 이때 Cloneable, Serializable은 예외다.
* 배열형 A[]에서 배열형 B[]로. 이때 A와 B사이에 string 변환을 제외한 변환이 존재하지 않는다.
  1. Assignment Compatibility
     1. 대입 호환성(assignment compatibility)이란

대입은 같은 타입끼리 일어날 수 있다. 즉 int형 값을 int형 변수에 대입할 수 있으며 String형 객체의 참조를 String형 변수에 대입할 수 있다. 형이 같지 않을 경우에도 대입을 할 수 있는 경우가 있는데 넓히는 형변환일 때이다. 이런식으로 대입이 가능한 것을 대입 호환성이라고 한다. 또는 대입될 수 있다(assignable)라고 말한다. 다음 예제는 호환되는 경우와 되지 않는 경우를 보여준다.

public class Example {

public static void main(String[] args) {

int i = 234; // OK.

long l = i; // OK.

Object o = new String(); // OK.

String s = new String(); // OK.

s = new Object(); // Error.

i = l; // Error.

Integer oi = i; // Error.

oi = new String(); // Error

}

}

호환되지 않는 경우는 명시적 형변환 즉 캐스팅을 해서 해결되는 경우가 있고 해결되지 않는 경우도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

long l = 2;

int i = (int)l; // OK.

String s = (String)new Object(); // Compile time OK. Runtime Exception.

}

}

결과는 다음과 비슷할 것이다.

java.lang.ClassCastException: java.lang.Object

at Example.main(Example.java:5)

Exception in thread "main"

컴파일이 잘되더라도 런타임에 문제가 생길 수가 있다. 그리고 아예 캐스팅이 안되는 경우도 있는데 금지된 형변환일 때가 그렇다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = 0;

Integer oi = (Integer)i; // Error.

oi = (Integer)new String(); // Error

}

}

아예 컴파일도 안된다.

* + 1. 대입 호환성이 적용되는 경우

대입 호환성은 이름으로 보듯이 주로 대입할 때 대입이 가능한지 여부를 가린다. 대입는 변수를 초기화하는데도 적용된다. = 연산자로 대입을 하지 않더라도 대입과 비슷한 일이 일어나는 곳이 있는데 그것은 매개변수에의 대입과 반환(return) 이다. 예외 상황이 던져질(throw) 때도 반환과 비슷하며 예외 상황이 잡힐(catch) 때는 catch 절의 매개변수에 대입된다. 이들 각각에 대해 알아보자.

매개 변수는 생성자, 메쏘드, catch 절 등에 쓰인다. 먼저 생성자와 메쏘드에 적용되는 경우에 대해 알아보자. 메쏘드와 생성자는 매개 변수와 대입 호환성의 문제에서는 차이가 없다. 여기에는 메쏘드에 대한 예제만 있다.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(400, 300);

}

public void paint(Graphics g) {

double x1 = 10.0;

double y1 = 10.0;

double x2 = 300.0;

double y2 = 200.0;

g.drawLine(x1, y1, x2, y2); // Error.

}

public static void main(String[] args) {

new Example().show();

}

}

컴파일이 안될 것이다.

왜냐하면 매개변수가 int형으로 선언되었기 때문이다. 그러므로 다음과 같이 고쳐 주어야 한다.

import java.awt.\*;

public class Example extends Frame {

public Example() {

setSize(400, 300);

}

public void paint(Graphics g) {

double x1 = 10.0;

double y1 = 10.0;

double x2 = 300.0;

double y2 = 200.0;

g.drawLine((int)x1, (int)y1, (int)x2, (int)y2); // OK.

}

public static void main(String[] args) {

new Example().show();

}

}

매개 변수의 경우에 예외가 있는데 그것에 대해서는 기본형을 다룬 장과 클래스와 인터페이스 – 못다한 이야기를 다룬 장에서 중복정의를 다룬 부분을 참조하라.

반환되는 값은 메쏘드의 반환형과 대입 호환성이 있어야 한다.

public class Example {

public String method() {

Object o = new String();

return o; // Error.

}

}

컴파일이 되지 않을 것이다. 실제로 리턴되는 객체가 String형임에도 o의 타입이 Object형이기 때문이다. 그러므로 다음과 같이 고쳐주어야 한다.

public class Example {

public String method() {

Object o = new String();

return (String)o; // OK.

}

}

다음과 같은 경우도 호환성이 성립하므로 잘 컴파일된다.

public class Example {

public Object method() {

return new String();

}

}

예외 상황을 던질 때는 throws 절에 규정된 예외와 대입 호환성이 있어야 한다.

public class Example {

public void method() throws java.io.IOException {

throw new Exception(); // Error.

}

}

Exception 클래스가 IOException 클래스의 서브 클래스가 아니기 때문에 컴파일이 되지 않는다. 다음과 같이 고쳐보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

method();

} catch (IOException e) {

System.out.println("I/O 에서 예외 발생");

}

}

public static void method() throws IOException {

throw (IOException)new Exception(); // Compile ok. but...

}

}

컴파일은 잘 되지만 다음과 같은 예외가 발생할 것이다.

java.lang.ClassCastException: java.lang.Exception

at Example.method(Example.java:13)

at Example.main(Example.java:6)

Exception in thread "main"

Exception 클래스인 예외 상황을 던지고 싶으면 다음과 같이 해야 할 것이다.

public class Example {

public static void main(String[] args) {

try {

method();

} catch (Exception e) {

System.out.println("예외 발생");

}

}

public static void method() throws Exception {

throw new Exception();

}

}

결과는 다음과 같을 것이다.

예외 발생

예외를 잡을 때는 catch 절들 중 매개변수와 대입 호환성이 있는 catch절에 잡힌다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

method();

} catch (Exception e) {

System.out.println("예외 발생");

}

}

public static void method() throws IOException {

throw new IOException();

}

}

결과는 다음과 같을 것이다.

예외 발생

IOException이 Exception에 대입될 수 있기 때문에 Exception이 매개변수로 선언된 catch에 잡혔다.

* 1. 영역(scope)
     1. 영역

지역 변수의 영역은 지역 변수가 그 변수가 선언된 다음부터 그 변수가 포함된 가장 안쪽 블록 끝까지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

a = 13; // Error.

int a;

a = 1; // OK.

}

}

컴파일이 되지 않을 것이다. 지역 변수 a가 선언된 블록은 main 메쏘드의 블록이다. 그러므로 main 메쏘드의 끝까지 쓸 수 있다. 그러나 a가 선언되는 줄(line) 위에서 쓸 수는 없다.

이번에는 a를 더 안쪽 블록에 선언해 보자.

public class Example {

public static void main(String[] args) {

a = 13; // Error.

{ // 블록 A

int a;

a = 321; // OK.

}

a = 1; // Error.

}

}

a = 1; 도 컴파일타임 오류가 난다. 왜냐하면 a가 선언된 가장 안쪽 블록 A 밖에 있기 때문이다.

여기서 for 문에 선언된 변수는 약간 다른 규칙을 따른다.

public class Example {

public static void main(String[] args) {

i = 0; // Error.

for (int i = 0; i < 10/\*OK\*/; i++/\*OK\*/) {

i = 12; // OK.

}

i = 321; // Error.

}

}

초기화하는 부분에서 선언된 변수는 증감부분, 조건 부분에서도 쓰일 수 있다. 그러나 for문을 벗어나면 쓸 수 없다.

* + 1. 금지된 이름 충돌

변수 이름을 같은 걸 쓰는 걸 금지하는 경우가 있다. 다음 예를 보자.

public class Example {

public void method(int abc) {

int abc; // Error.

}

}

컴파일되지 않을 것이다. 매개변수와 지역 변수가 충돌하기 때문이다.

다음 예도 역시 컴파일 되지 않는다. 지역 변수와 지역 변수가 충돌하기 때문이다.

public class Example {

public void method() {

int abc;

int abc; // Error.

}

}

다음 예는 어떨까?

public class Example {

public void method() {

{

int abc;

}

int abc; // OK.

}

}

컴파일이 잘된다. 두 변수의 영역이 겹치는 부분이 없기 때문이다. 그렇다고 이런식의 사용을 권장하는 것은 아니다. 그러나 다음과 같은 경우는 권장할 만하다.

public class Example {

public void method() {

for (int i = 0; i < 10; i++) {

//...

}

for (int i = 0; i < 5; i++) {

//...

}

for (int i = 0; i < 100; i++) {

//...

}

}

}

다음 예를 보자.

public class Example {

public void method() {

int abc;

{

int abc; // Error.

}

}

}

이번에는 컴파일이 안된다. 영역이 겹치기 때문이다.

* + 1. 가리기(shawdowing)

다음 예제를 실행해 보자.

public class Example {

private static int staticField = 1234;

private int instanceField = 4321;

public static void main(String[] args) {

new Example().method();

}

public void method() {

int staticField = 1111111;

int instanceField = 2222222;

System.out.println(staticField);

System.out.println(instanceField);

}

}

결과는 다음과 같을 것이다.

1111111

2222222

지역 변수가 정적 필드와 인스턴스 필드를 가렸다. 가려진 필드를 보기 위해서는 다음과 같이 해주면 된다.

public class Example {

private static int staticField = 1234;

private int instanceField = 4321;

public static void main(String[] args) {

new Example().method();

}

public void method() {

int staticField = 1111111;

int instanceField = 2222222;

System.out.println(Example.staticField);

System.out.println(this.instanceField);

}

}

결과는 다음과 같을 것이다.

1234

4321

* + 1. 지역 변수 선언에 대한 조언

가능한한 영역을 작게 잡아라. 다음의 예를 보면 무슨 뜻인지 알 수 있을 것이다.

public class Example {

public void method() {

int important;

int temp;

// 여기에 수백라인의 코드가 있다고 상상하라.

{ // 이 블록 안에서만 temp변수를 사용한다.

// ...

}

// 여기에 수백라인의 코드가 있다고 상상하라.

}

}

위와 같은 코드는 다음과 같이 바꿀 수 있다.

public class Example {

public void method() {

int important;

// 여기에 수백라인의 코드가 있다고 상상하라.

{ // 이 블록 안에서만 temp변수를 사용한다.

int temp;

// ...

}

// 여기에 수백라인의 코드가 있다고 상상하라.

}

}

변수 important는 메쏘드 전체에 계속 등장하는 중요한 변수이다. 주연이라고 할 수있다. temp는 안쪽의 블록에서만 보이는 엑스트라다. important는 당연히 메쏘드의 맨 위에 선언되어야 한다. 그렇다면 temp는 어디에 선언하는 것이 좋을까? important와 마찬가지로 메쏘드의 맨 위에 선언하면 코드를 읽는 사람은 메쏘드의 코드를 볼 때 계속 temp를 신경써야 한다. 만약 temp를 안쪽 블록에 선언했다면 temp에 대해서는 그 블록 안에서만 신경쓰고 나머지 코드를 볼 때는 실제로 중요한 important 변수에만 신경을 쓸 수 있을 것이다.

마찬가지의 원리가 for문을 사용할 때도 적용된다. 다음 코드를 보자.

public class Example {

public void method() {

int i;

// 여기에도 많은 코드가 있다.

for (i = 0; i < 33; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (i = 0; i < 49; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (i = 0; i < 108; i++) {

//...

}

// 여기에도 많은 코드가 있다.

}

}

이것은 다음과 같이 쓸 수도 있다.

public class Example {

public void method() {

// 여기에도 많은 코드가 있다.

for (int i = 0; i < 33; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (int i = 0; i < 49; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (int i = 0; i < 108; i++) {

//...

}

// 여기에도 많은 코드가 있다.

}

}

변수 i를 메쏘드의 맨 위에 선언하면 for문의 밖에서도 i의 존재에 대해 신경을 써야 한다. for문 안에 변수 i를 선언하면 for문 밖에는 i가 존재하지 않으므로 신경쓸 필요가 없다. 그리고 다음과 같은 경우를 생각해 보자.

public class Example {

public void method() {

int i;

// 여기에도 많은 코드가 있다.

for (i = 0; i < 33; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (i = 0; i < 49; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (; i < 108; i++) { // i = 0 을 빼먹었다. 그러나 컴파일은 잘된다. 버그가 된다.

//...

}

// 여기에도 많은 코드가 있다.

}

}

for문안에 변수를 선언했다면 즉 다음과 같이 했다면 이 버그를 컴파일러 잡아준다.

public class Example {

public void method() {

// 여기에도 많은 코드가 있다.

for (int i = 0; i < 33; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (int i = 0; i < 49; i++) {

//...

}

// 여기에도 많은 코드가 있다.

for (int i; i < 108; i++) { // 컴파일 에러.

//...

}

// 여기에도 많은 코드가 있다.

}

}

객체 지향 설계에서 중요한 것 중에 하나가 최대한 자신의 속성 또는 행동을 밖에 알리지 않는 것이다. 클래스 수준에서는 자신의 주로 자신의 필드를 private으로 감추는 방법을 쓴다. 즉 클래스를 선언하는 블록 안에 있는 것을 최대한 밖에 알리지 않는 것이다. 지역 변수도 마찬가지이다. 지역 변수가 쓰이는 블록 밖으로 최대한 알리지 않는 것이 좋다. 그러기 위해서는 최대한 안쪽 블록에 지역 변수를 선언해야 한다.

1. Object 클래스
   1. Object클래스에 대하여
      1. Object 클래스의 소스 코드

다음은 Object.java에 있는 (주석을 제외한) 전체 소스이다.

package java.lang;

public class Object {

private static native void registerNatives();

static {

registerNatives();

}

public final native Class getClass();

public native int hashCode();

public boolean equals(Object obj) {

return (this == obj);

}

protected native Object clone() throws CloneNotSupportedException;

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

public final native void notify();

public final native void notifyAll();

public final native void wait(long timeout) throws InterruptedException;

public final void wait(long timeout, int nanos) throws InterruptedException {

if (timeout < 0) {

throw new IllegalArgumentException("timeout value is negative");

}

if (nanos < 0 || nanos > 999999) {

throw new IllegalArgumentException(

"nanosecond timeout value out of range");

}

if (nanos >= 500000 || (nanos != 0 && timeout == 0)) {

timeout++;

}

wait(timeout);

}

public final void wait() throws InterruptedException {

wait(0);

}

protected void finalize() throws Throwable { }

}

소스의 양이 작은 이유는 Object클래스가 하는 일이 없어서가 아니라 많은 메쏘드가 native로 구현되었기 때문이다. native에 대해서는 native 메쏘드를 다루는 책을 참조하라.

* + 1. Object 클래스는 모든 클래스의 조상이다.

자바의 모든 객체는 Object클래스를 상속받는다. 여기에는 배열도 포함된다. 배열을 다룬 부분을 참조하라.

만약 어떤 클래스를 정의할 때 수퍼 클래스를 명시적으로 지정해 주지 않는다면 그것은 Object클래스를 수퍼 클래스로 지정하는 것과 같다. 다음 두 클래스 정의는 똑같다.

class A {

}

class A extends Object {

}

Object클래스가 모든 클래스의 조상이기 때문에 인터페이스를 이용해 다음과 같은 식의 코드가 가능하다. 그러나 결코 이런 식으로 코딩하는 것을 권하는 것은 아니다.

import java.io.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(new String("1234"));

al.add(new Integer(12));

Serializable s = al;

System.out.println("s.toString() = " + s.toString());

System.out.println("s.hashCode() = " + s.hashCode());

ArrayList alAnother = new ArrayList(al);

System.out.println("s.equals(alAnother) = " + s.equals(alAnother));

}

}

결과는 다음과 같다.

s.toString() = [1234, 12]

s.hashCode() = 46793675

s.equals(alAnother) = true

물론 Serialable 인터페이스에는 toString, hashCode, equals등의 메쏘드가 정의되지 않았다. 다음은 Serialable.java에 있는 (주석을 제외한) 소스코드 전체이다.

package java.io;

public interface Serializable {

}

인터페이스는 자동으로 Object클래스에 있는 public메쏘드를 상속(?) 받는다.

Object클래스와 같이 모든 클래스의 조상 클래스를 왜 만들었을까? 그 이유중의 하나는 거의 모든 클래스에서 쓰일 듯한 기능(메쏘드)들을 하나로 모으기 위해서이다. 이것이 Object 클래스의 메쏘드들을 구성한다. 또 하나의 이유는 다음 예제를 보면 명백해진다.

import java.awt.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(new Point(10, 20));

al.add(new String("string"));

al.add(new Integer(12));

al.add(new Boolean(true));

al.add(al.clone());

System.out.println(al);

}

}

결과는 다음과 같다.

[java.awt.Point[x=10,y=20], string, 12, true, [java.awt.Point[x=10,y=20], string, 12, true]]

ArrayList객체에 갖가지 클래스의 객체를 add할 수 있다(기본형은 wrapper 클래스를 이용해서). 이것은 add메쏘드가 매개 변수로 Object형을 받기 때문이다. 이것은 많은 컬렉션 클래스(collection classes)에서 쓰인다.

다른 예를 들어 보자. 여태까지 우리는 System.out.print()안에 어떤 객체든지 써 왔다. 실제로 이것이 어떻게 작동하는 지 알아보자. 위의 예제에서

System.out.println(al);

가 어떻게 작동하는지 알아보자. System.out은 PrintStream클래스의 객체이다. 그러므로 PrintStream.println(Object)를 부르게 된다. PrintStream.java의 일부를 인용해 보자.

public class PrintStream extends FilterOutputStream {

public void println(Object x) {

synchronized (this) {

print(x);

newLine();

}

}

public void print(Object obj) {

write(String.valueOf(obj));

}

...

}

print 메쏘드를 거쳐 String.valueOf(Object)를 부르는 것을 알 수 있다. String.java의 일부를 인용해 보자.

public final class String implements java.io.Serializable, Comparable, CharSequence {

public static String valueOf(Object obj) {

return (obj == null) ? "null" : obj.toString();

}

...

}

결국 Object.toString()을 부르는 것을 알 수 있다. 즉 이런 것이 Object 객체의 역할 중의 하나이다.

* 1. equals()와 ==의 차이
     1. 참조형에서 ==의 의미

일단 null은 null과 같다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String s = null;

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

System.out.println(s == null);

}

}

결과는 다음과 같다.

true

true

그러나 같은 null이라도 아무곳에서 == 연산자를 사용할 수 있는 것은 아니다. 하나가 null 리터럴이거나 아니면 두 변수가 같은 객체를 가리킬 가능성이 있어야 한다. 다음 예제를 보자.

import java.awt.event.\*;

public class Example {

public static void main(String[] args) {

Object o = null;

Integer i = null;

String s = null;

Comparable c = null;

ActionListener al = null;

System.out.println(o == s); // OK. String은 Object의 서브클래스.

System.out.println(c == s); // OK. String은 Comparable을 구현한다.

System.out.println(s == i); // Error.

System.out.println(s == al); // Error.

System.out.println(c == al); // OK. 두개의 변수가 같은 객체를 가리키고 있는 가능성이 있다.

}

}

(오류가 나는 곳을 빼버리고 실행하면) 결과는 다음과 같다.

true

true

true

Comparable과 ActionListener가 ==로 비교가능하다는 것에 의아해 하는 사람도 있을 것이다. 이 의문은 ‘두 변수가 같은 객체를 가리키고 있을 가능성’이 무엇을 뜻하는지 알면 풀릴 것이다. 다음 예제를 보면 의문이 풀릴 것이다.

import java.awt.event.\*;

public class Example {

public static void main(String[] args) {

A a = new A();

Comparable c = a;

ActionListener al = a;

System.out.println(c == al); // 두개의 변수가 같은 객체를 가리키고 있다.

}

}

class A implements ActionListener, Comparable {

public void actionPerformed(ActionEvent ae) {

}

public int compareTo(Object o) {

return 0; // 실제 프로그래밍에서는 이러면 안된다.

}

}

결과는 다음과 같다.

true

사실 두개의 참조형 변수가 ==로 비교될 수 있는가 여부는 두 변수가 같은 객체를 가리킬 가능성이 있는가로 결정된다. 위에서는 null과 연관지어 생각해 보았는데 컴파일러는 null인지 아닌지 모른다(아니면 알면서도 모른척 한다).

그렇다면 null이 아닐 때는 어떨까? 당연히 하나는 null이고 다른 하나는 null이 아니라면 둘은 다르다. 즉

public class Example {

public static void main(String[] args) {

System.out.println(null == new Object());

}

}

을 실행하면 결과는

false

이다. 여기까지는 너무나 당연하다.

만약 두개의 변수가 모두 null이 아니고 객체를 가리키고 있다면 어떨까? 이 때는 두 개의 변수가 같은 객체를 가리키고 있을 때 true가 된고 아닐 때는 false가 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String a = new String("one");

String b = a;

String c = new String("one");

System.out.println(a == b);

System.out.println(a == c);

}

}

결과는 다음과 같다.

true

false

위의 예제를 그림으로 본다면 그림 Object 클래스 – 1이 된다.

---------------------------------------- 그림 Object 클래스 – 1 -------------------------------------

이것을 동일성(identity) 체크라고 한다. 즉 객체의 상태를 비교하는 것이 아니라 정확하게 두 변수가 같은 메모리 공간을 가리키는가를 테스트하는 것이다.

* + 1. equals()와 동등성(equality)

equals메쏘드는 사용자가 재정의할 수 있다. 사용자는 어떻게든 재정의할 수 있다. 그리고 재정의하지 않는다면 수퍼 클래스에서 정의한 것을 사용한다. Object에 이르기까지 어떤 수퍼 클래스도 재정의하지 않았다면 Object클래스에 정의된 것이 쓰인다. Object클래스에는 어떻게 정의되어 있는지 소스 코드를 보자.

public boolean equals(Object obj) {

return (this == obj);

}

그냥 ==을 사용하고 있다. 그러나 이것은 equals메쏘드를 만들 때의 의도가 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

IntValue ivA = new IntValue(123);

IntValue ivB = new IntValue(123);

IntValue ivC = ivA;

System.out.println("ivA.equals(ivB) = " + ivA.equals(ivB));

System.out.println("ivA.equals(ivC) = " + ivA.equals(ivC));

}

}

class IntValue {

private int value;

public IntValue(int value) {

this.value = value;

}

}

결과는 다음과 같다.

ivA.equals(ivB) = false

ivA.equals(ivC) = true

그러므로 사용자는 대부분의 경우 equals메쏘드를 재정의해야 한다. 다음 예제는 equals 메쏘드를 재정의한다.

public class Example {

public static void main(String[] args) {

IntValue ivA = new IntValue(123);

IntValue ivB = new IntValue(123);

IntValue ivC = ivA;

IntValue ivD = new IntValue(-32);

System.out.println("ivA.equals(ivB) = " + ivA.equals(ivB));

System.out.println("ivA.equals(ivC) = " + ivA.equals(ivC));

System.out.println("ivA.equals(ivD)= " + ivA.equals(ivD));

}

}

class IntValue {

private int value;

public IntValue(int value) {

this.value = value;

}

public boolean equals(Object obj) {

if (obj instanceof IntValue) {

IntValue other = (IntValue)obj;

if (other.value == value)

return true;

else

return false;

} else {

return false;

}

}

}

결과는 다음과 같다.

ivA.equals(ivB) = true

ivA.equals(ivC) = true

ivA.equals(ivD)= false

equals메쏘드는 동등성(equality)를 체크해야 한다. 무엇이 같은가(equal) 하는 것은 사용자가 정의하기 나름이다. 예를 들어 다음과 같이 정의해도 문법적으로 문제가 되지 않는다.

public class Example {

public static void main(String[] args) {

IntValue a = new IntValue(0);

IntValue b = new IntValue(0);

IntValue c = new IntValue(1);

IntValue d = new IntValue(-1);

IntValue e = new IntValue(1);

IntValue f = new IntValue(12345);

IntValue g = new IntValue(-654451);

String str = new String("string");

System.out.println("a.equals(b) = " + a.equals(b));

System.out.println("c.equals(d) = " + c.equals(d));

System.out.println("c.equals(e) = " + c.equals(e));

System.out.println("f.equals(g) = " + f.equals(g));

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

}

}

class IntValue {

private int value;

public IntValue(int value) {

this.value = value;

}

// equals 메쏘드를 재정의하는 독특한(?) 방법.

public boolean equals(Object obj) {

if (obj instanceof IntValue) {

IntValue other = (IntValue)obj;

if (value > 0) {

if (other.value < 0)

return true;

else

return false;

} else if (value < 0) {

if (other.value > 0)

return true;

else

return false;

} else

return false;

} else {

return true;

}

}

}

결과는 다음과 같다.

a.equals(b) = false

c.equals(d) = true

c.equals(e) = false

f.equals(g) = true

c.equals(str) = true

그러나 이것은 ‘같다’의 의미와 맞지 않는다. API Specification(정식 명칭은 Java 2 Platform, Standard Edition, v 1.x.x API Specification)을 보면 equals 메쏘드가 만족해야 할 조건을 다음과 같이 정리했다.

reflexive : 자기 자신과는 같아야 한다. 즉 (a==b)가 true이면 a.equals(b)도 true여야 한다.

symmetric : a.equals(b)가 true이면 b.equals(a)도 true여야 한다.

transitive : a.equals(b)가 true이고 b.equals(c)가 true이면 a.equals(c)도 true여야 한다.

consistant : a와 b의 (equals 메쏘드와 관련된)상태가 변하지 않는한 a.equals(b)는 항상 같은 값을 반환해야 한다.

null과 non-null : a가 null이 아니면 a.equals(null)은 false를 반환해야 한다.

hashCode메쏘드와의 관계 : a.equals(b)가 true라면 a.hashCode()와 b.hashCode()의 값이 같아야 한다.

그리고 물론 같은 객체(equals메쏘드가 true를 반환하는)는 의미상 같아야 한다.

equals메쏘드는 반드시 하나의 클래스의 객체끼리만 비교해야만 하는 것은 아니다. 만약 의미상 같다는 것이 성립할 수 있다면 이것은 서로 다른 클래스의 객체에도 적용할 수 있다. 특히 비교 되는 객체의 클래스들이 클래스 계층도(class hierachy)에서 밀접한 관계가 있을 때 그러기가 쉽다. 즉 하나가 다른 하나의 수퍼 클래스이거나 둘이 같은 클래스를 수퍼클래스로 공유할 때를 말한다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

IntValue ivA = new IntValue(123);

LongValue lvA = new LongValue(123);

LongValue lvB = new LongValue(2);

System.out.println("ivA.equals(lvA) = " + ivA.equals(lvA));

System.out.println("ivA.equals(lvB) = " + ivA.equals(lvB));

System.out.println("lvA.equals(lvB) = " + lvA.equals(lvB));

}

}

class IntValue {

private int value;

public IntValue(int value) {

this.value = value;

}

public int getValue() {

return value;

}

public boolean equals(Object obj) {

if (obj instanceof IntValue) {

IntValue other = (IntValue)obj;

if (other.value == value)

return true;

else

return false;

} else if (obj instanceof LongValue) {

LongValue other = (LongValue)obj;

long otherValue = other.getValue();

if ( (otherValue <= (long)Integer.MAX\_VALUE) &&

(otherValue >= (long)Integer.MIN\_VALUE) ) {

if (value == (int)otherValue)

return true;

else

return false;

} else {

return false;

}

} else {

return false;

}

}

}

class LongValue {

private long value;

public LongValue(int value) {

this.value = value;

}

public long getValue() {

return value;

}

public boolean equals(Object obj) {

if (obj instanceof LongValue) {

LongValue other = (LongValue)obj;

if (other.value == value)

return true;

else

return false;

} else if (obj instanceof IntValue) {

IntValue other = (IntValue)obj;

int otherValue = other.getValue();

if ( (value <= (long)Integer.MAX\_VALUE) &&

(value >= (long)Integer.MIN\_VALUE) ) {

if ((int)value == otherValue)

return true;

else

return false;

} else {

return false;

}

} else {

return false;

}

}

}

결과는 다음과 같다.

ivA.equals(lvA) = true

ivA.equals(lvB) = false

lvA.equals(lvB) = false

* 1. hashCode()
     1. hashCode 메쏘드는 어디에 쓰이나

hashCode 메쏘드가 계산해 주는 해시 코드(hash code)는 주로 해시 테이블(hash table)에 쓰인다. 해시 코드와 해시 테이블에 대해서는 자료 구조론 책을 참조하기 바란다.

다음은 자바판 해시 테이블이라고 할 수 있는 HashMap.java에서 인용한 것이다.

public class HashMap extends AbstractMap implements Map, Cloneable, java.io.Serializable {

public Object put(Object key, Object value) {

Entry tab[] = table;

int hash = (key == null ? 0 : key.hashCode()); // 여기서 hashCode 메쏘드를 사용한다.

...

}

public Object get(Object key) {

Entry e = getEntry(key); // getEntry 메쏘드를 부른다.

return (e == null ? null : e.value);

}

Entry getEntry(Object key) {

Entry tab[] = table;

int hash = (key == null ? 0 : key.hashCode()); // 여기서 hashCode 메쏘드를 사용한다.

...

}

...

}

* + 1. hashCode와 equals의 관계

equals 메쏘드를 다룬 부분에서도 말했듯이 hashCode 메쏘드와 equals 메쏘드는 상호 일관성이 있어야 한다. 즉 (a.equals(b) == true)라면 (a.hashCode() == b.hashCode())여야 한다는 말이다.

서로 다른(unequal) 객체가 반드시 hashCode()의 반환값이 다를 필요는 없다. 그러나 서로 다른 객체가 같은 해시 코드의 값을 리턴하는 경향이 있다면 해시 테이블을 사용하는데 있어서 속도에 문제가 생길 것이다. 왜 그런지는 역시 자료구조론 책을 참조하기 바란다.

* + 1. hashCode() 재정의하기

Object클래스에 정의된 hashCode메쏘드는 객체의 identity가 다르면 다른 값을 리턴한다. 이것은 Object클래스에 정의된 equals 메쏘드가 ==을 사용하는 것과 상통한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

AnotherString a = new AnotherString("haha");

AnotherString b = new AnotherString("haha");

System.out.println("a.hashCode() = " + a.hashCode());

System.out.println("b.hashCode() = " + b.hashCode());

}

}

class AnotherString {

private String value;

public AnotherString(String str) {

value = str;

}

}

결과는 다음과 같다.

a.hashCode() = 6129586

b.hashCode() = 7852340

AnotherString에서 hashCode메쏘드를 재정의하지 않았기 때문에 Object에서 정의된 것을 사용한다. 두 객체의 값이 같음(equal)에도 불구하고 hashCode 메쏘드는 다른 값을 리턴했다. 이것은 위에서 살펴보았듯이 바람직한 결과가 아니다. hashCode를 재정의해보자.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

AnotherString a = new AnotherString("haha");

AnotherString b = new AnotherString("haha");

AnotherString c = new AnotherString("hahaha");

AnotherString d = new AnotherString("ha");

System.out.println("a.hashCode() = " + a.hashCode());

System.out.println("b.hashCode() = " + b.hashCode());

System.out.println("c.hashCode() = " + c.hashCode());

System.out.println("d.hashCode() = " + d.hashCode());

}

}

class AnotherString {

private String value;

public AnotherString(String str) {

value = str;

}

public int hashCode() {

int sum = 0;

for (int i = 0; i < value.length(); i++) {

char ch = value.charAt(i);

sum += ch \* (i + 1);

}

return sum;

}

}

결과는 다음과 같다.

a.hashCode() = 998

b.hashCode() = 998

c.hashCode() = 2100

d.hashCode() = 298

이번에는 표준 API의 String은 어떻게 했는지 살펴보자. 다음은 String.java에서 인용한 것이다.

public final class String implements java.io.Serializable, Comparable, CharSequence {

// offset은 0, count는 문자열의 길이라고 생각하자.

// 그게 불만이면 소스를 다 분석해 보든지...

public int hashCode() {

int h = hash;

if (h == 0) {

int off = offset;

char val[] = value;

int len = count;

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

h = 31\*h + val[off++];

}

hash = h;

}

return h;

}

private int offset;

private int count;

private char value[]; // 여기에 실제 문자열이 저장된다.

private int hash = 0; // 이것은 한번 계산하면 재활용하기 위해서이다.

...

}

대충 어떻게 계산하는지 알수 있을 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str = new String("12ab");

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

int hash = '1' \* 31 + '2';

hash = hash \* 31 + 'a';

hash = hash \* 31 + 'b';

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

}

}

결과는 다음과 같다.

str.hashCode() = 1510914

hash = 1510914

String.hashCode()을 보면 어떻게 해시 코드를 계산하면 좋은지 알 수 있을 것이다.

* 1. toString()
     1. toString 메쏘드

toString 메쏘드는 그 객체에 대한 정보를 보여주는 역할을 한다. 다음 예제를 보자.

import java.awt.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

String str = new String("u32ueo");

Integer i = new Integer(1234);

Point p = new Point(23, -5);

ArrayList al = new ArrayList();

Object o = new Object();

Runtime rt = Runtime.getRuntime();

al.add(str);

al.add(i);

al.add(p);

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

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

System.out.println("p.toString() = " + p.toString());

System.out.println("al.toString() = " + al.toString());

System.out.println("o.toString() = " + o.toString());

System.out.println("rt.toString() = " + rt.toString());

}

}

결과는 다음과 비슷할 것이다.

str.toString() = u32ueo

i.toString() = 1234

p.toString() = java.awt.Point[x=23,y=-5]

al.toString() = [u32ueo, 1234, java.awt.Point[x=23,y=-5]]

o.toString() = java.lang.Object@20c10f

rt.toString() = java.lang.Runtime@62eec8

toString 메쏘드는 자바 언어가 스트링을 다루는 방식과 밀접한 관계가 있다.

public class Example {

public static void main(String[] args) {

Object o = new Object();

String a = "" + o;

String b = "" + o.toString();

String c = "" + 123;

String d = "" + new Integer(123);

String e = "" + new Integer(123).toString();

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

System.out.println("o = " + o.toString());

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

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

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

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

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

}

}

결과는 다음과 비슷할 것이다.

o = java.lang.Object@310d42

o = java.lang.Object@310d42

a = java.lang.Object@310d42

b = java.lang.Object@310d42

c = 123

d = 123

e = 123

자바의 문자열 연결 연산자(string concatination operator)인 +가 객체를 만나면 그 객체의 toString 메쏘드를 부르는 것이다.

* + 1. Object.toString()

다음은 Object클래스의 toString 메쏘드의 전체 소스 코드이다.

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

클래스이름에서 @를 붙이고 해시코드 16진수로 보여준다. 다음 예제에서 클래스 A는 toString을 재정의하지 않았기 때문에 Object에 정의된 toString 메쏘드를 사용한다.

public class Example {

public static void main(String[] args) {

A a = new A();

System.out.println("a.toString() = " + a.toString());

}

}

class A {

public int hashCode() {

return 0xBaba;

}

}

결과는 다음과 같다.

a.toString() = A@baba

* + 1. toString 메쏘드 재정의하기

toString 메쏘드를 재정의하는 것은 정말 쉽다. 그냥 그 클래스의 상태를 묘사하는 String 객체를 리턴하면 된다. 다음은 Point.java에서 인용한 것이다.

public class Point extends Point2D implements java.io.Serializable {

public String toString() {

return getClass().getName() + "[x=" + x + ",y=" + y + "]";

}

...

}

* 1. clone()와 복제
     1. 복제(clone)이란?

프로그래밍하다 보면 객체를 복제하는 일이 많은 것이다. 복제된 객체란 identity로 따지면 다르지만 equality로 따지면 같은 객체를 말한다. 그리고 물론 같은 클래스의 객체여야 한다.

또 하나의 조건은 원본과 복제본은 독립적이어야 한다는 것이다(사실 이것은 identity 다르다는 것에 나온다). 즉 원본이 바뀌어도 복제본은 바뀌지 말아야 하고 복제본이 바뀌어도 원본에는 영향을 주지 말아야 한다. 이것에 대해서는 얕은 복제와 깊은 복제를 다루는 부분을 참조하라.

clone메쏘드를 정의하는데 위의 조건을 따르지 않아도 컴파일타임 오류가 나거나 하지는 않는다. 이것은 equals메쏘드와 비슷하다. 프로그래머가 원하면 제멋대로 정의할 수 있다. 그러나 그것은 아주 안좋은 생각이다.

배열의 복제에 대해서는 배열을 다루는 부분을 참조하라.

* + 1. 객체를 복제하는 여러가지 방법

가장 무식한 방법부터 시작해 보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.intValue = 10;

a.byteValue = 5;

a.stringValue = new String("str");

System.out.println("a.intValue = " + a.intValue);

System.out.println("a.byteValue = " + a.byteValue);

System.out.println("a.stringValue = " + a.stringValue);

System.out.println();

A aCloned = new A();

aCloned.intValue = a.intValue;

aCloned.byteValue = a.byteValue;

aCloned.stringValue = new String(a.stringValue);

System.out.println("aCloned.intValue = " + aCloned.intValue);

System.out.println("aCloned.byteValue = " + aCloned.byteValue);

System.out.println("aCloned.stringValue = " + aCloned.stringValue);

System.out.println();

System.out.println("(a==aCloned) = " + (a==aCloned));

System.out.println("(a.stringValue==aCloned.stringValue) = " + (a.stringValue==aCloned.stringValue));

}

}

class A {

public int intValue;

public byte byteValue;

public String stringValue;

}

결과는 다음과 같다.

a.intValue = 10

a.byteValue = 5

a.stringValue = str

aCloned.intValue = 10

aCloned.byteValue = 5

aCloned.stringValue = str

(a==aCloned) = false

(a.stringValue==aCloned.stringValue) = false

어쨌든 복제가 되긴 했지만 복제할 때마다 네 줄의 코드가 필요하다. 게다가 클래스 A에서는 데이터를 숨기지 않았다.

이것을 개선해 보자. 데이터를 숨기고(private으로) 복제하는 것을 하나의 메쏘드로 바꾸어 보자.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = a.복제하자();

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public A 복제하자() { // 자바에서는 한글도 된다. 그러나 별로 권할만하지는 않다.

return new A(intValue, byteValue, stringValue);

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

결과는 다음과 같다.

a = [intValue=10, byteValue=5, stringValue=str]

aCloned = [intValue=10, byteValue=5, stringValue=str]

(a==aCloned) = false

훨씬 좋아졌다. 그러나 이 방식보다는 Object.clone()을 이용하는 것이 더 좋다.

복제하는 또다른 방식은 복제 생성자(copy constructor)를 만드는 것이다. 이것은 몇몇 클래스에서 선호되는 방식이다. 예를 들어 String클래스가 이 방식을 사용한다. 다음은 String클래스의 복제생성자의 소스 코드이다.

public final class String implements java.io.Serializable, Comparable, CharSequence {

public String(String original) {

this.count = original.count;

if (original.value.length > this.count) {

this.value = new char[this.count];

System.arraycopy(original.value, original.offset, this.value, 0, this.count);

} else {

this.value = original.value;

}

}

...

}

String클래스는 Object클래스의 clone 메쏘드를 재정의하지 않고 복제 생성자를 사용한다.

우리의 예에서도 복제생성자를 만들어 보자.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = new A(a);

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public A(A a) {

this.intValue = a.intValue;

this.byteValue = a.byteValue;

this.stringValue = new String(a.stringValue);

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

결과는 다음과 같다.

a = [intValue=10, byteValue=5, stringValue=str]

aCloned = [intValue=10, byteValue=5, stringValue=str]

(a==aCloned) = false

* + 1. Object.clone()을 재정의해보자.

보통의 경우에 가장 좋은 방식은 clone 메쏘드를 이용하는 것이다. 위의 예제를 이렇게 고쳐 보자. 이 예제에서는 Object클래스의 clone 메쏘드를 재정의했다.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = (A)a.clone(); // clone()이 Object형을 반환하기 때문에 명시적 형변환이 필요하다.

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public Object clone() { // 여기만 바뀌었다.

return new A(intValue, byteValue, stringValue);

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

결과는 위와 같다.

그러나 이런식으로 복제하는 것은 원래 clone메쏘드를 만든 사람의 의도가 아니다. Object의 clone 메쏘드를 이용해 보자. 클래스 A에서 clone 메쏘드를 정의하지 않았으니까 상속받은 Object.clone()을 이용할 수 있을 것이다.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = (A)a.clone(); // Error.

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

그러나 컴파일이 안된다. Object.clone() 메쏘드가 protected로 선언되었기 때문이다. 이것을 해결해 보자.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = (A)a.clone();

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public Object clone() {

return super.clone(); // Error.

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

이번에는 Object.clone 메쏘드가 CloneNotSupportedException을 throw할 수 있기 때문에 컴파일타임 오류가 난다. try-catch를 이용해서 이 문제를 해결하자. 이해가 안가는 사람은 예외상황을 다루는 부분을 참조하라.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = (A)a.clone();

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A {

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public Object clone() {

Object obj = null;

try {

obj = super.clone();

} catch (CloneNotSupportedException e) {

System.out.println(e);

} finally {

return obj;

}

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

결과는 다음과 비슷할 것이다.

a = [intValue=10, byteValue=5, stringValue=str]

java.lang.CloneNotSupportedException: A

aCloned = null

(a==aCloned) = false

또 실패다. 이번에는 컴파일은 되는데 예외상황(exception)이 발생한다. 다음 예제는 제대로 작동할 것이다. Cloneable 인터페이스를 구현했다.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5, "str");

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

A aCloned = (A)a.clone();

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

System.out.println("(a==aCloned) = " + (a==aCloned));

}

}

class A implements Cloneable { // 여기를 바꾸었다.

private int intValue;

private byte byteValue;

private String stringValue;

public A(int intValue, byte byteValue, String stringValue) {

this.intValue = intValue;

this.byteValue = byteValue;

this.stringValue = stringValue;

}

public Object clone() {

Object obj = null;

try {

obj = super.clone();

} catch (CloneNotSupportedException e) {

System.out.println(e);

} finally {

return obj;

}

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + ", stringValue="

+ stringValue + "]");

}

}

결과는 다음과 같다.

a = [intValue=10, byteValue=5, stringValue=str]

aCloned = [intValue=10, byteValue=5, stringValue=str]

(a==aCloned) = false

왜 이렇게 복잡하게 만들었을까? 이제부터 이것에 대해 알아보자.

* + 1. Cloneable 인터페이스

여담부터 하면 Cloneable은 잘못된 철자법이고 Clonable이 맞다고 한다. 그래서 언젠가는 Cloneable인터페이스가 deprecated(deprecated가 궁금한 사람은 그것을 다룬 부분을 참조하라)될 수도 있다는데. 과연 그럴지는 모르겠다.

다음은 Cloneable.java의 (주석을 제와한) 전체 소스이다.

package java.lang;

public interface Cloneable {

}

아무것도 없는 썰렁한 인터페이스이다. 이런것을 marker interface라고한다. java.io.Serializable도 마찬가지이다(이것에 대해서는 직렬화를 다룬 장을 참조하라). marker interface란 아무런 메쏘드도 정의되지 않았으나 자바 컴파일러나 가상머신이 그 클래스의 어떤 특성을 알아내기 위해 사용하는 것이다.

원래의 의도대로 clone 메쏘드를 재정의하기 위해서는 이 인터페이스(Cloneable)을 반드시 구현(implements)해야 한다. 그러나 아래에서 보게 되듯이 Cloneable 인터페이스를 구현했다고 해서 반드시 clone 메쏘드를 사용할 수 있는 것은 아니다.

* + 1. Object.clone()이 하는 일 – 얕은 복제(shallow copy)

Object.clone()은 먼저 그 클래스가 Cloneable 인터페이스를 구현하는지 알아본다. 만약 구현하지 않는다면 CloneNotSupportedException이 발생한다(throw). 위에 예 중에서 그런 예가 있었다.

만약 Cloneable 인터페이스를 구현했으면 Object 클래스에 정의된 clone 메쏘드는 객체를 새로 만들어서 원본의 필드값을 그대로 복사하는 일을 한다. 그리고 생성된 객체를 반환한다. 이것이 무엇을 뜻하는지 알아보자. 먼저 필드가 기본형으로만 이루어진 객체에 대한 예이다.

public class Example {

public static void main(String[] args) {

A a = new A(10, (byte)5);

A aCloned = (A)a.clone();

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

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

System.out.println("(a==aCloned) = " + (a==aCloned));

a.setIntValue(1111);

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

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

}

}

class A implements Cloneable {

private int intValue;

private byte byteValue;

public A(int intValue, byte byteValue) {

this.intValue = intValue;

this.byteValue = byteValue;

}

public void setIntValue(int intValue) {

this.intValue = intValue;

}

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

public String toString() {

return ("[intValue=" + intValue + ", byteValue=" + byteValue + "]");

}

}

결과는 다음과 같다.

a = [intValue=10, byteValue=5]

aCloned = [intValue=10, byteValue=5]

(a==aCloned) = false

a = [intValue=1111, byteValue=5]

aCloned = [intValue=10, byteValue=5]

모든게 잘 되고 있다. Object에 있는 clone 메쏘드가 잘하고 있는 것이다. 위의 예제에서 try 블럭에서 예외상황이 발생할 일이 없다. 왜냐하면 클래스 A가 Cloneable 인터페이스를 구현했고 A의 바로 위 수퍼 클래스는 Object이기 때문이다. 그래도 예외상황이 발생한다면 그것은 가상머신이 맛이 갔거나 하는 심각한 문제일 것이다. 그래서 InternalError 객체를 만들어서 던져준(throw) 것이다.

문제는 참조형일때인데 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A("one");

A aCloned = (A)a.clone();

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

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

a.append(" two");

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

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

}

}

class A implements Cloneable {

private StringBuffer sb;

public A(String str) {

sb = new StringBuffer(str);

}

public void append(String str) {

sb.append(str);

}

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

public String toString() {

return sb.toString();

}

}

결과는 다음과 같다.

a = one

aCloned = one

a = one two

aCloned = one two

원본의 상태가 바뀌자 복사본의 상태도 바뀐다. 이런 것을 얕은 복제라고 한다. 그림 Object 클래스 – 2를 보자.

-------------------------------- 그림 Object 클래스 – 2 ---------------------------------

* + 1. 깊은 복제(deep copy)

깊은 복제란 원본과 복사본이 완전히 독립적이어서 하나가 바뀌어도 다른 하나에 영향을 미치지 않는 것을 말한다. 다음 예제는 깊은 복제에 대한 예이다.

public class Example {

public static void main(String[] args) {

A a = new A("one", 124);

A aCloned = (A)a.clone();

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

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

a.append(" two");

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

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

}

}

class A implements Cloneable {

private int number;

private StringBuffer sb;

public A(String str, int i) {

sb = new StringBuffer(str);

number = i;

}

public void append(String str) {

sb.append(str);

}

public Object clone() {

try {

A a = (A)super.clone();

a.sb = new StringBuffer(sb.toString()); // 따로 처리해 주어야 한다.

return a;

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

public String toString() {

return sb.toString() + ", " + number;

}

}

결과는 다음과 같다.

a = one, 124

aCloned = one, 124

a = one two, 124

aCloned = one, 124

기본형인 number는 그대로 Object가 처리하는데로 두면 되지만 sb는 깊은 복제를 위해 따로 처리해 주어야 한다.

대부분의 경우에는 깊은 복제를 써야 한다. 그러나 얕은 복제가 적당할 때가 있다. 예를 들어 필드가 가리키는 객체가 절대 변하지 않을 때는 얕은 복제를 쓰면 메모리가 절약된다.

그리고 직렬화 메커니즘을 이용해서 깊은 복제를 할 수 있다. 그것을 다룬 장을 참조하라.

다른 사람들이 만든 clone 메쏘드를 사용할 때에는 그 메쏘드가 깊은 복제를 지원하는지 얕은 복제를 지원하는 잘 확인하고 써야 한다. 예를 들어 java.util 패키지의 있는 컬렉션 클래스들은 대부분 얕은 복제를 한다. 배열의 복제도 얕은 복제이다.

필드가 가리키는 객체도 역시 참조형 필드가 있을 때에는 그것을 모두 고려해서 깊은 복제를 해 주어야 한다. 만약 하나의 객체로부터 출발하는 객체들의 그물(web of objects)이 엄청나게 많다면 그 많은 객체들이 모두 복제되어야 깊은 복제가 되는 것이다.

* + 1. clone()과 blank final

blank final에 대해서는 그것을 다루는 부분을 참조하라.

다음 예제를 보자.

class A implements Cloneable {

private final StringBuffer sb;

public A(String str) {

sb = new StringBuffer(str);

}

public Object clone() {

try {

A a = (A)super.clone();

a.sb = new StringBuffer(sb.toString()); // Error.

return a;

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

}

sb가 blank final이기 때문에 clone 메쏘드에서는 값이 정해질 수 없다. 이것은 어쩔 수 없다. 이럴 때는 복제 생성자(copy constructor)를 사용하면된다. 다음 예제를 보자.

class A {

private final StringBuffer sb;

public A(String str) {

sb = new StringBuffer(str);

}

public A(A a) { // 복제 생성자.

sb = new StringBuffer(a.sb.toString());

}

}

* + 1. 상속과 복제

clone 메쏘드 내에서 new를 사용하는 것은 좋지 않다. super.clone()를 사용해서 결국 Object.clone()이 객체를 만들게 하는 것이 좋다. 왜냐하면 new를 사용하면 생성되는 객체의 타입이 컴파일 타임에 정해지기 때문이다. 그러면 서브 클래스에서는 super.clone()을 사용할 수 없다. 그렇다면 서브 클래스에서 모든 일을 다 처리해 주어야 한다.

수퍼 클래스가 clone 메쏘드를 지원한다고 서브 클래스가 반드시 지원해야만 하는 것은 아니다. 경우에 따라서는 서브 클래스에서는 금지시키는 것이 좋을 때가 있다. 그러려면 수퍼 클래스의 clone 메쏘드에 throws CloneNotSupportedException이 선언되어 있어야 한다. 다음 예를 보자.

class SuperClass implements Cloneable {

public Object clone() throws CloneNotSupportedException {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

}

class SubClass {

public Object clone() throws CloneNotSupportedException {

throw new CloneNotSupportedException();

}

}

만약 SuperClass의 clone 메쏘드에서 throws CloneNotSupportedException을 빼버리면 SubClass에서 CloneNotSupportedException을 던질수(throw) 없다. 자세한 것은 예외 상황을 다루는 부분을 참조하라.

경우에 따라서는 자신은 복제가 가능하지 않지만 서브 클래스에서는 복제를 지원하게 할 수도 있다. clone 메쏘드를 protected로 지정하면 된다. 다음 예제를 보자.

class SuperClass implements Cloneable {

protected Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

}

class SubClass {

public Object clone() {

try {

return super.clone();

} catch (CloneNotSupportedException e) {

// cannot happen

throw new InternalError(e.toString());

}

}

}

SuperClass의 clone 메쏘드는 사용할 수 없지만 SubClass의 clone 메쏘드는 사용할 수 있다.

* 1. 기타

여기에 기타로 분류된 것은 중요하지 않기 때문이 아니다. 다른 맥락에서 다루는 것이 훨씬 낫기 때문이다. 특히 쓰레드와 관련된 것은 쓰레드를 어느 정도 이해하지 못하면 이해할 수 없다.

* + 1. finalize()

finalize 메쏘드에 대해서는 쓰레기 수거 (garbage collection)를 다루는 부분을 참조하라.

* + 1. getClass()

getClass 메쏘드에 대해서는 리플렉션을 다루는 부분을 참조하라.

* + 1. notify(), notifyAll()

notify, notifyAll 메쏘드에 대해서는 쓰레드를 다루는 부분을 참조하라.

* + 1. wait(), wait(long), wait(long, int)

wait 메쏘드에 대해서는 쓰레드를 다루는 부분을 참조하라.

1. 예외 상황
   1. 예외 상황(exception condition)과 예외 상황 메커니즘(exception mechanism)
      1. 예외 상황이란?

프로그램이 항상 정상적으로 작동할 수 있는 것은 아니다. 그 이유는 여러가지이다.

프로그램에 버그가 있을 수 있다. 예를 들어 0으로 나누거나 배열 인덱스가 배열의 범위를 넘어설 수 있다.

기본적인 하드웨어(CUP, 메모리, 하드디스크 등)에 문제가 생길 수 있다.

네트워크에 문제가 생길 수 있다.

찾고 있는 파일이 없을 수 있다.

사용자가 입력값을 잘못 입력할 수 있다.

자원이 부족할 수 있다.(예를 들어 메모리나 하드디스크가 부족해질 수 있다)

자바 실행 환경(예를 들어 가상 머신)에 문제가 있을 수 있다.

누군가가 전원을 뽑거나(이 누군가는 당신일 가능성이 크다) 컴퓨터를 폭파할 수 있다.

그 외에도 여러가지 문제로 프로그램의 정상적인 작동이 방해를 받을 수 있다. 이렇게 여러 가지 이유로 예외적인 상황이 발생하는 것을 예외 상황(exception condition또는 그냥 exception)이라고 한다.

* + 1. 이전의 처리 방식

예외 상황을 처리하는 방법에는 여러가지가 있다.

제일 간단한 방법은 처리를 안하는 것이다. 그러면 보통 프로그램이 다운되거나 종료될 것이다. 이것은 옛날에 선호되던 방식이다. 그리고 지금도 많은 사람들이 이것을 선호한다. 그러나 이것은 별로 안좋은 방법이다. 만약 예외 상황이 발생하더라도 프로그램이 그 예외 상황을 잘 극복할 수 있다면 더 좋을 것이다. 때에 따라 예외 상황을 수수방관할 수 밖에 없는 경우도 있다. 예를 들어 가상머신에 문제가 생겼다면 또는 CPU가 맛이 갔다면 또는 누군가 전원을 뽑아 버렸다면 또는 소행성이 지구와 충돌했다면 프로그래머로서의 당신은 무엇을 할 수 있을 것이라고 생각하는가? 예외 상황을 잘 다스린다면 더 유연한 프로그램이 될 수 있지만 그것에는 한계가 있다는 것도 염두에 두어야 한다. 모든 예외 상황을 잘 처리할 수는 없는 것이다(이것은 검사되지 않는 예외unchecked exception가 있는 이유이다).

객체 지향 프로그래밍이 나오기 전에는 함수의 반환값(return value)으로 이상한 값을 사용함으로써 예외 상황이 발생했음을 알렸다. 지금도 일부에서는 그 방법이 쓰이고 있다. 예를 들어 정상적인 반환값이 모두 양수라면 음수를 리턴하는 식이다. 또는 참조형으로는 null을 리턴함으로써 이상이 발생했음을 알릴 수도 있다. 그러나 이 방법에는 한계가 있다. 만약 void형 메쏘드(함수)라면? 만약 int를 반환하는 함수에서 모든 int값이 정상적인 리턴값이 될 수 있다면? 그리고 예외 상황을 알리는 값(예를 들어 -1)은 너무나 빈약한 정보를 제공한다.

* + 1. 예외 상황 메커니즘(exception mechanism)

이전의 처리 방식의 문제점을 해결하기 위해 만들어진 것이 exception mechanism이다. 이전의 처리 방식의 문제점의 근본원인은 반환(return) 경로가 하나라는 것이다. 지금까지 만들어진 거의 대부분의 프로그래밍 언어에서 함수(메쏘드)는 0개 1개 또는 여러개 심지어 어떤 언어에서는 가변의 갯수를 매개변수로 받아들이지만 반환값은 없거나(void형) 단 하나이다. 만약 반환값이 두개 이상이라면 문제가 해결될 수 있다. 하나는 함수가 구한 값을 반환하고 다른 하나는 발생한 예외 상황을 반환하면 된다. 사실 이것이 예외 상황 메커니즘에서 쓰이는 방식이다. 자바에는 두 개의 반환 경로가 있다. 그리고 그것에 해당하는 키워드도 각각 있다. 그것이 바로 return과 throw이다. return은 메쏘드가 구한 값을 리턴하고 예외 상황이 발생하면 return 대신 throw가 예외 상황이 발생했음을 알린다. 경로가 두개이기 때문에 throw가 넘겨주는 값은 return이 넘겨주는 값의 형 즉 반환형과 독립적일 수 있다. 반환형이 int형이라도 throw는 객체형을 사용함으로써 예외 상황에 대한 풍부한 정보를 넘겨줄 수 있는 것이다.

여기서 throw라는 단어에 대해 몇마디 하고 넘어가자. throw를 보통 ‘던지다’라고 번역한다. 이 책도 그렇게 번역한다. 실제로 throw를 키워드로 쓴 사람이 어떤 의미로 throw를 사용한 건지는 잘 모르겠다. 영어 사전을 보면 throw의 뜻으로 ‘팽개치다’, ‘내던지다’, ‘버리다’, ‘(말이 기수를)낙마시키다’, ‘좌초시키다’, ‘놀라게하다’라는 뜻도 있다. 이런 의미들은 ‘던지다’ 보다는 throw 키워드의 의미를 더 잘 전달해 준다. 그리고 ‘throw up’은 ‘토하다’라는 뜻이다. 여기서 좀 지저분한 비유를 들어보자. 우리가 음식을 먹으면 그 음식은 두 가지 경로 중 하나로 나아간다. 먼저 정상적으로 소화된다면 응아가 되어 정상적인 경로를 통해 배출된다. 이것을 return문이라고 생각할 수 있다. 만약 음식을 소화하는 과정에 문제가 생기면 우리는 토하게 된다. 이것을 비정상적(예외 상황) 경로인 throw 문이라고 생각할 수 있다.(여기서 설사는 무시한다)

* 1. 예외 상황 처리하기
     1. try-catch 문

다음 예제를 보자.

import java.net.\*;

public class Example {

public static void main(String[] args) {

InetAddress ia = InetAddress.getByName("java.sun.com");

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

}

}

컴파일이 안될 것이다. 컴파일타임 오류 메시지는 다음과 비슷할 것이다.

"Example.java": Error #: 360 : unreported exception: java.net.UnknownHostException; must be caught or declared to be thrown at line 5, column 32

다음은 java.io.InetAddress 클래스의 소스 코드에서 인용한 것이다.

public class InetAddress implements java.io.Serializable {

public static InetAddress getByName(String host) throws UnknownHostException {

return InetAddress.getAllByName(host)[0];

}

...

}

다음은 API Specification의 InetAddress.getByName 메쏘드에서 인용한 것이다.

...

Throws:

UnknownHostException - if no IP address for the host could be found.

SecurityException - if a security manager exists and its checkConnect method doesn't allow the operation

일단 SecurityException은 무시하자. InetAddress.getByName 메쏘드를 정의하는 코드에는 throws UnknownHostException가 포함되어 있다. 이 것은 이 메쏘드는 UnkownHestException이라는 예외 상황이 발생할 수 있으니 처리해 주어야 한다는 뜻이다. 그리고 API spec에서 알 수 있듯이 그 예외 상황을 IP 주소를 찾을 수 없을 때 발생한다.

자 이제 이 예외 상황을 처리하도록 하자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

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

} catch (UnknownHostException uhe) {

System.out.println("Example.main() uhe = " + uhe);

}

}

}

(만약 당신의 컴퓨터에서 인터넷을 사용할 수 있다면)결과는 다음과 비슷할 것이다.

ia = java.sun.com/192.18.97.71

인터넷을 사용할 수 없도록 하자. 예를 들어 랜(lan)선을 뽑아버리고 실행하자. 결과는 다음과 비슷할 것이다.

Example.main() uhe = java.net.UnknownHostException: java.sun.com

이번에는 인터넷을 이용할 돈이 없는 사람들을 위한 예제이다(그들도 자바를 배울 권리는 있다).

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileInputStream fis = new FileInputStream("c:/temp/BavaLang.txt");

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

} catch (FileNotFoundException fnfe) {

System.out.println("Example.main() fnfe = " + fnfe);

}

}

}

c:/temp디렉토리(폴더)에 BavaLang.txt라는 파일이 존재하는 상태에서 실행하면 결과는 다음과 비슷할 것이다.

fis = java.io.FileInputStream@5d87b2

그 파일이 없는 상태에서 또는 아예 temp디렉토리가 없는 상태에서 실행하면 결과는 다음과 비슷할 것이다.

Example.main() fnfe = java.io.FileNotFoundException: c:/temp/BavaLang.txt (지정된 파일을 찾을 수 없습니다)

이 예제도 소스 코드와 API spec을 뒤져보면 앞의 예제와 마찬가지일 것이다. 인터넷이 안되는 사람은 나중에 인터넷을 연결하거든 위의 인터넷 예제를 꼭 테스트해보기 바란다.

위의 두 예제로 알 수 있는 것을 정리해 보자.

어떤 메쏘드에 throws 절이 있으면 throws 키워드 바로 다음에 있는 예외 상황 클래스가 가리키는 예외 상황을 처리해 주어야 한다. 그렇지 않으면 컴파일타임 오류가 난다.(항상 그런 건 아니다. 그것에 대해서는 다음에 설명할 것이다.)

예외 상황을 처리하기 위해서는 try-catch문을 써야 한다.

catch다음에는 그 예외 상황 클래스를 괄호 안에 감싸서 써야 한다. 위의 예에서는 catch (UnknownHostException uhe)처럼 해 주었다.

만약 예외 상황이 발생하지 않으면 try 블록이 실행되고 catch 블럭은 무시된다. 반대로 예외 상황이 발생하면 try 블록의 실행이 중단되고(try 블록의 System.out.print 메쏘드는 불리지 않았다) catch 블럭이 실행된다.

위에서 자바에는 반환 경로가 둘이라고 했다. try 문은 이 반환 경로중 return을 통한 정상적인 경로와 throw를 통한 예외 상황의 경로중 어떤 경로를 통해서 반환이 되었는지를 판단하여 try 블록의 문장을 계속 실행할 것인지 catch 블럭으로 제어(controll)을 넘길 것인지를 판단하는 것이다.

지금까지 살펴본 것이 자바의 예외 상황 처리 방법이다.

예외 상황이 발생하는 경우와 발생하지 않는 경우에는 해야 할 일이 다를 것이다. 예외 상황이 발생하는 경우의 처리는 catch 블럭에 정상적인 실행일 경우에 할 일은 try 블럭에 코드를 넣으면 된다.

위의 예제를 좀더 그럴 듯하게 하면 다음과 같다.

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

System.out.println("당신은 인터넷을 사용하실 수 있습니다.");

} catch (UnknownHostException uhe) {

System.out.println("돈을 안내서 인터넷이 끊긴 것 같습니다.");

}

}

}

결과는 둘 중에 하나일 것이다.

당신은 인터넷을 사용하실 수 있습니다.

돈을 안내서 인터넷이 끊긴 것 같습니다.

물론 실제 프로그램에서는 더 복잡한 일을 할 것이다.

* + 1. try-catch-finally문

경우에 따라서는 예외가 발생하든 안하는 반드시 실행해야 하는 코드가 있을 수 있다. 그럴 때 쓸 수 있는 것이 finally이다. finally 블럭에 있는 것은 무슨일이 있더라도 실행된다.

다음 예제는 finally의 사용법을 보여준다.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

System.out.println("당신은 인터넷을 사용하실 수 있습니다.");

} catch (UnknownHostException uhe) {

System.out.println("돈을 안내서 인터넷이 끊긴 것 같습니다.");

} finally {

System.out.println("인터넷이 되든 안되는 밥은 먹어야 살 수 있습니다.");

}

}

}

정상적인 실행의 결과는 다음과 같다.

당신은 인터넷을 사용하실 수 있습니다.

인터넷이 되든 안되는 밥은 먹어야 살 수 있습니다.

예외 상황의 결과는 다음과 같다.

돈을 안내서 인터넷이 끊긴 것 같습니다.

인터넷이 되든 안되는 밥은 먹어야 살 수 있습니다.

* + 1. 처리되지 않은 예외 상황

예외 상황이 처리되지 않으면 그 메쏘드를 부른 메쏘드로 던져진다. 다음 예제를 보자.

import java.awt.\*;

import java.io.\*;

public class Example {

public static void main(String[] args) {

System.out.println("start of main");

methodA();

System.out.println("start of main");

}

public static void methodA() {

System.out.println("start of A");

methodB();

System.out.println("end of A");

}

public static void methodB() {

System.out.println("start of B");

methodC();

System.out.println("end of B");

}

public static void methodC() {

System.out.println("start of C");

methodD();

System.out.println("end of C");

}

public static void methodD() {

throw new NullPointerException();

}

}

결과는 다음과 비슷할 것이다..

start of main

start of A

start of B

start of C

java.lang.NullPointerException

at Example.methodD(Example.java:30)

at Example.methodC(Example.java:25)

at Example.methodB(Example.java:19)

at Example.methodA(Example.java:13)

at Example.main(Example.java:9)

Exception in thread "main"

결국은 main도 처리할 수 없자 프로그램이 비정상적으로 끝나 버렸다. 즉

System.out.println("start of main");

실행되지 않았다. 그리고 예외상황에 대한 메시지를 보여준다. 자세한 것은 쓰레드를 다루는 부분을 참조하라.

* 1. 예외 상황 발생시키기
     1. 왜 예외 상황을 발생시켜야 하는가?

예외 상황을 처리하는 일은 해야만 하는 일이지만 짜능나는 일이 아닐 수 없다. 그런데 왜 예외 상황을 발생 시켜야 하는가? 우리가 예외 상황을 발생시키면 기존의 예외 상황에다 예외 상황을 추가함으로써 상황을 더 예외적으로 만드는 것이(상황을 악화시키는 것이) 아닐까? 당신은 throw문을 배우기 전에 이런 생각을 먼저 해 보아야 한다. 딴 사람이 throw하는 것을 처리하는 것도 바쁜데 왜 내가 throw를 해서 그것을 처리해야 할 사람(그 사람이 남이면 다행이지만 그 사람은 불행이도 나일 수도 있다)을 괴롭혀야 하나? 차라리 throw문을 안 배워서 세상을 어지럽지히 않고 착하게 사는 것이 낫지 않을까?

먼저 throw을 하도록 코딩을 하는 사람은 왜 그래야만 했는지를 생각해 보자. InetAddress.getByName 메쏘드는 왜 UnknownHostException을 던져야(throw) 했는가? getByName 메쏘드는 호스트를 찾을 수 없으면 지가 알아서 처리하지 않고 왜 그 메쏘드를 부르는 사람을 귀찮게 하나? 그 이유는 다음의 두 예를 보면 알 수 있다.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

System.out.println("당신은 인터넷을 사용하실 수 있습니다. 참 잘났습니다.");

} catch (UnknownHostException uhe) {

System.out.println("돈을 안내서 인터넷이 끊긴 것 같습니다. 돈안내고 뭐했습니까?");

}

}

}

두번째 예제는 다음과 같다.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

System.out.println("인터넷이 되면 뭐하냐. 맨날 게임만 하는데...");

} catch (UnknownHostException uhe) {

System.out.println("세상에는 인터넷보다 소중한 것이 참 많이 있습니다.");

}

}

}

두 예제 다 InetAddress.getByName 메쏘드를 사용한다. 문제는 예외 상황이 발생했을 때 둘은 다른 일을 한다는 것이다. 예외 상황이 발생했을 때 무슨일을 해야 하는지를 getByName 메쏘드는 알 수가 없다. 그렇기 때문에 그 메쏘드를 부른 메쏘드(caller)에게 처리를 떠넘기는 것이다.

당신이 만약 API를 만든다면 예외 상황의 처리를 떠넘겨야 할일이 많을 것이다. 그렇지 않더라도 자신이 만든 클래스들 중 어떤 클래스에서 예외 상황을 처리해야 적당한지를 결정해야 한다. 그리고 적당하지 않은 클래스의 메쏘드에서 예외 상황을 만났다면 그것을(예외 상황의 처리를) 다른 클래스의 메쏘드로 떠넘겨야 한다. 그리하여 당신도 throw문의 사용법을 배워야 하는 것이다.

* + 1. throw와 throws

먼저 throw의 용법을 알아보자. 다음은 제곱근을 구하는 메쏘드이다. 만약 매개변수가 음수라면 IllegalArgumentException을 던진다.

import java.net.\*;

public class Example {

public static void main(String[] args) {

System.out.println(calculateSquareRoot(2));

System.out.println(calculateSquareRoot(-2));

}

public static double calculateSquareRoot(double d) {

if (d < 0) {

IllegalArgumentException e = new IllegalArgumentException();

throw e;

}

return Math.sqrt(d);

}

}

결과는 다음과 비슷할 것이다.

1.4142135623730951

java.lang.IllegalArgumentException

at Example.calculateSquareRoot(Example.java:11)

at Example.main(Example.java:6)

Exception in thread "main"

지금은 단독으로 throw를 사용했다. 그러나 보통 throw는 throws와 쌍을 이룬다.(두 경우의 차이는 다음에 다룬다) 다음은 FileInputStream.java에서 인용한 것이다.

public class FileInputStream extends InputStream {

public final FileDescriptor getFD() throws IOException {

if (fd != null)

return fd;

throw new IOException();

}

...

}

메쏘드의 헤더(header) 부분에 throws IOException을 넣음으로써 이 메쏘드가 IOException을 던질 수 있음을 알린다. 그리고 메쏘드의 몸체(body)에 throw문을 사용한다.

* 1. 예외 상황 클래스
     1. Throwable 클래스

Object가 모든 클래스의 아담(최고 조상)이라면 예외상황 클래스의 아담은 Throwable 이다. 이것은 모든 예외 상황 클래스가 Throwable클래스이거나 Throwable의 서브 클래스이어야 한다. 그러므로 다음 코드는 컴파일 되지 않는다.

public void a() throws String { // String은 Throwable의 서브 클래스가 아니다.

try {

int i = 1 / 0;

} catch (String str) { // String은 Throwable의 서브 클래스가 아니다.

}

throw new String(); // String은 Throwable의 서브 클래스가 아니다.

}

* + 1. 검사되지 않는 예외 상황(unchecked exception)

예외 상황에는 검사되는 예외 상황과 검사되지 않는 예외 상황이 있다. 여기서는 먼저 검사되지 않는 예외 상황 클래스에 대해 알아본다.

Error 클래스와 그것의 서브 클래스, RuntimeException 클래스와 그것의 서브 클래스는 검사되지 않는 예외 상황 클래스이다. 이 클래스의 예외 상황을 처리하지 않아도 즉 try문으로 감싸지 않아도 컴파일타임 오류가 나지 않는다.

Error 클래스는 심각한 시스템의 에러를 나타낸다. 예를 들어 OutOfMemeryError는 메모리가 부족함을 나타내고, VirtualMachineError는 가상 머신에 문제가 발생함을 나타낸다.

RuntimeException은 보통 프로그래머의 실수로 생기는 예외 상황이다. 즉 버그에서 생기는 예외 상황이라고 할 수 있다. 예를 들어 NullPointerException은 메쏘드나 필드 등을 사용하려고 하는데 객체를 가리켜야 할 변수가 아무것도 안 가리킴을(null 임을) 나타내고, ArrayOutOfBoundException은 배열 인덱스가 잘못되었음을 나타낸다.

* + 1. 검사되는 예외 상황(checked exception)

검사되는 예외 상황은 검사되지 않는 예외 상황을 제외한 모든 예외 상황을 말한다.

검사되는 예외 상황은 반드시 프로그래머에 의해서 명시적으로 처리되어야 한다. 그렇지 않으면 컴파일타임 오류가 난다. 다시 말해 try문으로 감싸주거나 다른 메쏘드에 그 처리를 떠넘겨야 한다.

try문으로 감싸는 것은 해 보았으니까 떠넘기는 방법을 알아보자. 다음은 FileInputStream.java에서 인용한 것이다.

public class FileInputStream extends InputStream {

public int read(byte b[], int off, int len) throws IOException {

return readBytes(b, off, len);

}

private native int readBytes(byte b[], int off, int len) throws IOException;

...

}

readBytes 메쏘드는 검사되는 예외 상황 클래스인 IOException을 던질 수 있다. 왜냐하면 throws IOException이라고 선언했기 때문이다. read 메쏘드는 readBytes 메쏘드를 부른다. 그러므로 read 메쏘드는 try 문을 사용하거나 다른 메쏘드에 그 처리를 떠넘겨야 한다. read는 떠넘기는 것을 선택했다. 방법은 간단하다. 자기 자신도 throws IOException을 선언한 것이다.

* + 1. 컴파일러는 왜 검사되지 않는 예외 상황에 관대한가?

먼저 에러(Error 클래스로 대표되는)에 대해 알아보자. 에러는 어디서든 발생할 수 있다. 예를 들어 어떤 메쏘드 안에 다음과 같은 코드가 있다고 해 보자.

int i;

아주 평범한 변수 선언이다. 그러나 이 한 줄을 실행하는데 스택이 부족해질 수도 있고 가상머신이 맛이 갈 수도 있다. 기타 여러 가지 원인으로 에러가 발생할 수 있는 것이다. 만약 이것을 처리하겠다고 다음과 같이 한다고 생각해 보자.

try {

int i;

} catch (StackOverflowError sofe) {

...

} catch (VirtualMachineError vme) {

...

}

과연 이런 방식으로 코딩하는 것이 가능할지도 의문이고 가능하더라도 전혀 실효성이 없다. 게다가 위에서는 ...으로 표시했는데 스택이 부족하거나 가상머신에 에러가 생겼을 때 그것을 (예를 들어 네크워크 장애를 처리할 때처럼)매끄럽게 처리할 수 있을까? 에러는 보통 처리가 불가능하기 때문에 catch해 준다고 해도 별 의미가 없다.

런타임 예외상황(RuntimeException 클래스로 대표되는)도 명시적으로 핸들링하지 않아도 된다. 런타임 예외상황은 프로그램의 버그인 경우가 많다. 예를 들어 0으로 나누거나 배열 인덱스를 잘못 설정하는 경우등이다. 이런 것의 처리를 컴파일 타임에 체크한다는 것은 런타임 버그를 컴파일 타임에 잡아내겠다는 생각이다. 또는 버그가 생겼을 때를 매끄럽게 처리하겠다는 것이다. 아직은 프로그램 공학이 그렇게 발전하지 않았다. 그리고 에러와 마찬가지로 try 문의 홍수를 만나게 될 것이다. 예를 들어 a.method()처럼 변수(a)가 가리키는 객체의 인스턴스 메쏘드를 부르는 모든 경우에 NullPointerException을 잡아(catch) 주는 try-catch문을 추가해야 한다.

결론적으로 검사되지 않는 예외상황인 에러(Error)와 런타임 예외 상황(RuntimeException)은 컴파일러가 신경쓰지 않는다.

* + 1. 사용자 정의형 예외 상황 클래스

프로그래머는 자신의 예외 상황 클래스를 만들어 사용할 수 있다. 단 하나의 조건은 그 클래스가 Throwable의 후손이어야 한다는 것이다. 그러나 보통 Throwable을 직접 상속받지는 않는다. 보통 Exception 클래스나 Exception 클래스의 후손 클래스(예를들어 입출력에서는 IOException)를 상속받는다. 기존에 있는 예외 상황 클래스와 다른 예외 상황을 정의하고 싶을 때 자신이 예외 상황 클래스를 새로 만들어서 사용하면 된다.

* 1. 예외 상황 메커니즘의 올바른 사용법
     1. 예외 상황이 발생하면 적절한 처리를 해 주어야 한다.

자바에서는 검사되는 예외 상황을 적절히 처리하지 않으면 컴파일타임 오류가 난다. 그것은 적절히 처리하라는 뜻이지 다음과 같이 하라는 뜻이 아니다. 만약 당신이 다음과 같은 코드를 남발한다면 예외 상황 메커니즘을 만든 사람은 상당히 슬퍼할 것이다.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

} catch (UnknownHostException e) {

// 아무 처리를 안해준다면 예외 상황 메커니즘이 무슨 소용인가?

}

}

}

* + 1. 예외 상황 클래스의 수퍼 클래스의 사용을 자제해야 한다.

다음 예제를 보자.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

int i = 1 / 0;

} catch (Exception e) {

System.out.println("인터넷 안된다니까!!!");

}

}

}

인터넷이 되든 안되든 결과는 다음과 같다.

인터넷 안된다니까!!!

잘못된 정보를 얻을 수 있다. 즉 문제는 0으로 나눈 것이었는데 죄없는 전용선이 욕을 먹을 수 있다는 얘기다. UnknownHostException의 수퍼 클래스인 Exception을 사용했기 때문이다. UnknownException을 사용해보자.

import java.net.\*;

public class Example {

public static void main(String[] args) {

try {

InetAddress ia = InetAddress.getByName("java.sun.com");

int i = 1 / 0;

} catch (UnknownHostException e) {

System.out.println("인터넷 안된다니까!!!");

}

}

}

인터넷이 된다면(그리고 sun사가 아직 안 망했다면) 결과는 다음과 비슷할 것이다. 제대로 된 정보를 얻는다는 것을 알 수 있다.

java.lang.ArithmeticException: / by zero

at Example.main(Example.java:7)

Exception in thread "main"

인터넷이 안되는 사람은 위와 비슷한 결과를 볼 수 없는데 이제 많이 배웠으니까 혼자서 적절한 예제를 만들어 테스트해보기 바란다.

* + 1. 누가 처리할지를 잘 판단해야 한다.

예외 상황을 누가(어느 클래스가 또는 어느 메쏘드가) 처리할 것인가를 결정하는 것은 아주 중요하다. 그 결정을 할 때의 고려해야 할 것은 누가 예외 상황을 적절히 처리할 정보를 가지고 있는가이다.

* 1. try-catch-finally문의 해부
     1. try

다음 예제는 컴파일이 되지 않는다.

public class Example {

public static void main(String[] args) {

try { // try에는 catch또는 finally가 하나는 있어야 한다.

}

;

catch (Exception e) { // try 없는 catch는 안된다.

}

;

finally { // try 없는 finally는 안된다.

}

try // 블럭이 있어야 한다.

A.method()

catch (Exception e) // 블럭이 있어야 한다.

System.out.println("Error Occured");

finally // 블럭이 있어야 한다.

System.out.println("finally");

}

}

try문은 세가지 형식을 취할 수 있다. try-catch-finally, try-catch, try-finally가 그것이다. 셋 모두 합법적이다.

* + 1. 무엇을 catch할 수 있고 무엇을 catch 해야 하나?

어이가 없겠지만 다음 예제는 컴파일 타임 오류가 나지 않는다.

public class Example {

public static void main(String[] args) {

try {

} catch (Exception e) {

}

}

}

그러나 다음 예제는 컴파일타님 오류가 난다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

} catch (IOException e) {

}

}

}

IOException은 검사되는 예외 상황이다. 컴파일러는 검사되는 예외 상황을 처리해 주어야 하는데 안 하는 경우에도 문제삼지만 발생할 일도 없는 검사되는 예외 상황을 처리해 주는 것도 문제삼는다. 이것은 throws와 throw의 관계와도 관계된다. 다음 예제를 보자.

class A {

public void method() {

throw new IOException();

}

}

컴파일타임 오류가 날 것이다. 왜냐하면 method 메쏘드에는 throws IOException을 선언해주지 않았기 때문이다. 어떤 메쏘드는 throws에 선언해 준 검사되는 예외 상황만 throw할 수 있다. 다음 예를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

A.method();

} catch (IOException e) { // 에러.

}

}

}

class A {

public static void method() {

}

}

메쏘드 A.method에는 검사되는 예외 상황이 선언되어 있지 않기 때문에 던져질 수도 없다. 그러므로 위의 try 블럭에서는 검사되는 예외 상황인 IOException이 발생할 가능성이 전혀 없는 것이다. 그러므로 컴파일타임 오류가 난다.

반대로 검사되는 예외 상황이 발생할 가능성이 있으면 반드시 어떤식으로든 처리해 주어야 한다. 즉 try문으로 감싸거나 그 메쏘드를 부른 메쏘드로 떠넘겨야 한다.

맨 위에 Exception이 나오는 예제에서 컴파일타임 오류가 안나는 이유는 무엇일까? Exception 클래스의 서브 클래스 중에는 RuntimeException이 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

A.method();

} catch (Exception e) {

System.out.println(e);

}

}

}

class A {

public static void method() {

throw new NullPointerException();

}

}

결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

NullPointerException은 RuntimeException의 서브 클래스이다. catch (Exception e)는 Exception의 서브 클래스인 RuntimeException을 그리고 RuntimeException의 서브 클래스인 NullPointerException을 잡을 수 있다. 그리고 RuntimeException은 throws에 선언된 것과 상관 없이 던져질 수 있다. 그러므로 위에서 A.method()가 검사되는 예외 상황을 던질 수 없는 메쏘드임에도 불구하고 catch (Exception e)이 컴파일러의 불만을 사지 않는 것이다. 맨 위의 예가 컴파일타임 오류가 나지 않는 이유가 어느 정도는 설명이 되었을 것이다. 그러나 그 예에서는 try 블럭에 아무것도 없다. 그럼에도 컴파일타임 오류가 나지 않은 이유는 나도 모른다.

* + 1. catch의 순서

catch가 여러 개 있다면 발생한 예외 상황은 순서대로 catch에 있는 매개변수(즉 예외 상황 클래스)들을 검사하여 자신에게 적용되는 것을 찾는다면 그곳에 들어간다. 그리고 그 블럭을 마치면 그 밑에 있는 다른 catch절들은 거들떠보지도 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

throw new IndexOutOfBoundsException();

} catch (ArrayIndexOutOfBoundsException e) { // 형(type)이 안 맞는다.

System.out.println("ArrayIndexOutOfBoundsException : " + e);

} catch (IndexOutOfBoundsException e) {

System.out.println("IndexOutOfBoundsException : " + e);

} catch (RuntimeException e) { // 앞에 있는 catch에 들어갔으므로 무시된다.

System.out.println("RuntimeException : " + e);

} catch (Exception e) { // 앞에 있는 catch에 들어갔으므로 무시된다.

System.out.println("Exception : " + e);

}

}

}

결과는 다음과 같다.

IndexOutOfBoundsException : java.lang.IndexOutOfBoundsException

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

int i = 1 / 0;

} catch (Throwable th) {

} catch (Exception e) { // 여기엔 도달할 수 없다.

}

}

}

컴파일이 안될 것이다. 만약 예외 상황이 생긴다면 Throwable이 다 그 예외 상황을 몽땅 흡수하기 때문에 Exception에게는 기회를 주지 않는다.

* + 1. catch 블럭 또는 finally 블럭에서 예외 상황이 발생하면?

catch블럭 또는 finally 블럭에서도 예외 상황은 발생할 수 있다. 그러면 새로 발생한 예외 상황이 전달된다. 이전에 발생한 예외 상황은 실종된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

methodA();

} catch (Exception e) {

System.out.println("main() Exception when calling methodA : " + e);

}

try {

methodB();

} catch (Exception e) {

System.out.println("main() Exception when calling methodB : " + e);

}

}

public static void methodA() {

try {

methodC();

} catch (Exception e) {

System.out.println("methodA() Exception when calling methodC : " + e);

methodD();

}

}

public static void methodB() {

try {

methodC();

} finally {

System.out.println("methodB() in finally");

methodD();

}

}

public static void methodC() {

throw new NullPointerException();

}

public static void methodD() {

throw new ArrayIndexOutOfBoundsException();

}

}

결과는 다음과 비슷할 것이다.

methodA() Exception when calling methodC : java.lang.NullPointerException

main() Exception when calling methodA : java.lang.ArrayIndexOutOfBoundsException

methodB() in finally

main() Exception when calling methodB : java.lang.ArrayIndexOutOfBoundsException

main메쏘드까지 왔을 때는 NullPointerException이 실종된 것을 알 수 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

throw new NullPointerException();

} catch (Exception e) {

System.out.println("catch : " + e);

} finally {

System.out.println("finally");

}

}

}

결과는 다음과 비슷할 것이다.

catch : java.lang.NullPointerException

finally

이것을 조금 바꿔보자.

public class Example {

public static void main(String[] args) {

try {

throw new NullPointerException();

} catch (Exception e) {

System.out.println("catch : " + e);

throw new NullPointerException();

} finally {

System.out.println("finally");

}

}

}

결과는 다음과 비슷할 것이다.

catch : java.lang.NullPointerException

finally

java.lang.NullPointerException

at Example.main(Example.java:7)

Exception in thread "main"

catch 블럭에서 또 예외 상황이 발생해서 프로그램이 비정상적으로 종료되었지만 finally 블럭은 그래도 실행되었다. 다음절에서도 finally 블럭의 실행되고자 하는 의지를 볼 수 있을 것이다.

* + 1. finally와 break, return, continue

다음 예제를 실행해 보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

try {

System.out.println(i);

if (i > 1)

break;

} finally {

System.out.println("finally never missed!!!");

}

}

}

}

결과는 다음과 같다.

0

finally never missed!!!

1

finally never missed!!!

2

finally never missed!!!

“2”를 화면에 뿌린다음에 break가 실행된다. 그러나 for문에서 빠져나오기 전에 finally절에 있는 것이 실행된다. 이것은 return문도 마찬가지이다.

public class Example {

public static void main(String[] args) {

try {

System.out.println("before return");

return;

} finally {

System.out.println("finally never missed!!!");

}

}

}

결과는 다음과 같다.

before return

finally never missed!!!

conitue도 마찬가지이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 3; i++) {

try {

if (i > 0)

continue;

} finally {

System.out.println("finally never missed!!!");

}

System.out.println("after try-finally");

}

}

}

결과는 다음과 같다.

finally never missed!!!

after try-finally

finally never missed!!!

finally never missed!!!

* + 1. try 문의 중첩

try 문은 중첩될 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

try {

throw new Exception();

} catch (RuntimeException e) {

System.out.println("RuntimeException : " + e);

}

} catch (Throwable th) {

System.out.println("Throwable : " + th);

}

}

}

결과는 다음과 비슷할 것이다.

Throwable : java.lang.Exception

안쪽 catch에서 잡히지 않자 바깥쪽 catch에서 잡힌다. 안쪽에서 잡힌다면 바깥쪽에서는 정상적으로 실행된 것으로 간주될 것이다. 위의 예제를 약간 바꾸었다.

public class Example {

public static void main(String[] args) {

try {

try {

throw new Exception();

} catch (Exception e) {

System.out.println("Exception : " + e);

}

} catch (Throwable th) {

System.out.println("Throwable : " + th);

}

}

}

결과는 다음과 같다.

Exception : java.lang.Exception

* 1. throw와 throws의 해부
     1. throw

다음 예제를 실행해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

methodA();

} catch (IOException e) {

System.out.println(e);

}

}

public static void methodA() throws IOException {

throw getException();

}

public static IOException getException() {

return new IOException();

}

}

결과는 다음과 비슷할 것이다.

java.io.IOException

throw 다음에 메쏘드 호출이 올 수도 있다는 것을 보여준다. 물론 그 메쏘드의 반환형이 IOException으로 문맥에 적절하기 때문이다.

throw null;

과 같은 것은 허용되지 않는다. 이것은 다음 예가 허용되는 것과 다르다.

public String method() {

return null;

}

* + 1. throw와 return

다음 예제는 컴파일이 안될 것이다.

public class Example {

public String method(boolean b) {

if (b) {

} else {

return "return value";

}

}

}

왜냐하면 b가 true일 때는 String형을 반환하지 않기 때문이다. 그러나 다음 예제는 컴파일이 된다.

public class Example {

public String method(boolean b) {

if (b) {

throw new NullPointerException();

} else {

return "return value";

}

}

}

throw가 실행되면 어짜피 return을 통한 경로는 무시되기 때문이다. throw와 return의 두개의 반환 경로가 있지만 둘은 배타적이다. 즉 둘 중에 하나가 유효하면 나머지 하나는 무시된다.

* + 1. throws

다음 예제를 컴파일해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

methodB();

}

public void methodA() throws IOException, Exception, Throwable {

}

public static void methodB() throws RuntimeException, Error {

}

}

무사히 컴파일된다. throws IOException, Exception, Throwable과 같이 해도 컴파일타임 오류는 안난다. 그러나 좋은 스타일은 아니다. Throwable이 나머지 둘 모두의 수퍼 클래스이기 때문이다. throws에 검사되지 않는 예외상황을 쓸 수도 있다. 그러나 main 메쏘드에서 methodB를 부르는 것에서 알 수 있듯이 try로 싸지 않아도 된다.

* 1. 상속과 예외 상황
     1. 클래스의 상속과 예외 상황

checked exception이 선언된 메쏘드를 재정의할 때는 주의할 것이 있다. 수퍼 클래스에서 정의 된 exception보다 더 많이 exception을 정의할 수 없다는 것이다. 이것은 코드의 재사용성을 높이기 위한 것이다. 다음 예를 보자.

import java.io.\*;

class SuperClass {

public void method() throws IOException {

}

}

class SubClass extends SuperClass {

public void method() throws Exception { // Error.

}

}

컴파일이 안되는 것을 알 수있다. 왜 이런 것이 컴파일이 안되는지 알아보자. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

SuperClass superClass = new SuperClass();

try {

superClass.method();

} catch (IOException ioe) {

System.out.println("Example.main() IOException occured!");

}

}

}

class SuperClass {

public void method() throws IOException {

throw new IOException();

}

}

결과는 다음과 같다.

Example.main() IOException occured!

다음 예제를 보자.

import java.awt.\*;

import java.io.\*;

public class Example {

public static void main(String[] args) {

SuperClass superClass = new SubClass();

try {

superClass.method();

} catch (IOException ioe) {

System.out.println("Example.main() IOException occured!");

}

}

}

class SuperClass {

public void method() throws IOException {

throw new IOException();

}

}

class SubClass extends SuperClass {

public void method() throws Exception { // 컴파일타임 오류가 안난다고 가정하면

throw new AWTException("AWT Exception");

}

}

* + 1. 생성자와 예외 상황

다음 예제를 보자.

import java.io.\*;

class SuperClass {

public SuperClass() throws IOException {

}

}

class SubClass extends SuperClass {

public SubClass() { // Error

}

}

컴파일이 되지 않을 것이다. 수퍼 클래스의 생성자의 throws 절에 규정된 예외 상황을 서브 클래스의 생성자에도 throws 절에 규정해야 한다. 왜 그런지 알아보자. 다음 예제를 보자.

import java.io.\*;

class SuperClass {

public SuperClass() throws IOException {

}

}

class SubClass extends SuperClass {

public SubClass() {

try { // 이런 식으로 예외를 잡을 수 없다.

super();

} catch (IOException e) {

}

}

}

try-catch 로 수퍼 클래스의 생성자에서 발생한 예외를 잡을 수 없다. 만약 수퍼 클래스의 생성자에서 발생한 예외를 잡을 수 있었다면 깨진 객체가 생성될 수 있었을 것이다. 왜냐하면 수퍼 클래스의 생성자에서 예외가 발생했다면 필드가 제대로 초기화되지 않았을 가능성이 있기 때문이다. 수퍼 클래스의 생성자에서 예외가 발생하면 서브 클래스의 생성자도 그 예외를 던져야 하며 결국은 new 문에서 예외상황이 발생하여 객체가 생성되지 않아야 하는 것이다. 그러므로 수퍼 클래스의 생성자의 throws 절에 규정된 예외를 서브 클래스의 생성자에서도 반드시 규정해야 하는 것이다. 다음 예제처럼 더 많은 예외를 규정해도 된다.

import java.io.\*;

class SuperClass {

public SuperClass() throws IOException {

}

}

class SubClass extends SuperClass {

public SubClass() throws Exception {

}

}

다음 예제는 무사히 컴파일 될 것이다.

import java.io.\*;

class SuperClass {

public SuperClass() throws IOException {

}

public SuperClass(int a) {

}

}

class SubClass extends SuperClass {

public SubClass() throws IOException {

}

public SubClass(int a) {

super(a);

}

}

조금 이상한 예제이지만 다음 예제도 컴파일이 잘 된다.

import java.io.\*;

class SuperClass {

public SuperClass() throws IOException {

}

public SuperClass(int a) {

}

}

class SubClass extends SuperClass {

public SubClass() {

super(12);

}

public SubClass(int a) throws IOException {

super();

}

}

서브 클래스의 생성자가 수퍼 클래스의 생성자들 중 어떤 것을 쓰느냐에 따라 그 수퍼 클래스의 생성자에 맞게 throws절을 규정해 주면 되는 것이다.

이런 문제는 디폴트 생성자를 사용할 때에도 조심해야 한다. 다음 예제를 보자.

class SuperClass {

SuperClass() throws Exception {

}

}

class SubClass extends SuperClass {

}

컴파일이 되지 않을 것이다. SubClass의 디폴트 생성자에는 당연히 throws절이 선언되어 있지 않기 때문이다. 이럴 때는 명시적으로 생성자를 선언해서 throws절에서 최소한 Exception 예외를 선언해야 한다.

* + 1. 인터페이스의 구현

인터페이스와 그것을 구현한 클래스와의 관계는 수퍼 클래스와 서브 클래스와 마찬가지이다. 즉 인터페이스의 어떤 메쏘드에서 선언한 예외상황보다 더 많은 예외상황을 그 메쏘드를 구현한 메쏘드가 throw할 수는 없다.

* + 1. 두개의 인터페이스에서 똑같은 메쏘드를 구현하는 경우

이런 일은 흔한 일도 아니고 좋은 스타일은 아니지만 일단 다루어 보자. 다음 예제를 보자.

import java.awt.\*;

import java.io.\*;

interface A {

void method() throws IOException;

}

interface B {

void method() throws AWTException;

}

무사히 컴파일 된다.

다음 예제를 보자.

import java.awt.\*;

import java.io.\*;

public class Example implements A, B {

public void method() {

}

}

interface A {

void method() throws IOException;

}

interface B {

void method() throws AWTException;

}

역시 무사히 컴파일 된다.

그렇다면 다음은 어떨까?

import java.awt.\*;

import java.io.\*;

public class Example implements A, B {

public void method() throws IOException {

}

}

interface A {

void method() throws IOException;

}

interface B {

void method() throws AWTException;

}

컴파일타임 오류가 발생한다. 위에서 살펴 보았듯이 서브 타입은 수퍼 타입보다 더 적은 검사되는 예외 상황을 던질수 있어야 한다. 위의 예에서는 그렇지가 못하다. B.method()에서는 IOException이 throws에 선언되지 않은 것이다.

* 1. 필드 초기화와 예외 상황
     1. 정적 필드의 초기화와 예외상황

정적 필드를 초기화하는 곳에서는 검사되는 예외상황이 발생하면 안된다. 왜냐하면 그것을 try-catch로 잡아 줄 수 없기 때문이다. 즉 정적 필드 초기화문(initializer for static variable)과 정적 초기화 블록(static initializer)에서는 검사되는 예외상황이 발생하면 안된다는 것이다. 다음 예제를 보자.

public class Example {

static int staticField = exceptionMaker(); // Error.

static {

exceptionMaker(); // Error.

}

static int exceptionMaker() throws Exception {

throw new Exception();

}

}

컴파일이 되지 않을 것이다.

그렇다면 검사되지 않는 예외 상황은 어떻게 다음 예제를 보자.

public class Example {

static {

exceptionMaker();

}

public static void main(String[] args) {

}

static void exceptionMaker() {

throw new RuntimeException();

}

}

컴파일은 되지만 결과는 다음과 비슷할 것이다.

java.lang.ExceptionInInitializerError

Caused by: java.lang.RuntimeException

at Example.exceptionMaker(Example.java:10)

at Example.<clinit>(Example.java:3)

Exception in thread "main"

* + 1. 인스턴스 필드의 초기화와 예외상황

위와 비슷한 예제이다.

public class Example {

int instanceField = exceptionMaker(); // Error.

{

exceptionMaker(); // Error.

}

int exceptionMaker() throws Exception {

throw new Exception();

}

}

컴파일이되지 않을 것이다. 그러나 인스턴스 필드의 초기화는 정적 필드의 초기화와 다른 점이 있다. 인스턴스 필드의 초기화에서 발생하는 예외 상황을 try-catch로 잡아줄 수 있는 방법이 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

A a = new A();

} catch (Exception e) {

System.out.println("이렇게 잡으면 된다 : " + e);

}

}

}

class A {

int instanceField = exceptionMaker();

{

exceptionMaker();

}

A() throws Exception { // 여기를 주목하라.

}

int exceptionMaker() throws Exception {

throw new Exception();

}

}

결과는 다음과 비슷할 것이다.

이렇게 잡으면 된다 : java.lang.Exception

생성자에서 throws절로 발생할 수 있는 예외 상황을 규정해 주고 있다. 이 때 주의할 것은 모든 생성자에 throws 절로 규정해 주어야 한다는 것이다. 왜냐하면 인스턴스 필드 초기화문과 인스턴스 초기화 블록은 어떤 생성자를 사용하더라도 실행되기 때문이다.

* + 1. 익명 클래스의 필드의 초기화와 예외상황

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new Outer().method();

}

}

class Outer {

void method() {

try {

new Object() {

int instanceField = exceptionMaker();

int exceptionMaker() throws Exception {

throw new Exception();

}

};

} catch (Exception e) {

System.out.println("여기서 잡을 수 있다");

}

}

}

익명 클래스는 객체가 단 한번 생성되기 때문에 통제가 가능한 것이다.

1. 문자열
   1. 문자열 리터럴(string literal)

문자열 리터럴은 큰 따옴표 “”를 사용해서 나타낸다.

줄바꿈을 나타내고 싶으면 “\n”등을 사용해야 한다. 즉 다음과 같이 할 수 없다.

public class Example {

public static void main(String[] args) {

String str = "first line

second line";

}

}

컴파일타임 오류가 날 것이다.

문자열이 너무 길 때에는 + 연산자를 사용하면 될 것이다.

* 1. 문자열 연결 연산자 +
     1. 숫자를 문자열(string)로 변환하는 방법

숫자(정수형, 실수형)가 String으로 항상 자동변환되지는 않는다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

str = 123; // Error.

method(123); // Error.

}

public static void method(String str) {

}

}

컴파일이 되지 않을 것이다.

이럴 때는 포장(wrapper) 클래스를 사용하는 방법이 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

str = Integer.toString(123);

System.out.println(str);

Integer oi = new Integer(32221);

method(oi.toString());

}

public static void method(String str) {

System.out.println(str);

}

}

결과는 다음과 같을 것이다.

123

32221

또는 String.valueOf 메쏘드를 사용할 수도 있다.

public class Example {

public static void main(String[] args) {

String str = String.valueOf(true);

System.out.println(str);

str = String.valueOf(12345);

System.out.println(str);

str = String.valueOf(new Object());

System.out.println(str);

}

}

결과는 다음과 비슷할 것이다.

true

12345

java.lang.Object@75da06

또는 좀더 편하게(그러나 약간은 편법이라는 느낌을 준다) 다음과 같이 사용할 수도 있다.

public class Example {

public static void main(String[] args) {

String str;

str = "" + 1234;

System.out.println(str);

method("" + 4312);

}

public static void method(String str) {

System.out.println(str);

}

}

결과는 다음과 같을 것이다.

1234

4312

* + 1. + 연산자

+ 연산자의 operand 둘 중에 하나 이상이 String 객체라면 + 연산자는 문자열 연결(string concatination) 연산자가 된다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

str = new String("str") + "ing";

System.out.println(str);

str = "string" + 123;

System.out.println(str);

str = 1234 + "string";

System.out.println(str);

str = "string" + new Object();

System.out.println(str);

boolean b = false;

str = "aoeu" + b;

System.out.println(str);

Object o = null;

str = "aoeuidh" + o;

System.out.println(str);

}

}

결과는 다음과 같을 것이다.

string

string123

1234string

stringjava.lang.Object@75da06

aoeufalse

aoeuidhnull

* + 1. + 연산자의 연산 순서

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("1 + 1 = " + 1 + 1);

System.out.println("1 + 1 = " + (1 + 1));

System.out.println(1 + 1 + " = 1 + 1");

}

}

결과는 다음과 같을 것이다.

1 + 1 = 11

1 + 1 = 2

2 = 1 + 1

숫자의 더하기 연산자인 +와 문자열 연결 연산자인 +가 섞여 있어도 왼쪽부터 차례대로 연산된다. 예를 들어

“1 + 1 = ” + 1 + 1

에서 먼저 앞쪽의

“1 + 1 = ” + 1

이 연산되어

“1 + 1 = 1” + 1

이 된다. 그러므로 원하는 값을 얻기 위해서 ()를 사용해서 숫자가 먼저 계산되게 하였다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("3 \* 3 = " + 3 \* 3);

}

}

결과는 예상했던 대로 다음과 같을 것이다.

3 \* 3 = 9

이번에는 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("3 - 3 = " + 3 - 3);

}

}

아예 컴파일이 안된다. 그 이유를 알아보자. 위의 곱하기 \* 예제에서는 \*가 +보다 우선 순위가 높기 때문에 곱하기가 먼저 되었다. 그러나 빼기는 –는 더하기 + 와 우선순위가 같기 때문에 먼저 더하기가 연산된다. 더하기가 먼저 연산되면 다음과 같은 상태가 된다.

“3 – 3 = 3” – 3

그러나 문자열에는 빼기 – 연산자를 적용할 수 없다. 그러므로 컴파일타임 오류가 나는 것이다. 원하는 값을 얻으려면 괄호를 사용해야 할 것이다.

* + 1. 객체와 + 연산자

문자열 연결 연산자 +는 숫자뿐만 아니라 객체에도 사용할 수 있다. 그 때에는 객체의 toString() 메쏘드가 실행된다. 정말 실행되는지 실험을 해 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

String str = "Who are you? " + a;

System.out.println(str);

System.out.println();

a = null;

str = "Who are you? " + a;

System.out.println(str);

}

}

class A {

public String toString() {

System.out.println("A.toString()");

return "I'm an Object of Type A";

}

}

결과는 다음과 같을 것이다.

A.toString()

Who are you? I'm an Object of Type A

Who are you? null

* 1. String 클래스
     1. 생성자

String 객체를 생성하는 방법은 여러가지가 있다. 먼저 문자열 리터럴을 쓰면 객체가 생성된다. 그리고 문자열 연결 연산자 +를 사용해도 객체가 생성된다. += 연산자도 마찬가지이다. 그리고 객체의 toString 메쏘드를 불러도 객체가 생성된다. 실제로 코드에는 toString이 없어도 많은 경우에 toString 메쏘드가 불려진다. 그리고 또 사용할 수 있는 방법이 명시적으로 생성자를 사용하는 것이다.

다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

byte[] bytes = new byte[10];

char[] chars = new char[10];

for (int i = 0; i < 10; i++) {

bytes[i] = (byte)('a' + i);

chars[i] = (char)('a' + i);

}

int offset = 3;

int length = 5;

int count = 3;

String str;

str = new String();

System.out.println("empty : " + str);

str = new String(bytes);

System.out.println("bytes : " + str);

str = new String(bytes, offset, length);

System.out.println("bytes, offset, length : " + str);

try {

str = new String(bytes, offset, length, "US-ASCII");

} catch (UnsupportedEncodingException e) {

System.out.println(e);

}

System.out.println("bytes, offset, length, US-ASCII : " + str);

try {

str = new String(bytes, "US-ASCII");

} catch (UnsupportedEncodingException e) {

System.out.println(e);

}

System.out.println("bytes, US-ASCII : " + str);

str = new String(chars);

System.out.println("chars : " + str);

str = new String(chars, offset, count);

System.out.println("chars, offset, count : " + str);

str = new String(str);

System.out.println("Copy Constructor : " + str);

str = new String(new StringBuffer(str));

System.out.println("StringBuffer : " + str);

}

}

결과는 다음과 같을 것이다.

empty :

bytes : abcdefghij

bytes, offset, length : defgh

bytes, offset, length, US-ASCII : defgh

bytes, US-ASCII : abcdefghij

chars : abcdefghij

chars, offset, count : def

Copy Constructor : def

StringBuffer : def

그 외에도 deprecated된 생성자가 있다. 다음이 그것들이다.

String(byte[] ascii, int hibyte)

String(byte[] ascii, int hibyte, int offset, int count)

자바에서는 2바이트 유니코드를 지원한다(4바이트 유니코드가 쓰이게 되면 자바가 어떻게 거기에 적응할까? 4바이트 유니코드로의 이행은 언제일지는 몰라도 필연적이다. 6만여 문자는 지구상의 문자를 제대로 나타내기에는 너무나 부족하다. 특히 한글과 중국어 같은 문자 때문에). 그것을 지원하기 위해 기본형인 char형도 있다. 그런데 아직도 여전히 1바이트로 문자를 나타내는 코드가 쓰이고 있다. 그리고 문자열을 나타내는 방식이 여러가지이다. 이것을 지원하기 위한 생성자가 다음과 같은 것들이다.

String(byte[] bytes)

String(byte[] bytes, String charsetName)

String(byte[] bytes, int offset, int length)

String(byte[] bytes, int offset, int length, String charsetName)

charsetName을 지정하는 것과 지정하지 않는 것이 있다. 지정하지 않으면 디폴트를 사용한다. 예제를 보자. 이 예제를 실행하려면 1.4버전이 필요하다.

import java.io.\*;

import java.nio.charset.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

String str;

byte[] bytes = new byte[256];

for (int i = 0; i < 256; i++)

bytes[i] = (byte)i;

str = new String(bytes);

System.out.println("default : " + str);

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

SortedMap sm = Charset.availableCharsets();

Collection values = sm.values();

for (Iterator i = values.iterator(); i.hasNext(); ) {

Charset chs = (Charset)i.next();

String strCharset = chs.name();

try {

str = new String(bytes, strCharset);

} catch (UnsupportedEncodingException e) {

System.out.println(e);

}

System.out.println(strCharset + " : " + str);

}

}

}

결과는 실제로 실행해서 확인해 보기 바란다. 운영체제에 따라 다른 결과가 나올 것이다.

* + 1. String 객체는 변할 수 없다(immutable)

String객체는 변할 수 없는 객체다. 만약 변할 수 있는 문자열이 필요하다면 StringBuffer를 써야 한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str = "1234";

str += "ABCD";

System.out.println(str);

}

}

결과는 다음과 같을 것이다.

1234ABCD

위의 예제에서 먼저 문자열 “1234” 을 저장하는 String 객체가 만들어지고 그 참조가 str 변수에 저장된다. 그 다음에 문자열 “1234ABCD”을 저장하는 String 객체가 만들어지고 그 참조가 str 변수에 저장된다.

그러므로 다음과 같은 식으로 프로그래밍한다면 객체를 계속해서 만들어 내야 하므로 속도에 상당한 영향을 미치게 된다.

public class Example {

public static void main(String[] args) {

String str = "";

for (int i = 0; i < 10; i++) {

str += i;

System.out.println(str);

}

}

}

결과는 다음과 같을 것이다.

0

01

012

0123

01234

012345

0123456

01234567

012345678

0123456789

* + 1. ==와 equals

==와 equals의 일반적인 차이에 대해서 알고 싶으면 Object 클래스를 다룬 부분을 보라.

초보자들은 다음과 같은 예제에서 흔히 ==를 사용한다. 대부분의 경우 잘 실행되긴 하지만 좋은 방법이 아니다. 예제를 보자.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

public Example() {

setSize(300, 300);

setLayout(new FlowLayout());

Button b = new Button("CreateOverlord");

b.setActionCommand("CreateOverlord");

b.addActionListener(this);

add(b);

}

public void actionPerformed(ActionEvent e) {

if (e.getActionCommand() == "CreateOverlord") { // == 사용

System.out.println("Not enough mineral !!!");

}

}

public static void main(String[] args) {

new Example().show();

}

}

버튼을 누르면 콘솔에 다음과 같이 뿌려질 것이다.

Not enough mineral !!!

예제를 다음과 같이 고쳤다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private String strCreate = "Create";

private String strOverlord = "Overlord";

public Example() {

setSize(300, 300);

setLayout(new FlowLayout());

Button b = new Button("CreateOverlord");

b.setActionCommand(strCreate + strOverlord);

b.addActionListener(this);

add(b);

}

public void actionPerformed(ActionEvent e) {

if (e.getActionCommand() == strCreate + strOverlord) { // == 사용

System.out.println("Not enough mineral !!!");

}

}

public static void main(String[] args) {

new Example().show();

}

}

버튼을 눌러도 반응이 없을 것이다. 만약 strCreate와 strOverlord를 final로 선언했다면 버튼을 누르면 메시지가 보일 것이다. 실제로 그렇게 고쳐서 실행해 보자. 문제는 문자열을 == 로 비교할 때는 컴파일 타임에 문자열의 값을 알 수 있는가에 없는가에 따라 결과가 바뀔 수 있다는 것이다. intern을 다루는 부분도 참조하기 바란다.

이번에는 equals 메쏘드를 사용했다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private String strCreate = "Create";

private String strOverlord = "Overlord";

public Example() {

setSize(300, 300);

setLayout(new FlowLayout());

Button b = new Button("CreateOverlord");

b.setActionCommand(strCreate + strOverlord);

b.addActionListener(this);

add(b);

}

public void actionPerformed(ActionEvent e) {

String actionCommand = e.getActionCommand();

if (actionCommand.equals(strCreate + strOverlord)) { // equals 사용

System.out.println("Not enough mineral !!!");

}

}

public static void main(String[] args) {

new Example().show();

}

}

버튼을 누르면 다시 메시지가 보일 것이다.

이번에는 비교 속도에 초점을 맞춘 예제를 보여준다.

public class Example {

public static void main(String[] args) {

String strA = "";

for (int i = 0; i < 10; i++)

strA += i;

String strB = strA;

System.out.println("strA.length() = " + strA.length());

long preTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

if (strA == strB)

;

System.out.println("== : " + (System.currentTimeMillis()-preTime));

preTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

if (strA.equals(strB))

;

System.out.println("equals : " + (System.currentTimeMillis()-preTime));

strB = "";

for (int i = 0; i < 10; i++)

strB += i;

preTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

if (strA == strB)

;

System.out.println("== : " + (System.currentTimeMillis()-preTime));

preTime = System.currentTimeMillis();

for (long l = 0; l < 10000000; l++)

if (strA.equals(strB))

;

System.out.println("equals : " + (System.currentTimeMillis()-preTime));

}

}

결과는 다음과 비슷할 것이다.

strA.length() = 10

== : 220

equals : 600

== : 160

equals : 4010

220과 160이 서로 다른 것은 우연적인 상황 때문이다. 하여튼 ==로 비교하는 것이 빠르다. 600과 4010이 모두 equals를 사용하는데도 커다란 차이가 난다. 이것은 String.equals 메쏘드의 소스 코드를 보면 이해가 갈 것이다. 600이 나왔을 때는 두 변수 strA, strB가 같은 객체를 가리킨다. 다음은 String.java에서 인용한 것이다.

...

public final class String implements java.io.Serializable, Comparable, CharSequence {

...

public boolean equals(Object anObject) {

if (this == anObject) {

return true;

}

if (!(anObject instanceof String)) {

return false;

}

String anotherString = (String)anObject;

return count == anotherString.count && compareArraysToCount(value, offset, anotherString.value, anotherString.offset);

}

...

}

* + 1. intern 메쏘드

다음 예제를 보자. 잘 이해가 가지 않으면 Object 클래스를 다룬 장에서 ==와 equals의 차이를 다룬 부분을 참조하라.

public class Example {

public static void main(String[] args) {

String strA = "string";

String strB = "string";

System.out.println(strA == strB);

strB = "str" + "ing";

System.out.println(strA == strB);

strB = "str";

strB += "ing";

System.out.println(strA == strB);

}

}

결과는 다음과 같을 것이다.

true

true

false

String 객체를 변할 수 없다(immutable). 컴파일러는 컴파일 타임에 두 개의 문자열이 똑같다(equals)는 것을 알게 되면 String 객체를 두 개를 만들지 않고 하나만 만든다. 문자열이 변한다면 문제가 생기겠지만 문자열이 변할 수 없기 때문에 이렇게 해도 문제가 생기지 않는다. 이렇게 하면 메모리를 절약할 수 있다. 그래서 위의 예제에서 두 번의 true가 결과로 나온 것이다. 마지막에도 strB변수가 가리키는 객체가 다른 것과 같은 “string”이지만 이것은 컴파일 타임에 결정되는 것이 아니다(영리한 컴파일러가 이것을 알지라도). 그래서 identity를 체크하는 테스트에서 false가 나온 것이다.

intern 메쏘드는 만약 똑 같은 즉 equals 메쏘드를 호출했을 때 true를 리턴하는 객체가 이미 있다면 그 객체에 연결해 준다. 다음 예제를 보면 무슨 뜻인지 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

String strA = "string";

String strB = "str";

strB += "ing";

System.out.println(strA == strB);

strB = strB.intern();

System.out.println(strA == strB);

}

}

결과는 다음과 같을 것이다.

false

true

여기서 주의할 것은

strB = strB.intern();

이라고 한 것이다. 만약 strB에 다시 대입하지 않고 그냥

strB.intern();

이라고 했다면 결과는 다르게 나왔을 것이다.

==를 이용해서 비교하는 것이 equals를 이용해서 비교하는 것보다 훨씬 빠르기 때문에 intern을 이용해서 프로그램의 속도를 빠르게 할 수 있다. 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList(10000);

Random r = new Random();

for (int i = 0; i < 10000; i++) {

String str = "string intern example - ";

for (int j = 0; j < 3; j++) {

str += (char)(Math.abs(r.nextInt() % 10) + '0');

}

al.add(str);

}

String strFind = "string intern example - 123";

int count = 0;

long preTime = System.currentTimeMillis();

for (int i = 0; i < 1000; i++) // 컴퓨터의 성능에 따라 반복 횟수를 조정하라.

for (Iterator it = al.iterator(); it.hasNext(); ) {

String str = (String)it.next();

if (str.equals(strFind)) // equals 로 비교.

count++;

}

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

System.out.println("time = " + (System.currentTimeMillis() - preTime));

}

}

결과는 다음과 비슷할 것이다.

count = 7000

time = 15380

이번에는 ==를 이용해서 비교한다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList(10000);

Random r = new Random();

for (int i = 0; i < 10000; i++) {

String str = "string intern example - ";

for (int j = 0; j < 3; j++) {

str += (char)(Math.abs(r.nextInt() % 10) + '0');

}

al.add(str);

}

String strFind = "string intern example - 123";

int count = 0;

long preTime = System.currentTimeMillis();

for (int i = 0; i < 1000; i++)

for (Iterator it = al.iterator(); it.hasNext(); ) {

String str = (String)it.next();

if (str == strFind) // 여기를 바꿨다.

count++;

}

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

System.out.println("time = " + (System.currentTimeMillis() - preTime));

}

}

결과는 다음과 비슷할 것이다.

count = 0

time = 3510

시간은 절약이 됐는데 결과가 엉망이다. 다음 예제는 intern을 사용한다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList(10000);

Random r = new Random();

for (int i = 0; i < 10000; i++) {

String str = "string intern example - ";

for (int j = 0; j < 3; j++) {

str += (char)(Math.abs(r.nextInt() % 10) + '0');

}

str = str.intern(); // 여기에 주목하라.

al.add(str);

}

String strFind = "string intern example - 123"; // 리터럴이기 때문에 intern 할 필요 없다.

int count = 0;

long preTime = System.currentTimeMillis();

for (int i = 0; i < 1000; i++)

for (Iterator it = al.iterator(); it.hasNext(); ) {

String str = (String)it.next();

if (str == strFind)

count++;

}

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

System.out.println("time = " + (System.currentTimeMillis() - preTime));

}

}

결과는 다음과 비슷할 것이다.

count = 8000

time = 2260

* 1. StringBuffer 클래스
     1. StringBuffer 객체는 변할 수 있다.

String 객체와 달리 StringBuffer 객체를 변할 수 있다. 사실 이것이 StringBuffer 클래스가 만들어진 가장 큰 이유일 것이다.

public class Example {

public static void main(String[] args) {

long preTime = System.currentTimeMillis();

String str = "";

for (int i = 0; i < 34567; i++) {

str += '0'; // 여기서 객체가 계속 만들어진다.

}

System.out.println("Time(String) : " + (System.currentTimeMillis() - preTime));

preTime = System.currentTimeMillis();

StringBuffer sb = new StringBuffer(100000);

for (int i = 0; i < 34567; i++) {

sb.append('0');

}

System.out.println("Time(StringBuffer) : " + (System.currentTimeMillis() - preTime));

}

}

결과는 다음과 비슷할 것이다.

Time(String) : 24280

Time(StringBuffer) : 0

놀라운 차이다. StringBuffer로 했을 때는 1 milli초도 안 걸리는 걸 String으로 하니까 24초나 걸렸다.

1. 리플렉션
   1. 리플렉션이란
      1. 리플렉션의 의미

리플렉션(reflection)은 introspection이라고 한다. 둘 다 반성 또는 내성(내적 성찰)이란 뜻이 있다. 반성 또는 내성이란 자신에 대해 남이 아니라 바로 자기 자신이 생각하는 것이다. 보통 프로그래밍을 할 때는 어떤 타입의 멤버 또는 수퍼 타입에 어떤 것이 있는지 알아보기 위해서는 소스 코드를 본다. 또는 소스 코드에 대한 문서를 본다. 그리고 필요한 멤버를 사용하는 것이다. 그러나 그럴 수가 없는 경우가 있다. 우리가 만는 프로그램에서 사용할 타입을 모를 수도 있는 것이다. 그 좋은 예가 디컴파일러(decompiler)인데 디컴파일러 프로그램 자체가 컴파일될 때는 장차 그 프로그램(디컴파일러)이 실행될 때 자신이 디컴파일할 클래스들을 전혀 모르는 것이다. 그 때에는 그 정보를 동적으로 알아내는 수밖에 없다. 그 때 사용하는 것이 리플렉션이다. 만약 우리가 클래스 A를 사용해야 하는데 그 클래스에 대해 잘 모른다고 생각해 보자. 클래스 A의 입장에서는 우리가 남이다. 우리는 클래스 A에게 내성(introspection)하게 하고 그 결과를 우리에게 알려달라고 부탁하는 것이다. 즉 클래스 A가 자기 자신에 대해 성찰하는 것이다.

* + 1. 리플렉션 클래스들

리플렉션 클래스 또는 인터페이스들은 거의 모두 java.lang.reflect 패키지에 있다. 예외는 Class, Package 클래스이다. 이 두 클래스는 java.lang 패키지에 있다.

리플렉션에서 가장 중요하고 기본이 되는 클래스는 Class 클래스이다. Class 클래스는 어떤 클래스 또는 인터페이스에 대한 정보를 가지고 있다. 거기다가 배열과 기본형을 나타낼 수도 있다. 심지어 void를 나타내는 Class 객체도 있다. Type이라는 이름이 더 적당한 것 같으나 무슨 이유인지 Class라고 이름 붙였다.

Package클래스는 패키지를 나타낸다.

Modifier 클래스는 변경자와 관련이 있다.

Member 인터페이스는 클래스나 인터페이스의 멤버들이 공유하는 것을 나타내는 인터페이스이다. Method, Field, Constructor가 이 인터페이스를 구현한다. 여기서 Constructor는 멤버가 아니라는 것에 주의하라. 그리고 멤버 클래스와 멤버 인터페이스가 있지만 Class 클래스는 Member 인터페이스를 구현하지 않는다.

AccessibleObject는 Method, Field, Constructor의 수퍼 클래스이다. 이것에 대해서는 접근 제한을 다루는 절을 참조하라.

Proxy 클래스는 조금 어려운 클래스이다. 그것을 다루는 부분을 참조하라. 여기서 InvocationHandler인터페이스도 같이 다룬다.

ReflectPermission도 접근 제한과 관련이 있는데 이 책에서는 다루지 않는다.

* + 1. 리플렉션은 어디에 사용되나

리플렉션은 보통 프로그램에서는 거의 쓰이지 않는다. 그러나 리플렉션이 꼭 필요한 곳이 있다. 먼저 직력화(serialization)에서 쓰인다. 직렬화를 다루는 장을 참조하라. JavaBeans에서도 쓰인다. 비주얼 베이직이나 델파이를 사용해 본 사람은 버튼, 텍스트 박스 등을 드래그해서 프로그램에 붙여 쓸 수 있다는 것을 알 것이다. 자바에서 이와 비슷한 것을 구현하기 위해 필요한 것이 자바 빈즈이다. 이 것에 대해서는 이 책에서 다루지 않는다. RMI에서도 리플렉션이 쓰인다. RMI를 이용한 프로그래밍에서 쓰이는 것이 아니라 RMI가 작동할 수 있도록 하는데 필요하다. 이 책에서는 이것에 대해서도 다루지 않는다. 디버거(debugger)나 디컴파일러(decompiler)를 만들 때도 리플렉션이 필요할 것이다. 그 외에도 여러가지 특수한 상황에 리플렉션이 사용될 수 있다.

리플렉션을 사용하면 코드를 알아보기 힘들어진다. 리플렉션은 위에서의 예처럼 불가피할 때만 사용하자. 쉬운 방법이 있을 때는 그냥 그 쉬운 방법을 쓰면 된다.

* 1. instanceof 연산자
     1. instanceof 연산자란?

instanceof 연산자는 리플렉션과 밀접한 관련이 있기 때문에 이 장에서 같이 다룬다. instanceof 연산자의 의미를 살펴보자. 글자 그대로 어떤 객체가 어떤 타입의 인스턴스인가를 알아보는 것이다. 연산값은 boolean 형이다. 연산자의 왼쪽에서 보통 어떤 객체를 가리키는 참조형 변수가 오고 오른쪽에는 타입이 온다.

* + 1. instanceof와 null

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str = new String();

boolean strIsString = str instanceof String;

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

str = null;

strIsString = str instanceof String;

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

}

}

결과는 다음과 같다.

strIsString = true

strIsString = false

위의 예제에서 str 변수가 String 형이라는 것은 컴파일타임에 알 수 있지만 실제로 String 형 객체를 가리키는지 null인지는 런타임에나 알 수 있다. instanceof 연산자는 실제로 str 변수가 무엇을 기리키고 있는지를 런타임에 알 수 있게 한다. 위의 예제에서는 str 이 무엇을 가리키고 있는지는 컴파일타임에도 알 수 있다고 생각하는 사람도 있을 것이다. 실제로 영리한 컴파일러라면 그것을 알 수도 있을 것이다. 그러나 그럴 수 없는 경우도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

String str;

// 난수에 의해 결정된다.

if ( (new java.util.Random().nextInt()%2) == 0 ) {

str = null;

} else {

str = new String();

}

boolean strIsString = str instanceof String;

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

}

}

결과는 다음과 같을지도 모른다.

strIsString = false

여러 번 실행해 보자. 컴파일타임에 알 수 없다는 것이 이해가 갈 것이다.

단지 null인지 아닌지를 알고 싶으면 다음과 같은 코드가 더 나을 것이다.

if (str != null) {

…

}

* + 1. 컴파일타임 오류

instanceof 연산자를 아무데나 사용할 수 있는 것은 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

boolean b;

b = (new String() instanceof String); // OK. 변수를 안 써도 된다.

b = (3 instanceof int); // 에러. 기본형에는 사용할 수 없다.

b = (new int[3] instanceof int[]); // OK. 배열에도 사용할 수 있다.

}

}

기본형에는 사용할 수 없다는 것을 알 수 있다.

다음 예제를 보자.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

Object obj;

Component comp;

boolean b;

b = (obj instanceof Object); // Error. obj 가 초기화되지 않았다.

obj = new Object();

b = (obj instanceof Object); // OK.

b = (obj instanceof Component); // OK. obj 에는 Component 객체가 올 수도 있다.

comp = new Panel();

b = (comp instanceof Object); // OK.

b = (comp instanceof Component); // OK.

b = (comp instanceof Frame); // OK. comp 에는 Frame 객체가 올 수도 있다.

b = (comp instanceof String); // Error. comp 에 String 객체를 대입할 수는 절대 없다.

}

}

두 군데서 컴파일타임 오류가 난다.

* + 1. instanceof와 상속

instanceof가 필요한 이유가 무엇일까? 위에서처럼 단지 null인지 아닌지 알아보기 위해 굳이 연산자 하나를 만들지는 않았을 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Shape[] shapeArray = new Shape[3];

shapeArray[0] = new Triangle();

shapeArray[1] = new Rectangle();

shapeArray[2] = new Circle();

for (int i = 0; i < shapeArray.length; i++) {

if (shapeArray[i] instanceof Triangle)

((Triangle)shapeArray[i]).drawTriangle();

if (shapeArray[i] instanceof Rectangle)

((Rectangle)shapeArray[i]).drawRectangle();

if (shapeArray[i] instanceof Circle)

((Circle)shapeArray[i]).drawCircle();

}

}

}

class Shape {

}

class Triangle extends Shape {

public void drawTriangle() {

System.out.println("화면에 삼각형이 그려진다고 상상하자.");

}

}

class Rectangle extends Shape {

public void drawRectangle() {

System.out.println("상상이 안되면 매직으로 화면에 직사각형을 그려보자.");

}

}

class Circle extends Shape {

public void drawCircle() {

System.out.println("그게 싫으면 이 예제를 Swing을 사용하는 프로그램으로 바꿔서 원을 그려보자.");

}

}

결과는 다음과 같을 것이다.

화면에 삼각형이 그려진다고 상상하자.

상상이 안되면 매직으로 화면에 직사각형을 그려보자.

그게 싫으면 이 예제를 Swing을 사용하는 프로그램으로 바꿔서 원을 그려보자.

그러나 더 나은 방법은 재정의 또는 추상 메쏘드의 구현을 사용하는 것이다. 위의 예제는 다음과 같이 하는 것이 더 낫다.

public class Example {

public static void main(String[] args) {

Shape[] shapeArray = new Shape[3];

shapeArray[0] = new Triangle();

shapeArray[1] = new Rectangle();

shapeArray[2] = new Circle();

for (int i = 0; i < shapeArray.length; i++)

shapeArray[i].draw();

}

}

interface Shape {

void draw();

}

class Triangle implements Shape {

public void draw() {

System.out.println("화면에 삼각형이 그려진다고 상상하자.");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("상상이 안되면 매직으로 화면에 직사각형을 그려보자.");

}

}

class Circle implements Shape {

public void draw() {

System.out.println("그게 싫으면 이 예제를 Swing을 사용하는 프로그램으로 바꿔서 원을 그려보자.");

}

}

* + 1. instanceof의 그럴듯한 사용 예

다음 예제를 보자. 이 예제에서 Example 클래스를 제외한 타입들은 당신이 만들었고 Example 클래스는 누군가 다른 사람이 나중에 만들었다고 상상하라. 패키지도 달라져야 하겠지만 편의상 디폴트 패키지(unnamed package)에 넣었다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

ArrayList al = new ArrayList();

al.add(new Triangle());

al.add(new Rectangle());

al.add(new Circle());

al.add(new String()); // 실수.

Shapes.drawShapes(al);

}

}

class Shapes {

public static void drawShapes(ArrayList shapes) {

for (Iterator i = shapes.iterator(); i.hasNext(); ) {

Object obj = i.next();

if (obj instanceof Shape) {

Shape sh = (Shape)obj;

sh.draw();

} else {

throw new IllegalArgumentException("Shape 넣으라고 했잖아.");

}

}

}

}

interface Shape {

void draw();

}

class Triangle implements Shape {

public void draw() {

System.out.println("화면에 삼각형이 그려진다고 상상하자.");

}

}

class Rectangle implements Shape {

public void draw() {

System.out.println("상상이 안되면 매직으로 화면에 직사각형을 그려보자.");

}

}

class Circle implements Shape {

public void draw() {

System.out.println("그게 싫으면 이 예제를 Swing을 사용하는 프로그램으로 바꿔서 원을 그려보자.");

}

}

결과는 다음과 비슷할 것이다.

화면에 삼각형이 그려진다고 상상하자.

상상이 안되면 매직으로 화면에 직사각형을 그려보자.

그게 싫으면 이 예제를 Swing을 사용하는 프로그램으로 바꿔서 원을 그려보자.

java.lang.IllegalArgumentException: Shape 넣으라고 했잖아.

at Shapes.drawShapes(Example.java:24)

at Example.main(Example.java:12)

Exception in thread "main"

drawShapes 메쏘드는 ArrayList를 받는다. ArrayList에는 Object 형으로 저장되기 때문에 어떤 객체도 저장될 수 있다. 그러므로 Shape라는 것을 보장하기 위해서는 instanceof를 이용해야 하는 것이다.

* 1. class 리터럴

class 리터럴에 대해 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Class cls;

cls = boolean.class;

System.out.println(cls);

cls = int[].class;

System.out.println(cls);

cls = String.class;

System.out.println(cls);

cls = String[][].class;

System.out.println(cls);

cls = Cloneable.class;

System.out.println(cls);

cls = Cloneable[].class;

System.out.println(cls);

}

}

결과는 다음과 같을 것이다.

boolean

class [I

class java.lang.String

class [[Ljava.lang.String;

interface java.lang.Cloneable

class [Ljava.lang.Cloneable;

class 리터럴은 기본형, 배열형, 클래스형, 인터페이스형에 대한 Class 객체를 나타낸다.

class 리터럴은 타입 이름에만 쓸 수 있다. 변수명 등에는 쓸 수 없다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Class cls;

boolean b = true;

cls = b.class; // Error

cls = 3.class; // Error

String str = new String();

cls = str.class; // Error

cls = new String().class; // Error

}

}

* 1. Class 클래스
     1. Class 객체 얻기

Class 객체를 얻는 방법에는 여러가지가 있다. 그 중 하나는 위에서 본 class 리터럴을 이용하는 것이다. 또 다른 방법은 Class.forName 메쏘드를 사용하는 것이다. Object.getClass 메쏘드를 사용할 수도 있다. 또 하나는 이미 얻은 Class 객체에서 수퍼 타입이나 멤버 타입을 얻을 수도 있다.

그리고 하나의 클래스에 속하는 여러 객체가 생성되어도 하나의 Class 객체만 존재한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Class clsLiteral, clsForName, clsGetClassA, clsGetClassB, clsGetClassC;

String strB, strC;

clsLiteral = String.class;

clsForName = null;

try {

clsForName = Class.forName("java.lang.String");

} catch (ClassNotFoundException e) {

System.out.println(e);

}

clsGetClassA = new String().getClass();

strB = new String("B");

strC = new String("C");

clsGetClassB = strB.getClass();

clsGetClassC = strC.getClass();

System.out.println(clsLiteral == clsForName);

System.out.println(clsLiteral == clsGetClassA);

System.out.println(clsLiteral == clsGetClassB);

System.out.println(clsLiteral == clsGetClassC);

}

}

결과는 다음과 같을 것이다.

true

true

true

true

class.forName 메쏘드에서는 fully qualified name을 사용해야 한다. 그러므로 다음과 같이 하면 예외상황이 발생한다.

public class Example {

public static void main(String[] args) {

try {

Class.forName("String");

} catch (ClassNotFoundException e) {

System.out.println(e);

}

}

}

결과는 다음과 비슷할 것이다.

java.lang.ClassNotFoundException: String

Object.getClass 메쏘드는 런타임 객체에 해당하는 Class 객체를 리턴한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Object obj;

obj = null;

try {

System.out.println(obj.getClass());

} catch (Exception e) {

System.out.println(e);

}

obj = new Object();

System.out.println(obj.getClass());

obj = new String();

System.out.println(obj.getClass());

obj = new int[3];

System.out.println(obj.getClass());

System.out.println(new Integer(3).getClass());

}

}

결과는 다음과 비슷할 것이다.

java.lang.NullPointerException

class java.lang.Object

class java.lang.String

class [I

class java.lang.Integer

* + 1. 타입에 대해 알아보기

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

try {

Class clsMember = Class.forName("java.lang.reflect.Member");

showClassInformation(clsMember);

Class clsClass = clsMember.getClass();

showClassInformation(clsClass);

Class clsVoid = void.class;

showClassInformation(clsVoid);

Class clsIntArray = int[].class;

showClassInformation(clsIntArray);

} catch (ClassNotFoundException e) {

System.out.println(e);

}

}

public static void showClassInformation(Class cls) {

System.out.println(cls.getName() + "은 인터페이스인가 : " + cls.isInterface());

System.out.println(cls.getName() + "은 기본형인가 : " + cls.isPrimitive());

System.out.println(cls.getName() + "은 배열인가 : " + cls.isArray());

System.out.println(cls.getName() + "은 기본형인가 : " + cls.isPrimitive());

System.out.println(cls.getName() + "은 기본형인가 : " + cls.isPrimitive());

System.out.println(cls.getName() + "은 기본형인가 : " + cls.isPrimitive());

}

}

결과는 다음과 같을 것이다.

java.lang.reflect.Member은 인터페이스인가 : true

java.lang.reflect.Member은 기본형인가 : false

java.lang.reflect.Member은 배열인가 : false

java.lang.reflect.Member은 기본형인가 : false

java.lang.reflect.Member은 기본형인가 : false

java.lang.reflect.Member은 기본형인가 : false

java.lang.Class은 인터페이스인가 : false

java.lang.Class은 기본형인가 : false

java.lang.Class은 배열인가 : false

java.lang.Class은 기본형인가 : false

java.lang.Class은 기본형인가 : false

java.lang.Class은 기본형인가 : false

void은 인터페이스인가 : false

void은 기본형인가 : true

void은 배열인가 : false

void은 기본형인가 : true

void은 기본형인가 : true

void은 기본형인가 : true

[I은 인터페이스인가 : false

[I은 기본형인가 : false

[I은 배열인가 : true

[I은 기본형인가 : false

[I은 기본형인가 : false

[I은 기본형인가 : false

이번에는 isAssignableFrom 메쏘드에 대해 알아보자. 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class clsObject = Object.class;

Class clsMember = Member.class;

Class clsAccessibleObject = AccessibleObject.class;

Class clsField = Field.class;

Class clsString = String.class;

System.out.println("Object 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : " + clsAccessibleObject.isAssignableFrom(clsObject));

System.out.println("Member 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : " + clsAccessibleObject.isAssignableFrom(clsMember));

System.out.println("AccessibleObject 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : " + clsAccessibleObject.isAssignableFrom(clsAccessibleObject));

System.out.println("Field 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : " + clsAccessibleObject.isAssignableFrom(clsField));

System.out.println("String 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : " + clsAccessibleObject.isAssignableFrom(clsString));

System.out.println();

System.out.println("Object 형의 객체는 Member 형 변수에 대입할 수 있나 : " + clsMember.isAssignableFrom(clsObject));

System.out.println("Member 형의 객체는 Member 형 변수에 대입할 수 있나 : " + clsMember.isAssignableFrom(clsMember));

System.out.println("AccessibleObject 형의 객체는 Member 형 변수에 대입할 수 있나 : " + clsMember.isAssignableFrom(clsAccessibleObject));

System.out.println("Field 형의 객체는 Member 형 변수에 대입할 수 있나 : " + clsMember.isAssignableFrom(clsField));

System.out.println("String 형의 객체는 Member 형 변수에 대입할 수 있나 : " + clsMember.isAssignableFrom(clsString));

}

}

결과는 다음과 같다.

Object 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : false

Member 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : false

AccessibleObject 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : true

Field 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : true

String 형의 객체는 AccessibleObject 형 변수에 대입할 수 있나 : false

Object 형의 객체는 Member 형 변수에 대입할 수 있나 : false

Member 형의 객체는 Member 형 변수에 대입할 수 있나 : true

AccessibleObject 형의 객체는 Member 형 변수에 대입할 수 있나 : false

Field 형의 객체는 Member 형 변수에 대입할 수 있나 : true

String 형의 객체는 Member 형 변수에 대입할 수 있나 : false

이번에는 inInstance 메쏘드에 대해 알아보자.

import java.awt.\*;

public class Example {

public static void main(String[] args) {

Class clsComponent = Component.class;

System.out.println("Object 형 객체는 Component 형 변수에 대입될 수 있다. : " + clsComponent.isInstance(new Object()));

System.out.println("Panel 형 객체는 Component 형 변수에 대입될 수 있다. : " + clsComponent.isInstance(new Panel()));

System.out.println("String 형 객체는 Component 형 변수에 대입될 수 있다. : " + clsComponent.isInstance(new String()));

}

}

결과는 다음과 같다.

Object 형 객체는 Component 형 변수에 대입될 수 있다. : false

Panel 형 객체는 Component 형 변수에 대입될 수 있다. : true

String 형 객체는 Component 형 변수에 대입될 수 있다. : false

* + 1. 수퍼 타입 얻기

수퍼 타입에는 수퍼 인터페이스와 수퍼 클래스가 있다. Class 클래스에는 두가지를 얻을 수 있는 메쏘드가 있다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class clsField = Field.class;

Class[] superInterfaces = clsField.getInterfaces();

for (int i = 0; i < superInterfaces.length; i++)

System.out.println(superInterfaces[i].getName());

Class superClass = clsField.getSuperclass();

System.out.println(superClass.getName());

}

}

결과는 다음과 같을 것이다.

java.lang.reflect.Member

java.lang.reflect.AccessibleObject

* + 1. 멤버 타입 얻기

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Class clsSubClass = SubClass.class;

Class[] memberTypes = clsSubClass.getClasses();

for (int i = 0; i < memberTypes.length; i++)

System.out.println(memberTypes[i].getName());

System.out.println();

memberTypes = clsSubClass.getDeclaredClasses();

for (int i = 0; i < memberTypes.length; i++)

System.out.println(memberTypes[i].getName());

}

}

class SuperClass {

public interface SuperPublicNestedInterface {}

protected interface SuperProtectedNestedInterface {}

interface SuperPackageNestedInterface {}

private interface SuperPrivateNestedInterface {}

public class SuperPublicNestedClass {}

protected class SuperProtectedNestedClass {}

class SuperPackageNestedClass {}

private class SuperPrivateNestedClass {}

}

class SubClass extends SuperClass {

public interface SubPublicNestedInterface {}

protected interface SubProtectedNestedInterface {}

interface SubPackageNestedInterface {}

private interface SubPrivateNestedInterface {}

public class SubPublicNestedClass {}

protected class SubProtectedNestedClass {}

class SubPackageNestedClass {}

private class SubPrivateNestedClass {}

}

결과는 다음과 같을 것이다.

SubClass$SubPublicNestedClass

SubClass$SubPublicNestedInterface

SuperClass$SuperPublicNestedClass

SuperClass$SuperPublicNestedInterface

SubClass$SubPrivateNestedClass

SubClass$SubPackageNestedClass

SubClass$SubProtectedNestedClass

SubClass$SubPublicNestedClass

SubClass$SubPrivateNestedInterface

SubClass$SubPackageNestedInterface

SubClass$SubProtectedNestedInterface

SubClass$SubPublicNestedInterface

반대로 멤버 타입을 알면 그 멤버를 정의한 외부 타입을 알 수도 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Class clsNestedInterface = OuterClass.NestedInterface.class;

System.out.println(clsNestedInterface);

Class clsOuterClass = clsNestedInterface.getDeclaringClass();

System.out.println(clsOuterClass);

}

}

class OuterClass {

public interface NestedInterface {}

}

결과는 다음과 같을 것이다.

interface OuterClass$NestedInterface

class OuterClass

* 1. Modifier 클래스

Modifier는 타입에도 쓰이고 멤버에도 쓰인다. 그러므로 Class 객체와 Member를 구현한 클래스의 객체와 관련이 된다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

int modifiers = A.class.getModifiers();

System.out.println(Modifier.isAbstract(modifiers));

System.out.println(Modifier.isFinal(modifiers));

System.out.println(Modifier.isPublic(modifiers));

}

}

abstract class A {}

결과는 다음과 같을 것이다.

true

false

false

다음 예제처럼 좀더 지저분하게 테스트할 수도 있다.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

int modifiers = A.class.getModifiers();

System.out.println( (Modifier.ABSTRACT & modifiers) != 0);

System.out.println( (Modifier.FINAL & modifiers) != 0 );

System.out.println( (Modifier.PUBLIC & modifiers) != 0 );

}

}

abstract class A {}

결과는 다음과 같을 것이다.

true

false

false

* 1. Package 클래스

Package 클래스는 패키지에 대한 정보를 담고 있다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Package p = Class.class.getPackage();

System.out.println(p.getImplementationTitle());

System.out.println(p.getImplementationVendor());

System.out.println(p.getImplementationVersion());

System.out.println(p.getName());

System.out.println(p.getSpecificationTitle());

System.out.println(p.getSpecificationVendor());

System.out.println(p.getSpecificationVersion());

System.out.println(p.isSealed());

}

}

결과는 다음과 비슷할 것이다.

Java Runtime Environment

Sun Microsystems, Inc.

1.4.0-beta

java.lang

Java Platform API Specification

Sun Microsystems, Inc.

1.4

false

* 1. Field 클래스
     1. 필드 이름 알아내기

필드 이름 모두 보이기 예제이다.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

A a = new A();

Class cls = a.getClass();

Field[] fields = cls.getDeclaredFields();

for (int i = 0; i < fields.length; i++)

System.out.println(fields[i]);

}

}

class A {

int a;

private int b;

protected byte c;

public Object o;

}

결과는 다음과 같을 것이다.

int A.a

private int A.b

protected byte A.c

public java.lang.Object A.o

* + 1. 필드의 값

이번에는 필드의 내용까지 보이는 예를 만들어보자. 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

A a = new A();

Class cls = a.getClass();

Field[] fields = cls.getDeclaredFields();

for (int i = 0; i < fields.length; i++)

try {

System.out.println(fields[i].getName() + " = " + fields[i].get(a));

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

int a = 2;

private boolean b = true;

protected byte c = -5;

public Object o = new String("qwerty");

}

결과는 다음과 같을 것이다.

a = 2

java.lang.IllegalAccessException

c = -5

o = qwerty

예외 상황이 발생하는 것에 대해서는 이 장의 뒷 부분을 참조하라.

필드가 기본형일 때에는 그 기본형의 포장(wrapper) 클래스를 반환한다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class c = A.class;

Field f = c.getField("primitiveField");

int i = f.getInt(new A());

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

Integer oi = (Integer)f.get(new A());

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

} catch (NoSuchFieldException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

public int primitiveField = 10;

}

결과는 다음과 같을 것이다.

i = 10

oi = 10

포장 클래스에 대해서는 기본형을 다루는 부분을 참조하라.

필드에 값을 대입할 수도 이것은 API Specification을 보고 직접 만들어보기 바란다.

* 1. Method 클래스
     1. 메쏘드의 이름 알아내기

모든 메쏘드 이름 보기 예제이다.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

A a = new A();

Class cls = a.getClass();

Method[] methods = cls.getDeclaredMethods();

for (int i = 0; i < methods.length; i++)

System.out.println(methods[i]);

}

}

class A {

void methodA() {}

private void methodB() {}

protected void methodC() {}

public void methodO() {}

}

결과는 다음과 같을 것이다.

void methodA()

private void A.methodB()

protected void A.methodC()

public void A.methodO()

* + 1. 메쏘드 실행하기

이번에는 메쏘드를 실제로 부르는 예제이다.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

A a = new A();

Object[] parameters = {};

Class cls = a.getClass();

Method[] methods = cls.getDeclaredMethods();

for (int i = 0; i < methods.length; i++)

try {

methods[i].invoke(a,parameters);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

void methodA() {

System.out.println("A.methodA()");

}

private void methodB() {

System.out.println("A.methodB()");

}

protected void methodC() {

System.out.println("A.methodC()");

}

public void methodO() {

System.out.println("A.methodO()");

}

}

결과는 다음과 같을 것이다.

A.methodA()

java.lang.IllegalAccessException

A.methodC()

A.methodO()

여기서도 같은 예외 상황이 발생한다.

이번에는 메쏘드에 매개변수를 넘기고 반환값(return value)을 받아 보자. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class[] parameters = {int.class};

Object[] arguments = {new Integer(123)};

try {

Class c = A.class;

Method m = c.getMethod("doubleValue", parameters);

Integer oi = (Integer)m.invoke(new A(), arguments);

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

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

public int doubleValue(int value) {

return value \* 2;

}

}

결과는 다음과 같을 것이다.

oi = 246

* + 1. InvocationTargetException

메쏘드를 실행하다 보면 예외 상황이 발생할 수도 있다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class[] parameters = {int.class};

Object[] arguments = {new Integer(123)};

try {

Class c = A.class;

Method m = c.getMethod("doubleValue", parameters);

Integer oi = (Integer)m.invoke(new A(), arguments);

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

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

System.out.println(e.getCause());

System.out.println(e.getTargetException());

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

public int doubleValue(int value) {

throw new ArrayIndexOutOfBoundsException();

}

}

결과는 다음과 비슷할 것이다.

java.lang.reflect.InvocationTargetException

java.lang.ArrayIndexOutOfBoundsException

java.lang.ArrayIndexOutOfBoundsException

* 1. Constructor 클래스
     1. 객체 생성하기

리플렉션을 이용해 객체를 생성하는 방법에는 여러 가지가 있다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class c = A.class;

A a = (A)c.newInstance();

System.out.println(a);

Constructor conNoArguments = c.getConstructor(new Class[0]);

a = (A)conNoArguments.newInstance(new Object[0]);

System.out.println(a);

Class[] parameters = {String.class};

Constructor conOneArgument = c.getConstructor(parameters);

Object[] arguments = {"One Argument"};

a = (A)conOneArgument.newInstance(arguments);

System.out.println(a);

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

} catch (InstantiationException e) {

System.out.println(e);

}

}

}

class A {

String field;

public A() {

field = new String("Default");

}

public A(String str) {

field = str;

}

public String toString() {

return field;

}

}

결과는 다음과 같을 것이다.

Default

Default

One Argument

생성자에서도 예외상황이 발생할 수 있다. InvocationTargetException은 메쏘드 호출에서와 마찬가지이다.

* + 1. 중첩 클래스의 객체 생성하기

먼저 정적 중첩 클래스의 객체를 생성해 보자. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class cOuter = OuterClass.class;

Class[] classes = cOuter.getDeclaredClasses();

Class cNested = classes[0];

Class[] parameters = {String.class};

Constructor conOneArgument = cNested.getConstructor(parameters);

Object[] arguments = {"One Argument"};

OuterClass.NestedClass nc = (OuterClass.NestedClass)conOneArgument.newInstance(arguments);

System.out.println(nc);

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

} catch (InstantiationException e) {

System.out.println(e);

}

}

}

class OuterClass {

public static class NestedClass {

private String field;

public NestedClass(String str) {

field = str;

}

public String toString() {

return field;

}

}

}

결과는 다음과 같을 것이다.

One Argument

최상위 클래스와 특별히 다를 것이 없다.

이번에는 내부 클래스(non-static nested class)의 객체를 생성해 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class cOuter = OuterClass.class;

Class[] classes = cOuter.getDeclaredClasses();

Class cNested = classes[0];

Class[] parameters = {OuterClass.class, String.class}; // OuterClass.class 추가됨.

Constructor conOneArgument = cNested.getConstructor(parameters);

Object[] arguments = {new OuterClass("OuterString"), "InnerString"}; // OuterClass의 객체가 추가됨.

OuterClass.NestedClass nc = (OuterClass.NestedClass)conOneArgument.newInstance(arguments);

System.out.println(nc);

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

} catch (InstantiationException e) {

System.out.println(e);

}

}

}

class OuterClass {

private String outerField;

public OuterClass(String str) {

outerField = str;

}

public class NestedClass {

private String innerField;

public NestedClass(String str) {

innerField = str;

}

public String toString() {

String str;

str = "this = " + innerField;

str += ", outer.this = " + outerField;

return str;

}

}

}

결과는 다음과 같을 것이다.

this = InnerString, outer.this = OuterString

* 1. Array 클래스

배열을 다룬 장을 참조하라.

* 1. 리플렉션과 접근 제한
     1. IllegalAccessException

다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class c = A.class;

Field f = c.getDeclaredFields()[0]; // 이런거(...()[0]) 따라하지 말것.

f.get(new A("aoeuidhtns"));

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

private String privateField;

public A(String str) {

privateField = str;

}

}

결과는 다음과 비슷할 것이다.

java.lang.IllegalAccessException: Class Example can not access a member of class A with modifiers "private"

privateField가 private으로 선언했기 때문에 접근할 수 없다는 것이다. 이것은 메쏘드와 생성자도 마찬가지이다.

* + 1. AccessibleObject와 접근 제한의 우회

AccessibleObject는 Field, Method, Constructor의 수퍼 클래스이다. 이 클래스의 메쏘드를 이용해 위에 나오는 제한을 우회할 수 있다.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class c = A.class;

Field f = c.getDeclaredFields()[0]; // 이런거(...()[0]) 따라하지 말것.

f.setAccessible(true); // 여기에 주의할 것.

String str = (String)f.get(new A("aoeuidhtns"));

System.out.println(str);

} catch (IllegalAccessException e) {

System.out.println(e);

}

}

}

class A {

private String privateField;

public A(String str) {

privateField = str;

}

}

결과는 다음과 비슷할 것이다.

aoeuidhtns

위와 같은 결과가 나오지 않을 수도 있다. 이것은 Security Manager와 관련있는데 이 책의 범위를 넘어선다. 알고 싶은 사람은 자바 보안을 다루는 책을 참고하기 바란다.

그건 그렇고 위와 같이 접근 제한을 우회하는 것은 특별한 경우가 아니면 좋은 생각이 아니다. private, proteted 등의 키워드를 만든 사람이 아주 슬퍼할 것이다.

* 1. Proxy 클래스
     1. Proxy 에 대해

Proxy클래스는 JDK 1.3 버전에 추가되었다. Proxy 클래스는 주어진 인터페이스를 구현한 클래스를 동적으로 생성하게 해 준다. 내용이 상당히 어려우니 이해가 안가도 너무 좌절하지 않아도 된다. 거의 쓰이지는 않지만 상당히 흥미있는 기능이다.

* + 1. Proxy 클래스

다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class proxyClass = Proxy.getProxyClass(InterfaceA.class.getClassLoader(), new Class[] {InterfaceA.class});

System.out.println("Proxy class : " + proxyClass);

Class superClass = proxyClass.getSuperclass();

System.out.println("Super class : " + superClass);

Class[] superInterfaces = proxyClass.getInterfaces();

for (int i = 0; i < superInterfaces.length; i++)

System.out.println("Super interface : " + superInterfaces[i]);

Field[] fields = proxyClass.getDeclaredFields();

for (int i = 0; i < fields.length; i++)

System.out.println("Field : " + fields[i]);

Method[] methods = proxyClass.getDeclaredMethods();

for (int i = 0; i < methods.length; i++)

System.out.println("Method : " + methods[i]);

}

}

interface InterfaceA {

void methodA();

String methodB();

}

결과는 다음과 비슷할 것이다.

Proxy class : class $Proxy0

Super class : class java.lang.reflect.Proxy

Super interface : interface InterfaceA

Field : private static java.lang.reflect.Method $Proxy0.m4

Field : private static java.lang.reflect.Method $Proxy0.m3

Field : private static java.lang.reflect.Method $Proxy0.m0

Field : private static java.lang.reflect.Method $Proxy0.m2

Field : private static java.lang.reflect.Method $Proxy0.m1

Method : public final int $Proxy0.hashCode()

Method : public final boolean $Proxy0.equals(java.lang.Object)

Method : public final java.lang.String $Proxy0.toString()

Method : public final void $Proxy0.methodA()

Method : public final java.lang.String $Proxy0.methodB()

Proxy.getProxyClass 메쏘드가 Class 객체를 만들어서 반환해준다. 클래스 이름은 조금 이상한 $Proxy0이다. $가 붙는 것은 중첩 타입에서 보았을 것이다. 지금까지는 클래스를 만들려면 항상 소스 코드에서 클래스를 선언해 주어야 했는데 여기서는 $Proxy0이라는 클래스가 동적으로 만들어진 것이다. $Proxy0 클래스는 getProxyClass 메쏘드의 매개변수로 넣어준 인터페이스인 InterfaceA를 구현한다.

* + 1. InvocationHandler 인터페이스

Proxy 클래스(Proxy.getProxyClass 메쏘드로 만들어진 클래스) 의 객체를 생성해 보자. 그러려면 생성자가 있어야 한다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

Class proxyClass = Proxy.getProxyClass(InterfaceA.class.getClassLoader(), new Class[] {InterfaceA.class});

Constructor[] constructors = proxyClass.getConstructors();

for (int i = 0; i < constructors.length; i++)

System.out.println(constructors[i]);

}

}

interface InterfaceA {

void methodA();

String methodB();

}

결과는 다음과 비슷할 것이다.

public $Proxy0(java.lang.reflect.InvocationHandler)

생성자는 InvocationHandler를 인수로 받는 것 단 하나 있다.

다음은 InvocationHandler.java의 (주석을 제외한) 전체 소스 코드이다.

package java.lang.reflect;

public interface InvocationHandler {

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;

}

프록시 객체의 주어진 인터페이스의 메쏘드가 불려지면 InvocationHandler.invoke 메쏘드가 불려진다. 다음 예제를 보자.

import java.lang.reflect.\*;

public class Example {

public static void main(String[] args) {

try {

Class proxyClass = Proxy.getProxyClass(InterfaceA.class.getClassLoader(), new Class[] {InterfaceA.class});

Constructor constructor = proxyClass.getConstructor(new Class[] {InvocationHandler.class});

Object obj = constructor.newInstance(new Object[] {new InvocationHandlerImpl()});

InterfaceA ia = (InterfaceA)obj;

ia.methodA();

ia.methodB();

} catch (NoSuchMethodException e) {

System.out.println(e);

} catch (InvocationTargetException e) {

System.out.println(e);

} catch (IllegalAccessException e) {

System.out.println(e);

} catch (InstantiationException e) {

System.out.println(e);

}

}

}

interface InterfaceA {

void methodA();

String methodB();

}

class InvocationHandlerImpl implements InvocationHandler {

public Object invoke(Object proxy, Method method, Object[] args) {

System.out.println("InvocationHandlerImpl.invoke() method = " + method);

return null;

}

}

결과는 다음과 비슷할 것이다.

InvocationHandlerImpl.invoke() method = public abstract void InterfaceA.methodA()

InvocationHandlerImpl.invoke() method = public abstract java.lang.String InterfaceA.methodB()

실제로 일이 벌어지는 것은 invoke 메쏘드인 것이다. 이 invoke 메쏘드를 여러 가지 방식으로 구현해서 여러 가지 일을 할 수 있는 것이다.

* 1. 연습 프로그램
     1. 간단한 디컴파일러

다음 예제는 리플렉션을 이용해서 클래스의 수퍼 타입, 멤버 타입, 그리고 그 모든 타입의 멤버를 알아내는 프로그램이다. 코드를 보기 전에 먼저 실행을 해보고 한번 똑 같은 기능을 하는 프로그램을 만들어 보기 바란다. javax.swing.JTree를 입력하고 실행해 보라. 중첩 타입의 중첩 타입도 보여주는 것을 알 수 있다. 재귀 호출을 사용하고 있는데 그것에 대해서는 자료구조론 책을 참조하라.

import java.awt.\*;

import java.awt.event.\*;

import java.lang.reflect.\*;

import java.util.\*;

import javax.swing.\*;

public class SimpleDecompiler extends JFrame implements ActionListener, WindowListener {

public SimpleDecompiler() {

setSize(700, 500);

setLocation(200, 50);

getContentPane().setLayout(new BorderLayout());

JPanel pnl = new JPanel();

pnl.setLayout(new GridLayout(2, 1));

pnl.add(m\_lbl);

pnl.add(m\_tf);

getContentPane().add(pnl, BorderLayout.NORTH);

JScrollPane sp = new JScrollPane(m\_ta);

getContentPane().add(sp, BorderLayout.CENTER);

m\_tf.addActionListener(this);

m\_tf.setBackground(new Color(212, 208, 200));

m\_ta.setBackground(new Color(212, 208, 200));

addWindowListener(this);

}

public static Vector getAllSuperTypes(Class cls) {

Vector vctClasses = new Vector();

vctClasses.add(cls);

while (true) {

cls = cls.getSuperclass();

if (cls == null) {

break;

} else {

vctClasses.add(cls);

}

}

HashSet hsTemp = new HashSet();

for (Iterator itr = vctClasses.iterator(); itr.hasNext(); ) {

cls = (Class)itr.next();

Class[] acls = cls.getInterfaces();

for (int i = 0; i < acls.length; i++) {

hsTemp.add(acls[i]);

}

}

vctClasses.addAll(hsTemp);

return vctClasses;

}

public static void main(String args[]) {

Frame f = new SimpleDecompiler();

f.show();

}

public void actionPerformed(ActionEvent ae) {

m\_ta.setText("");

Class cls = null;

try {

cls = Class.forName(m\_tf.getText());

} catch (ClassNotFoundException cnfe) {

System.out.println("ReflectionToSuper.main() cnfe " + cnfe);

return;

}

showTypeDefinition(cls, 0);

}

public static void showTypeDefinition(Class cls, int iDepth) {

int iModifiers;

Class clsType;

Class[] aclsParameters;

Class[] aclsTypes;

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

m\_ta.append("\t");

}

iModifiers = cls.getModifiers();

m\_ta.append(Modifier.toString(iModifiers));

if (false == cls.isInterface()) {

m\_ta.append(" class");

}

m\_ta.append(" " + cls.getName());

Class clsSuper = cls.getSuperclass();

if (true == cls.isInterface()) {

aclsTypes = cls.getInterfaces();

if (aclsTypes.length > 0) {

m\_ta.append(" extends");

}

for (int j = 0; j < aclsTypes.length; j++) {

if (j != 0) {

m\_ta.append(",");

}

m\_ta.append(" " + aclsTypes[j].getName());

}

} else {

if (null != clsSuper) {

m\_ta.append(" extends");

m\_ta.append(" " + clsSuper.getName());

}

aclsTypes = cls.getInterfaces();

if (aclsTypes.length > 0) {

m\_ta.append(" implements");

}

for (int j = 0; j < aclsTypes.length; j++) {

if (j != 0) {

m\_ta.append(",");

}

m\_ta.append(" " + aclsTypes[j].getName());

}

}

m\_ta.append(" {\n");

Constructor[] acns = cls.getDeclaredConstructors();

for (int i = 0; i < acns.length; i++) {

for (int j = 0; j < iDepth; j++) {

m\_ta.append("\t");

}

iModifiers = acns[i].getModifiers();

m\_ta.append("\t" + Modifier.toString(iModifiers));

m\_ta.append(" " + acns[i].getName() + "(");

aclsParameters = acns[i].getParameterTypes();

for (int j = 0; j < aclsParameters.length; j++) {

if (j != 0) {

m\_ta.append(", ");

}

m\_ta.append(aclsParameters[j].getName());

}

m\_ta.append(")");

aclsTypes = acns[i].getExceptionTypes();

if (aclsTypes.length != 0) {

m\_ta.append(" throws");

}

for (int j = 0; j < aclsTypes.length; j++) {

if (j != 0) {

m\_ta.append(",");

}

m\_ta.append(" " + aclsTypes[j].getName());

}

m\_ta.append(";\n");

}

Method[] amth = cls.getDeclaredMethods();

for (int i = 0; i < amth.length; i++) {

for (int j = 0; j < iDepth; j++) {

m\_ta.append("\t");

}

iModifiers = amth[i].getModifiers();

m\_ta.append("\t" + Modifier.toString(iModifiers));

clsType = amth[i].getReturnType();

m\_ta.append(" " + clsType.getName());

m\_ta.append(" " + amth[i].getName() + "(");

aclsParameters = amth[i].getParameterTypes();

for (int j = 0; j < aclsParameters.length; j++) {

if (j != 0) {

m\_ta.append(", ");

}

m\_ta.append(aclsParameters[j].getName());

}

m\_ta.append(")");

aclsTypes = amth[i].getExceptionTypes();

if (aclsTypes.length != 0) {

m\_ta.append(" throws");

}

for (int j = 0; j < aclsTypes.length; j++) {

if (j != 0) {

m\_ta.append(",");

}

m\_ta.append(" " + aclsTypes[j].getName());

}

m\_ta.append(";\n");

}

Field[] afld = cls.getDeclaredFields();

for (int i = 0; i < afld.length; i++) {

for (int j = 0; j < iDepth; j++) {

m\_ta.append("\t");

}

iModifiers = afld[i].getModifiers();

m\_ta.append("\t" + Modifier.toString(iModifiers));

clsType = afld[i].getType();

m\_ta.append(" " + clsType.getName());

m\_ta.append(" " + afld[i].getName() + ";\n");

}

Class[] aclsMemberType = cls.getDeclaredClasses();

for (int i = 0; i < aclsMemberType.length; i++) {

showTypeDefinition(aclsMemberType[i], iDepth + 1);

}

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

m\_ta.append("\t");

}

m\_ta.append("}\n");

}

public void windowOpened(WindowEvent we) {

}

public void windowClosing(WindowEvent we) {

System.exit(0);

}

public void windowClosed(WindowEvent we) {

}

public void windowActivated(WindowEvent we) {

}

public void windowDeactivated(WindowEvent we) {

}

public void windowIconified(WindowEvent we) {

}

public void windowDeiconified(WindowEvent we) {

}

private static JLabel m\_lbl = new JLabel("full name(예 java.awt.Panel)을 입력하고 엔터를 치시오.");

private static JTextField m\_tf = new JTextField();

private static JTextArea m\_ta = new JTextArea();

}

java.awt.Panel을 입력했을 때의 결과이다.

public class java.awt.Panel extends java.awt.Container implements javax.accessibility.Accessible {

public java.awt.Panel(java.awt.LayoutManager);

public java.awt.Panel();

public javax.accessibility.AccessibleContext getAccessibleContext();

java.lang.String constructComponentName();

public void addNotify();

private static final java.lang.String base;

private static int nameCounter;

private static final long serialVersionUID;

protected class java.awt.Panel$AccessibleAWTPanel extends java.awt.Container$AccessibleAWTContainer {

protected java.awt.Panel$AccessibleAWTPanel(java.awt.Panel);

public javax.accessibility.AccessibleRole getAccessibleRole();

private final java.awt.Panel this$0;

}

}

1. 쓰레드(thread)
   1. 쓰레드란
      1. 프로세스(process)와 쓰레드

프로세스와 쓰레드에 대해 예를 들어 설명해 보겠다. 만약 당신의 컴퓨터에 워드 프로세서와 브라우저가 돌고 있다면 워드 프로세서와 브라우저는 두 개의 독립된 프로세스이다. 쓰레드는 프로세스보다 더 작은 단위이다. 워드 프로세서는 프린트를 하면서 맞춤법 테스트를 할 수 있다. 그러면 프린트하는 쓰레드와 맞춤법 테스트하는 쓰레드가 돌고 있는 것이다.

프로세스에서는 각각의 프로세스가 아예 그 코드가 차지하는 메모리가 다르다. 그러나 쓰레드는 같은 메모리에 있는 코드를 바탕으로 여러 개를 동시에 실행하는 것이다. 좀더 일반적으로 말하면 프로세스는 많은 부분의 자원을 독립적으로 가지고 있는 반면 쓰레드는 많은 부분의 자원을 공유한다. 그렇기 때문에 한 프로세스를 실행하다가 동시에 돌고 있는 다른 프로세스를 실행하기 위해서는 많은 비용이 드는 바면 쓰레드에서는 비교적 저렴한 비용으로 쓰레드 사이를 왔다갔다 할 수 있다. 이것이 쓰레드가 각광받은 이유 중에 하나이다.

* 1. 쓰레드 만들기
     1. Thread 클래스

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thOne = new UserThread("One");

thOne.run();

Thread thTwo = new UserThread("Two");

thTwo.run();

Thread thThree = new UserThread("Three");

thThree.run();

}

}

class UserThread extends Thread {

private String field;

public UserThread(String str) {

field = str;

}

public void run() {

for (int i = 0; i < 3; i++) {

System.out.println(field + " : " + i);

try {

Thread.sleep(1000); // 1000 밀리초(1초) 쉰다.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 같을 것이다. 여기서는 시간적인 요소도 중요하니 한번 실행해 보기 바란다.

One : 0

One : 1

One : 2

Two : 0

Two : 1

Two : 2

Three : 0

Three : 1

Three : 2

위의 예제는 하나의 쓰레드로 돈다(garbage collection 쓰레드 등은 무시한다). 순차적으로 실행되는 것을 볼 수 있을 것이다. 위의 예제에서는 Thread를 상속받은게 아무런 의미가 없다. 이 예제는 쓰레드의 특성을 보여주기 위한 예제이다. run 메쏘드를 직접 호출하면 Thread 클래스의 기능을 전혀 사용할 수 없다. 그럼 위의 예제를 약간 고쳐 보자.

public class Example {

public static void main(String[] args) {

Thread thOne = new UserThread("One");

thOne.start(); // run -> start

Thread thTwo = new UserThread("Two");

thTwo.start(); // run -> start

Thread thThree = new UserThread("Three");

thThree.start(); // run -> start

}

}

class UserThread extends Thread {

private String field;

public UserThread(String str) {

field = str;

}

public void run() {

for (int i = 0; i < 3; i++) {

System.out.println(field + " : " + i);

try {

Thread.sleep(1000); // 1000 밀리초(1초) 쉰다.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

One : 0

Two : 0

Three : 0

One : 1

Two : 1

Three : 1

One : 2

Two : 2

Three : 2

이번에는 run 메쏘드를 직접 호출하지 않고 start 메쏘드를 호출했다. 이것이 Thread 클래스를 사용하는 방법이다. 실행시간이 약 9초이던 것이 약 3초로 단축되었다. 그리고 순차적으로 실행되던 것이 동시적으로 실행된다.

* + 1. Runnable 인터페이스

위의 예제는 Thread를 직접 상속받았다. 이번에는 다른 방법을 사용해보자. 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thOne = new Thread( new UserThread("One") );

thOne.start();

Thread thTwo = new Thread( new UserThread("Two") );

thTwo.start();

Thread thThree = new Thread( new UserThread("Three") );

thThree.start();

}

}

class UserThread implements Runnable {

private String field;

public UserThread(String str) {

field = str;

}

public void run() {

for (int i = 0; i < 3; i++) {

System.out.println(field + " : " + i);

try {

Thread.sleep(1000); // 1000 밀리초(1초) 쉰다.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

One : 0

Two : 0

Three : 0

Two : 1

One : 1

Three : 1

Two : 2

Three : 2

One : 2

UserThread가 Thread를 상속하는 대신 Runnable 인터페이스를 구현했다. Runnable 인터페이스의 메쏘드는 run 메쏘드 단 하나이다. 그리고 Thread의 생성자에 UserThread 객체를 인자(argument)로 넘겨주었다. 좀더 번거로워졌다. 그러나 이 방법이 더 많이 쓰인다. 그 이유는 무엇일까? 그 이유 중 가장 큰 이유는 자바에서는 하나의 클래스만 상속받을 수 있기 때문이다. UserThread가 만약 다른 클래스를 상속받아야만 한다면 선택의 여지가 없다. Thread를 상속할 수 없기 때문에 Runnable 인터페이스를 구현할 수밖에 없는 것이다.

위의 예제를 다시 조금 고쳐 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread("OneObject");

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

Thread thThree = new Thread(r);

thThree.setName("Three");

thThree.start();

}

}

class UserThread implements Runnable {

private String field;

public UserThread(String str) {

field = str;

}

public void run() {

for (int i = 0; i < 3; i++) {

String str = Thread.currentThread().getName();

System.out.println(field + " : " + str + " : " + i);

try {

Thread.sleep(1000); // 1000 밀리초(1초) 쉰다.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

OneObject : One : 0

OneObject : Two : 0

OneObject : Three : 0

OneObject : Two : 1

OneObject : One : 1

OneObject : Three : 1

OneObject : Two : 2

OneObject : Three : 2

OneObject : One : 2

앞에서는 UserThread 객체가 세 개 만들어졌는데 이번에는 하나만 만들었다. 만약 UserThread가 Thread 클래스를 상속했다면 이것도 불가능했을 것이다. 여기서 주목할 것은 세 개의 쓰레드가 하나의 UserThread객체를 공유하고 있다는 것이다.

혹시 Thread 클래스를 상속해도 된다고 우기는 사람이 있을지도 모르겠다. 다음 예제처럼 하면 가능하기는 하다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread("OneObject");

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

Thread thThree = new Thread(r);

thThree.setName("Three");

thThree.start();

}

}

class UserThread extends Thread { // Thread 상속

private String field;

public UserThread(String str) {

field = str;

}

public void run() {

for (int i = 0; i < 3; i++) {

String str = Thread.currentThread().getName();

System.out.println(field + " : " + str + " : " + i);

try {

Thread.sleep(1000); // 1000 밀리초(1초) 쉰다.

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

OneObject : One : 0

OneObject : Two : 0

OneObject : Three : 0

OneObject : One : 1

OneObject : Two : 1

OneObject : Three : 1

OneObject : One : 2

OneObject : Two : 2

OneObject : Three : 2

Thread 클래스가 Runnable 인터페이스를 구현하기 때문에 가능하긴 하다. 그러나 이건 낭비다. 거기다가 보는 사람을 헷갈리게 한다.

* 1. Daemon 쓰레드
     1. daemon 쓰레드

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("End of main()");

}

}

결과는 다음과 같다.

End of main()

여기서 중요한 것은 main 메쏘드가 끝나면서 프로그램이 종료된다는 것이다. 그럼 다음 예제는 어떨까?

public class Example {

public static void main(String[] args) {

new UserThread().start();

Thread thr = Thread.currentThread();

System.out.println(thr.getName() + " : End of main()");

}

}

class UserThread extends Thread {

public void run() {

int i = 0;

while (true) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

Thread thr = Thread.currentThread();

System.out.println(thr.getName() + " : " + i);

i++;

}

}

}

결과는 다음과 비슷할 것이다.

main : End of main()

Thread-0 : 0

Thread-0 : 1

Thread-0 : 2

Thread-0 : 3

Thread-0 : 4

…

프로그램은 종료되지 않는다. main 쓰레드가 종료되었지만 Thread-0 쓰레드가 살아있기 때문이다.

이제 데몬(daemon) 쓰레드에 대해 알아보자. 다음 예제는 데몬 쓰레드를 보여준다.

public class Example {

public static void main(String[] args) {

UserThread userThread = new UserThread();

userThread.setDaemon(true);

userThread.start();

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

System.out.println(e);

}

Thread thr = Thread.currentThread();

System.out.println(thr.getName() + " : " + thr.isDaemon());

System.out.println(thr.getName() + " : End of main()");

}

}

class UserThread extends Thread {

public void run() {

int i = 0;

while (true) {

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

Thread thr = Thread.currentThread();

System.out.println(thr.getName() + " : " + i);

i++;

}

}

}

결과는 다음과 비슷할 것이다.

Thread-0 : 0

Thread-0 : 1

Thread-0 : 2

main : false

main : End of main()

Thread-0 : 3

데몬이 아닌 main 쓰레드가 종료되자 데몬인 Thread-0 쓰레드가 살아있음에도 프로그램은 종료된다.

* 1. sleep, yield, jion
     1. seep

sleep 메쏘드는 일정 시간동안 실행을 중지하는 메쏘드이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

System.out.println("wait three seconds");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

}

System.out.println("after three seconds");

}

}

결과는 다음과 같을 것이다.

wait three seconds

after three seconds

첫 줄이 뿌려진 다음에 약 3초 후에 다음 줄이 뿌려질 것이다.

sleep 메쏘드는 InterruptedException이라는 예외상황이 발생할 수 있기 때문에 이것을 꼭 try-catch로 잡아주어야 한다. 그렇지 않으면 컴파일이 되지 않을 것이다. 아래에 보면 stop 메쏘드를 다루는 곳에 sleep 된 쓰레드를 interupt 메쏘드로 깨우는 예제가 있다.

* + 1. yield

yield 메쏘드는 다른 쓰레드에게 양보하게 하는 메쏘드이다. 양보하는 것은 좋은 습관이다. 쓰레드 프로그래밍을 할 때 문제가 되는 것 중에 하나가 하나의 쓰레드가 CPU 타임을 몽땅 잡아먹는 것이다. 보통 쓰레드는 운영체제에서 정해준 시간만큼 일하고 양보를 한다. 그러나 yield 메쏘드를 사용하면 그 메쏘드를 만나는 즉시 양보를 하는 것이다.

* + 1. join

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thr = new UserThread();

thr.start();

System.out.println("before join()");

try {

thr.join();

} catch (InterruptedException e) {

}

System.out.println("after join()");

}

}

class UserThread extends Thread {

public void run() {

System.out.println("birth of user thread");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

}

System.out.println("death of user thread");

}

}

결과는 다음과 비슷할 것이다.

before join()

birth of user thread

death of user thread

after join()

join 메쏘드를 실행하면 현재 쓰레드(위의 예제에서는 main 메쏘드의 쓰레드)는 그 쓰레드(위의 예제에서는 thr 쓰레드)가 죽을 때까지 기다린다. 그래서 약 3초 후에 “death of user thread”가 뿌려진 다음에 즉 thr 쓰레드가 죽은 다음에 그 다음 줄이 실행되어 “after join()”이 뿌려진 것이다. 다른 말로 하면 thr 쓰레드가 일을 다 끝마친 다음까지 기다린 후에 자신의 일을 계속하는 것이다. 만약 thr 쓰레드의 일이 현재 자신의 일에 중요한 전제가 된다면 그 쓰레드의 일이 끝나기를 기다려야 할 것이다.

만약 그 쓰레드가 일을 끝내기를 무한정 기다릴 수 없다면 시간을 정해서 그 시간이 지나면 그 쓰레드가 안 죽어도 일을 처리할 수 있다. 다음 예제를 보자. 위의 예제가 조금 바뀌었다.

public class Example {

public static void main(String[] args) {

Thread thr = new UserThread();

thr.start();

System.out.println("before join()");

try {

thr.join(3000);

} catch (InterruptedException e) {

}

System.out.println("after join()");

}

}

class UserThread extends Thread {

public void run() {

System.out.println("birth of user thread");

try {

Thread.sleep(300000);

} catch (InterruptedException e) {

}

System.out.println("death of user thread");

}

}

결과는 다음과 비슷할 것이다.

before join()

birth of user thread

after join()

정해 놓은 시간 3초가 지나도 thr 쓰레드가 죽지 않자 더 이상 기다리지 않고 계속 자신의 일을 처리한다.

join 메쏘드도 sleep 메쏘드처럼 InterruptedException을 처리해 주어야 한다.

* 1. ThreadGroup 클래스
     1. 현재 돌고 있는 쓰레드들 알아내기

현재 돌고 있는 쓰레드에 무엇이 있는지 알아보는 예제다. 재귀 호출을 사용했는데 어렵지 않으니 잘 보면 쉽게 알 수 있을 것이다.

public class Example {

public static void main(String[] args) {

ThreadGroup thg = Thread.currentThread().getThreadGroup();

ThreadGroup thgTop = getTopThreadGroup(thg);

thgTop.list();

}

public static ThreadGroup getTopThreadGroup(ThreadGroup thg) {

ThreadGroup thgParent = thg.getParent();

if (thgParent == null)

return thg;

else

return getTopThreadGroup(thgParent); // 재귀호출

}

}

결과는 다음과 비슷할 것이다.

java.lang.ThreadGroup[name=system,maxpri=10]

Thread[Reference Handler,10,system]

Thread[Finalizer,8,system]

Thread[Signal Dispatcher,10,system]

Thread[CompileThread0,10,system]

java.lang.ThreadGroup[name=main,maxpri=10]

Thread[main,5,main]

* 1. priority
     1. priority란

쓰레드들은 평등할 수도 있고 평등하지 않을 수도 있다. 때에 따라서는 쓰레드들이 불평등해야 더 좋은 경우도 있다. 어떤 일은 급한 일이고 어떤 일은 천천히 해도 되는 일이라면 급한 일에 우선권을 주는 것이 좋다. 쓰레드에 우선권을 나타내는 숫자를 부여할 수 있는데 이런 것을 priority라고 한다.

* + 1. setPriority, getPriority

setPriority 메쏘드는 해당 쓰레드의 priority를 부여하는 메쏘드이고 getPriority는 부여된 priority를 구하는 메쏘드이다. ThreadGroup에는 setMaxPriority, getMaxPriority 메쏘드가 있는데 그 그룹에 속하는 쓰레드가 가질 수 있는 priority의 최대값을 설정하고 구하는 메쏘드들이다. priority값은 Thread.MIN\_PRIORITY에서 Thread.MAX\_PRIORITY까지의 값이 지정될 수 있다. 그리고 지정이 안된다면 Thread.NORM\_PRIORITY 값이 디폴트로 설정된다. 다음 예제는 여러 쓰레드에 서로 다른 priority 값을 지정하여 실행하는 예제이다.

import java.math.\*;

public class Example {

public static void main(String[] args) {

UserThread[] threads = new UserThread[Thread.MAX\_PRIORITY - Thread.MIN\_PRIORITY + 1];

for (int i = 0, p = Thread.MIN\_PRIORITY; p <= Thread.MAX\_PRIORITY; i++, p++) {

threads[i] = new UserThread();

threads[i].setPriority(p);

threads[i].start();

}

try {

Thread.sleep(300000);

} catch (InterruptedException e) {

}

for (int i = 0, p = Thread.MIN\_PRIORITY; p <= Thread.MAX\_PRIORITY; i++, p++) {

System.out.println(threads[i].toString());

}

}

}

class UserThread extends Thread {

private long bigValue = 0;

private long value = 0;

private boolean finished = false;

public void run() {

while (!finished) {

value++;

if (value == Long.MAX\_VALUE) {

value = 0;

bigValue++;

}

}

}

public String toString() {

finished = true;

return "priority : " + getPriority() + ", bigValue = " + bigValue + ", value : " + value;

}

}

약 5분간 기다리면 다음과 비슷한 결과가 나올 것이다.

priority : 1, bigValue = 0, value : 37126228

priority : 2, bigValue = 0, value : 40176651

priority : 3, bigValue = 0, value : 44195318

priority : 4, bigValue = 0, value : 42958854

priority : 5, bigValue = 0, value : 51451123

priority : 6, bigValue = 0, value : 71164570

priority : 7, bigValue = 0, value : 61408641

priority : 8, bigValue = 0, value : 215872213

priority : 9, bigValue = 0, value : 160913257

priority : 10, bigValue = 0, value : 14699066730

priority가 높을수록 더 많은 시간을 할당받았다는 것을 알 수 있다.

만약 priority를 설정하는 부분 즉

threads[i].setPriority(p);

을 없애버리고 실행하면 결과는 다음과 비슷할 것이다. 위에 비해 결과가 상당히 평등하다.

priority : 5, bigValue = 0, value : 1494289235

priority : 5, bigValue = 0, value : 1434519910

priority : 5, bigValue = 0, value : 1450214126

priority : 5, bigValue = 0, value : 1466789413

priority : 5, bigValue = 0, value : 1453546994

priority : 5, bigValue = 0, value : 1471783463

priority : 5, bigValue = 0, value : 1466201278

priority : 5, bigValue = 0, value : 1414939573

priority : 5, bigValue = 0, value : 1426676353

priority : 5, bigValue = 0, value : 1421307186

여기서 주의할 것은 쓰레드를 관리하는 것은 결국은 운영체제이기 때문에 운영체제에서 priority를 지원하지 않으면 setPriority 메쏘드는 아무런 의미가 없다는 것이다.

* 1. 경합상태
     1. 경합상태(race condition)이란

쓰레드 프로그래밍을 하다보면 부딪히는 첫번째 문제가 경합상태(race condition)이다. 여러 쓰레드가 하나의 자원 예를 들어 변수를 공동으로 사용함으로써 문제가 생기는 것이다. 경합상태 때문에 생기는 버그는 상당히 까다로운 버그이다. 아무런 문제 없이 잘 돌다가 아주 가끔 가다 문제가 생기기 때문이다. 그리고 예측을 할 수 없다는 것도 커다란 문제이다. 동시에 여러 쓰레드가 돌면 어떤 쓰레드가 어떤 변수의 값을 언제 바꾸는지 예측하기가 거의 불가능하다.

비유를 들어서 설명해 보겠다. 여러 사람이 하나의 칠판을 공유한다고 생각해 보자. 즉 하나의 칠판을 여러 사람이 동시에 쓰는 것이다. 만약 어떤 사람이 열심히 수학 문제를 풀고 있는데 다른 사람이 자기가 쓰겠다고 그 중의 일부를 지워버린다면 어떤 문제가 발생할까? 수학 문제를 푸는 사람의 일이 방해를 받는다.

또다른 비유를 들자면 화장실을 들 수 있다. 화장실에서 어떤 사람이 일을 보고 있는데 누군가 화장실 안으로 들어닥친다면 참으로 곤란할 것이다.

위의 두 예에서 칠판, 또는 화장실을 변수라고 생각하고 사람들을 쓰레드라고 생각하면 된다. 여러 쓰레드가 동시에 돌고 있는데 하나의 변수를 사용하려고 경쟁(race)하기 때문에 문제가 발생하는 것이다.

* + 1. 경합상태(race condition)의 예

이번에는 실제 프로그램에서 경합상태가 어떻게 발생하는지 보자. 예제를 보자. 이 예제는 순차적으로 하나씩 실행한다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

for (int i = 1; i <= 10; i++) {

Thread th = new Thread(r);

th.setName("" + i);

th.run(); // 비정상적인 사용이지만 테스트를 위해.

}

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

for (int i = 0; i < 10000000; i++)

field++;

String str = Thread.currentThread().getName();

System.out.println(str + " : " + field);

}

}

결과는 다음과 같을 것이다.

main : 10000000

main : 20000000

main : 30000000

main : 40000000

main : 50000000

main : 60000000

main : 70000000

main : 80000000

main : 90000000

main : 100000000

main 쓰레드 하나에서 다 도는 것을 볼 수 있다. 이번에는 한 줄만 고친 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

for (int i = 1; i <= 10; i++) {

Thread th = new Thread(r);

th.setName("" + i);

th.start(); // 이제 정상적인 사용.

}

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

for (int i = 0; i < 10000000; i++)

field++;

String str = Thread.currentThread().getName();

System.out.println(str + " : " + field);

}

}

결과는 다음과 비슷할 것이다.

4 : 25973658

5 : 23871036

6 : 24264832

9 : 20820827

1 : 20863655

7 : 21419704

2 : 24356547

10 : 22354437

8 : 24469510

3 : 25779854

순서야 어떻든 10000000 \* 10번 ++연산자를 사용했기 때문에 결국 field의 값이 100000000이 되어야 한다고 생각하는 사람이 있을 것이다. 이런 기대를 무너뜨리는 것이 경합 상태이다.

이제 다른 예제를 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

long temp;

String str = Thread.currentThread().getName();

temp = field;

Thread.yield(); // 다른 쓰레드에게 양보한다.

temp++;

field = temp;

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

}

}

결과는 아마 다음과 같을 것이다.

Two : field = 1

One : field = 1

field++가 총 두 번 실행되었는데 값은 1밖에 증가되지 않았다. 무슨 일이 일어나고 있는지 보기 위해 중간 중간에 변수의 내용을 콘솔(console)에 뿌려보겠다. 위의 예제를 약간 고쳤다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

long temp;

String str = Thread.currentThread().getName();

System.out.println(str + " : field(before ++) = " + field);

temp = field;

System.out.println(str + " : temp(before ++) = " + temp);

Thread.yield(); // 다른 쓰레드에게 양보한다.

temp++;

System.out.println(str + " : temp(after ++) = " + temp);

field = temp;

System.out.println(str + " : field(after ++) = " + field);

}

}

결과는 아마 다음과 같을 것이다.

One : field(before ++) = 0

One : temp(before ++) = 0

Two : field(before ++) = 0

Two : temp(before ++) = 0

One : temp(after ++) = 1

One : field(after ++) = 1

Two : temp(after ++) = 1

Two : field(after ++) = 1

먼저 쓰레드 One이 실행되고 temp에 field의 값인 0이 대입된다. 그 다음에 yield 메쏘드가 실행되기 때문에 다른 쓰레드인 Two에게 양보한다. 쓰레드 Two가 실행되고 temp에 field의 값인 0이 대입된다. 다시 yield 메쏘드가 실행되기 때문에 다른 쓰레드인 One에게 양보한다. 쓰레드 One은 temp를 1증가시키고 그것은 field에 대입한다. 그러므로 field의 값은 1이 되었다. 이제 One은 실행을 마친다. 그러므로 이젠 Two가 실행된다. Two는 temp를 1 증가시키고 그것을 field에 대입한다. 그러므로 field의 값은 1이 된다. 이것이 race condition의 메커니즘이다.

* 1. synchronized와 lock
     1. lock

위에서 화장실에 비유했던 것을 기억할 것이다. 화장실에서 난감한 일을 막는 방법은 무엇인가? 화장실의 문을 잠글 수 있게 하는 것이다. 그리고 노크를 한다. 쓰레드의 문제점은 여러 쓰레드가 하나의 객체를 공유한다는데 있다. 그 해결책은 객체마다 잠금 장치(lock)를 달아 두는 것이다. 그리고 화장실에 사람이 들어가면 문을 잠그듯이 어떤 객체를 어떤 쓰레드가 쓰고 있으면 객체를 잠그는 것이다. 그리고 다른 쓰레드는 화장실에서처럼 줄을 선다. 그리고 맨 앞에 있는 쓰레드가 노크를 한다. 만약 아무도 없다는 것이 확인되면 객체를 잠그고 자신이 사용한다.

* + 1. synchronized 문

객체의 문을 잠글 수 있도록 하는 것이 바로 synchronized 키워드이다. 경합 상태(race condition)가 일어났던 위의 두 예제를 synchronized문을 이용해서 고쳐 보겠다. 먼저 첫번째 예제를 고친 것이다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

for (int i = 1; i <= 10; i++) {

Thread th = new Thread(r);

th.setName("" + i);

th.start(); // 이제 정상적인 사용.

}

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

synchronized (this) {

for (int i = 0; i < 10000000; i++)

field++;

String str = Thread.currentThread().getName();

System.out.println(str + " : " + field);

}

}

}

결과는 다음과 비슷할 것이다.

2 : 10000000

3 : 20000000

4 : 30000000

5 : 40000000

6 : 50000000

7 : 60000000

8 : 70000000

9 : 80000000

10 : 90000000

1 : 100000000

이번에는 두번째 예제를 고친 것이다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

private long field = 0;

public void run() {

long temp;

String str = Thread.currentThread().getName();

synchronized (this) {

System.out.println(str + " : field(before ++) = " + field);

temp = field;

System.out.println(str + " : temp(before ++) = " + temp);

Thread.yield(); // 다른 쓰레드에게 양보한다.

temp++;

System.out.println(str + " : temp(after ++) = " + temp);

field = temp;

System.out.println(str + " : field(after ++) = " + field);

}

}

}

결과는 다음과 같을 것이다.

One : str 설정됨

One : field(before ++) = 0

One : temp(before ++) = 0

Two : str 설정됨

One : temp(after ++) = 1

One : field(after ++) = 1

Two : field(before ++) = 1

Two : temp(before ++) = 1

Two : temp(after ++) = 2

Two : field(after ++) = 2

이 예의 실행 순서를 보자. 일단 쓰레드 One이 실행된다. str에 현재 쓰레드의 이름이 설정된다. 다음에 synchronized (this)를 만난다. 화장실에서 노크를 한다고 생각하면 된다. 아무도 없다. 화장실에 들어간다. 그리고 문을 잠근다. temp에 field의 값인 0이 대입된다. yield 메쏘드가 실행된다. 그러므로 다른 쓰레드인 Two가 실행된다. 먼저 str 값이 설정된다. 그리고 synchronized (this)를 만난다. 노크를 한다. 누가(One) 있다. Two는 화장실에 들어갈 수 없다. 할일이 없는 Two는 양보한다. 이제 다시 One이 실행된다. temp가 1 증가되고 그 값이 field에 대입된다. One은 실행을 마친다. 그러므로 화장실은 다시 열린 상태이다. 볼일을 다 보면 화장실을 열어놓고 가기 때문이다. 이젠 Two가 실행된다. 아까 해결이 안되었던 synchronized (this)를 다시 만한다. 즉 다시 노크하는 것이다. 이번에는 화장실이 비어있다. temp 에 field의 값을 1을 대입한다. temp를 1증가시키고 그 값을 field에 대입한다. Two도 실행을 마친다.

* + 1. synchronized 메쏘드

위의 예제 중 하나를 다음과 같이 고쳤다.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread();

for (int i = 1; i <= 10; i++) {

Thread th = new Thread(r);

th.setName("" + i);

th.start();

}

}

}

class UserThread implements Runnable {

private long field = 0;

public synchronized void run() {

for (int i = 0; i < 10000000; i++)

field++;

String str = Thread.currentThread().getName();

System.out.println(str + " : " + field);

}

}

결과는 다음과 비슷할 것이다.

1 : 10000000

2 : 20000000

3 : 30000000

4 : 40000000

5 : 50000000

6 : 60000000

7 : 70000000

8 : 80000000

9 : 90000000

10 : 100000000

synchronized 메쏘드는 synchronized문을 간편하게 바꾼것이다. 예를 들어

public void synchronized method() {

// 실행 부분.

}

는

public void method() {

synchronized (this) {

// 실행 부분.

}

}

와 동등하다.

이번에는 하나의 클래스에 두 개 이상의 synchronized 메쏘드가 있다면 어떻게 작동하는지 알아보자. lock되는 것은 객체지 메쏘드가 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread(); // 바로 이 객체가 lock 된다.

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("One")) {

one();

}

if (str.equals("Two")) {

two();

}

}

public synchronized void one() {

String str = Thread.currentThread().getName();

System.out.println(str + " : one() first");

Thread.yield();

System.out.println(str + " : one() second");

}

public synchronized void two() {

String str = Thread.currentThread().getName();

System.out.println(str + " : two() first");

Thread.yield();

System.out.println(str + " : two() second");

}

}

결과는 다음과 같을 것이다.

One : one() first

One : one() second

Two : two() first

Two : two() second

또는 다음과 같을 수도 있다.

Two : two() first

Two : two() second

One : one() first

One : one() second

먼저 쓰레드 One이 실행된다(경우에 따라서는 쓰레드 Two가 먼저 실행될 수도 있다. 그러면 두번째에 나온 결과처럼 될 것이다.). 쓰레드 이름이 “One”이기 때문에 메쏘드 one이 실행된다. 이때 one 메쏘드가 synchronized로 선언되었기 때문에 객체를 잠근다. 먼저 “One : one() first”가 화면에 뿌려진다. 그 다음에 yield 메쏘드가 실행되기 때문에 쓰레드 Two에게 양보한다. 쓰레드 Two는 메쏘드 two를 실행한다. 메쏘드 two도 synchronized로 선언되었기 때문에 객체가 잠겨 있는지 알아본다. 물론 One이 잠궜기 때문에 메쏘드 two는 실행될 수 없다. 다시 메쏘드 one이 계속 실행된다. 즉 “One : one() second”가 화면에 뿌려진다. 이젠 메쏘드 one의 실행이 끝나고 걸었던 lock이 풀린다. 쓰레드 One의 실행이 종료된다. 이제 쓰레드 Two가 실행되고 아까 막혔던 곳에서 다시 출발한다. 즉 다시 lock이 걸려 있는지 체크한다. lock은 풀렸다. 그러므로 메쏘드 two는 실행될 수 있다. 먼저 쓰레드 Two는 lock을 건다. 그 다음에 “Two : two() first”가 화면에 뿌려진다. 다음에 yield 메쏘드가 실행되기 때문에 양보한다. 다른 쓰레드가 없기 때문에 쓰레드 Two는 메쏘드 two를 계속 실행한다. 즉 “Two : two() second”가 화면에 뿌려진다. 이제 메쏘드 two도 끝나고 lock이 풀린다. 이젠 쓰레드 Two도 끝난다.

위의 예제에서 synchronized를 빼 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread(); // 바로 이 객체가 lock 된다.

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("One")) {

one();

}

if (str.equals("Two")) {

two();

}

}

public void one() {

String str = Thread.currentThread().getName();

System.out.println(str + " : one() first");

Thread.yield();

System.out.println(str + " : one() second");

}

public void two() {

String str = Thread.currentThread().getName();

System.out.println(str + " : two() first");

Thread.yield();

System.out.println(str + " : two() second");

}

}

결과는 다음과 같을 것이다.

One : one() first

Two : two() first

One : one() second

Two : two() second

또는 다음과 같을 수도 있다.

Two : two() first

One : one() first

Two : two() second

One : one() second

왜 이런 결과가 나왔는지 잘 따져보기 바란다. 이번에는 둘 중에 하나만 synchronized를 빼 보자.

public class Example {

public static void main(String[] args) {

Runnable r = new UserThread(); // 바로 이 객체가 lock 된다.

Thread thOne = new Thread(r);

thOne.setName("One");

thOne.start();

Thread thTwo = new Thread(r);

thTwo.setName("Two");

thTwo.start();

}

}

class UserThread implements Runnable {

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("One")) {

one();

}

if (str.equals("Two")) {

two();

}

}

public synchronized void one() {

String str = Thread.currentThread().getName();

System.out.println(str + " : one() first");

Thread.yield();

System.out.println(str + " : one() second");

}

public void two() {

String str = Thread.currentThread().getName();

System.out.println(str + " : two() first");

Thread.yield();

System.out.println(str + " : two() second");

}

}

결과는 다음과 같을 것이다.

One : one() first

Two : two() first

One : one() second

Two : two() second

또는 다음과 같을 수도 있다.

Two : two() first

One : one() first

Two : two() second

One : one() second

* + 1. this가 아닌 다른 객체를 잠그는 synchronized 문

synchronized문에서 잠글 수 있는 객체는 this만이 아니다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

LockedObject lo = new LockedObject(); // LockedObject의 객체

A a = new A(lo);

a.start();

B b = new B(lo);

b.start();

}

}

class A extends Thread {

private LockedObject lo;

public A(LockedObject lo) {

this.lo = lo;

}

public void run() {

synchronized (lo) {

System.out.println("A.run() first");

Thread.yield();

System.out.println("A.run() second");

}

}

}

class B extends Thread {

private LockedObject lo;

public B(LockedObject lo) {

this.lo = lo;

}

public void run() {

synchronized (lo) {

System.out.println("B.run()");

}

}

}

class LockedObject {

}

결과는 다음과 같을 것이다.

A.run() first

A.run() second

B.run()

먼저 A.run()이 실행된다. synchronized (lo) 문을 실행하면서 LockedObject 객체를 잠근다. 그 다음에 “A.run() first”가 화면에 뿌려진다. yield가 실행되어 B.run()이 실행된다. synchronized (lo) 문을 만난다. lo 객체가 잠겼는지 체크한다. 잠겼다. 그러므로 synchronized문을 실행하지 못한다. 다시 A.run()의 나머지 부분이 실행된다. “A.run() second”가 화면에 뿌려지고 A.run()에 있는 synchronized 문이 끝난다. 그러므로 객체에 대한 lock이 풀린다. 다시 B.run()의 synchronized 문이 실행되는데 이 때는 lock이 풀렸으므로 실행될 수 있다. 즉 “B.run()”이 화면에 뿌려진다.

위의 예제에서 synchronized를 없애고 실행해 보면 위의 예제를 이해하는데 도움이 될 것이다.

public class Example {

public static void main(String[] args) {

LockedObject lo = new LockedObject(); // LockedObject의 객체

A a = new A(lo);

a.start();

B b = new B(lo);

b.start();

}

}

class A extends Thread {

private LockedObject lo;

public A(LockedObject lo) {

this.lo = lo;

}

public void run() {

System.out.println("A.run() first");

Thread.yield();

System.out.println("A.run() second");

}

}

class B extends Thread {

private LockedObject lo;

public B(LockedObject lo) {

this.lo = lo;

}

public void run() {

System.out.println("B.run()");

}

}

class LockedObject {

}

결과는 다음과 같을 때도 있을 것이다.

A.run() first

B.run()

A.run() second

결과가 위와 같이 나오지 않을 수도 있다. B.run()이 먼저 실행되는 경우에는 synchronized를 하나 안하나 결과가 마찬가지일 것이다. 그러나 A.run()이 먼저 실행되는 경우에는 synchronized 문을 사용한 경우에는 첫번째 예제의 결과대로만 나올 것이다. synchronized 문을 사용하지 않은 경우에는 대체로 두번째 예제의 결과대로 나올 것이다.

synchronized 문으로 잠글 수 있는 것은 객체다. 기본형 변수를 잠글 수는 없다.

* + 1. 잠기는 것은 객체이지 클래스가 아니다.

자바 lock은 하나의 객체마다 있다. 잠기는 것은 클래스도 아니고 메쏘드도 아니다. 바로 객체가 잠기는 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

LockedObject loA = new LockedObject(); // loA

LockedObject loB = new LockedObject(); // loB

A a = new A(loA);

a.start();

B b = new B(loB);

b.start();

}

}

class A extends Thread {

private LockedObject loA;

public A(LockedObject lo) {

loA = lo;

}

public void run() {

synchronized (loA) {

System.out.println("A.run() first");

Thread.yield();

System.out.println("A.run() second");

}

}

}

class B extends Thread {

private LockedObject loB;

public B(LockedObject lo) {

loB = lo;

}

public void run() {

synchronized (loB) {

System.out.println("B.run()");

}

}

}

class LockedObject {

}

결과는 다음과 같을 때가 많을 것이다.

A.run() first

B.run()

A.run() second

A.run() 메쏘드에서 loA를 lock시켰지만 B.run() 메쏘드에 있는 synchronized 문은 그대로 실행된다. 왜냐하면 거기서는 loB가 lock되었는지를 체크하기 때문이다. loA, loB 두 객체는 모두 LockedObject 클래스의 객체지만 idenity가 다른 두 개의 객체이다.

* + 1. 클래스를 잠그기

그러나 어떤 때는 클래스를 잠그는 것이 필요할 때도 있을지 모른다. Class 객체를 이용하는 것도 하나의 방법이 될 수도 있다. 하나의 클래스마다 Class 객체는 단 하나만 존재하기 때문에 가능하다. 즉 클래스 A를 잠그고 싶으면

synchronized (A.class) {

…

}

와 같이 하면 된다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.start();

B b = new B();

b.start();

}

}

class A extends Thread {

public void run() {

synchronized (LockedClass.class) {

System.out.println("A.run() first");

Thread.yield();

System.out.println("A.run() second");

}

}

}

class B extends Thread {

private LockedClass lc = new LockedClass();

public void run() {

synchronized (lc) {

System.out.println("B.run()");

}

}

}

class LockedClass {

}

결과는 다음과 같이 나올 때도 있을 것이다.

A.run() first

B.run()

A.run() second

LockedClass.class와 lc는 두 개의 별개의 객체다. 그러므로 LockedClass의 Class 객체에 lock을 건다고 해서 LockedClass의 객체인 lc의 lock과는 무관한 것이다.

static 메쏘드를 synchronized화 할 수 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new ClassLock(true).start();

new ClassLock(false).start();

}

}

class ClassLock extends Thread {

private boolean callStatic;

public ClassLock(boolean callStatic) {

this.callStatic = callStatic;

}

public void run() {

if (callStatic)

synchStatic();

else

synchInstance();

}

public static synchronized void synchStatic() {

System.out.println("ClassLock.synchStatic() first");

Thread.yield();

System.out.println("ClassLock.synchStatic() second");

}

public synchronized void synchInstance() {

System.out.println("ClassLock.synchInstance()");

}

}

결과가 다음과 같을 때도 있을 것이다.

ClassLock.synchStatic() first

ClassLock.synchInstance()

ClassLock.synchStatic() second

위와 마찬가지로 정적 메쏘드와 인스턴스 메쏘드가 서로 독립적임을 알 수 있다.

다음 예를 보자.

public class Example {

public static void main(String[] args) {

new ClassLock(true).start();

new ClassLock(false).start();

}

}

class ClassLock extends Thread {

private boolean callStatic;

public ClassLock(boolean callStatic) {

this.callStatic = callStatic;

}

public void run() {

if (callStatic)

synchStatic();

else {

synchronized (ClassLock.class) {

System.out.println("ClassLock.run()");

}

}

}

public static synchronized void synchStatic() {

System.out.println("ClassLock.synchStatic() first");

Thread.yield();

System.out.println("ClassLock.synchStatic() second");

}

}

결과는

ClassLock.synchStatic() first

ClassLock.synchStatic() second

ClassLock.run()

또는

ClassLock.run()

ClassLock.synchStatic() first

ClassLock.synchStatic() second

둘중의 하나일 것이다. 정적 메쏘드를 synchronized로 하자 synchronized 문이 영향을 받는 것을 알 수 있다. 하나의 쓰레드가 lock을 걸자 다른 쓰레드는 lock이 걸려서 실행할 수 없었다.

* + 1. synchronized의 ‘중첩’

synchronized문 또는 synchronized 메쏘드 안에 또 synchronized 문이 올 수 있다. 즉 중첩될 수 있다. 그리고 synchronized 문 또는 synchronized 메쏘드 안에서 synchronized 메쏘드를 부를 수도 있다. 만약 잠기는 객체가 서로 다르다면 하나의 쓰레드가 여러 개의 객체를 잠그는 것이 된다. 같은 객체를 여러 번 잠근다면 잠근 횟수가 기억된다. 그리고 synchronized 문 또는 synchronized 메쏘드를 빠져나올 때마다 그 증가되었던 잠근 횟수가 감소한다.

예를 들어 다음과 같이 사용할 수 있다.

public class Example {

public static void main(String[] args) {

ClassLock cl = new ClassLock(null);

new ClassLock(cl).start();

}

}

class ClassLock extends Thread {

private ClassLock cl;

public ClassLock(ClassLock cl) {

this.cl = cl;

}

public synchronized void run() { // this 객체를 잠근다.

synchronized (cl) { // cl 객체를 잠근다.

method(); // this 객체를 한번더 잠근다.

}

}

public synchronized void method() {

}

}

* 1. 교착 상태
     1. 교착 상태(dead lock)란

미국의 어떤 주에는 다음과 같은 철도법이 있(었)다고 한다. 두 철도가 교차하는 곳에서의 규정이다. 이 때 어떤 열차의 기관사가(예를 들어 남북 방향의 철로에 있는) 다른 열차를(예를 들어 동서 방향의 철로에 있는) 본다면 그 열차가 지나갈 때가지 정차하고 있어야 한다. 문제는 동서 방향의 철로에 있는 기차의 기관사도 똑 같은 규칙에 의해 정차하고 있을 것이라는 것이다. 두 기관사가 만약 규정을 글자그대로 지킨다면 두 기차는 영원히 정차하고 있어야 할 것이다. 물론 사람은 융통성이 있기 때문에 어떻게든 이 문제를 해결할 것이다. 그러나 컴퓨터는(적어도 지금 우리가 다루고 있는 컴퓨터는) 그런 융통성이 없다. 시키면 시키는 대로 하는 것이 컴퓨터다. 그러므로 이와 비슷한 상황이 발생하면 컴퓨터는 진짜로 영원히 정지해 버린다.

교착 상태(dead lock)이 발생하는 시나리오는 대충 이렇다. 누군가가 먼저 쓰레드를 이용한 프로그램을 만든다. 그 다음에 경합 상태(race condition)가 발생한다는 것을 발견한다. 그래서 그 해결책으로 synchronized를 이용한다. 그리고 프로그램을 실행하는 중에 다운됐다는 것을 발견한다.

* + 1. 교착 상태(dead lock)의 예

이제 실제로 교착 상태가 발생하는 예를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

B b = new B();

a.setB(b);

b.setA(a);

a.start();

b.start();

}

}

class A extends Thread {

private B field;

public synchronized void run() {

System.out.println("A.run() first");

Thread.yield();

System.out.println("A.run() second");

field.b(); // 여기서 멈춘다

System.out.println("A.run() third");

}

public void setB(B b) {

field = b;

}

public synchronized void a() {

System.out.println("A.a()");

}

}

class B extends Thread {

private A field;

public synchronized void run() {

System.out.println("B.run() first");

Thread.yield();

System.out.println("B.run() second");

field.a(); // 여기서 멈춘다

System.out.println("B.run() third");

}

public void setA(A a) {

field = a;

}

public synchronized void b() {

System.out.println("B.b()");

}

}

결과는 다음과 같을 것이다.

A.run() first

B.run() first

A.run() second

B.run() second

먼저 A.run()이 실행된다. 그러면서 클래스 A의 객체가 잠긴다. 그 다음에 “A.run() first”가 화면에 뿌려진다. 그 다음에 yield가 실행되기 때문에 다른 쓰레드에게 양보한다. 그래서 B.run()이 실행된다. 그러면서 클래스 B의 객체가 잠긴다. 그 다음에 “B.run() first”가 화면에 뿌려진다. 그 다음에 yield가 실행되기 때문에 다른 쓰레드에게 양보한다. A.run()의 다음 줄이 실행된다. 즉 “A.run() second”가 화면에 뿌려진다. 그 다음에 field.b()가 실행되어야 한다. 그러나 field 즉 클래스 B의 객체는 잠겨져 있다. 그러므로 실행될 수 없다. 그래서 다른 쓰레드로 컨트롤이 넘어간다. 즉 B.run()의 다름 줄이 실행된다. 즉 “B.run() second”가 화면에 뿌려진다. 그 다음에 field.a()가 실행되어야 한다. 그러나 field 즉 클래스 A의 객체는 잠겨져 있다. 그러므로 실행될 수 없다. 클래스 A의 객체의 lock이 풀리려면 A.run()이 모두 실행되어야 한다. 그러나 클래스 B의 객체가 lock이 걸려 있기 때문에 A.run() 메쏘드에 있는 field.b()가 실행될 수 없다. 마찬가지로 클래스 B의 객체의 lock이 풀리려면 B.run()이 모두 실행되어야 한다. 그러나 클래스 A의 객체가 lock이 걸려 있기 때문에 B.run() 메쏘드에 있는 field.a()가 실행될 수 없다. 위에 있는 기관사의 예와 비슷하다.

* + 1. 교착 상태(dead lock)를 어떻게 막을 것인가

첫째, 교착 상태의 발생 여부를 컴파일러가 알 수 없다. 미래에는 어떨지 모르지만 현재의 컴파일러는 그렇게 똑똑하지 못하다. 두번째, 교착 상태를 막으려면 어떻게 해야 하는가 하는 만병통치약이 없다. 이것은 클래스 설계를 잘하려면 어떻게 해야 하는가하는 만병 통치약이 없는 것과 마찬가지이다. 그것은 경험만이 해 줄 수 있는 것이다. 그리고 쓰레드만을 전문적으로 다루는 책도 많은 도움이 될 것이다.

만약 프로그램에 결함이 있다면 경합 상태나 교착 상태가 예측할 수 없게 일어난다. 그래서 얼마동안 잘 돌다가 갑자기 데이터가 이상해지거나(경합 상태) 갑자기 프로그램이 다운되어 버리는(교착 상태) 것이다.

* 1. 쓰레드와 UI
     1. 반응하지 않는 UI

다음 예제를 보자.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private Button stopButton;

private boolean stop = false;

public Example() {

setSize(300, 200);

stopButton = new Button("Stop");

stopButton.addActionListener(this);

setLayout(new FlowLayout());

add(stopButton);

addWindowListener( // 이 부분을 이해하고 싶으면 중첩 타입을 다룬 부분을 참조하라.

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == stopButton) {

System.out.println("Stop Button Clicked");

stop = true;

}

}

public static void main(String[] args) {

Example e = new Example();

e.show();

e.think();

}

public void think() {

System.out.println("I'm Thinking");

for (long i = 0; i < 100; i++) {

if (stop)

break;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(i);

}

System.out.println("Thinking finished");

}

}

몇 초 후에 Stop 버튼을 클릭한다면 결과는 다음과 비슷할 것이다.

I'm Thinking

0

1

2

Stop Button Clicked

3

Thinking finished

예상대로일 것이다.

이번에는 시작하는 버튼도 넣어 보자.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private Button startButton;

private Button stopButton;

private boolean stop = false;

public Example() {

setSize(300, 200);

startButton = new Button("Start");

startButton.addActionListener(this);

stopButton = new Button("Stop");

stopButton.addActionListener(this);

setLayout(new FlowLayout());

add(startButton);

add(stopButton);

addWindowListener( // 이 부분을 이해하고 싶으면 중첩 타입을 다룬 부분을 참조하라.

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

System.exit(0);

}

}

);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == startButton) {

System.out.println("Start Button Clicked");

think();

}

if (e.getSource() == stopButton) {

System.out.println("Stop Button Clicked");

stop = true;

}

}

public static void main(String[] args) {

Example e = new Example();

e.show();

}

public void think() {

System.out.println("I'm Thinking");

for (long i = 0; i < 100; i++) {

if (stop)

break;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(i);

}

System.out.println("Thinking finished");

}

}

Start 버튼을 클릭했다면 결과는 다음과 같을 것이다.

Start Button Clicked

I'm Thinking

0

1

2

3

4

5

…

Stop 버튼이 먹통이 되어 버렸다. 물론 Start 버튼도 그렇다. 먼저 예제에서는 잘 됐는데 무슨 일이 벌어진 걸까? 먼저 첫번째 예제를 약간 고쳤다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private Button stopButton;

private boolean stop = false;

public Example() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Constructor");

setSize(300, 200);

stopButton = new Button("Stop");

stopButton.addActionListener(this);

setLayout(new FlowLayout());

add(stopButton);

addWindowListener( // 이 부분을 이해하고 싶으면 중첩 타입을 다룬 부분을 참조하라.

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Window Closing");

System.exit(0);

}

}

);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == stopButton) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Stop Button Clicked");

stop = true;

}

}

public static void main(String[] args) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : main()");

Example e = new Example();

e.show();

e.think();

}

public void think() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : I'm Thinking");

for (long i = 0; i < 100; i++) {

if (stop)

break;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(i);

}

System.out.println("Thinking finished");

}

}

몇 초후에 Stop버튼을 누르고 윈도우 닫기(우상단의 x버튼)버튼을 눌렀다면 결과는 다음과 비슷할 것이다.

Thread[main,5,main] : main()

Thread[main,5,main] : Constructor

Thread[main,5,main] : I'm Thinking

0

1

2

Thread[AWT-EventQueue-0,6,main] : Stop Button Clicked

3

Thinking finished

Thread[AWT-EventQueue-0,6,main] : Window Closing

두개의 쓰레드가 돌고 있다(그 외에도 몇 개의 쓰레드가 더 돌고 있지만 여기서는 무시한다). think 메쏘드는 main쓰레드에서 돌고 버튼 이벤트를 받는 쓰레드는 AWT-EventQueue-0 쓰레드다. main 쓰레드가 바쁘게 돌고 있었지만 AWT-EventQueue-0 쓰레드는 그것과 독립된 쓰레드이기 때문에 금방 반응할 수 있었다. 대부분의 운영체제는 time slicing을 통해서 여러 쓰레드에게 CPU 시간을 대체로 골고루 나누어 주기 때문이다. 그럼 두번째 예제도 약간 고쳐 보자.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private Button startButton;

private Button stopButton;

private boolean stop = false;

public Example() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Constructor");

setSize(300, 200);

startButton = new Button("Start");

startButton.addActionListener(this);

stopButton = new Button("Stop");

stopButton.addActionListener(this);

setLayout(new FlowLayout());

add(startButton);

add(stopButton);

addWindowListener( // 이 부분을 이해하고 싶으면 중첩 타입을 다룬 부분을 참조하라.

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Window Closing");

System.exit(0);

}

}

);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == startButton) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Start Button Clicked");

think();

}

if (e.getSource() == stopButton) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Stop Button Clicked");

stop = true;

}

}

public static void main(String[] args) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : main()");

Example e = new Example();

e.show();

}

public void think() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : I'm Thinking");

for (long i = 0; i < 100; i++) {

if (stop)

break;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(i);

}

System.out.println("Thinking finished");

}

}

Start 버튼을 클릭했다면 결과는 다음과 비슷할 것이다.

Thread[main,5,main] : main()

Thread[main,5,main] : Constructor

Thread[AWT-EventQueue-0,6,main] : Start Button Clicked

Thread[AWT-EventQueue-0,6,main] : I'm Thinking

0

1

2

3

4

5

…

여기서는 think 메쏘드가 main 쓰레드가 아니라 AWT-EventQueue-0 쓰레드에서 돌고 있다. 왜냐하면 think 메쏘드를 main 메쏘드가 아니라 actionPerformed 메쏘드에서 불렀기 때문이다. think 메쏘드가 AWT-EventQueue-0 쓰레드를 완전히 잡고 있기 때문에 Start, Stop, x 버튼 모두 반응할 수가 없는 것이다. 최소화, 최대화 버튼은 작동하는데 그것은 운영체제 수준에서 작동시키는 듯하다.

* + 1. 반응하는 UI

그럼 위의 예제를 UI가 반응할 수 있도록 고쳐 보자. 핵심은 UI 쓰레드 즉 AWT-EventQueue-0 쓰레드가 아닌 다른 쓰레드에서 think 메쏘드를 돌리는 것이다.

import java.awt.\*;

import java.awt.event.\*;

public class Example extends Frame implements ActionListener {

private Button startButton;

private Button stopButton;

private boolean stop = false;

public Example() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Constructor");

setSize(300, 200);

startButton = new Button("Start");

startButton.addActionListener(this);

stopButton = new Button("Stop");

stopButton.addActionListener(this);

setLayout(new FlowLayout());

add(startButton);

add(stopButton);

addWindowListener( // 이 부분을 이해하고 싶으면 중첩 타입을 다룬 부분을 참조하라.

new WindowAdapter() {

public void windowClosing(WindowEvent e) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Window Closing");

System.exit(0);

}

}

);

}

public void actionPerformed(ActionEvent e) {

if (e.getSource() == startButton) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Start Button Clicked");

Thread thrUser = new Thread(new UserThread(this));

thrUser.start();

}

if (e.getSource() == stopButton) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : Stop Button Clicked");

stop = true;

}

}

public static void main(String[] args) {

Thread thr = Thread.currentThread();

System.out.println(thr + " : main()");

Example e = new Example();

e.show();

}

public void think() {

Thread thr = Thread.currentThread();

System.out.println(thr + " : I'm Thinking");

for (long i = 0; i < 100; i++) {

if (stop)

break;

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(i);

}

System.out.println("Thinking finished");

}

}

class UserThread implements Runnable {

Example e;

public UserThread(Example e) {

this.e = e;

}

public void run() {

e.think();

}

}

결과는 다음과 비슷할 것이다.

Thread[main,5,main] : main()

Thread[main,5,main] : Constructor

Thread[AWT-EventQueue-0,6,main] : Start Button Clicked

Thread[Thread-3,6,main] : I'm Thinking

0

1

2

Thread[AWT-EventQueue-0,6,main] : Stop Button Clicked

3

Thinking finished

Thread[AWT-EventQueue-0,6,main] : Window Closing

이 예제는 쓰레드의 작동을 설명하기 위해 만들어진 예제이다. 이 예제의 클래스 설계는 배울만한 것이 못된다. 이제 think 메쏘드가 Thread-3 쓰레드에서 돈다.

* 1. wait, notify, notifyAll
     1. Object 클래스

Object 클래스에는 wait, notify, notifyAll이란 메쏘드들이 있다. 이 메쏘드들은 쓰레드와 관련이 있기 때문에 Object 클래스를 다루는 장이 아니라 여기서 다룬다.

* + 1. wait와 synchronized

wait 메쏘드는 synchronized 메쏘드 또는 synchronized 문 내에서 또는 그렇지 않더라도 객체에 대한 lock을 가진 상태에서 불러야 한다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

new WaitNotify().start();

}

}

class WaitNotify extends Thread {

public void run() {

method();

}

public void method() {

try {

wait();

} catch (InterruptedException e) {

System.out.println();

}

}

}

결과는 다음과 비슷할 것이다.

java.lang.IllegalMonitorStateException: current thread not owner

at java.lang.Object.wait(Native Method)

at java.lang.Object.wait(Object.java:425)

at WaitNotify.method(Example.java:14)

at WaitNotify.run(Example.java:9)

다음 예를 보자.

public class Example {

public static void main(String[] args) {

new A().start();

new B().start();

new C().start();

}

}

class A extends Thread {

public void run() {

method();

}

public synchronized void method() {

try {

System.out.println("A.method() first");

wait(); // synchronized 메쏘드 안이다.

System.out.println("A.method() second");

} catch (InterruptedException e) {

System.out.println();

}

}

}

class B extends Thread {

public void run() {

method();

}

public void method() {

synchronized (this) {

try {

System.out.println("B.method() first");

wait(); // synchronized 문 안이다.

System.out.println("B.method() second");

} catch (InterruptedException e) {

System.out.println();

}

}

}

}

class C extends Thread {

public synchronized void run() {

method();

}

public void method() {

try {

System.out.println("C.method() first");

wait(); // method() 가 synchronized 메쏘드 안에서 불렸다. 즉 lock을 가지고 있는 상태다.

System.out.println("C.method() second");

} catch (InterruptedException e) {

System.out.println();

}

}

}

결과는 다음과 비슷할 것이다. 그리고 프로그램은 종료되지 않은 상태일 것이다.

A.method() first

B.method() first

C.method() first

lock을 가지고 있는 것만으로는 부족하다. this에 대한 lock을 가지고 있어야 한다. 다음 예를 보자.

public class Example {

public static void main(String[] args) {

B b = new B(null);

new B(b).start();

}

}

class B extends Thread {

B field;

public B(B b) {

field = b;

}

public void run() {

method();

}

public void method() {

synchronized (field) { // this 가 아니다.

try {

System.out.println("B.method() first");

wait();

System.out.println("B.method() second");

} catch (InterruptedException e) {

System.out.println();

}

}

}

}

결과는 다음과 비슷할 것이다.

B.method() first

java.lang.IllegalMonitorStateException: current thread not owner

at java.lang.Object.wait(Native Method)

at java.lang.Object.wait(Object.java:425)

at B.method(Example.java:23)

atB.run(Example.java:16)

notify와 notifyAll도 마찬가지이다. 즉 this에 대한 lock을 가지고 있는 상태여야 한다. 비슷한 코드로 실험해 보기 바란다.

* + 1. notify, notifyAll

wait는 notify또는 notifyAll과 짝을 이뤄서 사용한다. wait는 글자 그대로 기다리게 하는 메쏘드이다. 그럼 무엇을 기다리는가? 바로 누군가가 notify 또는 notifyAll하기를 기다리는 것이다. notify 메쏘드는 wait 메쏘드를 실행해서 blocked상태에 들어간 쓰레드 중 하나를 깨운다. notifyAll 메쏘드는 wait 메쏘드를 실행해서 blocked 상태에 들어간 쓰레드 모두를 깨운다. 여기서 깨운다고 바로 그 쓰레드가 실행되는 것은 아니다. thread scheduler가 CPU 시간을 주어야 실제로 실행되는 것이다. notify나 notifyAll이 해 줄 수 있는 것은 blocked 상태를 runnable 상태로 바꾸어 주는 것이다.

* + 1. wait의 필요성

다음 예를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

IncrementDecrement id = new IncrementDecrement();

Thread thrInc = new Thread(id);

thrInc.setName("Inc");

thrInc.start();

Thread thrDec = new Thread(id);

thrDec.setName("Dec");

thrDec.start();

}

}

class IncrementDecrement implements Runnable {

private long totalSum = 0;

private long incSum = 0;

private long incCount = 0;

private long decSum = 0;

private long decCount = 0;

private Random random = new Random();

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("Inc"))

increment();

else if (str.equals("Dec"))

decrement();

else

System.out.println("바보냐 !!!");

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

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

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

System.out.println("incSum + decSum = " + (incSum+decSum));

System.out.println();

}

public void increment() {

while (true) {

int r = Math.abs(random.nextInt());

incCount++;

incSum += r;

totalSum += r;

if (incCount >= 4000000) // 결과가 '제대로' 안나오면 숫자를 조정하라.

return;

}

}

public void decrement() {

while (true) {

int r = -1 \* Math.abs(random.nextInt());

decCount++;

decSum += r;

totalSum += r;

if (decCount >= 4000000) // 결과가 '제대로' 안나오면 숫자를 조정하라.

return;

}

}

}

결과는 다음과 비슷할 것이다.

incSum = 4295854022581256

decSum = -3726522737250237

totalSum = 575097234249696

incSum + decSum = 569331285331019

incSum = 4295854022581256

decSum = -4293945573088571

totalSum = 7674398411362

incSum + decSum = 1908449492685

여기서 문제는 totalSum과 incSum + decSum이 서로 다르다는 것이다. 코드를 잘 보면 둘이 같아야 되도록 코딩되어 있다. 실제로 위에서 4000000을 아주 작은 수 예를 들어 400으로 줄이면 totalSum과 incSum + decSum이 같게 나올 것이다. 이것이 서로 다른 이유는 race condition 때문이다. 그리고 그 해결책은 synchronized다.

위의 예제를 synchronized를 이용해서 조금 바꾸었다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

IncrementDecrement id = new IncrementDecrement();

Thread thrInc = new Thread(id);

thrInc.setName("Inc");

thrInc.start();

Thread thrDec = new Thread(id);

thrDec.setName("Dec");

thrDec.start();

}

}

class IncrementDecrement implements Runnable {

private long totalSum = 0;

private long incSum = 0;

private long incCount = 0;

private long decSum = 0;

private long decCount = 0;

private Random random = new Random();

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("Inc"))

increment();

else if (str.equals("Dec"))

decrement();

else

System.out.println("바보냐 !!!");

}

public synchronized void increment() {

while (true) {

int r = Math.abs(random.nextInt());

incCount++;

incSum += r;

totalSum += r;

if (incCount >= 4000000) { // 결과가 '제대로' 안나오면 숫자를 조정하라.

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

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

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

System.out.println("incSum + decSum = " + (incSum+decSum));

System.out.println();

return;

}

}

}

public synchronized void decrement() {

while (true) {

int r = -1 \* Math.abs(random.nextInt());

decCount++;

decSum += r;

totalSum += r;

if (decCount >= 4000000) { // 결과가 '제대로' 안나오면 숫자를 조정하라.

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

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

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

System.out.println("incSum + decSum = " + (incSum+decSum));

System.out.println();

return;

}

}

}

}

결과는 다음과 비슷할 것이다.

incSum = 4295203441205639

decSum = 0

totalSum = 4295203441205639

incSum + decSum = 4295203441205639

incSum = 4295203441205639

decSum = -4292860672357415

totalSum = 2342768848224

incSum + decSum = 2342768848224

두 값은 같게 나왔는데 두 쓰레드가 동시적으로 실행되는 것이 아니라 하나가 완전히 마친후 다른 하나가 실행된다. increment 메쏘드나 decrement 메쏘드에 yield 또는 sleep 메쏘드를 넣어도 마찬가지이다. 왜냐하면 yield든 sleep이든 lock을 포기하지 않기 때문이다.

* + 1. wait – notify의 사용

위의 예제를 wait – notify 쌍을 이용해서 고쳤다.

import java.util.\*;

public class Example {

public static void main(String[] args) {

IncrementDecrement id = new IncrementDecrement();

Thread thrInc = new Thread(id);

thrInc.setName("Inc");

thrInc.start();

Thread thrDec = new Thread(id);

thrDec.setName("Dec");

thrDec.start();

}

}

class IncrementDecrement implements Runnable {

private long totalSum = 0;

private long incSum = 0;

private long incCount = 0;

private long decSum = 0;

private long decCount = 0;

private Random random = new Random();

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("Inc"))

increment();

else if (str.equals("Dec"))

decrement();

else

System.out.println("바보냐 !!!");

}

public synchronized void increment() {

while (true) {

int r = Math.abs(random.nextInt());

incCount++;

incSum += r;

totalSum += r;

if (incCount >= 40000) { // 결과가 '제대로' 안나오면 숫자를 조정하라.

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

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

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

System.out.println("incSum + decSum = " + (incSum+decSum));

System.out.println();

return;

}

notify();

if (decCount < 40000 - 1) {

try {

wait();

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

public synchronized void decrement() {

while (true) {

int r = -1 \* Math.abs(random.nextInt());

decCount++;

decSum += r;

totalSum += r;

if (decCount >= 40000) { // 결과가 '제대로' 안나오면 숫자를 조정하라.

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

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

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

System.out.println("incSum + decSum = " + (incSum+decSum));

System.out.println();

return;

}

notify();

if (incCount < 40000 - 1) {

try {

wait();

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

}

결과는 다음과 비슷할 것이다.

incSum = 42917050589561

decSum = -42682333330235

totalSum = 234717259326

incSum + decSum = 234717259326

incSum = 42918311323695

decSum = -42682333330235

totalSum = 235977993460

incSum + decSum = 235977993460

위의 예제에서 두 군데 있는 notify();를 제거해 보자. 그러면 교착 상태(dead lock)의 또 다른 예제가 된다. notify();를 둘 중에 하나만 제거해도 교착 상태가 된다. 이 예제에서는 두 개의 쓰레드 즉 thrInc와 thrDec가 돈다(garbage collection 쓰레드 등은 무시한다). 하나의 쓰레드가 wait 메쏘드를 실행하면 즉 blocked 상태에 들어가면 다른 쓰레드가 notify 메쏘드를 불러서 깨우는 것을 알 수 있다. 만약 notify();가 하나라도 없으면 예를 들어 decrement()에 있는 것을 없애면 thrInc를 깨울 수 없다. 그러면 곧 thrDec가 blocked 상태(wait)로 들어가기 때문에 thrDec도 실행을 멈춰 버린다. thrInc가 정지해 버렸기 때문에 thrDec를 깨울 수 없다.

* 1. deprecated – stop, suspend, resume
     1. deprecated

이 절에 있는 메쏘드들은 deprecated되었다. 즉 사용하지 말 것을 강력하게 권고하는 메쏘드들이다. 이 절을 읽지 않고 그냥 이 메쏘드들을 사용하지 않을 수도 있다. 그러나 이미 누가 사용한 것을 유지 보수해야할 경우도 있다. 그리고 이 절은 읽음으로써 쓰레드의 메커니즘을 더 잘 알 수 있을 것이다.

* + 1. stop

stop 메쏘드는 실행중이던 쓰레드를 정지시키는 메쏘드이다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thr = new Stopped();

thr.start();

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

thr.stop();

}

}

class Stopped extends Thread {

private int count = 0;

public void run() {

while (true) {

System.out.println(count++);

try {

Thread.sleep(999);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

0

1

2

3

4

5

* + 1. stop이 deprecated 된 이유는

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thr = new Stopped();

thr.start();

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

thr.stop();

}

}

class Stopped extends Thread {

private int count = 0;

public void run() {

try {

while (true) {

System.out.println(count++);

try {

Thread.sleep(999);

} catch (InterruptedException e) {

System.out.println(e);

}

}

} catch (Error e) {

System.out.println(e);

throw e; // 이게 없으면 쓰레드가 안죽는다.

}

}

}

결과는 다음과 비슷할 것이다.

0

1

2

3

4

java.lang.ThreadDeath

stop 메쏘드를 실행하면 실제로 일어나는 일은 Error의 서브 클래스인 ThreadDeath가 발생하는 것이다.

stop 메쏘드는 위험하다. 왜냐하면 stop을 당하는 쓰레드는 불시에 stop당하고 그 상황에서 아무것도 할 수 없기 때문이다. 이것은 아래에 나오는 interrupt 메쏘드의 사용법과 다르다. interrupt를 당한 쓰레드는 수습을 할 수 있는 기회가 있다. 여러 쓰레드가 접근하는 중요한 변수는 일관성을 유지하기 위해서 synchronized로 보호한다. 그러나 불시에 stop당한 쓰레드 때문에 그 일관성이 깨지는 것이다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Stopped s = new Stopped();

Thread thrCheck = new Thread(s);

thrCheck.setName("Check");

thrCheck.setPriority(10);

thrCheck.start();

Thread[] threadArray = new Thread[1000];

for (int i = 0; i < 1000; i++) {

threadArray[i] = new Thread(s);

threadArray[i].setPriority(5);

threadArray[i].start();

}

System.out.println("All Threads Started");

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

for (int i = 0; i < 1000; i++) {

threadArray[i].stop();

}

System.out.println("All Threads Stoped");

}

}

class Stopped implements Runnable {

private long a = 0;

private long b = 0;

public void run() {

String str = Thread.currentThread().getName();

if (str.equals("Check")) {

while (true)

check();

} else {

while (true)

increment();

}

}

public synchronized void increment() {

for (int i = 0; i < 10000000; i++)

a++;

for (int i = 0; i < 10000000; i++)

b++;

notifyAll();

try {

wait();

} catch (InterruptedException e) {

System.out.println(e);

}

}

public synchronized void check() {

notifyAll();

String str = Thread.currentThread().getName();

System.out.println(str + " : a = " + a + ", b = " + b);

if (a != b)

System.exit(0);

try {

wait();

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

결과는 다음과 비슷할 것이다.

Check : a = 0, b = 0

All Threads Started

Check : a = 20000000, b = 20000000

Check : a = 40000000, b = 40000000

Check : a = 60000000, b = 60000000

Check : a = 80000000, b = 80000000

Check : a = 100000000, b = 100000000

Check : a = 120000000, b = 120000000

All Threads Stoped

Check : a = 178896447, b = 142815594

이 예에서는 a와 b의 값을 같게 했다. 그러기 위해서 a와 b에 접근하는 메쏘드를 synchronized로 만들었다. 그런데 stop이 그것을 깬 것이다. 왜 그럴까? 만약

for (int i = 0; i < 10000000; i++)

a++;

for (int i = 0; i < 10000000; i++)

b++;

를 실행하는 동안에 stop 당한다면 a와 b의 값이 달라진다. 그리고 이 쓰레드는 ThreadDeath 가 발생하여 죽었기 때문에 이 쓰레드가 가지고 있는 lock이 풀리기 때문에 synchronized 메쏘드인 check 메쏘드가 손상된 객체에 접근할 수 있었던 것이다.

* + 1. stop의 대체 – 변수 사용

stop 메쏘드를 사용하지 말고 객체에 변수를 하나 만들어서 계속 실행해야 하는지 여부를 저장하는 방법을 쓰면 된다. 아니면 interrupt 메쏘드를 사용한다. 먼저 변수를 사용하는 방법을 보자.

public class Example {

public static void main(String[] args) {

Stopped s = new Stopped();

s.start();

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

s.stopPlease();

}

}

class Stopped extends Thread {

private int count = 0;

private boolean stop = false;

public void run() {

while (true) {

if (stop)

break;

System.out.println(count++);

try {

Thread.sleep(999);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

public void stopPlease() {

stop = true;

}

}

결과는 다음과 비슷할 것이다.

0

1

2

3

4

5

* + 1. stop의 대체 – interrupt 메쏘드

이번에는 interrupt 메쏘드를 사용한 예이다.

public class Example {

public static void main(String[] args) {

Stopped s = new Stopped();

s.start();

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

s.interrupt();

}

}

class Stopped extends Thread {

private int count = 0;

public void run() {

while (true) {

System.out.println(count++);

try {

Thread.sleep(999);

} catch (InterruptedException e) {

System.out.println("interrupted");

break;

}

}

}

}

결과는 다음과 비슷할 것이다.

0

1

2

3

4

5

interrupted

sleep 메쏘드가 없어도 interrupt 메쏘드를 사용할 수 있다.

public class Example {

public static void main(String[] args) {

Stopped s = new Stopped();

s.start();

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

s.interrupt();

}

}

class Stopped extends Thread {

public void run() {

while (true) {

if (Thread.interrupted()) {

System.out.println("interrupted");

break;

}

}

}

}

결과는 다음과 비슷할 것이다.(약 5초후에)

interrupted

변수를 사용하는 방법을 쓸 수 있다면 굳이 interrupt를 사용할 필요는 없다. 특히 sleep과 같이 사용할 때는 InterruptedException 즉 예외 상황이 발생하는데 정상적인 상황임에도 예외상황을 사용하는 것은 별로 좋은 생각이 아니다.

* + 1. suspend, resume

다음 예제는 suspend와 resume의 사용법을 보여준다.

public class Example {

public static void main(String[] args) {

Thread th = new Thread(new Suspended());

th.start();

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

System.out.println(e);

}

th.suspend();

try {

Thread.sleep(4000);

} catch (InterruptedException e) {

System.out.println(e);

}

th.resume();

}

}

class Suspended implements Runnable {

private int count = 0;

public void run() {

while (count < 60) {

System.out.print(count + " ");

count++;

if (count % 10 == 0)

System.out.println();

try {

Thread.sleep(100);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 같을 것이다. (여기서는 시간적 요인이 중요하다)

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

20 21 22 23 24 25 26 27 28 29

30 31 32 33 34 35 36 37 38 39

40 41 42 43 44 45 46 47 48 49

50 51 52 53 54 55 56 57 58 59

* + 1. suspend, resume이 deprecated 된 이유는

suspend, resume 메쏘드는 교착 상태(dead lock)가 생기게하는 경향이 있다(deadlock-prone). 만약 suspend된 쓰레드가 lock을 가지고 있는 상태라면 그 lock은 그 쓰레드가 resume될 때까지 풀릴 수 없다. 쓰레드가 suspended되었음에도 lock은 계속 가지고 있는 것이다. 이런 이유 때문에 교착 상태가 생길 가능성이 크다. 이것은 wait와 다르다. wait는 뭔가를 기다릴 때 lock을 포기하고 기다리기 때문이다. 그러므로 suspend – resume를 사용하지 말로 wait – notify를 사용하는 것이 좋다.

1. 스트림
   1. 스트림(stream)이란
      1. 스트림이란

입출력은 여러 가지에 할 수 있다. 콘솔(console), 파일, 네트워크 등등. 이 모든 것의 공통점은 데이터가 순서대로 나열될 수 있다는 것이다. 이것을 추상화한 것이 스트림이다. 물의 흐름처럼 데이터가 흐르는 것이다.

* + 1. 스트림의 종류

자바 스트림에는 byte 스트림과 char 스트림이 있다. byte 스트림이 1바이트 값의 나열이라면 char 스트림은 char 형 즉 2바이트 값의 나열이다.

스트림에는 입력 스트림(input stream)과 출력 스트림(output stream)이 있다. 예를 들어 파일에서 데이터를 읽어온다면 입력 스트림이고 파일에 데이터를 쓴다면 출력 스트림이다.

스트림을 나타내기 위해 네 개의 추상 클래스가 있다. 그것은 다음과 같다.

InputStream : byte input stream

OutputStream : byte output stream

Reader : character input stream

Writer : character output stream

InputStream과 OutputStream은 JDK 1.0 버전에 만들어졌고 Reader와 Writer는 1.1 버전에 추가되었다. Reader와 Writer는 유니코드를 지원하기 위해 만들어졌다. 그리고 설계면에서도 Reader, Writer가 약간 개선되었다. 개선된 것에는 어떤 메쏘드를 추상메쏘드로 할 것인가하는 문제, 쓰레드 문제 등이 있다. 이것은 아래 내용을 읽으면 알 수 있을 것이다.

자바의 스트림에 대한 클래스나 인터페이스는 대부분 java.io 패키지에 있다.

* + 1. 열기와 닫기

스트림을 사용하려면 그 스트림을 열어야(open) 한다. 그리고 스트림을 다 사용했으면 닫아야(close) 한다.

스트림 객체를 만들면 그 스트림이 열린다. 다음은 FileInputStream.java에서 인용한 것이다.

...

public class FileInputStream extends InputStream {

...

public FileInputStream(String name) throws FileNotFoundException {

SecurityManager security = System.getSecurityManager();

if (security != null) {

security.checkRead(name);

}

fd = new FileDescriptor();

open(name); // 여기에 주목하라.

}

...

private native void open(String name) throws FileNotFoundException;

...

}

생성자에서 open 메쏘드를 부른다. open 메쏘드는 native 메쏘드이기 때문에 어떻게 작동하는지는 알 수 없지만 파일을 열어 주는 일을 하는 것임에 틀림없다. 사실 open 메쏘드가 native인 이유는 파일을 여는 일이 플랫폼에 따라 다를 것이기 때문이다.

스트림을 다 사용한 다음에는 반드시 닫아 주어야 한다. 즉 close 메쏘드를 불러 주어야 한다. 파일 같은 자원은 상당히 희소한 자원이기 때문이다. finalize 메쏘드에서도 close 메쏘드를 불러주더라도 그것에 의존하면 안된다. 그 이유는 finalize 메쏘드가 불려지는 방식에 있다. 자세한 것은 쓰레기 수거(garbage collection)를 다루는 부분을 참조하라. 그러므로 사용을 마친 즉시 close 메쏘드를 불러서 스트림을 닫아주어야 한다.

* + 1. 차단(block)

block이란 실행이 진행되지 않고 멈추는 것을 말한다. 중괄호 {}도 블록(block)이라고 하는데 전혀 다른 것을 가리킨다. 헷갈리지 말기 바란다. 많은 입출력에 관련된 메쏘드가 입출력이 안되면 block된다. 어떤 프로그램이 당신이 키보드로 무언가를 입력할 때까지 아무 일도 안하고 기다리고 있다면 그것이 block된 것이다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

System.out.println("Start of main()");

try {

System.in.read();

} catch (IOException e) {

System.out.println(e);

}

System.out.println("End of main()");

}

}

결과는 다음과 비슷할 것이다.

Start of main()

123456

End of main()

여기서 123456은 사용자가 입력한 것이다. 그리고 엔터를 쳐넣어야 한다. 여기서 중요한 것은 엔터키를 치기 전까지는 System.in.read() 메쏘드가 반환(return)하지 않는다는 것이다. 만약 영원히 엔터키를 치지 않는다면 System.in.read() 메쏘드도 영원히 반환하지 않고 멈춰선다. 이런 현상을 block되었다고 한다. (어떤 개발툴에서는 결과가 보이는 부분을 클릭한 후 입력을 해야 한다)

* 1. InputStream 클래스
     1. InputStream의 서브클래스

다음은 InputStream의 서브 클래스들이다.

InputStream

ByteArrayInputStream

FileInputStream

FilterInputStream

BufferedInputStream

DataInputStream

LineNumberInputStream

PushbackInputStream

ObjectInputStream

PipedInputStream

SequenceInputStream

StringBufferInputStream

* + 1. InputStream의 소스

다음은 InputStream.java의 (주석을 제외한) 전체 소스코드이다.

package java.io;

public abstract class InputStream {

private static final int SKIP\_BUFFER\_SIZE = 2048;

private static byte[] skipBuffer;

public abstract int read() throws IOException;

public int read(byte b[]) throws IOException {

return read(b, 0, b.length);

}

public int read(byte b[], int off, int len) throws IOException {

if (b == null) {

throw new NullPointerException();

} else if ((off < 0) || (off > b.length) || (len < 0) ||

((off + len) > b.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return 0;

}

int c = read();

if (c == -1) {

return -1;

}

b[off] = (byte)c;

int i = 1;

try {

for (; i < len ; i++) {

c = read();

if (c == -1) {

break;

}

if (b != null) {

b[off + i] = (byte)c;

}

}

} catch (IOException ee) { // 예외가 무시된다.

}

return i;

}

public long skip(long n) throws IOException {

long remaining = n;

int nr;

if (skipBuffer == null)

skipBuffer = new byte[SKIP\_BUFFER\_SIZE];

byte[] localSkipBuffer = skipBuffer;

if (n <= 0) {

return 0;

}

while (remaining > 0) {

nr = read(localSkipBuffer, 0, (int) Math.min(SKIP\_BUFFER\_SIZE, remaining));

if (nr < 0) {

break;

}

remaining -= nr;

}

return n - remaining;

}

public int available() throws IOException {

return 0;

}

public void close() throws IOException {}

public synchronized void mark(int readlimit) {}

public synchronized void reset() throws IOException {

throw new IOException("mark/reset not supported");

}

public boolean markSupported() {

return false;

}

}

* + 1. InputStream.read()

이 클래스에서 추상 메쏘드는 단 하나 read() 메쏘드이다. 이 메쏘드를 구현해야 구체 클래스(concrete class)로 사용할 수 있다. 즉 객체를 생성할 수 있다. read() 메쏘드의 반환값은 byte 형이 아니라 int 형이다. byte 스트림임을 생각할 때 조금 이상한 일이다. 그 이유는 스트림의 끝에 도달했을 때를 나타내기 위해서이다. 이 메쏘드는 스트림의 끝에 도달하면 –1을 반환한다. 만약 정상적으로 값을 읽어올 때는 0~255의 값을 반환한다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 256; i++)

System.out.println(i + " -> " + (byte)i);

}

}

결과는 다음과 같을 것이다.

0 -> 0

1 -> 1

2 -> 2

3 -> 3

4 -> 4

…

126 -> 126

127 -> 127

128 -> -128

129 -> -127

130 -> -126

131 -> -125

…

251 -> -5

252 -> -4

253 -> -3

254 -> -2

255 -> -1

반환되는 int 값은 위에서처럼 byte값으로 해석할 수 있다.

다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

InputStream in = new FileInputStream("C:/BavaLang/Example.java");

while (true) {

int i = in.read();

if (i == -1) // 파일의 끝일 때

break;

System.out.print((char)i + "(" + i + ")");

}

in.close();

} catch (FileNotFoundException e) {

System.out.println("C:/BavaLang 에 Example.java 라는 파일이 있어야 합니다.");

} catch (IOException e) {

System.out.println(e);

}

}

}

결과는 다음과 비슷할 것이다. 이 때 위의 소스 파일인 Example.java가 C:/BavaLang 디렉토리에 있어야 한다.

i(105)m(109)p(112)o(111)r(114)t(116) (32)j(106)a(97)v(118)a(97).(46)i(105)o(111).(46)\*(42);(59)(13)

(10)(13)

(10)p(112)u(117)b(98)l(108)i(105)c(99) (32)c(99)l(108)a(97)s(115)s(115) (32)E(69)x(120)a(97)m(109)p(112)l(108)e(101) (32){(123)(13)

(10) (9)p(112)u(117)b(98)l(108)i(105)c(99) (32)s(115)t(116)a(97)t(116)i(105)c(99) (32)v(118)o(111)i(105)d(100) (32)m(109)a(97)i(105)n(110)((40)S(83)t(116)r(114)i(105)n(110)g(103)[(91)](93) (32)a(97)r(114)g(103)s(115))(41) (32){(123)(13)

(10) (9) (9)t(116)r(114)y(121) (32){(123)(13)

(10) (9) (9) (9)I(73)n(110)p(112)u(117)t(116)S(83)t(116)r(114)e(101)a(97)m(109) (32)i(105)n(110) (32)=(61) (32)n(110)e(101)w(119) (32)F(70)i(105)l(108)e(101)I(73)n(110)p(112)u(117)t(116)S(83)t(116)r(114)e(101)a(97)m(109)((40)"(34)C(67):(58)/(47)B(66)a(97)v(118)a(97)L(76)a(97)n(110)g(103)/(47)E(69)x(120)a(97)m(109)p(112)l(108)e(101).(46)j(106)a(97)v(118)a(97)"(34))(41);(59)(13)

(10) (9) (9) (9)w(119)h(104)i(105)l(108)e(101) (32)((40)t(116)r(114)u(117)e(101))(41) (32){(123)(13)

(10) (9) (9) (9) (9)i(105)n(110)t(116) (32)i(105) (32)=(61) (32)i(105)n(110).(46)r(114)e(101)a(97)d(100)((40))(41);(59)(13)

(10) (9) (9) (9) (9)i(105)f(102) (32)((40)i(105) (32)=(61)=(61) (32)-(45)1(49))(41)(13)

(10) (9) (9) (9) (9) (9)b(98)r(114)e(101)a(97)k(107);(59)(13)

(10) (9) (9) (9) (9)S(83)y(121)s(115)t(116)e(101)m(109).(46)o(111)u(117)t(116).(46)p(112)r(114)i(105)n(110)t(116)((40)((40)c(99)h(104)a(97)r(114))(41)i(105) (32)+(43) (32)"(34)((40)"(34) (32)+(43) (32)i(105) (32)+(43) (32)"(34))(41)"(34))(41);(59)(13)

(10) (9) (9) (9)}(125)(13)

(10) (9) (9) (9)i(105)n(110).(46)c(99)l(108)o(111)s(115)e(101)((40))(41);(59)(13)

(10) (9) (9)}(125) (32)c(99)a(97)t(116)c(99)h(104) (32)((40)F(70)i(105)l(108)e(101)N(78)o(111)t(116)F(70)o(111)u(117)n(110)d(100)E(69)x(120)c(99)e(101)p(112)t(116)i(105)o(111)n(110) (32)e(101))(41) (32){(123)(13)

(10) (9) (9) (9)S(83)y(121)s(115)t(116)e(101)m(109).(46)o(111)u(117)t(116).(46)p(112)r(114)i(105)n(110)t(116)l(108)n(110)((40)"(34)C(67):(58)/(47)B(66)a(97)v(118)a(97)L(76)a(97)n(110)g(103) (32)¿(191)¡(161) (32)E(69)x(120)a(97)m(109)p(112)l(108)e(101).(46)j(106)a(97)v(118)a(97) (32)¶(182)?(243)´(180)?(194) (32)Æ(198)?(196)?(192)?(207)?(192)?(204) (32)?(192)?(214)¾(190)?(238)¾(190)ß(223) (32)?(199)?(213)´(180)?(207)´(180)?(217).(46)"(34))(41);(59)(13)

(10) (9) (9)}(125) (32)c(99)a(97)t(116)c(99)h(104) (32)((40)I(73)O(79)E(69)x(120)c(99)e(101)p(112)t(116)i(105)o(111)n(110) (32)e(101))(41) (32){(123)(13)

(10) (9) (9) (9)S(83)y(121)s(115)t(116)e(101)m(109).(46)o(111)u(117)t(116).(46)p(112)r(114)i(105)n(110)t(116)l(108)n(110)((40)e(101))(41);(59)(13)

(10) (9) (9)}(125)(13)

(10) (9)}(125)(13)

(10)}(125)

read() 메쏘드는 스트림에 데이터가 없으면 block된다. 즉 입력 스트림에 데이터가 생길 때까지 (또는 예외 상황이 발생할 때까지) 기다린다.

* + 1. InputStream.read(byte[] b, int off, int len), InputStream.read(byte[] b)

read(byte[] b, int off, int len) 메쏘드는 인수로 넘어오는 byte 형 배열에 최대 len만큼의 데이터를 스트림으로부터 채워넣는다. 인덱스 off부터 시작해서. 배열의 나머지 부분은 바뀌지 않는다. 데이터를 읽다가 스트림의 끝을 만나면 거기까지 읽는다. 그리고 실제로 읽은 바이트수를 반환한다. 만약 처음부터 스트림의 끝을 만나면 –1을 반환한다.

이 메쏘드도 block된다. 이것은 InputStream의 소스 코드를 보면 명백히 알 수 있다. read(byte[], int, int) 메쏘드가 read() 메쏘드를 이용하기 때문이다. read() 메쏘드가 block된다면 그 메쏘드를 부르는 read(byte[], int, int) 메쏘드가 block되는 것은 당연하다.

이 메쏘드의 예외 상황의 발생에 대해서 주의해야 할 점이 있다. 그것은 하나라도 데이터를 받으면 그 다음에 발생하는 예외 상황은 무시된다는 것이다. 이것은 소스 코드를 보아도 알 수 있다. catch 블록에 아무것도 안하는 곳이 있다. 즉 발생한 예외를 rethrow하지 않고 무시하는 것이다.

read() 메쏘드를 구현했다면 read(byte[]) 메쏘드나 read(byte[], int, int) 메쏘드를 오버라이딩하지 않아도 사용할 수는 있다. 그러나 read(byte[], int, int)를 재정의하면 더 빠르게 작동하게 할 수도 있으므로 재정의하는 것이 좋다.

read(byte[] b)는 소스코드에서도 알 수 있듯이 read(b, 0, b.length)와 같다. 즉 가능하면 배열을 다 채우는 것이다.

* + 1. InputStream.close()

이 메쏘드는 스트림을 닫아주는 역할을 해야 한다. 이 때 닫은 스트림을 다시 닫으면 아무런 효과도 없어야 한다.

InputStream 클래스에서의 이 메쏘드의 구현을 보면 아무것도 없다. 이것은 당연하다. 스트림을 닫는 것은 스트림에 따라 예를 들어 파일이냐 콘솔이냐에 따라 다르기 때문이다. 문제는 이 메쏘드가 추상 메쏘드가 아니기 때문에 이 메쏘드를 재정의 않아도 컴파일타임 오류가 나지 않는다는 것이다. 이것은 문제가 있다. 스트림을 닫는다는 것은 선택이 아니라 필수여야 하기 때문이다. 이것은 나중에 만들어진 Reader와 Writer 클래스에서 수정되었다. Reader.close()와 Writer.close() 는 추상 메쏘드로 선언되었다.

* + 1. InputStream.skip(long n)

skip 메쏘드는 인자(argument)로 넘어온 개수 만큼 무시하는 것이다. 그러나 그 개수만큼 무시되라는 보장은 없다. 예를 들어 인수의 개수를 무시하기 전에 스트림이 끌날 수 있다. 실제로 무시된 개수가 반환된다. 리턴값이 음수면 전혀 무시되지 않았음을 나타낸다. 소스 코드를 보면 read(byte[], int, int) 메쏘드를 이용해 구현되었음을 알 수 있다.

* + 1. InputStream.available()

이 메쏘드는 block되지 않고도 읽을 수 있는 바이트수를 반환해야 한다. InputStream 클래스에서는 소스에서 볼 수 있듯이 무조건 0을 반환한다. 이것을 재정의하지 않으면 혼돈이 생길수 있을 것이다. 그럼에도 이 메쏘드는 추상 메쏘드가 아니기 때문에 문제의 소지가 있다. Reader 클래스에는 이 메쏘드를 아예 없애 버렸다.

* + 1. InputStream.mark(int readlimit), InputStream.reset(), InputStream.markSupported()

markSupported 메쏘드는 해당 클래스가 mark와 reset 메쏘드를 지원하는지 여부를 알려준다. InputStream 클래스에서는 false를 리턴한다. mark와 reset 메쏘드를 지원한다면 이 메쏘드를 재정의해서 true를 리턴하게 해야 할 것이다.

스트림은 특별한 경우가 아니면 순서가 있는 정보를 순서대로 읽는다. 그런데 경우에 따라서는 전에 읽었던 것을 다시 보고 싶을 수도 있을 것이다. 이럴 때 쓸 수 있는 것이 mark와 reset 메쏘드이다. mark로 표시를 해놓고 필요할 때 reset으로 표시한 곳으로 되돌아가는 것이다. 이 때 mark메쏘드의 매개변수 readlimit는 mark를 한후 몇 바이트가 지나기 전까지 그 표시를 기억하게 할 것인가를 정하는 것이다. 만약 표시를 한 후 readlimit보다 더 많은 수의 정보를 읽었다면 그 표시는 무효가 된다.

* 1. OutputStream 클래스
     1. OutputStream의 서브 클래스

다음은 InputStream의 서브 클래스들이다.

ByteArrayOutputStream

FileOutputStream

FilterOutputStream

BufferedOutputStream

DataOutputStream

PrintStream

ObjectOutputStream

PipedOutputStream

* + 1. OutputStream의 소스 코드

다음은 OutputStream.java의 (주석을 제외한) 전체 소스코드이다.

package java.io;

public abstract class OutputStream {

public abstract void write(int b) throws IOException;

public void write(byte b[]) throws IOException {

write(b, 0, b.length);

}

public void write(byte b[], int off, int len) throws IOException {

if (b == null) {

throw new NullPointerException();

} else if ((off < 0) || (off > b.length) || (len < 0) ||

((off + len) > b.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return;

}

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

write(b[off + i]);

}

}

public void flush() throws IOException {

}

public void close() throws IOException {

}

}

* + 1. OutputStream.write(int b)

이 클래스에서 추상 메쏘드는 단 하나 write (int b) 메쏘드이다. 이 메쏘드를 구현해야 구체 클래스(concrete class)로 사용할 수 있다. 즉 객체를 생성할 수 있다. write (int b) 메쏘드의 매개변수 b는 byte 형이 아니라 int 형이다. 이것은 InputStream.read()의 반환값이 int형인 것과 마찬가지이다. 여기서는 이유는 약간 다르다. 그 이유는 프로그래머의 편의를 위함이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

byte b = 12;

FakeOutputStream out = new FakeOutputStream();

out.write(b + 1); // Error.

}

}

class FakeOutputStream {

public void write(byte b) {

}

}

컴파일이 되지 않을 것이다. 왜냐하면 (b + 1)의 타입이 byte가 아니라 int형이기 때문이다. 자세한 것은 기본형의 형변환을 다루는 부분을 참조하라. 그러므로 위의 예제는 다음과 고쳐야 쓸 수 있다.

public class Example {

public static void main(String[] args) {

byte b = 12;

FakeOutputStream out = new FakeOutputStream();

out.write((byte)(b + 1)); // OK.

}

}

class FakeOutputStream {

public void write(byte b) {

}

}

캐스팅을 해주었다. 이런 귀찮은 점을 완화하려고 아예 int형을 인수로 받는 것이다. 물론 실제 스트림에 쓰여지는 값은 가장 낮은 8 비트값이 쓰인다. 이 메쏘드도 스트림에 쓸 때까지 block된다.

* + 1. OutputStream.write(byte[] b, int off, int length), OutputStream.write(byte[] b)

write(byte[] b, int off, int len) 메쏘드는 인수로 넘어오는 byte 형 배열에서 len만큼의 데이터를 스트림으로 보낸다. 인덱스 off부터 시작해서.

이 메쏘드도 block된다. 이것은 OutputStream의 소스 코드를 보면 명백히 알 수 있다. write (byte[], int, int) 메쏘드가 write(int b) 메쏘드를 이용하기 때문이다. write(int b) 메쏘드가 block된다면 그 메쏘드를 부르는 write(byte[], int, int) 메쏘드가 block되는 것은 당연하다.

write(int b) 메쏘드를 구현했다면 write(byte[]) 메쏘드나 write(byte[], int, int) 메쏘드를 재정의하지 않아도 사용할 수는 있다. 그러나 write(byte[], int, int)를 재정의하면 더 빠르게 작동하게 할 수도 있으므로 재정의하는 것이 좋다.

write(byte[] b)는 소스코드에서도 알 수 있듯이 write(b, 0, b.length)와 같다. 즉 배열의 전부를 스트림에 보내는 것이다.

* + 1. OutputStream.close()

OutputStream.close()는 InputStream.close()와 비슷하다. 그것을 다룬 부분을 참조하라.

이 메쏘드에서는 스트림을 닫기 전에 flush() 메쏘드를 불러줘야 할 것이다.

* + 1. OutputStream.flush()

flush는 버퍼(buffer)와 관련이 있다. 이 메쏘드는 버퍼에 있는 것을 즉시 해당 스트림에 보낸다. 자세한 것은 버퍼를 다루는 부분을 참조하라.

이 메쏘드도 추상 메쏘드가 아닌데 여기서 문제가 발생할 수 있다. Writer.flush() 메쏘드는 추상 메쏘드로 선언되어 있다.

* 1. Reader 클래스
     1. Reader의 서브 클래스

다음은 Reader의 서브 클래스들이다.

BufferedReader

LineNumberReader

CharArrayReader

FilterReader

PushbackReader

InputStreamReader

FileReader

PipedReader

StringReader

* + 1. Reader의 소스 코드

다음은 Reader.java의 (주석을 제외한) 전체 소스코드이다.

package java.io;

public abstract class Reader {

protected Object lock;

protected Reader() {

this.lock = this;

}

protected Reader(Object lock) {

if (lock == null) {

throw new NullPointerException();

}

this.lock = lock;

}

public int read() throws IOException {

char cb[] = new char[1];

if (read(cb, 0, 1) == -1)

return -1;

else

return cb[0];

}

public int read(char cbuf[]) throws IOException {

return read(cbuf, 0, cbuf.length);

}

abstract public int read(char cbuf[], int off, int len) throws IOException;

private static final int maxSkipBufferSize = 8192;

private char skipBuffer[] = null;

public long skip(long n) throws IOException {

if (n < 0L)

throw new IllegalArgumentException("skip value is negative");

int nn = (int) Math.min(n, maxSkipBufferSize);

synchronized (lock) {

if ((skipBuffer == null) || (skipBuffer.length < nn))

skipBuffer = new char[nn];

long r = n;

while (r > 0) {

int nc = read(skipBuffer, 0, (int)Math.min(r, nn));

if (nc == -1)

break;

r -= nc;

}

return n - r;

}

}

public boolean ready() throws IOException {

return false;

}

public boolean markSupported() {

return false;

}

public void mark(int readAheadLimit) throws IOException {

throw new IOException("mark() not supported");

}

public void reset() throws IOException {

throw new IOException("reset() not supported");

}

abstract public void close() throws IOException;

}

* + 1. Reader.read(char[] cbuf, int off, int len), Reader.read(char[] cbuf), Reader.read()

Reader 클래스의 read 메쏘드들은 InputStream 클래스의 read 메쏘드들과 비슷하다. 여기서는 차이점에 대해서만 얘기하겠다. InputStream에서는 read() 메쏘드가 추상 메쏘드였고 다른 read 메쏘드는 이 메쏘드를 바탕으로 쓰여졌다. 여기서는 반대로 read(char[] cbuf, int off, int len) 메쏘드가 추상 메쏘드이다. 파일 등 대부분의 스트림에서 한 두 바이트 단위로 입출력하는 것보다는 더 큰 단위로 입출력하는 것이 훨씬 효율적이기 때문에 이런 방식으로 바꾸었다.

* + 1. Reader.close()

InputStream 클래스와는 다르게 Reader.close() 메쏘드는 추상 메쏘드이다.

* + 1. Reader.skip(long n)

이 메쏘드는 InputStream의 skip 메쏘드와 비슷하다.

* + 1. Reader.ready()

InputStream에는 available 메쏘드가 있었다. available 메쏘드는 얼마나 많은 데이터가 스트림에 있는가를 알려주는 반면 이 메쏘드는 데이터가 하나라도 있는가만을 알려준다. Reader 클래스에서는 false를 리턴한다.

이 메쏘드가 false를 반환한다고 반드시 block되는 것은 아니다. 그 순간에 데이터가 준비될 수도 있기 때문이다.

* + 1. Reader.mark(int readAheadLimit), Reader.reset(), Reader.markSupported()

이 메쏘드들은 InputStream에 있는 메쏘드들과 비슷하다.

* 1. Writer 클래스
     1. Writer의 서브 클래스

다음은 Writer의 서브 클래스들이다.

BufferedWriter

CharArrayWriter

FilterWriter

OutputStreamWriter

FileWriter

PipedWriter

PrintWriter

StringWriter

* + 1. Writer의 소스 코드

다음은 Writer.java의 (주석을 제외한) 전체 소스코드이다.

package java.io;

public abstract class Writer {

private char[] writeBuffer;

private final int writeBufferSize = 1024;

protected Object lock;

protected Writer() {

this.lock = this;

}

protected Writer(Object lock) {

if (lock == null) {

throw new NullPointerException();

}

this.lock = lock;

}

public void write(int c) throws IOException {

synchronized (lock) {

if (writeBuffer == null){

writeBuffer = new char[writeBufferSize];

}

writeBuffer[0] = (char) c;

write(writeBuffer, 0, 1);

}

}

public void write(char cbuf[]) throws IOException {

write(cbuf, 0, cbuf.length);

}

abstract public void write(char cbuf[], int off, int len) throws IOException;

public void write(String str) throws IOException {

write(str, 0, str.length());

}

public void write(String str, int off, int len) throws IOException {

synchronized (lock) {

char cbuf[];

if (len <= writeBufferSize) {

if (writeBuffer == null) {

writeBuffer = new char[writeBufferSize];

}

cbuf = writeBuffer;

} else {

cbuf = new char[len];

}

str.getChars(off, (off + len), cbuf, 0);

write(cbuf, 0, len);

}

}

abstract public void flush() throws IOException;

abstract public void close() throws IOException;

}

* + 1. Writer.write(char[] cbuf, int off, int len), Writer.write(char[] cbuf), Writer.write(int c)

이것에 대해서는 OutputStream의 write 메쏘드들과 Reader의 read 메쏘드들에 대해 참조하라.

* + 1. Writer.write(String str, int off, int len), Writer.write(String str)

이 메쏘드는 다른 write 메쏘드와 비슷하다. 단지 String 객체를 지원하는 면이 다르다.

* + 1. Writer.close()

Reader.close() 메쏘드를 참조하라. 필요하다면 flush() 메쏘드를 불러 주어야 한다.

* + 1. Writer.flush()

InputStream.flush() 메쏘드를 참조하라.

* 1. 필터(filter)
     1. 필터란

담배에 보면 필터가 있다. 이 필터는 원래 담배 연기에서 타르 등을 많이 걸러준다. 담배 연기를 변형시키는 것이다. 담배 연기를 스트림에 흘러가는 데이터라고 생각하면 된다. 필터는 스트림의 데이터를 어떻게든 변화시켜준다.

* + 1. FilterInputStream

다음은 FilterInputStream의 (주석을 제외한) 전체 소스 코드이다.

package java.io;

public class FilterInputStream extends InputStream {

protected InputStream in;

protected FilterInputStream(InputStream in) {

this.in = in;

}

public int read() throws IOException {

return in.read();

}

public int read(byte b[]) throws IOException {

return read(b, 0, b.length);

}

public int read(byte b[], int off, int len) throws IOException {

return in.read(b, off, len);

}

public long skip(long n) throws IOException {

return in.skip(n);

}

public int available() throws IOException {

return in.available();

}

public void close() throws IOException {

in.close();

}

public synchronized void mark(int readlimit) {

in.mark(readlimit);

}

public synchronized void reset() throws IOException {

in.reset();

}

public boolean markSupported() {

return in.markSupported();

}

}

in이라는 필드에 생성자에서 인자로 받은 InputStream을 저장하고 모든 메쏘드가 in에 있는 스트림을 이용해서 작동한다. 단 하나의 예외는 read(byte[]) 메쏘드인데 이 메쏘드도 결국은 read(byte[], int, int) 메쏘드를 부르기 때문에 in의 스트림을 사용하는 것이다. close 메쏘드에서는 in을 닫는다. 필터 스트림을 닫으면 결국 필터에 연결된 스트림도 닫는 것이다.

자 이제 FilterInputStream을 사용해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileInputStream inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

FakeFilterInputStream inFilter = new FakeFilterInputStream(inFile);

while (true) {

int i = inFilter.read();

if (i == -1)

break;

System.out.print((char)i);

}

System.out.println();

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

class FakeFilterInputStream extends InputStream {

protected InputStream in;

protected FakeFilterInputStream(InputStream in) {

this.in = in;

}

public int read() throws IOException {

return in.read();

}

public int read(byte b[]) throws IOException {

return read(b, 0, b.length);

}

public int read(byte b[], int off, int len) throws IOException {

return in.read(b, off, len);

}

public long skip(long n) throws IOException {

return in.skip(n);

}

public int available() throws IOException {

return in.available();

}

public void close() throws IOException {

in.close();

}

public synchronized void mark(int readlimit) {

in.mark(readlimit);

}

public synchronized void reset() throws IOException {

in.reset();

}

public boolean markSupported() {

return in.markSupported();

}

}

결과는 다음과 비슷할 것이다. 이 때 C:/BavaLang 디렉토리에 Zealot.txt 파일이 있어야 한다. 파일에 있는 내용이 그대로 나올 것이다. byte 스트림임으로 파일 내용에 한글은 사용하지 말자.

My life for Aiur.

여기서 FakeFilterInputStream의 소스 코드는 FilterInputStream과 비교해 보자. 클래스 이름을 제외하고는 완전히 같다. 그런데 왜 이런 짓을 했는지 궁금한 사람도 있을 것이다. 그 이유는 그냥 FilterInputStream을 사용하면 생성자가 protected이기 때문에 사용을 할 수 없기 때문이다. 그래서 한 파일에 FakeFilterInputStream의 소스 코드를 넣음으로써 사용할 수 있게 했다. FakeFilterInputStream과 Example 클래스는 모두 디폴트 패키지(unnamed package)에 소속하기 때문에 protected 멤버도 접근할 수 있다. 잘 이해가 안가면 패키지를 다룬 장을 참조하라. 파일에 있는 내용이 그대로 나왔다. 이럴거면 필터를 사용할 필요가 없다.

이번에는 좀더 의미있는 필터를 만들어 보자. 이 필터는 코드를 1씩 증가시킨다. 그래서 예를 들어 M은 N이 된다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileInputStream inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

StupidEncrypter inEncrypter = new StupidEncrypter(inFile);

while (true) {

int value = inEncrypter.read();

if (value == -1)

break;

System.out.print((char)value);

}

System.out.println();

inEncrypter.close();

inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

inEncrypter = new StupidEncrypter(inFile);

byte[] b = new byte[20];

int readBytes = inEncrypter.read(b, 2, 13);

for (int i = 2; i < 2 + readBytes; i++)

System.out.print((char)b[i]);

System.out.println();

inEncrypter.close();

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

class StupidEncrypter extends FilterInputStream {

public StupidEncrypter(InputStream in) {

super(in);

}

public int read() throws IOException {

int i = super.read();

if (i == -1)

return -1;

else

return i + 1;

}

public int read(byte[] b, int off, int len) throws IOException {

int returned = super.read(b, off, len);

for (int i = off; i < off + len; i++) {

b[i]++;

}

return returned;

}

public int read(byte[] b) throws IOException {

return read(b, 0, b.length);

}

}

결과는 다음과 비슷할 것이다.

Nz!mjgf!gps!Bjvs/

Nz!mjgf!gps!B

한심한 수준이지만 나름대로 암호화했다.

FilterInputStream의 서브 클래스에는 BufferedInputStream, CheckedInputStream, CipherInputStream, DataInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream 등이 있다.

* + 1. FilterOutputStream

다음은 FilterOutputStream의 (주석을 제외한) 전체 소스 코드이다.

package java.io;

public class FilterOutputStream extends OutputStream {

protected OutputStream out;

public FilterOutputStream(OutputStream out) {

this.out = out;

}

public void write(int b) throws IOException {

out.write(b);

}

public void write(byte b[]) throws IOException {

write(b, 0, b.length);

}

public void write(byte b[], int off, int len) throws IOException {

if ((off | len | (b.length - (len + off)) | (off + len)) < 0)

throw new IndexOutOfBoundsException();

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

write(b[off + i]);

}

}

public void flush() throws IOException {

out.flush();

}

public void close() throws IOException {

try {

flush();

} catch (IOException ignored) {

}

out.close();

}

}

필터를 닫으면 필터에 연결된 스트림도 닫힌다. 그리고 flush도 마찬가지이다. 필터를 flush하면 결국 필터에 연결된 스트림도 flush되는 것이다. 만약 여러 개의 필터가 연결되어 있다면 끝에 연결된 스트림도 flush 되는 것이다. 또한 close 메쏘드를 보면 닫기 전에 flush를 해주는 것도 알 수 있다. 여기서 주의할 것은 flush하다가 예외 상황이 발생하더라도 무시된다는 것이다. catch절에서 하는 일이 없다. 이것은 문제의 소지가 있다. FilterWriter는 좀 다른 방법으로 처리한다.

다음과 같은 필터의 서브 클래스를 만든다면 좋아할 사람이 없을 것이다.

public class ILoveBug extends FilterOutputStream {

public ILoveBug(OutputStream out) {

super(out);

}

public void flush() throws IOException {

}

public void close() throws IOException {

}

}

중간에 이 필터를 끼워넣으면 flush하는 것과 닫는 것이 잘 안될 것이다.

FilterOutputStream의 서브 클래스에는 BufferedOutputStream, CheckedOutputStream, CipherOutputStream, DataOutputStream, DeflaterOutputStream, DigestOutputStream, PrintStream 등이 있다.

* + 1. FilterReader

FilterInputStream, FilterOutputStream은 추상 클래스가 아닌데 FilterReader는 추상 클래스이다. 이것은 FilterWriter도 마찬가지이다. 사실 FilterInputStream, FilterOutputStream이 생성자를 protected로 만듦으로서 사실상 사용하지 못하게 했음에도 추상 클래스로 선언한 것은 문제가 있다.

FilterReader의 서브 클래스에는 PushbackReader 등이 있다.

* + 1. FilterWriter

FilterWriter의 소스 코드를 보면 close 메쏘드가 FilterOutputStream과는 다른 것을 알 수 있다. 여기서는 flush를 부르지 않는다. 그럼 남아 있는 데이터가 버려진단 말인가? 그렇지 않다. 필터에 연결된 스트림의 close 메쏘드를 부른다. 그 연결된 스트림이 또한 필터라면 그것은 다시 그것에 연결된 스트림의 close 메쏘드를 부를 것이다. 그럼 결국 마지막에는 필터가 아닌 스트림의 close 메쏘드가 불려진다. 그 클래스에서 flush를 불러줄 것이다. 만약 그 클래스를 만든 사람이 어리석은 사람이 아니라면. 그리고 그것을 구현하는 사람이 flush를 실행할 때 예외 상황이 발생한다면 어떻게 할지를 알 수 있을 것이다. 이것은 FilterOutputStream에서처럼 필터가 독단적으로 flush를 부르고 예외 상황이 발생하면 무시하는 것보다 낫다. 필터보다는 마지막에 있는 스트림이 파일인지, 네트워크인지 등등에 대해 알 수 있기 때문에 예외 상황에 대해 더 적당한 선택을 할 수 있다.

FilterReader, FilterWriter는 FilterInputStream, FilterOutputStream에 비해 서브 클래스들이 적다. 그들은 왜 후손을 적게 남겼을까? 그 이유 중에 하나는 압축이나 암호화 같은 기능을 하는 필터가 byte 스트림에만 있기 때문이다. 이것은 그리 놀라운 일이 아니다. 또 하나의 이유는 좀 더 의아심을 불러 일으키는데 BufferedInputStream은 FilterInputStream을 상속받는데 BufferedReader는 FilterReader를 상속받지 않는다. 비슷한 예가 여러 개 있다.

* + 1. 여러 개의 필터의 연결

예제를 보자. 이 예제는 두 개의 필터를 연결해서 사용한다. 하나는 위에서 만들었던 StupidEncrypter이고 나머지 하나는 CaseReverser로 대소문자를 뒤바꿔준다. byte 스트림이기 때문에 유니코드 문자를 제대로 지원하지는 못한다. 영어만 사용한다면 별 문제는 없을 것이다. 억울하면 character 스트림으로도 한번 만들어 보라.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileInputStream inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

StupidEncrypter inEncrypter = new StupidEncrypter(inFile);

CaseReverser inReverser = new CaseReverser(inEncrypter);

while (true) {

int value = inReverser.read();

if (value == -1)

break;

System.out.print((char)value);

}

System.out.println();

inReverser.close();

inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

inEncrypter = new StupidEncrypter(inFile);

inReverser = new CaseReverser(inEncrypter);

byte[] b = new byte[20];

int readBytes = inReverser.read(b, 2, 13);

for (int i = 2; i < 2 + readBytes; i++)

System.out.print((char)b[i]);

System.out.println();

inReverser.close();

inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

inEncrypter = new StupidEncrypter(new CaseReverser(inFile));

while (true) {

int value = inEncrypter.read();

if (value == -1)

break;

System.out.print((char)value);

}

System.out.println();

inEncrypter.close();

inFile = new FileInputStream("C:/BavaLang/Zealot.txt");

inEncrypter = new StupidEncrypter(new CaseReverser(inFile));

b = new byte[20];

readBytes = inEncrypter.read(b, 2, 13);

for (int i = 2; i < 2 + readBytes; i++)

System.out.print((char)b[i]);

System.out.println();

inEncrypter.close();

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

class StupidEncrypter extends FilterInputStream {

public StupidEncrypter(InputStream in) {

super(in);

}

public int read() throws IOException {

int i = super.read();

if (i == -1)

return -1;

else

return i + 1;

}

public int read(byte[] b, int off, int len) throws IOException {

int returned = super.read(b, off, len);

for (int i = off; i < off + len; i++) {

b[i]++;

}

return returned;

}

public int read(byte[] b) throws IOException {

return read(b, 0, b.length);

}

}

class CaseReverser extends FilterInputStream {

public CaseReverser(InputStream in) {

super(in);

}

public int read() throws IOException {

int i = super.read();

if (i == -1)

return -1;

else

return reverse((char)i);

}

public int read(byte[] b, int off, int len) throws IOException {

int returned = super.read(b, off, len);

for (int i = off; i < off + len; i++) {

b[i] = (byte)reverse((char)b[i]);

}

return returned;

}

public int read(byte[] b) throws IOException {

return read(b, 0, b.length);

}

private char reverse(char ch) {

if (Character.isLowerCase(ch))

return Character.toUpperCase(ch);

else if (Character.isUpperCase(ch))

return Character.toLowerCase(ch);

else

return ch;

}

}

Zealot.txt 파일에 “My life for Aiur.”가 있었다면 결과는 아마 다음과 같을 것이다. 당신의 시스템이 아스키 코드를 사용하지 않는다면 결과가 달라질 수도 있다 (메인 프레임이나 수퍼 컴퓨터를 사용한다면 그럴 가능성이 있다).

nZ!MJGF!GPS!bJVS/

nZ!MJGF!GPS!b

nZ!MJGF!GPS!bJVS/

nZ!MJGF!GPS!b

연결 순서와 관계 없이 결과가 똑같다. 그러나 만약 Zealot.txt 파일에 “All fleet is stabilized.”가 있었다면 결과는 아마 다음과 같을 것이다.

bMM!GMFFU!JT!TUBCJMJ{FE/

bMM!GMFFU!JT!

bMM!GMFFU!JT!TUBCJMJ[FE/

bMM!GMFFU!JT!

딱 한군데 다른데 그것은 ‘z’가 변환된 곳이다. 왜 그런지 알고 싶으면 아스키 코드표와 위의 소스코드를 보면서 곰곰히 생각해 보기 바란다.

여러 개의 필터를 연결해서 사용하면 다양한 결과를 얻을 수 있다.

* + 1. FilterXXX를 상속하지 않는 필터 스트림

필터 스트림(filter stream)은 다른 스트림에 연결되어서 무엇인가를 처리해 주는 스트림이다. 그리고 이 필터를 만들기 위해서 네 개의 클래스가 만들어졌다. FilterInputStream, FilterOutputStream, FilterReader, FilterWriter가 그것들이다. 그러나 모든 필터 스트림이 이것들을 상속하지는 않는다. 사실 많은 필터 스트림이 이들 클래스의 서브 클래스가 아니다. 특히 character 스트림에서 그것 필터 스트림들이 많다.

BufferedReader, BufferedWriter, LineNumberReader, PrintWriter, StreamTokenizer 등은 필터이지만 FilterXXX 클래스를 상속하지 않는다. 특히 흥미로운 것은 BufferedInputStream, BufferedOutputStream, LineNumberInputStream, PrintStream등은 FilterXXX 클래스를 상속한다는 것이다. 왜 많은 character 필터 스트림은 FilterXXX 클래스를 조상으로 모시는 것을 거부하는 걸까? FilterXXX 클래스가 필터 스트림을 위해 만들어진 클래스라는 점을 생각할 때 조금 이상한 일이다. 왜 상속 구조를 깨뜨렸는지는 잘 모르겠다. BufferedReader와 BufferedWriter는 out 필드를 private으로 선언해서 서브클래스로부터도 숨기는게 이유가 될 수 있을지도 모른다. 그러면 좀더 안전해지니까. 그리고 StreamTokenizer 같은 경우는 InputSream과 Reader를 모두 처리할 수 있도록 했다. 그나마 InputStream을 받아들이는 생성자는 deprecated되었지만. 원래 FilterInputStream, FilterOutputStream을 만들었을 때의 의도는 그것으로부터 대부분의 필터 스트림을 상속하는 것이었을 것이고 실제로 그렇게 했는데 character 스트림에서 그것이 깨진 걸 보면 무언가 설계상에 문제가 있었던 것 같다.

* 1. 버퍼(buffer)
     1. 버퍼란

하드 디스크와 네크워크는 메모리에 비해서 엄청나게 느리다. 그리고 예를 들어 파일에 쓰거나 읽을 때는 한 바이트씩 찔끔찔끔 읽거나 쓰는 것보다 한꺼번에 여러 바이트(예를 들어 1024 바이트)를 처리하는 것이 훨씬 효율적이다. 왜 그런지는 하드웨어나 운영체제를 다룬 책을 보길 바란다. 아마 밥을 한 톨씩 먹는 것보다 한 숫가락씩 먹는 것이 효율적인것과 비슷한 원리일 것이다. 그래서 그런 입출력 장치에 쓰거나 읽을 때는 버퍼라는 곳에 쌓아 두었다가 어느 정도(위의 예에서는 1024바이트) 차면 한꺼번에 쓰거나 읽는 것이 효율적이다. 그러므로 버퍼를 쓴다고 모든 스트림의 처리가 효율적이 되는 것은 아니다. 예를 들어 ByteArrayInputStream처럼 메모리에서 바로 읽는 스트림일 경우에는 오히려 더 느려질 것이다. 버퍼가 유용하려면 버퍼를 만들고 버퍼에서 처리하는 비용을 보상해주고도 남을 정도로 효율성의 개선이 있어야 한다. 그러려면 입출력 장치와 메모리와의 속도차가 커야 한다. 그리고 입출력 장치의 성격이 한 두 바이트보다는 여러 바이트를 처리하는 것이 훨씬 효율적이어야 한다. 그런데 사실 대부분의 입출력 장치가 이 조건을 만족한다.

* + 1. 버퍼의 사용

다음 예제는 버퍼의 효율성을 보여 준다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileOutputStream outNoBuffer = new FileOutputStream("C:/BavaLang/NoBuffer");

long pre = System.currentTimeMillis();

for (long l = 0; l < 1000000; l++) {

outNoBuffer.write('a');

}

outNoBuffer.close();

System.out.println("No Buffer : " + (System.currentTimeMillis()-pre));

BufferedOutputStream outBuffer = new BufferedOutputStream(new FileOutputStream("C:/BavaLang/Buffer"));

pre = System.currentTimeMillis();

for (long l = 0; l < 1000000; l++) {

outBuffer.write('a');

}

outNoBuffer.close();

System.out.println("Buffer : " + (System.currentTimeMillis()-pre));

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

결과는 다음과 비슷할 것이다. 이때 C:/BavaLang이라는 디렉토리가 있어야 한다.

No Buffer : 7360

Buffer : 110

걸리는 시간에 엄청난 차이가 있다.

* + 1. flush

flush 메쏘드가 존재하는 이유가 바로 버퍼 때문이다. flush는 버퍼에 남아 있는 것을 연결된 스트림에 보내는 것을 말한다.

다음 예제는 flush를 안하면 어떤 일이 발생하는지를 보여 준다. 이 예제에서는 두 개의 스트림 객체를 다 사용하는데 그것은 권할 만한 것은 아니다. 단지 여기서는 설명을 위해서 그렇게 하는 것이다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileWriter outFile = new FileWriter("C:/BavaLang/Unit.txt");

BufferedWriter outBuffer = new BufferedWriter(outFile);

outBuffer.write("Who wanna ");

outFile.write("a piece of meat, ");

outBuffer.write("boy");

outBuffer.close();

} catch (IOException e) {

System.out.println(e);

}

}

}

실행하고 Unit.txt 파일을 보면 다음과 같을 것이다.

a piece of meat, Who wanna boy

그러므로 다음과 같이 고쳐 주어야 한다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileWriter outFile = new FileWriter("C:/BavaLang/Unit.txt");

BufferedWriter outBuffer = new BufferedWriter(outFile);

outBuffer.write("Who wanna ");

outBuffer.flush(); // 이것이 필요하다.

outFile.write("a piece of meat, ");

outBuffer.write("boy");

outBuffer.close();

} catch (IOException e) {

System.out.println(e);

}

}

}

실행하고 Unit.txt 파일을 보면 다음과 같을 것이다.

Who wanna a piece of meat, boy

* 1. 파이프(pipe)
     1. 파이프란

파이프는 일종의 버퍼라고 할 수 있다. 그 버퍼에 누군가가 데이터를 쓰면 다른 누군가가 그 버퍼에 있는 데이터를 읽어서 사용하는 것이다. 파이프를 사용할 때는 보통 두 개의 쓰레드에서 데이터를 주고 받는다. 하나의 쓰레드에서 사용할 수는 있으나 유용성은 없다. 그리고 두 개가 넘는 쓰레드에서도 사용할 수는 있겠지만 상당히 주의하지 않으면 위험한 일이 발생할 수도 있다.

* + 1. 파이프의 연결

두개의 파이프 스트림은 연결을 해야 사용할 수 있다.

파이프를 연결하는 방법에는 여거가지가 있다. 생성자를 이용할 수도 있고 connect 메쏘드를 이용할 수도 있다. 두 개의 파이프 스트림 중 어떤 것을 사용해도 된다.

* + 1. 파이프의 사용

다음 예제는 하나의 쓰레드에서 데이터를 주고 받는 예제이다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

PipedReader pipeIn = new PipedReader();

PipedWriter pipeOut = new PipedWriter();

pipeIn.connect(pipeOut);

pipeOut.write("정찰은 단순히 적의 위치만을 파악하는 것만이 다는 아니다" +

". 적의 위치 파악과 적의 빌드오더를 파악하고 더 나아가서는 " +

"적의 진출을 방해하고 초반 공격등을 위해서 필수적이다. ");

int i;

while ((i = pipeIn.read()) != -1)

System.out.print((char)i);

System.out.println();

} catch (IOException e) {

System.out.println(e);

}

}

}

결과는 아마 다음과 같을 것이다.

정찰은 단순히 적의 위치만을 파악하는 것만이 다는 아니다. 적의 위치 파악과 적의 빌드오더를 파악하고 더 나아가서는 적의 진출을 방해하고 초반 공격등을 위해서 필수적이다.

이번에는 두개의 쓰레드에서 데이터를 주고 받는 예제이다.

import java.io.\*;

import java.util.\*;

public class Example {

public static void main(String[] args) {

try {

PipedReader pipeIn = new PipedReader();

PipedWriter pipeOut = new PipedWriter();

pipeIn.connect(pipeOut);

new LetterConsumer(pipeIn).start();

new RandomProducer(pipeOut).start();

} catch (IOException e) {

System.out.println(e);

}

}

}

class RandomProducer extends Thread {

private PipedWriter out;

public RandomProducer(PipedWriter out) {

this.out = out;

}

public void run() {

Random r = new Random();

while (true) {

int i = r.nextInt(26);

try {

out.write(i + 'a');

} catch (IOException e) {

System.out.print(e);

}

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

class LetterConsumer extends Thread {

private PipedReader in;

public LetterConsumer(PipedReader in) {

this.in = in;

}

public void run() {

System.out.println("LetterConsumer Thread started");

while (true) {

try {

while (in.ready()) {

int i = in.read();

System.out.print((char)i);

}

System.out.println();

} catch (IOException e) {

System.out.println(e);

}

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

}

결과는 다음과 비슷할 것이다.

LetterConsumer Thread started

g

etmbn

tzhyt

rzzen

pdqse

…

* + 1. blocking

만약 파이프에서 읽으려는데 파이프에 데이터가 없거나 파이프에 쓰려고 하는데 파이프의 버퍼가 꽉찬 경우에는 block된다.

* 1. 스트림과 쓰레드
     1. byte 스트림

여기 있는 내용을 이해하려면 쓰레드에 대해 어느 정도 알고 있어야 한다. 잘 모르겠으면 쓰레드를 다룬 부분을 먼저 보기 바란다.

byte 스트림은 스트림 객체 자체의 lock을 이용한다. 주로 synchronized 메쏘드를 이용한다. 예를 들어 다음은 ByteArrayOutputStream.java에서 인용한 것이다.

package java.io;

public class ByteArrayOutputStream extends OutputStream {

...

public synchronized void write(int b) {

...

}

public synchronized void write(byte b[], int off, int len) {

if ((off < 0) || (off > b.length) || (len < 0) ||

((off + len) > b.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return;

}

int newcount = count + len;

if (newcount > buf.length) {

byte newbuf[] = new byte[Math.max(buf.length << 1, newcount)];

System.arraycopy(buf, 0, newbuf, 0, count);

buf = newbuf;

}

System.arraycopy(b, off, buf, count, len);

count = newcount;

}

public synchronized void writeTo(OutputStream out) throws IOException {

out.write(buf, 0, count);

}

public synchronized void reset() {

count = 0;

}

public synchronized byte toByteArray()[] {

...

}

public int size() {

return count;

}

public String toString() {

return new String(buf, 0, count);

}

public String toString(String enc) throws UnsupportedEncodingException {

return new String(buf, 0, count, enc);

}

public String toString(int hibyte) {

return new String(buf, hibyte, 0, count);

}

public void close() throws IOException {

}

}

많은 메쏘드가 synchronized로 선언되어 있다. 당신이 실제로 어떤 스트림 클래스를 만든다면 이렇게 만들어야 한다. 어떤 곳에서 race condition이 생길지를 잘 생각해서 synchronized시켜야 한다는 말이다. 만약 이렇게 하지 않으면 어떤 일이 발생할까? 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

OutputStream out = new GoodStream(1000);

for (int i = 0; i < 3; i++) {

String str = "<";

for (int j = 0; j < 10; j++) {

str += i;

}

str += ">";

Thread t = new Thread(new WordWritingThread(out, str));

t.start();

}

try {

Thread.sleep(10000); // 세 개의 쓰레드가 모두 끝날 때까지 충분히 잔다.

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(out);

}

}

class WordWritingThread implements Runnable{

private OutputStream out;

private String word;

public WordWritingThread(OutputStream out, String word) {

this.out = out;

this.word = word;

}

public void run() {

byte[] b = word.getBytes();

for (long l = 0; l < 5; l++) {

try {

out.write(b);

} catch (IOException e) {

System.out.println(e);

}

}

System.out.println(word + " finished");

}

}

class GoodStream extends ByteArrayOutputStream {

public GoodStream(int size) {

super(size);

}

public synchronized void write(byte b[], int off, int len) {

if ((off < 0) || (off > b.length) || (len < 0) ||

((off + len) > b.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return;

}

int newcount = count + len;

if (newcount > buf.length) {

byte newbuf[] = new byte[Math.max(buf.length << 1, newcount)];

System.arraycopy(buf, 0, newbuf, 0, count);

buf = newbuf;

}

////////////////////////////////////////////////////////

// 여기를 좀 고쳤다. 바로 아래 주석처리된 것이 원래 ByteArrayOutputStream에 쓰인 코드이다.

// System.arraycopy(b, off, buf, count, len);

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

buf[count + i] = b[off + i];

try {

Thread.sleep(10);

} catch (InterruptedException e) {

System.out.println(e);

}

}

////////////////////////////////////////////////////////

count = newcount;

}

}

결과는 다음과 비슷할 것이다.

<0000000000> finished

<1111111111> finished

<2222222222> finished

<0000000000><1111111111><2222222222><0000000000><1111111111><2222222222><0000000000><1111111111><2222222222><0000000000><1111111111><2222222222><0000000000><1111111111><2222222222>

이것은 ByteArrayStreamOutput 클래스의 상당히 비효율적인 버전이다. 그러나 느린 것 빼면 잘 작동한다. 여기서 <>안에 묶인 것이 제대로 보존됨을 볼 수 있다. 이것은 write(byte[], int, int) 메쏘드를 한번 실행했을 때 스트림에 보낸 것이다. 제대로 보존되었다는 것이 무슨 뜻인지 잘 모르겠으면 다음 예제를 실행해 보면 된다. 위의 예제를 아주 약간 바꾸었다. write 메쏘드에서 synchronized 키워드를 빼 버린 것이다. 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

OutputStream out = new BadStream(1000);

for (int i = 0; i < 3; i++) {

String str = "<";

for (int j = 0; j < 10; j++) {

str += i;

}

str += ">";

Thread t = new Thread(new WordWritingThread(out, str));

t.start();

}

try {

Thread.sleep(10000); // 세 개의 쓰레드가 모두 끝날 때까지 충분히 잔다.

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println(out);

}

}

class WordWritingThread implements Runnable{

private OutputStream out;

private String word;

public WordWritingThread(OutputStream out, String word) {

this.out = out;

this.word = word;

}

public void run() {

byte[] b = word.getBytes();

for (long l = 0; l < 5; l++) {

try {

out.write(b);

} catch (IOException e) {

System.out.println(e);

}

}

System.out.println(word + " finished");

}

}

class BadStream extends ByteArrayOutputStream {

public BadStream(int size) {

super(size);

}

public void write(byte b[], int off, int len) { // 여기에 주목하라.

if ((off < 0) || (off > b.length) || (len < 0) ||

((off + len) > b.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return;

}

int newcount = count + len;

if (newcount > buf.length) {

byte newbuf[] = new byte[Math.max(buf.length << 1, newcount)];

System.arraycopy(buf, 0, newbuf, 0, count);

buf = newbuf;

}

////////////////////////////////////////////////////////

// 여기를 좀 고쳤다. 바로 아래 주석처리된 것이 원래 ByteArrayOutputStream에 쓰인 코드이다.

// System.arraycopy(b, off, buf, count, len);

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

buf[count + i] = b[off + i];

try {

Thread.sleep(10);

} catch (InterruptedException e) {

System.out.println(e);

}

}

////////////////////////////////////////////////////////

count = newcount;

}

}

결과는 다음과 비슷할 것이다.

<1111111111> finished

<2222222222> finished

<0000000000> finished

<2211011010><0001101101><0101110000><1010100100><1000222220>

데이터가 엉망으로 섞였을 뿐 아니라 위에서는 열 다섯개의 ‘단어’가 쓰여졌는데 여기서는 개수도 줄었다. count 필드와 관련해서도 race condition이 일어났기 때문이다.

* + 1. character 스트림

character 스트림은 byte 스트림과 약간 다르다. 위에서 Reader와 Writer의 소스 코드를 보면

protected Object lock;

으로 lock이라는 Object 형 protected 필드를 선언한 것을 볼 수 있을 것이다. 그리고 Reader와 Writer 클래스는 각각 두개의 생성자가 있다. 하나는 인자가 없는 생성자이고 하나는 Object형 인자가 있는 생성자이다. 소스 코드를 보면 인자가 없는 생성자는 lock 필드에 this를 대입하는 것을 알 수 있다. 특별히 지정하지 않으면 byte 스트림처럼 this를 잠그는 것이다. 그러나 인자를 받는 생성자를 잠글 객체를 지정할 수 있다. 한마디로 하면 character 스트림은 선택의 여지가 있다. 다시 말해서 더 유연하다.

다음은 CharArrayWriter.java에서 인용한 것이다.

package java.io;

public class CharArrayWriter extends Writer {

...

public CharArrayWriter() {

this(32);

}

public CharArrayWriter(int initialSize) {

if (initialSize < 0) {

throw new IllegalArgumentException("Negative initial size: "

+ initialSize);

}

buf = new char[initialSize];

}

public void write(int c) {

synchronized (lock) {

...

}

}

public void write(char c[], int off, int len) {

if ((off < 0) || (off > c.length) || (len < 0) ||

((off + len) > c.length) || ((off + len) < 0)) {

throw new IndexOutOfBoundsException();

} else if (len == 0) {

return;

}

synchronized (lock) {

...

}

}

public void write(String str, int off, int len) {

synchronized (lock) {

...

}

}

public void writeTo(Writer out) throws IOException {

synchronized (lock) {

out.write(buf, 0, count);

}

}

public void reset() {

count = 0;

}

public char toCharArray()[] {

synchronized (lock) {

...

}

}

public int size() {

return count;

}

public String toString() {

synchronized (lock) {

return new String(buf, 0, count);

}

}

public void flush() { }

public void close() { }

}

synchronized 메쏘드를 사용하지 않고 synchronized 문을 사용했다. 그리고 lock 필드를 잠근다. 이런 식으로 해야 한다. 다시 말하면 synchronized 메쏘드를 사용해서도 안되고 synchronized문에서 this를 잠그는 것도 안된다. synchronized 메쏘드가 결국은 this를 잠그는 것이기 때문에 둘은 마찬가지이다. 그 스트림의 사용자가 this를 잠그는 것을 선택할지 다른 객체를 잠그는 것을 선택할지 알 수 없기 때문이다. CharArrayWriter는 this를 lock으로 사용한다. 이것은 CharArrayWriter의 두 개의 생성자를 보면 알 수 있다. 그러나 StringWriter는 this가 아닌 다른 객체를 lock으로 사용한다. 다음은 StringWriter.java에서 인용한 것이다.

package java.io;

public class StringWriter extends Writer {

private StringBuffer buf;

public StringWriter() {

buf = new StringBuffer();

lock = buf; // 여기에 주목하라.

}

public StringWriter(int initialSize) {

if (initialSize < 0) {

throw new IllegalArgumentException("Negative buffer size");

}

buf = new StringBuffer(initialSize);

lock = buf; // 여기에 주목하라.

}

...

}

1. 직렬화
   1. 직렬화(serialization)란
      1. 직렬화와 desirialization

직렬화란 객체를 일련의 byte값으로 즉 byte 스트림으로 바꾸는 것이다. 사실 객체 자체가 메모리에 일련의 byte 값으로 저장되어 있다. 그럼 왜 직렬화를 할까? 먼저 하나의 객체는 메모리 상에 일련의 연속된 byte값으로 저장되어 있지만 문제가 그리 간단하지가 않다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.intValue = 123;

a.obj = new Object();

System.out.println(a.hashCode());

System.out.println(a.obj.hashCode());

}

}

class A {

public Object obj;

public int intValue;

}

결과는 다음과 비슷할 것이다.

3997960

971877

Object.hashCode()는 보통 메모리 번지를 이용해서 구현한다. 그러므로 그 값은 그 객체가 저장된 번지수를 가리킨다(대체로 그렇지만 아닐 수도 있다). 결과값에서 알수 있듯이 a와 a.obj는 메모리의 전혀 다른 부분에 저장되어 있다. 이것을 그림으로 나타내면 그림 X와 같다.

위의 예에서 변수 a가 가리키는 객체를 제대로 직렬화하려면 a.obj 객체도 포함되어야 한다. 이 예제에서는 클래스 A가 단순했지만 복잡한 클래스에서는 수십, 수백개의 흩어진 객체들을 모두 모아 직렬화해야 한다. 즉 단순히 그 객체가 차지하는 메모리에 있는 것만을 이용할 수 없다. 연결된 모든 객체(object graph, web of objects)를 다 찾아내서 직렬화해야 하는 것이다.

직렬화(serialization)한 것을 다른 매체 또는 메모리에 넣었다가 다시 객체로 사용하고 싶으면 그것을 객체로 복원해야 한다. 즉 직렬화의 반대과정이 필요하다. 이것을 역직렬화(deserialization)라고 한다.

* + 1. 직렬화의 쓰임

그렇다면 이렇게 객체에 대한 정보를 일렬로 늘어놓아서 어쩌겠다는 것인가? 이것은 여러 가지로 쓰일 수 있다. 객체를 파일 또는 데이터베이스에 저장하고 싶을 때, 네트워크로 객체를 보낼 때(예를 들어 RMI), JavaBeans 등에 쓰인다. 그리고 직렬화를 이용해 깊은 복제를 할 수도 있다. 깊은 복제에 대해서는 Object 클래스를 다루는 부분을 참조하라. 이 책은 RMI에 대해서는 다루지 않는다.

* 1. Default Serialization
     1. ObjectInputStream.readObject(), ObjectOutputStream.writeObject(Object obj)

다음 예제는 Serialization을 하는 것을 보여준다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

String str = "Birdie Poem\n\n" +

"Birdie, birdie in the sky\n" +

"Dropped some white stuff in my eye,\n" +

"I'm a big girl I won't cry,\n" +

"I'm just glad that cows don't fly.";

try {

FileOutputStream outFile = new FileOutputStream("C:/BavaLang/Serialize");

ObjectOutputStream out = new ObjectOutputStream(outFile);

out.writeObject(str);

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

실행을 하면 C:/BavaLang에 Serialize라는 파일이 생성될 것이다. 이 때 BavaLang이라는 디렉토리가 존재하고 있어야 한다.

이번에는 이 파일에 있는 객체를 복원해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

try {

FileInputStream inFile = new FileInputStream("C:/BavaLang/Serialize");

ObjectInputStream in = new ObjectInputStream(inFile);

String str = (String)in.readObject();

System.out.println(str);

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (ClassNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

결과는 다음과 같을 것이다.

Birdie Poem

Birdie, birdie in the sky

Dropped some white stuff in my eye,

I'm a big girl I won't cry,

I'm just glad that cows don't fly.

여기서는 파일을 사용했는데 네트워크 등 다른 것으로도 가능하다.

* + 1. Serializable 인터페이스

Serializable 인터페이스에 대해 알아보자. 다음은 Serializable.java의 (주석을 제외한)전체 소스 코드이다.

package java.io;

public interface Serializable {

}

아무것도 정의되지 않은 인터페이스이다. 이런 것을 marker 인터페이스라고 한다. 가상 머신은 이 인터페이스가 구현되어 있으면 직렬화를 할 수 있다는 것을 알아낸다. 이 인터페이스를 구현한 클래스의 서브클래스도 당연히 직렬화가 가능하다.

위의 예제에서 String 객체가 직렬화되었는데 String 클래스는 Serializable 인터페이스를 구현하고 있다. 그럼 Serializable 인터페이스를 구현하지 않은 클래스의 객체를 직렬화해 보자. Object 클래스는 Serializable 인터페이스를 구현하지 않는다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

Object obj = new Object();

try {

FileOutputStream outFile = new FileOutputStream("C:/BavaLang/Serialize");

ObjectOutputStream out = new ObjectOutputStream(outFile);

out.writeObject(obj);

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

결과는 다음과 비슷할 것이다.

java.io.NotSerializableException: java.lang.Object

여기서 중요한 것은 변수의 타입이 아니라 실제 객체가 문제가 즉 컴파일 타임 타입이 중요한 것이 아니라 런타임 타입이 중요하다. 다음 예제를 실행해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

Object obj = "Birdie Poem\n\n" +

"Birdie, birdie in the sky\n" +

"Dropped some white stuff in my eye,\n" +

"I'm a big girl I won't cry,\n" +

"I'm just glad that cows don't fly.";

try {

FileOutputStream outFile = new FileOutputStream("C:/BavaLang/Serialize");

ObjectOutputStream out = new ObjectOutputStream(outFile);

out.writeObject(obj);

} catch (FileNotFoundException e) {

System.out.println(e);

} catch (IOException e) {

System.out.println(e);

}

}

}

잘 실행될 것이다. 기존의 API에 있는 클래스들을 살펴보면 어떤 클래스는 Serializable 인터페이스를 구현했고 어떤 클래스는 그렇지 않다. 클래스의 성격에 따라 직렬화할만한 클래스는 Serializable 인터페이스를 구현했다. 배열도 직렬화할 수 있다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

Class c = int[].class;

Class[] interfaces = c.getInterfaces();

for (int i = 0; i < interfaces.length; i++)

System.out.println(interfaces[i]);

}

}

결과는 다음과 비슷할 것이다.

interface java.lang.Cloneable

interface java.io.Serializable

배열도 Serializable 인터페이스를 구현하는 것을 볼 수 있다. 이것은 상식적으로 생각할 때 당연하다.

* + 1. object graph(web of objects)

하나의 객체를 직렬화하는 것은 위에서 말했듯이 단순하지가 않다. 그 객체의 필드가 참조하는 객체도 직렬화해야 하기 때문이다. 물론 참조된 객체의 필드도 어떤 객체를 참조한다면 그 객체도 직렬화해야 한다. 이 과정은 처음에 직렬화하고자 했던 객체에서 연결된 모든 객체가 직렬화될 때까지 계속된다. 이것이 무엇을 뜻하는지 알아보자. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

A a = new A();

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(a);

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject();

System.out.print(reconstructedA.str);

System.out.print(reconstructedA.b.str);

System.out.print(reconstructedA.b.anotherObj);

System.out.print(reconstructedA.b.c.str);

System.out.print(((C)reconstructedA.b.obj).str);

}

}

class A implements Serializable {

B b = new B();

String str = "Birdie Poem\n\n";

}

class B implements Serializable {

C c = new C("I'm a big girl I won't cry,\n");

Object obj = new C("I'm just glad that cows don't fly.");

String str = "Birdie, birdie in the sky\n";

Object anotherObj = "Dropped some white stuff in my eye,\n";

}

class C implements Serializable {

String str;

C(String str) {

this.str = str;

}

}

결과는 다음과 같을 것이다.

Birdie Poem

Birdie, birdie in the sky

Dropped some white stuff in my eye,

I'm a big girl I won't cry,

I'm just glad that cows don't fly.

위의 예제에서 클래스 A, B, C는 모두 Serializable 인터페이스를 구현했다. 하나라도 구현하지 않은 클래스가 있으면 잘 작동하지 않을 것이다. 그 객체를 직렬화할 수 없기 때문이다. 못믿겠으면 한번 실험해 보기 바란다. 그리고 이 예제에서 B.obj 필드와 B.anotherObj 필드가 Object 형임에도(Object 클래스는 Serializable 인터페이스를 구현하지 않는다) 실제 가리키는 객체(런타임 객체)가 Serializable 인터페이스를 구현한 클래스의 객체이므로 오류 없이 실행되었다.

직렬화 메커니즘은 꽤 영리하다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

A a = new A();

C c = new C();

a.first = new B(c);

a.second = new B(c);

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(a);

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject();

System.out.print(a.first.c == a.second.c);

}

}

class A implements Serializable {

B first;

B second;

}

class B implements Serializable {

C c;

B(C c) {

this.c = c;

}

}

class C implements Serializable {

}

결과는 다음과 같을 것이다.

true

메모리의 위치는 다르지만 다이아몬드 모양의 object graph가 복원되는 것을 알 수 있다.

다음 예제를 보자. 위의 예제와 비슷하지만 writeObject 메쏘드를 두번 사용했다.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

C c = new C();

B first = new B(c, "First");

B second = new B(c, "Second");

System.out.println(first.c == second.c);

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(first);

out.writeObject(second);

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

B reconstructedFirst = (B)in.readObject();

B reconstructedSecond = (B)in.readObject();

System.out.println(reconstructedFirst.str);

System.out.println(reconstructedSecond.str);

System.out.println(reconstructedFirst.c == reconstructedSecond.c);

System.out.println(c == reconstructedSecond.c);

}

}

class B implements Serializable {

String str;

C c;

B(C c, String str) {

this.str = str;

this.c = c;

}

}

class C implements Serializable {

}

결과는 다음과 같을 것이다.

true

First

Second

true

false

객체들의 관계를 역시 잘 보존했다. 마지막에는 false가 나왔는데 deserialization은 새로운 객체를 만들어 내는 것이기 때문에 당연하다. 이 점 헷갈리지 말기 바란다.

* + 1. transient 필드와 static 필드

정적 필드는 직렬화되지 않으며 역직렬화할 때도 영향을 받지 않는다. 정적 필드는 하나의 객체에 속하는 것이 아니라 클래스에 속하는 변수이기 때문이다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

A a = new A();

A.staticField = 999;

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(a);

A.staticField = 666;

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject();

System.out.println(A.staticField);

}

}

class A implements Serializable {

static int staticField;

}

결과는 다음과 같을 것이다.

666

transient라는 키워드는 직렬화 때문에 만들어진 것이다. transient에는 일시적인, 덧없는, 순간적인 등등의 뜻이 있다. 이것은 파일이나 데이터베이스에 저장하기 위해 직렬화하는 경우를 생각하면 무슨 뜻인지 분명해질 것이다. transient 필드는 직렬화되지 않으므로 파일 등에 저장되지 않으므로 일시적인(transient) 것이다. transient 필드는 직렬화되지 않으므로 역직렬화될 때에는 그 변수의 타입에 맞는 디폴트 값(default value)으로 정해진다. 디폴트 값이란 배열이 만들어졌을 때나 필드가 만들어졌으로 특별히 초기화되지 않았을 때 정해지는 값이다. 숫자를 나타낸다면 0, 객체는 null, boolean일 때는 false이다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

A a = new A();

a.transientInt = 1234;

a.transientBoolean = true;

a.transientString = "String";

a.intValue = 1234;

a.booleanValue = true;

a.stringValue = "String";

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(a);

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject();

System.out.println(reconstructedA.transientInt);

System.out.println(reconstructedA.transientBoolean);

System.out.println(reconstructedA.transientString);

System.out.println(reconstructedA.intValue);

System.out.println(reconstructedA.booleanValue);

System.out.println(reconstructedA.stringValue);

}

}

class A implements Serializable {

transient int transientInt;

transient boolean transientBoolean;

transient String transientString;

int intValue;

boolean booleanValue;

String stringValue;

}

결과는 다음과 같을 것이다.

0

false

null

1234

true

String

필드를 transient로 선언하는 이유는 여러 가지가 있을 수 있다. 직렬화해서는 안되는 정보가 있을 수 있다. 예를 들어 비밀번호(password) 같은 정보가 파일에 저장되거나 네트워크로 왔다갔다 한다면 문제가 있을 것이다. 또는 직렬화하지 않는 것이 더 효율적일 수도 있다. 예를 들어 금방 조합될 수 있는 정보를 네트워크로 보내는 것은 비효율적이다.

* + 1. 직렬화와 생성자

다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

A a = new A();

a.instanceField = 999;

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

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

out.writeObject(a);

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

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

A reconstructedA = (A)in.readObject();

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

System.out.println(a.instanceField);

}

}

class A implements Serializable {

static int staticField = initializerOfStaticField();

int instanceField = initializerOfInstanceField();

static {

System.out.println("static initializer");

}

{

System.out.println("instance initializer");

}

A() {

System.out.println("costructor");

}

static int initializerOfStaticField() {

System.out.println("initializer of static field");

return 1234;

}

int initializerOfInstanceField() {

System.out.println("initializer of instance field");

return 4321;

}

}

결과는 다음과 같을 것이다.

initializer of static field

static initializer

initializer of instance field

instance initializer

costructor

------ Serialization started -------

------ Serialization finished -------

------- Deserialization finished ------

999

역직렬화하는 과정에서 생성자, 정적 초기화 블록, 인스턴스 초기화 블록, 정적 필드 초기화문, 인스턴스 변수 초기화문 어떤 것도 실행되지 않는다.

* + 1. 상속과 직렬화

직렬화되는 객체의 수퍼 클래스에 대해 알아보자. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

try {

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(s);

} catch (IOException e) {

System.out.println("serialization : " + e);

}

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

try {

ObjectInputStream in = new ObjectInputStream(inMemory);

in.readObject();

} catch (ClassNotFoundException e) {

System.out.println("deserialization : " + e);

} catch (IOException e) {

System.out.println("deserialization : " + e);

}

System.out.println("end of main()");

}

}

class SuperClass {

SuperClass(int i) {}

}

class SubClass extends SuperClass implements Serializable {

SubClass () {

super(123);

}

}

결과는 다음과 비슷할 것이다.

deserialization : java.io.InvalidClassException: SubClass; no valid constructor

end of main()

InvalidClassException은 IOException의 서브 클래스이다. 위의 예제에서 SuperClass가 Serializable 인터페이스를 구현하게 하면 즉

class SuperClass implements Serializable {

과 같이 고치면 결과는 다음과 같을 것이다.

end of main()

또는 SuperClass에 인자가 없는(no-argument) 생성자를 추가하면 즉

import java.io.\*;

public class Example {

public static void main(String[] args) {

SubClass s = new SubClass();

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

try {

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(s);

} catch (IOException e) {

System.out.println("serialization : " + e);

}

System.out.println("deserialization started");

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

try {

ObjectInputStream in = new ObjectInputStream(inMemory);

in.readObject();

} catch (ClassNotFoundException e) {

System.out.println("deserialization : " + e);

} catch (IOException e) {

System.out.println("deserialization : " + e);

}

System.out.println("end of main()");

}

}

class SuperClass {

int field = initializer();

{

System.out.println("SuperClass : instance initializer");

}

SuperClass() { // 인수가 없는(no-argument) 생성자 추가.

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

}

SuperClass(int i) {}

int initializer() {

System.out.println("SuperClass : initializer for instance field");

return 123;

}

}

class SubClass extends SuperClass implements Serializable {

SubClass () {

super(123);

}

}

와 같이 고치면 결과는 다음과 같을 것이다.

SuperClass : initializer for instance field

SuperClass : instance initializer

deserialization started

SuperClass : initializer for instance field

SuperClass : instance initializer

SuperClass.SuperClass()

end of main()

직렬화되는 객체가 역직렬화되기 위해서는 직렬화되는 객체의 수퍼 클래스가 Serializable 인터페이스를 구현하거나 인자가 없는(no-argument) 생성자를 제공해야 한다. 만약 수퍼 클래스가 Serializable 인터페이스를 구현하지 않으면 그 생성자가 실행된다. 위에서 보듯이 인스턴스 초기화 블록 등도 실행된다. 모든 객체의 수퍼 클래스는 Object 클래스인데 Object 클래스는 Serializable 인터페이스를 구현하지 않는다. 그러므로 Object 클래스의 인자가 없는 생성자가 실행된다.

* 1. Custom Serialization
     1. custom serialization의 필요성

대부분의 경우에 default serialization으로 충분하다. 그러나 항상 예외란 있는 법이다. 만약 transient 필드가 있다면 그 필드를 정확한 값으로 설정해 주어야 할 것이다. 이것은 default serialization으로는 할 수 없는 것이다. deserialization으로 객체를 복구하면 객체 간의 관계는 그대로 복구되지만 객체가 차지하는 메모리의 위치는 달라진다. 이것은 대부분의 경우에 별로 상관이 없다. 그러나 만약 그 클래스에서 Object.hashCode() 등을 사용한다면 문제가 될 수 있다. 또한 커다란 프로그램에서는 클래스들이 진화한다. 이런 것에 대응하기 위해서는 custom serialization이 필요하다.

* + 1. readObject(ObjectInputStream in), writeObject(ObjectOutputStream out)

일단 ObjectInputStream.readObject(), ObjectOutputStream.writeObject(Object obj)와 헷갈리지 말기 바란다.

만약 Serializable 인터페이스를 구현한 클래스에 readObject(ObjectInputStream) 메쏘드나 writeObject(ObjectOutputStream) 메쏘드가 있으면 default 직렬화가 실행되지 않고 이 메쏘드가 대신 실행된다.

이 메쏘드들은 private이어야 한다. 만약 private이 아니면 무시된다. private이기 때문에 보안 정책이 엄격하면 작동하지 않을 수도 있다. 이 책에서는 보안에 대해서는 다루지 않는다. readObject와 writeObject 중 하나만 제공할 수도 있다. 제공되지 않으면 그것은 default 직렬화가 실행된다. 그러나 이러면 사람을 헷갈리게 하므로 모두 제공하는 것이 좋다. 둘중의 하나를 default 직렬화를 사용하고 싶으면 아래 나오는 defaultReadObject, defaultWriteObject 메쏘드를 사용하면 된다.

이 메쏘드들이 구현되었다고 하더라도 Serializable 인터페이스를 구현하지 않으면 작동하지 않는다. 다음 예제를 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) {

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

try {

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(new A());

} catch (IOException e) {

System.out.println("serialization : " + e);

}

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

try {

ObjectInputStream in = new ObjectInputStream(inMemory);

in.readObject();

} catch (ClassNotFoundException e) {

System.out.println("deserialization : " + e);

} catch (IOException e) {

System.out.println("deserialization : " + e);

}

}

}

class A { // Serializable 을 구현하지 않았다.

private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {

}

private void writeObject(ObjectOutputStream out) throws IOException {

}

}

결과는 다음과 비슷할 것이다.

serialization : java.io.NotSerializableException: A

deserialization : java.io.WriteAbortedException: writing aborted; NotSerializableException: A

* + 1. Externalizable 인터페이스

다음은 Externalizable 클래스의 (주석을 제외한) 전체 소스 코드이다.

package java.io;

import java.io.ObjectOutput;

import java.io.ObjectInput;

public interface Externalizable extends java.io.Serializable {

void writeExternal(ObjectOutput out) throws IOException;

void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

}

* + 1. readObject, writeObject 대 readExternal, writeExternal

다음 예제를 실행해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

SubClass s = new SubClass();

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

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

out.writeObject(s);

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

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

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

in.readObject();

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

}

}

class SuperClass implements Serializable {

private void readObject(ObjectInputStream in) {

System.out.println("SuperClass.readObject");

}

private void writeObject(ObjectOutputStream out) {

System.out.println("SuperClass.writeObject");

}

}

class SubClass extends SuperClass implements Serializable {

private void readObject(ObjectInputStream in) {

System.out.println("SubClass.readObject");

}

private void writeObject(ObjectOutputStream out) {

System.out.println("SubClass.writeObject");

}

}

결과는 다음과 같을 것이다.

------ Serialization started -------

SuperClass.writeObject

SubClass.writeObject

------ Serialization finished -------

------ Deserialization started -------

SuperClass.readObject

SubClass.readObject

------ Deserialization finished -------

수퍼 클래스의 writeObject, readObject를 자동으로 불러준다.

다음 예제를 실행해 보자.

import java.io.\*;

public class Example {

public static void main(String[] args) throws Exception { // 귀찮아서 편법을 썼다. 따라하면 안된다.

SubClass s = new SubClass();

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

ObjectOutputStream out = new ObjectOutputStream(outMemory);

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

out.writeObject(s);

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

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

ObjectInputStream in = new ObjectInputStream(inMemory);

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

in.readObject();

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

}

}

class SuperClass implements Externalizable {

public SuperClass() {

System.out.println("SubClass : Constructor");

}

private void readObject(ObjectInputStream in) {

System.out.println("SuperClass.readObject");

}

private void writeObject(ObjectOutputStream out) {

System.out.println("SuperClass.writeObject");

}

public void readExternal(ObjectInput in) {

System.out.println("SuperClass.readExternal");

}

public void writeExternal(ObjectOutput out) {

System.out.println("SuperClass.writeObject");

}

}

class SubClass extends SuperClass implements Externalizable {

public SubClass() {

System.out.println("SubClass : Constructor");

}

private void readObject(ObjectInputStream in) {

System.out.println("SubClass.readObject");

}

private void writeObject(ObjectOutputStream out) {

System.out.println("SubClass.writeObject");

}

public void readExternal(ObjectInput in) {

System.out.println("SubClass.readExternal");

}

public void writeExternal(ObjectOutput out) {

System.out.println("SubClass.writeExternal");

}

}

결과는 다음과 같을 것이다.

SubClass : Constructor

SubClass : Constructor

------ Serialization started -------

SubClass.writeExternal

------ Serialization finished -------

------ Deserialization started -------

SubClass : Constructor

SubClass : Constructor

SubClass.readExternal

------ Deserialization finished -------

수퍼 클래스의 readExternal, writeExternal 메쏘드를 자동으로 불러주지 않기 때문에 프로그래머가 알아서 해주어야 한다. 가장 일반적인 방식은 서브 클래스의 readExternal에서

super.readExternal();

으로 수퍼 클래스의 readExternal을 불러주는 것일 것이다. writeExternal도 마찬가지로 해야 한다. 그리고 위의 결과에서 알 수 있듯이 객체를 복원할 때 인자가 없는(no-argument) 생성자를 먼저 부르고 readExternal를 부른다. 그리고 그 생성자는 public이어야 한다. 위에서는 readObject도 같이 제공했는데 무시된다. 즉 Externalizable을 구현하면 xxxExternal 메쏘드가 우선권이 있는 것이다.

* 1. Serialization을 이용한 깊은 복제
     1. 깊은 복제

직렬화는 직렬화되는 객체와 연결된 모든 객체들이 다 직렬화된다. 이것을 이용하면 깊은 복제를 할 수 있다. 깊은 복제에 대해 알고 싶으면 Object 클래스를 다루는 부분을 참조하라.

예제를 보자. 이것은 배열을 다룬 부분에서 좀더 고생해서 한 것과 마찬가지 결과가 나온다. 직렬화를 이용하면 복잡한 경우에 편리하긴 하지만 아무래도 속도면에서는 느릴 것이다. 여기서 물론 전제되어야 할 것은 연결된 객체가 모두 직렬화가 가능해야 한다는 것이다.

import java.io.\*;

public class Example {

public static void main(String[] args) {

int[][][] intArray = {

{ {1, 2, 3}, {4, 5, 6}, {7, 8, 9} },

{ {10, 11}, {12, 13}, null },

null,

{ {14}, {15, 16}, {} },

};

print(intArray);

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

int[][][] intArrayCloned = (int[][][])DeepCopier.deepCopy(intArray);

print(intArrayCloned);

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

intArray[0][0][0] = 1000;

print(intArrayCloned);

}

public static void print(int[][][] array) {

for (int i = 0; i < array.length; i++) {

int[][] subComponent = array[i];

if (subComponent == null)

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

else if (subComponent.length == 0)

System.out.println("array["+i+"].length = 0");

else

for (int j = 0; j < subComponent.length; j++) {

int[] subSubComponent = subComponent[j];

if (subSubComponent == null)

System.out.println("array["+i+"]["+j+"] = null");

else if (subSubComponent.length == 0)

System.out.println("array["+i+"]["+j+"].length = 0");

else

for (int k = 0; k < subSubComponent.length; k++) {

System.out.println("array["+i+"]["+j+"]["+k+"] = " + subSubComponent[k]);

}

}

}

}

}

class DeepCopier {

public static Object deepCopy(Object original) {

ByteArrayOutputStream outMemory = new ByteArrayOutputStream();

try {

ObjectOutputStream out = new ObjectOutputStream(outMemory);

out.writeObject(original);

} catch (IOException e) {

System.out.println(e);

}

ByteArrayInputStream inMemory = new ByteArrayInputStream(outMemory.toByteArray());

try {

ObjectInputStream in = new ObjectInputStream(inMemory);

Object cloned = in.readObject();

return cloned;

} catch (ClassNotFoundException e) {

System.out.println(e);

return null;

} catch (IOException e) {

System.out.println(e);

return null;

}

}

}

결과는 다음과 같을 것이다.

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

---------------------

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

---------------------

array[0][0][0] = 1

array[0][0][1] = 2

array[0][0][2] = 3

array[0][1][0] = 4

array[0][1][1] = 5

array[0][1][2] = 6

array[0][2][0] = 7

array[0][2][1] = 8

array[0][2][2] = 9

array[1][0][0] = 10

array[1][0][1] = 11

array[1][1][0] = 12

array[1][1][1] = 13

array[1][2] = null

array[2] = null

array[3][0][0] = 14

array[3][1][0] = 15

array[3][1][1] = 16

array[3][2].length = 0

1. 메모리 관리와 쓰레기 수거
   1. 쓰레기 수거(Garbage Collection)란
      1. 쓰레기 수거란

자바에는 new 키워드로 객체를 생성하는데 그것에 대응하는 delete 키워드가 없다. 이것은 한번 만들어진 객체가 불로장생한다는 것을 뜻하지는 않는다. 자바에서도 객체는 죽는다. 객체가 죽지 않는다면 인구가 너무 많아져서 살 곳이 부족해질 것이다. 즉 메모리가 부족해질 것이다. 자신의 임무를 다한 객체는 죽어야 한다. 자바에 delete 키워드가 없다는 것은 자바가 자동으로 객체를 죽여죽다는 얘기다. 즉 그 객체가 차지하던 메모리를 재할당(reclaim)하는 것이 자동으로 된다는 것이다. 이것을 하는 것이 쓰레기 수거기(garbage collector)이다. 그리고 이 과정을 쓰레기 수거(garbage collection)라고 한다. 한마디로 말하면 쓰레기 수거란 더 이상 필요 없어진 객체를 청소하여 그 메모리에 다른 객체를 생성할 수 있도록 하는 것이다.

* + 1. 쓰레기 수거의 장단점

C++과 같은 언어에서는 delete 키워드가 있어서 프로그래머가 직접 객체를 파괴한다. 이 방식에는 두가지 단점이 있다.

첫째로 객체를 파괴하는 것을 잊어버릴 수 있다는 것이다. 이런 실수를 하면 필요 없는 객체가 계속 존재하게 된다. 즉 메모리에 쓰레기가 존재하는 것이다. 이것을 메모리가 샌다(memory leak)고 한다. 이것은 상황에 따라 하찮은 문제일 수도 있고 심각한 문제일 수도 있다. 예를 들어 몇 바이트의 메모리가 샌다고 큰 일이 난 것은 아니다. 그러나 만약 이 프로그램이 서버 프로그램이고 몇 개월을 돌아야 한다고 생각해 보자. 그러면 티끌 모아 태산이라고 하듯이 몇 바이트가 수 개월 동안 새면 커다란 손실이 생길 수도 있다. 이런 것이 더 무서운데 만약 한꺼번에 많은 메모리가 샌다면 금방 버그를 찾아서 고칠 수 있지만 조금씩 샌다면 잘 돌다가 몇 개월 후 어느날 시스템이 다운되는 참으로 불행한 사태가 일어날 수도 있다.

둘째는 첫번째와 반대되는 실수인데 delete 해버린 객체를 사용하는 것이다. 즉 delete하지 말하야 할 것을 delete해버리는 경우이다. 이 때에 이 객체가 살고 있는 메모리에는 무엇이 있을지 예측할 수 없다. 따라서 이 객체를 사용할 때 무슨 일이 생길지도 예측할 수 없다. 이런 것을 dangling reference라고 한다.

자바는 delete를 프로그래머가 직접 하지 않고 쓰레기 수거기가 대신하기 때문에 이런 일이 일어나지 않는다. 물론 쓰레기 수거기에 버그가 없다는 가정하에서이다. 그리고 사실 쓰레기 수거기에 버그가 있을 가능성은 애플리케이션 프로그램에 버그가 있을 가능성보다 훨씬 적다.

하지만 이런 자동화에는 단점도 있다. 가장 큰 것은 속도 문제다. 쓰레기를 자동으로 처리하려면 그것을 처리하는 쓰레드가 하나 더 돌아야 한다. 그리고 이것은 상당히 시간을 많이 잡아먹는다. 왜냐하면 힙(heap)에 있는 모든 객체를 다 조사해서 쓰레기인지 아닌지를 알아내야 하기 때문이다. 이것은 특히 실시간(real time) 프로그래밍에서는 치명적인 약점이 될 수 있다.

* + 1. 쓰레기(garbage)란 무엇인가

쓰레기란 필요 없는 객체를 말한다. 그렇다면 정확히 어떤 객체가 쓰레기인가? 이 문제는 생각보다 그리 간단한 문제가 아니다.

먼저 아무도 참조해 주지 않는 객체는 쓰레기이다. 이것은 자명하다. 아무도 참조하지 않는다면 아무도 사용할 수 없다는 얘기다. 그러므로 아무 쓸모가 없을 수밖에 없다. 아무도 사용할 수 없는데 어디에 쓰겠는가? 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new String();

}

}

여기서 String 객체가 하나 만들어졌다. 그러나 아무도 이 객체를 참조하지 않는다. 그리하여 이 객체는 태어나자 마자 쓰레기가 되어 버렸다.

public class Example {

public static void main(String[] args) {

String str = new String();

str = null;

}

}

이번에는 String 객체가 만들어져서 지역 변수인 str에 그 참조가 대입된다. 즉 str이 그 객체를 참조하는 것이다. 그 다음 줄에 str에 null이 대입되었다. 이 순간에 String 객체를 참조하는 변수가 사라졌다. 즉 String 객체는 쓰레기가 된 것이다.

public class Example {

public static void main(String[] args) {

String str1 = new String();

String str2 = str1;

str1 = null;

}

}

이번에는 String 객체가 만들어지고 그 참조가 str1에 대입된다. 그리고 str1이 str2에 대입된다. 이제 두 개의 변수 즉 str1과 str2가 그 객체를 가리키고 있다. 그 다음에 str1에 null이 대입된다. 그러나 그 객체는 아직 쓰레기가 아니다. 왜냐 하면 str2가 아직도 그 객체를 가리키고 있기 때문이다.

* + 1. cycle(circular reference)

위에서 아무도 참조하지 않는 객체가 쓰레기라고 했다. 이것만이 쓰레기인 것은 아니다. 누군가 참조를 하더라도 쓰레기일 수가 있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A a = new A();

a.field = new String();

a = null;

}

}

class A {

public String field;

}

A 객체가 하나 만들어졌다. String 객체가 만들어지고 그 참조가 a.field즉 A 객체의 필드에 대입되었다. 그리고 a 에 null이 대입되었다.

이 때 A 객체를 참조하는 변수가 아무것도 없으므로 그 객체는 쓰레기이다. 그럼 String 객체는 어떤가? 이 객체를 A객체의 필드가 참조하지만 이미 A 객체가 쓰레기이다. 쓰레기는 누구도 접근할 수 없는 객체이다. 그러므로 이 쓰레기에 연결되어 있다고 하더라도 쓰레기 신세에서 벗어날 수 없다. 즉 String 객체도 덩달아 쓰레기가 된다.

그럼 다음 예제는 어떨까?

public class Example {

public static void main(String[] args) {

A a1 = new A();

A a2 = new A();

A a3 = new A();

a1.a = a2;

a2.a = a3;

a3.a = a1;

a1 = null;

a2 = null;

a3 = null;

}

}

class A {

public A a;

}

세 개의 A 객체가 만들어지고 그 것이 각각 지역 변수 a1, a2, a3에 대입되었다. 그리고 각각은 다른 객체의 필드에 대입되었다. 그 다음에 a1, a2, a3에 null이 대입되었다. 그럼 이 상태에서 세 객체는 쓰레기인가? 쓰레기 맞다. 세 객체 모두 누군가에 의해 참조되긴 하지만 코드에서 접근할 수가 없다. 이런 경우를 원환 참조(circular reference, cycle)이라고 한다.

* + 1. 잘 안보이는 참조

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Thread thr = new Thread();

thr = null;

}

}

thr = null;이 실행되는 순간 Thread 객체는 쓰레기가 되는 걸까? 그렇지 않다. 위의 코드상에는 분명히 Thread 객체는 아무도 참조해 주지 않아 버림받은 객체가 되었다. 그러나 사실은 누군가가 Thread 객체를 참조해 준다. 다음은 Thread.java와 ThreadGroup.java에서 인용한 것이다.

Thread.java

...

public class Thread implements Runnable {

...

public Thread() {

init(null, null, "Thread-" + nextThreadNum(), 0);

}

private void init(ThreadGroup g, Runnable target, String name, long stackSize) {

...

g.add(this); // 여기에 주목하라. g는 ThreadGroup형 변수.

}

...

}

ThreadGroup.java

...

public class ThreadGroup {

...

void add(Thread t) {

synchronized (this) {

...

threads[nthreads] = t; // 여기에 주목하라. 여기서 결국 참조가 생긴다.

...

}

}

...

}

Thread의 생성자에서 시작하여 init 메쏘드를 거쳐 결국 ThreadGroup의 add(Thread t) 메쏘드에서 threads[nthreads] = t;라는 줄에서 누군가에 의해 참조되는 것이다. 이 장의 쓰레드와 Garbage collection을 다룬 부분도 참조하라.

다음 예제도 마찬가지이다.

public class Example {

public static void main(String[] args) {

new java.awt.Frame().show();

}

}

Frame.show() 메쏘드에서 출발하면 아주 복잡한 과정을 거쳐서 어디엔가에 참조가 대입되는 것을 볼 수 있을 것이다.

* + 1. garbage collection 알고리즘

쓰레기 수거 알고리즘은 여러가지가 쓰일 수 있다. 그러므로 어떤 객체가 먼저 청소될지를 예측할 수 없다. 그리고 객체가 언제 청소될지도 알수 없다. 게다가 객체가 청소가 되지 않을 수 있다. 대부분의 garbage collector는 되도록이면 청소를 하지 않으려고 한다. 왜냐하면 청소하는 것이 시간을 많이 잡아먹기 때문이다. 이것을 설거지나 빨래에 비유하면 일주일 내내 하지 않다가 빨래통이 꽉찰 때(메모리가 부족해질 때)에 하는 것과 비슷하다.

* 1. Object.finalize()
     1. 쓰레기 수거기 와 finalize()

쓰레기 수거기는 버림받은 객체를 정리한다. 즉 그 객체가 차지하던 메모리를 재활용할 수 있도록 한다. 그 때 그 객체에게 마지막 기회를 주는 것이 finalize() 메쏘드이다. 쓰레기 수거기(garbage collector)는 쓰레기가 된 객체의 finalize() 메쏘드가 다 실행될 때까지 기다렸다가 그 객체가 차지하던 메모리를 정리하는 것이다. 이것은 C++의 destructor와 비슷하지만 다른 점이 있다.

finalize() 메쏘드가 언제 불릴지 심지어 불릴지 안 불릴지 여부도 예측할 수 없다. 쓰레기 수거기는 나름대로의 알고리즘을 이용해서 쓰레기를 청소한다. 그러므로 먼저 쓰레기가 되었다고 해서 먼저 청소되는 것이 아니다. 그리고 쓰레기가 되었다고 바로 청소되는 것도 아니다. 게다가 프로그램이 종료될 때까지 메모리에 여유가 있다면 아예 어떤 객체도 finalize() 메쏘드가 불리지 않을수도 있다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

new A();

System.out.println("end of Example.main()");

}

}

class A {

protected void finalize() {

System.out.println("A.finalize()");

}

}

결과는 아마 다음과 같을 것이다.

end of Example.main()

A.finalize()가 불리지 않았다. 대부분의 가상 머신은 프로그램이 끝날 때 객체의 쓰레기 수거를 하지 않는다. 그냥 운영체제로부터 받은 메모리를 그냥 그대로 돌려준다. 이렇게 하면 시간을 상당히 절약할 수 있다.

* + 1. finalize() 재정의하기

이 객체의 finalize()가 불렸다는 것은 이 객체의 필드가 가리키는 객체의 상태가 어떻게 될지 모른다는 것을 의미한다. 다음 예제를 보자.

class Garbage {

private Object field;

protected void finalize() {

// 여기서 field 가 가리키는 객체의 상태에 대해서는 예측할 수 없다.

}

}

위의 예제에서 만약 finalize() 메쏘드 안에서 field 가 가리키는(또는 가리켰던) 객체를 다루려고 한다면 어떤 일이 일어날지 모른다. 만약 가리키는 객체를 다른 변수도 가리키고 있다면 그 객체가 쓰레기가 아니므로 정상적인 상태일 것이다. 그러나 그 객체를 가리키고 있는 변수가 여기있는 field 하나 뿐이라면 그 객체도 이미 쓰레기이므로 그 곳의 메모리가 이미 재활용되었는지도 모른다.

finalize()에서는 수퍼 클래스의 finalize()를 불러주는 것이 좋다. 심지어 Object 클래스가 바로 위 수퍼 클래스라고 하더라도 그렇게 하는 것이 좋다. 다음에 클래스 상속구조가 바뀌더라도 문제가 발생하지 않는다. 즉 다음과 같이 해주는 것이 좋다.

class SuperClass {

protected void finalize() throws Throwable {

// ...

super.finalize();

}

}

class SubClass extends SuperClass {

protected void finalize() throws Throwable {

// ...

super.finalize();

}

}

* + 1. cleanup

자바에서는 객체를 자동으로 청소해주기 때문에 따라 cleanup할 필요가 없다. 그러나 다른 경우에는 수동으로 정리해 주어야 한다. 그런 경우에는 어떤 것들이 있을까?

객체는 메모리에 생성된다. 프로그래밍을 할 때에는 메모리 자원이 아닌 다른 것을 건드릴 필요가 있을 때가 많다. 그런 것 중에 대표적인 것이 파일과 데이터베이스이다. 파일과 데이터베이스를 열었다면 반드시 닫아주어야 한다. 이런 자원에 대한 cleanup(정리)를 finalize() 메쏘드에서 하겠다는 생각은 아주 안좋은 생각이다. 왜냐하면 finalize() 메쏘드가 언제 불릴지 예측할 수 없으며 아예 불리지 않을수도 있기 때문이다. 이런 경우에는 따로 cleanup 메쏘드를 만들어주는 것이 좋다. 다음 예제를 보자.

class A {

public void open() {

// 여기서 파일이나 DB 를 열어준다.

}

public void close() { // cleanup 을 해주는 메쏘드

// 여기서 파일이나 DB 를 닫아준다.

}

}

이런 경우에도 finalize() 메쏘드를 이용해서 유용한 일을 할 수는 있는데 다음 예제를 보면 될 것이다.

class A {

boolean bOpen = false;

public void open() {

// 여기서 파일이나 DB 를 열어준다.

bOpen = true;

}

public void close() { // cleanup 을 해주는 메쏘드

// 여기서 파일이나 DB 를 닫아준다.

bOpen = false;

}

protected void finalize() throws Throwable {

if (bOpen)

System.out.println("A.finalize() 파일(DB)를 닫아주지 않았습니다.");

}

}

위의 클래스 A를 사용할 때 파일(DB)를 닫는 것을 잊어버릴 수도 있을 것이다. 이런 버그를 운이 좋으면(A.finalize()가 쓰레기 수거기에 의해 실행된다면) 발견할 수 있을 것이다.

또는 아예 다음과 같이 finalize()에서 닫아줄 수도 있을 것이다.

class A {

boolean bOpen = false;

public void open() {

// 여기서 파일이나 DB 를 열어준다.

bOpen = true;

}

public void close() { // cleanup 을 해주는 메쏘드

// 여기서 파일이나 DB 를 닫아준다.

bOpen = false;

}

protected void finalize() throws Throwable {

if (bOpen)

close();

}

}

위의 두 예제 모두 파일(DB)를 닫는 것을 잊어버리는 버그를 완전히 해결해 주지는 못하지만 없는 것보다는 낫다.

메모리 자원중에도 쓰레기 수거기가 자동으로 처리하지 못하는 것이 있을 수 있다. 그것은 native 메쏘드에서 얻은 메모리이다. 예를 들어 C의 malloc() 함수로 메모리를 얻었다면 free()등의 함수로 정리해 주어야 한다.

* + 1. finalize()를 부르는 Thread

finalize()는 Garbage Collector가 부른다. 그럼 finalize()가 현재 쓰레드는 garbage collector 쓰레드인 것이다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

ThreadGroup thg = Thread.currentThread().getThreadGroup();

ThreadGroup thgTop = getTopThreadGroup(thg);

thgTop.list();

for (int i = 0; i < 5; i++) // 안되면 반복횟수를 늘려보라.

new A();

System.out.println("end of Example.main()");

}

public static ThreadGroup getTopThreadGroup(ThreadGroup thg) {

ThreadGroup thgParent = thg.getParent();

if (thgParent == null)

return thg;

else

return getTopThreadGroup(thgParent); // 재귀호출

}

}

class A {

private long[] field = new long[100000]; // 안되면 배열 크기를 줄이거나 늘려보라.

protected void finalize() {

Thread thr = Thread.currentThread();

System.out.println("A.finalize() current thread : " + thr);

}

}

결과는 다음과 비슷할 것이다.

java.lang.ThreadGroup[name=system,maxpri=10]

Thread[Reference Handler,10,system]

Thread[Finalizer,8,system]

Thread[Signal Dispatcher,10,system]

Thread[CompileThread0,10,system]

java.lang.ThreadGroup[name=main,maxpri=10]

Thread[main,5,main]

A.finalize() current thread : Thread[Finalizer,8,system]

A.finalize() current thread : Thread[Finalizer,8,system]

A.finalize() current thread : Thread[Finalizer,8,system]

A.finalize() current thread : Thread[Finalizer,8,system]

end of Example.main()

다섯개의 쓰레드(Reference Handler, Finalizer, Signal Dispatcher, CompileThread0, main)가 돌고 있는데 그 중에 Finalizer라는 쓰레드가 finalize() 메쏘드를 불렀다.

* + 1. finalize()의 예외 상황은 무시된다.

finalize() 메쏘드 안에서도 예외 상황을 발생할 수 있을 것이다. 이 메쏘드는 garbage collector가 부르는데 garbage collector는 이 예외상황을 그냥 무시한다.

* + 1. 부활(ressurection)

죽어가는 객체는 부활할 수 있다. finalize() 메쏘드에서 다시 참조를 만들면 된다. 다음 예제를 보자.

import java.util.\*;

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 5; i++) { // 잘 안되면 반복횟수를 늘려보라.

new Resurrection("" + i);

}

for (Iterator i = Resurrection.resurrected.iterator(); i.hasNext(); ) {

System.out.println("Example.main() " + i.next());

}

}

}

class Resurrection {

public static ArrayList resurrected = new ArrayList();

private long[] devourer = new long[100000]; // 잘 안되면 배열 크기를 늘리거나 줄려보라.

private String id;

public Resurrection(String str) {

id = str;

}

protected void finalize() {

System.out.println("Resurrection.finalize()");

resurrected.add(this); // 부활

}

public String toString() {

return "ID : " + id;

}

}

결과는 다음과 비슷할 것이다.

Resurrection.finalize()

Resurrection.finalize()

Resurrection.finalize()

Example.main() ID : 0

Example.main() ID : 1

Example.main() ID : 2

이번 실행 예에서는 세 개의 객체의 finalize() 메쏘드가 불렸고 그 세개의 객체가 모두 다 부활했다.

하나의 객체의 finalize() 메쏘드는 쓰레기 수거기에 의해서는 최대 단 한번만 불린다. 그러므로 부활한 객체가 정리될 때는 finalize() 메쏘는 불리지 않는다.

만약 부활이 필요하다면 객체 자신을 부활시키는 것보다 clone 메쏘드나 new 키워드를 이용해서 똑같은 객체를 복제하는 것이 낫다. 그러나 더 중요한 것은 이런 식으로 쓰레기와 쓰레기가 아닌 객체 사이를 왔다갔다 하는식으로 설계하는 것은 좋은 설계가 아니라는 것이다. 만약 이런식으로 해야겠다는 생각이 든다면 다른 방법이 없는지 먼저 생각해 보기 바란다.

* 1. Runtime, System 클래스와 메모리 관리
     1. System.gc(), System.runFinalization()

runFinalization 메쏘드는 쓰레기 객체의 finalize 메쏘드를 부르게 한다.

gc 메쏘드는 쓰레기를 정리하게 한다.

이 메쏘드들은 가상머신을 구현하는 사람 맘대로이다. 즉 쓰레기를 정리하라고 명령한다기보다는 부탁하는 것이다. 부탁은 들어줄 수도 안 들어줄 수도 있다. 즉 이 메쏘드를 부른다고 쓰레기 정리가 실제로 되라는 보장이 없다. 그러나 많은 객체를 한꺼번에 생성할 일이 있다면 그 전에 이 메쏘드를 부르는 것이 좋다. 부탁을 들어주면 좋고 안들어준다고 크게 손해보는 것은 없으니까.

* + 1. Runtime.freeMemory(), Runtime.totalMemory()

freeMemory 메쏘드는 현재의 사용할 수 있는 메모리의 양을 대략적으로 알려준다.

totalMemory 메쏘드는 운영체제로부터 받은 메모리의 양을 알려준다.

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

Runtime rt = Runtime.getRuntime();

long free = rt.freeMemory();

long total = rt.totalMemory();

System.out.println("free : " + free);

System.out.println("total : " + total);

System.out.println();

long[] devourer = new long[100000]; // 배열 크기를 조절해 보라.

free = rt.freeMemory();

total = rt.totalMemory();

System.out.println("free : " + free);

System.out.println("total : " + total);

}

}

결과는 다음과 비슷할 것이다.

free : 1766936

total : 2031616

free : 965696

total : 2031616

* 1. 쓰레드와 Garbage Collection
     1. 쓰레드와 Garbage Collection

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

new LostObject();

}

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println(e);

}

}

}

class LostObject {

private long[][] devourer = new long[1000][1000]; // 결과가 나오지 않으면 배열 크기를 바꿔라.

public void finalize() {

System.out.println("LostObject.finalize()");

}

}

결과는 다음과 비슷할 것이다.

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

LostObject.finalize()

위의 예제를 약간 고쳐보자. Thread를 상속받는다.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

new LostObject();

}

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println("End of main()");

}

}

class LostObject extends Thread {

private long[][] devourer = new long[1000][1000]; // 결과가 나오지 않으면 크기를 바꿔라.

public void finalize() {

System.out.println("LostObject.finalize()");

}

}

결과는

java.lang.OutOfMemoryError

Exception in thread "main"

또는

End of main()

와 같을 것이다. 배열 크기가 너무 크면 메모리가 부족하다고 할 것이고 너무 작으면 쓰레기 수거기가 쓰레기 청소를 할 필요성을 못 느낄 가능성이 크다. 하여튼 여기서 중요한 것은 Thread를 상속받는 객체는 객체의 참조(reference)를 저장하지 않았음에도 쓰레기 수거가 되지 않는다는 것이다. 그 이유는 위의 코드에서는 안 보이지만 Thread의 생성자의 어디에선가 참조를 저장하기 때문이다.

* + 1. Thread 객체는 언제부터 쓰레기 취급을 받나

위의 예제를 약간 바꾸었다.

public class Example {

public static void main(String[] args) {

for (int i = 0; i < 10; i++) {

new LostObject().start();

}

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println(e);

}

System.out.println("End of main()");

}

}

class LostObject extends Thread {

private long[][] devourer = new long[1000][1000];

public void run() {

System.out.println("LostObject.run()");

}

public void finalize() {

System.out.println("LostObject.finalize()");

}

}

결과는 다음과 비슷할 것이다.

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.finalize()

LostObject.run()

LostObject.run()

End of main()

쓰레드의 실행을 마치자 청소되는 것을 알 수 있다.

* 1. 스택(stack)과 힙(heap)
     1. 스택에 만들어지는 것들

지역 변수와 파라미터는 스택에 만들어지고 객체와 객체에 속하는 필드는 힙에 만들어진다. 이제부터 실제로 그런지 알아보자. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

local(1);

}

public static void local(long count) {

if (count % 100 == 0)

System.out.println(count + ", ");

local(count + 1);

}

}

결과는 다음과 비슷할 것이다.

100,

200,

300,

…

7700,

7800,

java.lang.StackOverflowError

at Example.local(Example.java:9)

at Example.local(Example.java:9)

at Example.local(Example.java:9)

…

재귀 호출을 약 7800번 한다음에 스택 오버플로우(stack overflow)가 발생한다. 이제 메쏘드에 지역 변수를 추가해 보자.

public class Example {

public static void main(String[] args) {

local(1);

}

public static void local(long count) {

long localVariable1;

long localVariable2;

long localVariable3;

long localVariable4;

long localVariable5;

long localVariable6;

long localVariable7;

long localVariable8;

long localVariable9;

long localVariable10;

if (count % 100 == 0)

System.out.println(count + ", ");

local(count + 1);

}

}

결과는 다음과 비슷할 것이다.

…

3300,

3400,

3500,

java.lang.StackOverflowError

at Example.local(Example.java:20)

at Example.local(Example.java:20)

at Example.local(Example.java:20)

…

이번에는 약 3500번에 스택 오버플로우가 났다. 이것으로 지역 변수가 스택에 쓰여지는 것이 입증된 것이다. 이번에는 매개변수도 테스트 해보자.

public class Example {

public static void main(String[] args) {

local(1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

}

public static void local(long count, long parameter1, long parameter2,

long parameter3, long parameter4, long parameter5, long parameter6,

long parameter7, long parameter8, long parameter9, long parameter10) {

if (count % 100 == 0)

System.out.println(count + ", ");

local(count + 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

}

}

결과는 다음과 비슷할 것이다.

…

1800,

1900,

2000,

java.lang.StackOverflowError

at Example.local(Example.java:12)

at Example.local(Example.java:12)

at Example.local(Example.java:12)

…

이번에는 약 2000 번에 스택 오버플로우가 났다. 매개변수 역시 스택에 만들어지는 것이다.

* + 1. 힙에 만들어지는 것들

이번에는 객체에 대해 알아보자.

public class Example {

public static void main(String[] args) {

local(1);

}

public static void local(long count) {

A a = new A();

if (count % 100 == 0)

System.out.println(count + ", ");

local(count + 1);

}

}

class A {

private long[] field = new long[10000]; // 배열 크기를 줄이거나 늘려보라.

}

결과는 다음과 비슷할 것이다.

100,

200,

300,

400,

500,

600,

700,

800,

java.lang.OutOfMemoryError

Exception in thread "main"

객체는 힙에 만들어지기 때문에 StackOverflowError가 아니라 OutOfMemoryError가 났다. 객체가 힙에 만들어지기 때문에 객체의 필드도 당연히 힙에 만들어진다.

* + 1. 메모리는 어떻게 할당되나

여기서는 지역 변수, 파라미터, 객체가 메모리에 어떻게 할당되나를 알아본다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

int i = 123;

StringBuffer abc = new StringBuffer("abc");

abc = method(i, abc);

}

public static StringBuffer method(int a, StringBuffer sb) {

String str = sb.toString();

StringBuffer buffer = new StringBuffer("buffer" + str);

return buffer;

}

}

먼저 main 메쏘드로 컨트롤이 넘어온다. 스택에 int형 변수 i가 만들어지고 123이 i에 대입된다. 스택에 StringBuffer 형 변수 abc가 만들어진다. 힙에 StringBuffer형 객체가 만들어진다. 그 객체는 문자열 “abc”를 담고 있다. 이 객체에 대한 참조가 변수 abc에 저장된다. 다른 말로 하면 변수 abc가 문자열 “abc”를 담고 있는 객체를 가리킨다.

컨트롤은 이제 method 메쏘드로 넘어온다. 스택에 매개변수 a와 sb가 만들어진다. 그리고 a에는 i의 값인 123이 복사된다. 그리고 sb에는 abc의 참조가 복사된다. 즉 sb는 문자열 “abc”를 담고 있는 객체를 가리킨다. 다음에 지역 변수 str와 buffer가 스택에 만들어진다. str는 sb.toString();이 리턴하는 String 객체를 가리킨다. StringBuffer 객체가 힙에 만들어진다. 이 객체는 문자열 “bufferabc”가 담겨있다. 이 객체에 대한 참조가 buffer 변수에 저장된다. 즉 buffer는 문자열 “bufferabc”가 담겨있는 객체를 가리킨다.

이제 return문을 만나서 method 메쏘드는 끝난다. 그러므로 매개변수 a, sb와 지역 변수 str, buffer의 수명이 끝난다. 그러므로 “abc”를 담고 있는 String 객체를 가리키는 변수는 없다. 이 객체를 쓰레기(garbage)가 된다. 이젠 컨트롤은 main 메쏘드로 돌아왔다. 변수 abc는 method 메쏘드가 리턴한 객체 즉 “bufferabc”를 담고 있는 객체를 가리킨다. 그러므로 “abc”를 담고 있는 StringBuffer 객체는 쓰레기가 된다.

* + 1. 메모리 할당은 가변적일 수 있다

자바 가상 머신은 운영체제로부터 메모리를 받아서 스택과 힙으로 사용한다. 이 때 메모리가 모자라면 메모리를 더 달라고 할 수도 있다. 이것은 가상 머신을 구현하는 사람이 결정할 문제이다. 즉 메모리양이 불변인지 가변인지는 가상머신에 달려있다. 다음 예제를 보자.

public class Example {

public static void main(String[] args) {

A[] array = new A[1000000];

for (int i = 0; i < array.length; i++) {

try {

array[i] = new A();

System.out.println("Total memory = " + Runtime.getRuntime().totalMemory());

System.out.println("Free memory = " + Runtime.getRuntime().freeMemory());

System.out.println("A.created = " + A.created);

} catch (OutOfMemoryError e) {

System.out.println(e);

break;

}

}

}

}

class A {

public static long created = 0;

private long[] devourer = new long[500000];

public A() {

created++;

}

}

결과는 어쩌면 다음과 비슷할지도 모른다. JDK 1.4(java 1.4.0-beta-b65)로 실행한 결과이다.

Total memory = 11517952

Free memory = 3360984

A.created = 1

Total memory = 14708736

Free memory = 2552632

A.created = 2

Total memory = 21835776

Free memory = 5679904

A.created = 3

Total memory = 21835776

Free memory = 1678960

A.created = 4

Total memory = 35954688

Free memory = 11798752

A.created = 5

Total memory = 35954688

Free memory = 7797808

A.created = 6

Total memory = 35954688

Free memory = 3796864

A.created = 7

Total memory = 55955456

Free memory = 19799424

A.created = 8

Total memory = 55955456

Free memory = 15798480

A.created = 9

Total memory = 55955456

Free memory = 11797536

A.created = 10

Total memory = 55955456

Free memory = 7796592

A.created = 11

Total memory = 55955456

Free memory = 3795648

A.created = 12

Total memory = 66846720

Free memory = 10690528

A.created = 13

Total memory = 66846720

Free memory = 6689584

A.created = 14

Total memory = 66846720

Free memory = 2688640

A.created = 15

java.lang.OutOfMemoryError

메모리가 부족하자 계속 메모리 양(Total memory)을 늘리고 있다.

1. Assertion(단언)
   1. assert의 문법
      1. assert 키워드

assert 키워드 다음에는 boolean 식이 온다. 만약 그 boolean 식의 값이 false 값이면 AssertionError가 발생한다. 사용 예는 아래에 있다.

* + 1. AssertionError 클래스

이름이 AssertionException이 아니라 AssertionError이다. 즉 수퍼 클래스가 Exception이 아니라 Error이다. Error는 보통 복구될 수 없는 심각한 상황을 나타낸다. 버그는 예외 상황보다는 심각한 에러에 더 가깝다. 그래서 assert를 만든 사람들은 assert가 실패하는 상황을 Error상황으로 받아들여주길 바란 것이다.

* 1. assert와 관련된 설정
     1. 버전 문제

Assertion는 JDK 1.4.0(JavaTM 2 Platform, Standard Edition, v 1.4.0)에 추가되었다. assert라는 키워드가 추가된 것이다. 그러므로 assert를 사용한 코드를 컴파일하거나 실행할 때는 1.4.0또는 그 이상의 버젼을 사용해야 한다(현재 1.4.0의 베타 버젼까지 나왔다).

* + 1. 호환성과 컴파일 옵션

assert라는 키워드가 버젼 1.4에 추가되다 보니 발생하는 문제가 있다. 이전에 assert를 식별자(identifier)로 사용했을 때이다. 이미 컴파일된 .class파일은 문제가 없다. 그래서 java.exe에 –source라는 옵션이 있다. 먼저 예제를 보자.

public class Example {

public static void main(String[] args) {

int assert; // -source 1.4를 사용할 경우 Error.

assert true; // 1.4이전 버젼을 사용하거나 –source 1.4 없이 컴파일할 경우 Error.

}

}

다음과 같이 두가지 경우로 컴파일해보자.

javac –source 1.4 Example.java

javac Example.java

-source 1.4를 사용할 경우와 안할 경우에 컴파일타임 오류가 나는 곳이 다를 것이다. assert를 사용하는 앞으로의 예제는 –source 1.4를 사용해서 컴파일해야 한다.

* + 1. 실행할 때의 assert의 on/off

다음 예제를 보자.

public class Example {

public static void main(String[] args) {

assert booleanMethod(); // assert 문

System.out.println("Example.main()");

}

public static boolean booleanMethod() {

System.out.println("Example.booleanMethod()");

return true;

}

}

다음 예제를 컴파일해 보자.

javac -source 1.4 Example.java

이제 실행해 보자.

java Example

결과는 다음과 같다.

Example.main()

assert 문이 전혀 실행되지 않았다. assert 문이 실행되게 하려면 다음과 같이 하면 된다.

java -ea Example

결과는 다음과 같다.

Example.booleanMethod()

Example.main()

개발중이거나 테스트 중일 때는 –ea 를 사용하고 즉 assert를 작동하게 하고 실제 배포 할 때는 –ea없이 하면 된다. assert가 작동되지 않는다면 실행시간이 절약될 것이다.

* + 1. 클래스 파일에서 assert를 아예 없애버리기

어떤 프로그램은 너무나 가난한 컴퓨터(예를 들어 가전제품 같이 메모리가 적은 데서)에서 실행되기 때문에 실행 파일의 크기를 줄이는데 목숨을 걸만할 때도 있다. 그럴때는 배포할 때에는 군더더기에 불과한 assert를 클래스 파일에서 제외하고 싶어질 것이다. 클래스(.class) 파일에서 assert 문을 제외하는 근본적인 방법은 없다. 그러나 if문을 사용하여 영리한 컴파일러가 없애게 할 수는 있다. 다음 예제를 보자.

public class Example {

public static boolean TEST = true;

public static void main(String[] args) {

if (TEST) {

assert false;

}

}

}

컴파일을 해보자.

javac -source 1.4 Example.java

이젠 실행해 보자.

java -ea Example

결과는 다음과 비슷할 것이다.

Exception in thread "main" java.lang.AssertionError

at Example.main(Example.java:3)

프로그램을 개발하거나 테스트 중일 때는 위와 같이 하면 된다. 이젠 배포(release)할 때는 다음과 같이 한 줄을 바꾸면 된다. 영리한 컴파일러는 TEST의 값이 false라는 것을 컴파일 타임에 알기 때문에 if문의 블록 안에 있는 것이 결코 실행되지 않는다는 것을 안다.

public class Example {

public static boolean TEST = false; // 여기만 바꾼다.

public static void main(String[] args) {

if (TEST) {

assert false;

}

}

}

위와 같이 컴파일하고 실행했음에도 AssertionError가 나지 않을 것이다. 만약 그래도 AssertionError가 난다면 즉 Assertion이 클래스 파일에서 제외되지 않았다면 멍청한 컴파일러를 만든 사람을 원망하기 바란다.

assert 키워드를 도입한 사람들은 이런식으로 Assertion 기능을 빼버리는 것을 바라지 않는다. 이것은 정말 메모리에 목숨을 걸어야 할때만 사용하자.

* + 1. assert를 무조건 실행되게 하기

어떤 때는 assert가 배포시에도 무조건 실행되게 하고 싶을 때가 있을 것이다. 그럴땐 다음과 같이 하면 된다. 예제를 보자.

public class Example {

static {

boolean assertsEnabled = false;

assert assertsEnabled = true;

if (!assertsEnabled)

throw new RuntimeException("Asserts must be enabled!!!");

}

public static void main(String[] args) {

System.out.println("OK");

}

}

assert가 실행되면 즉

assert assertsEnabled = true;

가 실행되면 assertsEnabled는 당연히 true가 된다.

다음 예제를 컴파일한후 다음과 같이 실행해 보자. 즉 assert를 켜고 실행해 보자.

java -ea Example

결과는 다음과 같을 것이다.

OK

assert를 끄고 실행해 보자. 즉

java -da Example

그러면 결과는 다음과 비슷할 것이다. 즉 실행을 거부한다.

Exception in thread "main" java.lang.ExceptionInInitializerError

Caused by: java.lang.RuntimeException: Asserts must be enabled!!!

at Example.<clinit>(Example.java:8)

* 1. assertion 의 사용에 대한 조언
     1. assert를 사용해야 할 곳

다음 예제를 보자.

import java.util.\*;

/\*\*

\* 다른 사람이 만들 클래스라고 생각하기 바람.

\*/

public class Example {

public static void main(String[] args) {

System.out.println(Factorizer.factorize(4053504));

}

}

/\*\*

\* 인수 분해를 하는 클래스.

\* 당신이 만든 클래스라고 생각하기 바람.

\*/

class Factorizer {

/\*\*

\* Integer 객체의 Collection을 리턴한다.

\*/

public static Collection factorize(int number) {

if (number < 2)

throw new IllegalArgumentException("2 이상의 숫자를 넣으시오."); // 예외 상황 사용

Collection factors = new HashSet();

int copyNumber = number;

for (int i = 2; i <= copyNumber; i++) {

if (copyNumber%i == 0) {

factors.add(new Integer(i));

copyNumber /= i;

i = 1;

continue;

}

}

assert isFactors(number, factors); // Assertion 사용

return factors;

}

public static boolean isFactors(int number, Collection factors) {

for (Iterator i = factors.iterator(); i.hasNext(); ) {

Integer oiFactor = (Integer)i.next();

int iFactor = oiFactor.intValue();

if (number%iFactor != 0) {

return false;

}

}

return true;

}

}

당신이 인수분해하는 클래스를 만들었고 누군가 다른 사람이 그것을 사용한다고 생각하자. 실행하면 결과는 다음과 같다.

[29, 7, 3, 13, 2]

모든 것이 잘 돌아가고 있다. 적어도 그렇게 보인다.

그럼 두 가지 경우를 생각해 보자. 먼저 당신이 뭔가를 실수했을 때이다. 위의 코드중

factors.add(new Integer(i));

을 실수로

factors.add(new Integer(i+1));

과 같이 했다고 생각하자. (Assertion을 켜고) 실행하면 다음과 비슷할 것이다.

Exception in thread "main" java.lang.AssertionError

at Factorizer.factorize(Example.java:18)

at Example.main(Example.java:10)

이번에는 사용자가 실수했을 경우를 생각해 보자. 위의 코드중

System.out.println(Factorizer.factorize(4053504));

을 실수로

System.out.println(Factorizer.factorize(-4053504));

와 같이 했다고 생각해 보자. 실행하면 결과는 다음과 비슷할 것이다.

Exception in thread "main" java.lang.IllegalArgumentException: 2 이상의 숫자를

넣으시오.

at Factorizer.factorize(Example.java:24)

at Example.main(Example.java:10)

왜 한 곳에서는 IllegalArgumentException을 사용했고 왜 한곳에서는 assert 키워드를 사용했는지를 생각해 보자.

assert는 당신이 만든 코드의 버그를 찾는데 주로 사용한다. 예외 상황은 당신이 만든 코드의 버그가 아니라 다른 사람이 당신이 만든 클래스를 잘못 사용했거나 네트워크 장애등의 예외상황이 발생했을 때를 알려주기 위해 사용한다.

* + 1. assert를 사용해서는 안되는 곳

위와 반대로 사용해도 문법적으로는 틀리지 않다. 위의 예제를 조금 바꾸었다.

/\*\*

\* 다른 사람이 만들 클래스라고 생각하기 바람.

\*/

public class Example {

public static void main(String[] args) {

System.out.println(Factorizer.factorize(4053504));

}

}

/\*\*

\* 인수 분해를 하는 클래스.

\* 당신이 만든 클래스라고 생각하기 바람.

\*/

class Factorizer {

/\*\*

\* Integer 객체의 Collection을 리턴한다.

\*/

public static Collection factorize(int number) {

assert (number >= 2);

Collection factors = new HashSet();

int copyNumber = number;

for (int i = 2; i <= copyNumber; i++) {

if (copyNumber%i == 0) {

factors.add(new Integer(i));

copyNumber /= i;

i = 1;

continue;

}

}

if ( false == isFactors(number,factors) )

throw new IllegalArgumentException("2 이상의 숫자를 넣으시오.");

return factors;

}

public static boolean isFactors(int number, Collection factors) {

for (Iterator i = factors.iterator(); i.hasNext(); ) {

Integer oiFactor = (Integer)i.next();

int iFactor = oiFactor.intValue();

if (number%iFactor != 0) {

return false;

}

}

return true;

}

}

그러나 이렇게 의미에 맞지 않게 사용하면 언젠가 피 볼 날이 있을 것이다.