원시 타입의 값과 래퍼 클래스의 객체간 자동적 변환을 지원하기 위한 Java의 확장

(Java Extension for supporting Automatic Transformation between Values of Primitive Types and Objects of Wrapper Classes)

김 성 기 † 김 상 철 †† 정 병 수 †††
(Sung-Ki Kim) (Sang-Chul Kim) (Byeong-Soo Jeong)

요 약 Java에서 원시 타입과 클래스 타입간에 호환성이 제공되지 않으므로 원시 값이 클래스 타입의 변수에 저장되지 못하며, 클래스 타입의 값이 원시 타입 변수에 저장되지 못한다. 객체가 필요한 곳에서 원시 값을 사용하기 위해서는 원시 값을 저장하는 래퍼 클래스의 객체 생성이 필요하며, 래퍼 클래스의 객체에서 저장된 원시 값을 꺼내기 위해서는 특정 메소드를 호출하여야 한다. 이러한 불편함과 변환시의 오류를 줄이기 위하여 본 논문에서는 Java의 원시타입과 래퍼 클래스 타입의 호환성 제공을 위한 변환방법을 제안하였다. 원시 타입과 래퍼 클래스 타입간의 호환성 뿐 아니라 원시 타입간의 호환성에 상응하는 래퍼 클래스간의 호환성도 제공하기 위해 래퍼 클래스 계층방법, java.lang.Number 클래스 이용방법, 래퍼 인터페이스 계층방법 등 3가지 변환방법을 제시하였다. 이들 방법의 비교분석과 성능측정을 통하여 래퍼 인터페이스 계층방법이 가장 우수한 방법임을 확인하였다.

Abstract Since there is no compatability between primitive types and class types in Java, values of primitive types cannot be assigned to variables of class types, and values of class types cannot be assigned to variables of primitive types. Primitive values must be converted to objects of wrapper classes and special methods must be called in order to extract the primitive values from those objects. In this paper we propose three methods which provide automatic transformation between primitive types and class types for their compatibility. Those methods support compatability not only between primitive types and class types but also between wrapper classes. The first method utilizes the hierarchy of wrapper classes, the second utilizes java.lang.Number class, and the third utilizes the hierarchy of wrapper interfaces. Through comparison and performance measurement, we confirm that the third method works better than the others.

1. 서 론

Java 프로그래밍 언어는 객체와 클래스, 상속, 추상화(abstraction), 캡슐화(encapsulation), 다형성(polymorphism) 등 객체 지향 패러다임의 중요 기능을 지원하면서 플랫폼에 독립적인 수행환경을 제공하고 인터넷 웹 페이지에 애플릿을 내장할 수 있는 장점으로 인하여 급속히 보급되며 발전되고 있다.

Java의 타입은 크게 문자, 정수, 실수 등 기본적인 값을 지원하기 위한 원시 타입(primitive type)과 객체에 대한 참조를 지원하는 참조 타입(reference type)으로 나누어진다[1]. 원시 타입에는 boolean, char, byte, int, long, float, double 타입이 있으며, 참조 타입에는 배열 객체를 참조하는 배열 타입과 클래스의 인스턴스 객체를 참조하는 클래스 타입이 있다. Java는 타입의 관점에서 다음의 취약점을 내포하고 있다. 첫째, '모든 것이

· 본 연구는 한국과학재단 목적기초연구(2001-1-30300-019-2)지원으로 수행되었음
† 종신회원 : 한신대학교 컴퓨터학과 교수
skkim@hucc.hanshin.ac.kr
†† 종신회원 : 한국외국어대학교 컴퓨터공학과 교수
kimsa@maincc.hufs.ac.kr
††† 종신회원 : 경희대학교 전자정보학부 교수
jeong@khu.ac.kr
논문접수 : 2000년 12월 7일
심사완료 : 2001년 8월 9일

객체이다'라는 객체 지향 패러다임의 근본과 달리, Java의 모든 것은 값과 객체로 구성되며 이들간에는 호환성이 제공되지 않는다. 그러므로 원시 값이 클래스 타입의 변수에 저장되지 못하며 클래스 타입의 값이 원시 타입 변수에 저장되지 못한다. 둘째, 파라메트릭 다형성(parametric polymorphism)이 타입 변수를 이용하지 않고, 범용 참조 타입(universal reference type)이면서 모든 클래스 타입의 상위타입(super type)인 Object 클래스를 이용하여 제한적으로 제공된다[2]. C++[3]이나 Eiffel[4] 등에서는 타입 변수를 이용한 포괄 클래스(generic class)를 선언하고 나중에 인스턴스화할 수 있다. 이에 반하여 Java에서는 포괄 클래스를 선언할 수 없으며, Object 타입의 변수에 하위 타입의 값을 저장하고 나중에 실제적으로 참조하는 타입의 값으로 명시적으로 캐스팅시키게 된다. Java에서 파라메트릭 다형성을 지원하기 위한 여러 방안이 제시되었다[2, 5, 6].

Java에는 boolean, char, int, double 등 원시 타입에 대응되는 Boolean, Character, Integer, Double 등의 래퍼(wrapper) 클래스들이 있다. 각 래퍼 클래스는 원시 타입에 대한 유용한 메소드를 제공하며, 각 타입에 대한 상수를 정의하고, 가장 중요한 것으로, 원시값을 저장하는 객체(앞으로 이를 래퍼 객체라 함)를 생성하고, 래퍼 객체에 대한 여러 메소드를 제공한다[1]. 예를 들어, int 타입에 대한 래퍼 클래스인 Integer 클래스에는 정수를 문자열로 변환하는 static 메소드 toString(), 문자열을 정수로 변환하는 static 메소드 parseInt() 등이 정의되며, 정수의 최소값을 나타내는 상수 MIN_VALUE, 최대값을 나타내는 상수 MAX_VALUE가 정의되어 있다. 또한 주어진 정수를 Integer 클래스의 래퍼 객체로 만드는 객체 생성자 Integer(), Integer 래퍼 객체의 정수값을 반환하는 메소드 intValue(), Integer 래퍼 객체의 정수값을 double 값으로 변환하여 반환하는 메소드 doubleValue() 등이 정의되어 있다.

래퍼 객체는 객체 참조가 필요한 곳에서 원시값이 사용될 수 있게 한다. 즉, 클래스 타입의 값이 요구되는 경우 원시값은 래퍼 객체로 캡슐화되어 사용된다. 특히 매개변수가 클래스 타입인 메소드의 실매개변수로 래퍼 객체가 이용될 수 있다. 그러므로 래퍼 클래스는 원시값과 Java 객체를 연결시키는 교량역할을 하는 클래스이다. 그림 1의 Java 프로그램은 int 값 1과 4를 Vector 객체의 원소로 첨가시키고 검색하기 위하여 래퍼 클래스 Integer를 이용하는 Java 프로그램이다.

그림 1 프로그램에서 사용된 Vector 객체는 모든 클래스의 객체를 원소로 저장할 수 있으며, 저장된 이들

```
import java.util.Vector;
class Example1 {
   public static void main(String a[]) {
      Vector intVector = new Vector();
      intVector.addElement(new Integer(1));
      intVector.addElement(new Integer(4));

      for (int i=0; i<intVector.size(); i++) {
               int n = ((Integer) intVector.elementAt
            (i)).intValue();
         System.out.println(i+1 +"-th element: " + n);
      }
   }
}
```

그림 1 여러 int 값을 벡터에 저장하고 검색하는 Java 프로그램

객체를 접근할 수 있다. Vector 클래스에서 객체를 원소로 저장하는 메소드인 addElement()와 특정 순서위치에 저장된 원소 객체를 접근하는 메소드인 elementAt()의 시그니처는 다음과 같다.

public synchronized void addElement(Object obj)
// obj가 참조하는 객체를 저장
public synchronized Object elementAt(int index)
// index 위치의 원소를 접근

addElement() 메소드의 매개변수의 타입은 최상위 클래스 타입인 Object 타입이므로 원시 값이 아닌 모든 클래스 타입의 값이 실매개변수가 될 수 있다. 또한 elementAt() 메소드의 반환값 타입은 Object 타입이며, 실제 반환된 객체는 해당 타입의 객체로 캐스팅된 후 그 타입의 값으로 사용될 수 있다.

그림 1에서 int 타입의 값 1, 4가 래퍼 클래스인 Integer의 객체로 변환된 후 addElement() 메소드의 실매개변수로 사용되었다. 이 작업은 실제로 저장하고자 하는 원시값을 intVector에 저장시키기 위해 반드시 필요한 작업이다.

intVector에 저장된 int 타입의 값을 출력하기 위해서, 벡터의 특정 위치에 저장된 원소인 Integer 클래스의 객체를 elementAt() 메소드를 이용하여 검색한다. 이 때 elementAt() 메소드는 Object 타입의 값을 반환하는데, 실제 반환되는 객체의 타입은 Object 타입의 하위 타입인 Integer 타입이다. 이 반환된 객체를 Integer 타입으로 캐스팅시켜 Integer 타입으로 변환한다. 이것은 반환된 객체에서 저장된 int 값을 얻는 메소드 intValue()를 호출하기 위해서 반드시 필요한 작업이다. 프로그램에서 ((Integer) intVector.elementAt(i)).intValue () 부분이 이 과정을 수행한다. intValue() 메소드를 이용하여

Integer 객체의 정수값이 구해지면 이를 출력한다.

그림 1 프로그램을 통하여 다음의 2가지 사실을 확인할 수 있다. 첫째, 원시 값이 클래스 타입의 변수에 저장되기 위해서 래퍼 객체가 생성되어야 한다. 둘째, 래퍼 객체의 값이 원시 타입의 변수에 저장되기 위해서 래퍼 객체에서 값을 반환하는 메소드가 호출되어야 한다. 이러한 객체 생성과 값 반환 메소드 호출은 번거롭고 오류를 발생하기 쉬운 작업이다.

본 논문에서는 원시 타입과 대응되는 래퍼 클래스간의 타입 호환성을 제공하는 방법을 제시하고자 한다. 그 결과 객체 참조가 필요한 곳에서 원시값을 사용할 수 있으며 래퍼 객체를 원시타입의 변수에 바로 대입할 수 있게 된다. 원시 타입과 래퍼 클래스간의 타입 호환성이 제공될 경우 이의 이점은 다음과 같다. 첫째, 명시적인 래퍼 객체의 생성, Object 타입 값의 래퍼 클래스 타입으로의 캐스팅, 원시 값으로 변환하는 메소드의 호출 등의 번거로움이 없어진다. 둘째, 래퍼 객체를 원시값으로 변환시킬 때의 오류의 가능성을 줄인다. 셋째, 원시값을 클래스 타입의 객체처럼, 래퍼 객체를 원시타입의 값처럼 자유롭게 이용할 수 있게된다. 이것은 객체 지향 패러다임이 추구하는 객체로의 통합에 더 다가가게 할 것이다.

본 논문은 Java에서 제공되지 않은 새로운 기능을 제공하는 것이다. 이와 비슷하게, 기존의 프로그래밍 언어를 확장하여 새로운 기능을 추가하는 많은 연구가 [2, 5, 6, 9, 10] 등에서 이루어졌다. Bennet[9]는 표준 C 언어에 집합 타입과 연산을 지원하기 위하여 집합의 각 원소에 대해 작동되는 foreach 연산을 설계하였다. 그는 집합 타입 변수와 foreach 연산의 구현에서 C 마크로(macro)를 활용한 프리프로세싱을 이용하였다. 또한 Bergin[10]은 Pascal을 확장하여 객체 참조와 다중 상속 등 객체 지향 기능을 지원하는 방안을 제시하였다.

한편 Java에 대한 확장도 활발히 연구되고 있다. Odersky와 Walder[6]는 Java에 파라메트릭 다형성, 고차 함수 그리고 대수 추상 데이타를 지원하기 위하여 Java를 확장한 언어 Pizza를 설계하였으며, Pizza를 Java로 변환하는 방법을 제시하였다. Solorzano와 Alagic[2] 그리고 Agesen 등[5]은 타입 매개변수를 사용하여 포괄 클래스와 인터페이스가 정의될 수 있도록 Java를 확장하였다.

이와 같이 프로그래밍 언어의 기능적 확장을 위한 연구가 많이 진행되었으나, 우리의 조사에 따르면, 원시타입과 래퍼 클래스간의 호환성을 지원하기 위한 연구는 아직 이루어지지 않은 상태이다. 그 이유는 아직 값과 객체의 하나로 통합할 필요성이 강하게 제기되지 않았으며, 직접 참조되는 값과 포인터를 이용하여 간접적으로 참조되는 객체의 근본적인 차이점 때문일 것이다. 우리는 객체 지향 패러다임에서 원칙적으로 모든 것은 객체로 캡슐화되어야 된다고 판단한다. 그러나 성능면을 고려해서 원시값은 캡슐화되지 않고서 직접 참조되는 특수한 객체로 다루어지고 있으며, 이로 인하여 객체와 원시값 간의 비호환성이 발생하고 있다고 본다. 값만을 다루는 기존의 FORTRAN, C 등의 언어에서 객체를 다루는 객체 지향 언어로 발전하는 과정에서 출현한 Java에서는 아직 값과 객체의 통합이 이루어지지 않은 상태이다. 내부 구현에서는 값과 객체를 구별하지만 사용 측면에서는 값과 객체를 통합한 보다 편리한 객체 지향적인 언어를 제공하기 위한 방안을 모색하는 것이 본 논문의 목적이다.

2. 원시 타입과 클래스 타입간의 호환성

원시 타입과 래퍼 클래스간의 호환성이 제공될 경우, 그림 1의 프로그램은 그림 2와 같이 표현되어 질 수 있다.

```
import java.util.Vector;
class Example1 {
   public static void main(String a[]) {
      Vector intVector = new Vector();
      intVector.addElement(1);
      intVector.addElement(4);

      for (int i=0; i<intVector.size(); i++) {
        int n = intVector.elementAt(i);
        System.out.println(i+1 +"-th element: " + n);
      }
   }
}
```

그림 2 원시 타입과 래퍼 클래스간의 타입 호환성이 제공되는 가상적 프로그램

그림 2에서 int 값 1과 4가 바로 addElement()의 실매개변수로 사용되었고, elementAt(i)의 반환값이 바로 int 변수 n에 저장되었다. 그림 2의 프로그램이 그림 1의 프로그램보다 휠씬 간단하며 작성도 용이함을 알 수 있다.

2.1 타입 비호환성에 의한 오류의 제거

그림 2의 프로그램을 Java 컴파일러로 컴파일하면 그림 3과 같은 컴파일 오류가 발생한다. 즉, 원시값을 클래스 타입의 변수에 저장하거나 클래스 타입의 값을 원시 타입의 변수에 저장하려고 할 때 이러한 오류가 발생하며, 이것은 원시 타입과 클래스 타입의 비호환성에 의한 결과이다.

```
Example2.java:7: Incompatible type for method.
        Can't convert int to java.lang.Object.
                        intVector.addElement(1);
                                             ^
Example2.java:10: Incompatible type for method.
        Can't convert int to java.lang.Object.
                        intVector.addElement(4);
                                             ^
Example2.java:13: Incompatible type for declaration.
        Can't convert java.lang.Object to int.
                        int n = intVector.elementAt(i);
                                                     ^
3 errors
```

그림 3 그림 2의 프로그램을 컴파일할 때의 오류 메시지

원시 타입과 래퍼 클래스 타입간의 호환성은 이들 타입간의 자동적인 변환코드를 첨가하므로 가능하다. Java에서는 이미 자동적 코드 첨가가 부분적으로 이루어지고 있다. 예를 들어, Java의 객체 생성자 구현에서 객체 생성자 호출이 생략되면 상위 클래스의 객체 생성자를 호출하는 코드(super();)가 자동적으로 첨가된다[7]. 이와 비슷하게 원시값과 래퍼 객체 사이에서 비호환적인 대입이 이루어질 경우에 자동적으로 변환 코드를 첨가하는 것이다.

자동적 변환 코드의 첨가가 이루어지면 그림 2의 프로그램은 그림 1의 프로그램으로 변환된다. 이러한 자동적인 변환을 위한 기본적인 방법이 2.2절과 2.3절에서 소개되며 3장에서는 기본적인 변환방법의 문제점과 개선된 변환방법을 소개한다.

2.2 원시값의 래퍼 객체 변환

변환 코드의 첨가는 원시값을 래퍼 객체로 변환하는 것과 래퍼 객체를 원시값으로 변환하는 것으로 나누어진다. 먼저 원시값을 래퍼 클래스의 변수에 저장할 때 필요한, 원시값을 래퍼 객체로 변환하는 방법을 소개한다.

그림 2 프로그램의 intVector.addElement(1)에서는 실매개변수 1이 Object 타입의 형식매개변수에 전달되어야 한다. 이 때에는 1의 타입이 int이므로 1을 대응되는 Integer 클래스의 객체로 변환시키는 코드를 첨가한다. 그 결과 주어진 메소드 호출은 intVector.addElement (new Integer(1))로 바뀌어지게 된다. 이러한 변환과정은 다음의 원시값의 변환규칙 TR_{prim}로 표현된다.

원시값의 변환규칙 TR_{prim}: o = v에서 v가 원시 타입 T의 값이며 T의 대응되는 래퍼 클래스가 C_T이고 변수 o가 클래스 타입 C의 변수일 때, C가 C_T의 상위 타입이면 o = **new** C_T(v)로 변환한다. 그렇지 않으면 오류를 발생한다. 이 규칙은 v가 실매개변수이며 o가 형식매개변수일 때에도 적용된다.

2.3 래퍼 객체의 원시 값 변환

래퍼 객체를 원시값으로 변환하는 것은 원시값을 래퍼 객체로 변환하는 것보다 복잡하다. 그림 2 프로그램의 문장 int n=intVector.elementAt(i);에서 intVector.elementAt(i)의 반환된 Integer 객체가 int 타입의 변수 n에 저장되어야 한다. 반환된 값의 타입이 int의 래퍼 클래스인 Integer이므로 intValue() 메소드를 이용하여 int 값을 추출하여 n에 저장하여야 한다. 그 결과 주어진 문장은 다음과 같이 변환된다.

int n=((**Integer**) intVector.elementAt(i)).**intValue()**;

이러한 변환과정은 다음의 첫 번째 래퍼 객체의 변환규칙인 $TR_{wrap}1$로 표현된다.

첫 번째 래퍼 객체의 변환규칙 $TR_{wrap}1$: **v** = **o**에서 o가 클래스 타입 C의 값이고 변수 v가 원시 타입 T의 변수이며 T의 대응 래퍼 클래스가 C_T이며 C_T의 객체에서 타입 T의 값을 구하는 메소드가 m()일 때, C가 C_T의 상위 타입 또는 하위 타입이면 **v** = ((**C_T**) **o**).**m()**으로 변환한다. 이 규칙은 **o**가 실매개변수이며 v가 형식매개변수일 때에도 적용된다.

TR_{prim}과 $TR_{wrap}1$이 적용되는 예는 다음과 같다.

Number nbr = 3;
  ⇒ Number nbr = **new Integer(3)**;
int n = nbr;
  ⇒ int n = ((**Integer**) nbr).**intValue()**;
Object obj = 45.6;
  ⇒ Object obj = **new Double(45.6)**;
double d = obj
  ⇒ double d = ((**Double**) obj).**doubleValue()**;

TR_{prim}과 $TR_{wrap}1$에서의 원시 타입, 대응되는 래퍼 클래스, 래퍼 객체에서 원시값을 구하는 메소드는 표 1과 같이 정의된다. TR_{prim}과 $TR_{wrap}1$을 적용하여 그림 2의 프로그램을 변환하면 그림 1의 프로그램으로 변환된다.

표 1 TR_{prim}과 $TR_{wrap}1$에서의 타입과 대응 래퍼 클래스

원시 타입	대응 래퍼 클래스	래퍼 클래스의 상위 클래스	래퍼 객체의 값 추출 메소드
boolean	Boolean	Object	booleanValue()
char	Character	Object	charValue()
byte	Integer	Number	byteValue()
short	Integer	Number	shortValue()
int	Integer	Number	intValue()
long	Long	Number	longValue()
float	Float	Number	floatValue()
double	Double	Number	doubleValue()

3. 래퍼 클래스간의 호환성 제공

3.1 래퍼 클래스간 비호환성의 문제점

래퍼 객체를 값으로 변환하는 규칙 $TR_{wrap}1$의 v=o에서 클래스 C의 값 o가 래퍼 클래스 타입 C_T로 캐스팅된다. 그런데 o의 타입 C가 C_T의 상위 타입이라 하더라도 실제로 참조하는 객체가 C_T로 캐스팅될 수 없는 객체인 경우에는 실행시간 오류가 발생될 수 있다. 그림 4는 규칙 $TR_{wrap}1$에 의해 변환될 경우 실행시간에 오류가 발생하는 예를 보인다.

```
import java.util.Vector;
class Example2 {
   public static void main(String a[]) {
      Vector aVector = new Vector();

      aVector.addElement(1);
      double d = aVector.elementAt(0);
       System.out.println("d = " + d);
   }
}
```

그림 4 $TR_{wrap}1$로 변환하면 문제 발생하는 프로그램

그림 4의 프로그램은 TR_{prim}과 $TR_{wrap}1$에 의해 그림 5와 같이 변환된다.

```
import java.util.Vector;
class Example2 {
   public static void main(String a[]) {
      Vector aVector = new Vector();

      aVector.addElement(new Integer(1));
      double d = ((Double) aVector.elementAt(0)).
                doubleValue();
       System.out.println("d = " + d);
   }
}
```

그림 5 TR_{prim}과 $TR_{wrap}1$에 의해 그림 4의 프로그램이 변환된 프로그램

그림 5의 프로그램은 컴파일시간의 타입 검사는 정상적으로 통과한다. 그러나 실행시 aVector.elementAt(0)의 결과로 aVector에 저장된 Integer 객체가 반환되므로 Double 타입으로 캐스트될 수 없다. 그러므로 실행중에 "java.lang.ClassCastException: java.lang.Integer" 오류가 발생한다. 이 오류는 Integer 클래스 타입이 Double 타입과 비호환적이므로 발생하는 오류이다.

Java의 원시 타입에서 byte는 short로, short는 int로, int는 long으로, long은 float로, float는 double로 자동적으로 타입 변환이 이루어진다. 그 반대의 경우에는 반드시 명시적인 캐스트가 필요하다. 그런데 표 1에서와 같이, 래퍼 클래스들은 Object 또는 Number를 상위 클래스로 가지므로 원시 타입에서 제공되는 호환성이 래퍼 클래스들 사이에서 제공되지 않는다.

3.2 래퍼 클래스 계층을 이용한 래퍼 클래스간의 호환성 제공

3.1절에서 래퍼 클래스간의 비호환성으로 인한 문제가 제기되었다. 그러므로 int 타입이 double 타입을 가지는 것과 마찬가지로 래퍼 클래스간의 호환성을 제공할 필요가 있다. 이를 위한 가장 쉬운 방법이 변환규칙 $TR_{wrap}1$을 수정하지 않고 java.lang 패키지의 래퍼 클래스들을 수정하여 래퍼 클래스 계층을 새로이 형성하는 것이다. 그림 6은 래퍼 클래스 사이의 호환성을 제공하기 위하여 기존의 java.lang 패키지의 래퍼 클래스들의 선언을 수정한 것이다.

```
public class Double extends Number implements Comparable {...}
public class Float extends Double implements Comparable {...}
public class Long extends Float implements Comparable {...}
public class Integer extends Long implements Comparable {...}
public class Short extends Integer implements Comparable {...}
public class Byte extends Short implements Comparable {...}
public final class Character extends Integer
          implements java.io.Serializable, Comparable {...}
public class Boolean implements Comparable { ... }
```

그림 6 래퍼 클래스간의 호환성을 위한 래퍼 클래스의 수정

그림 6과 같이 래퍼 클래스의 상위 클래스를 수정하면 원시 타입에서 제공되는 호환성이 래퍼 클래스에서도 제공되어진다. 그리하여 그림 5의 프로그램은 컴파일된 후 정상적으로 수행된다.

본 논문에서는 래퍼 클래스 호환성을 위해 래퍼 클래스를 수정하는 방법을 **래퍼 클래스 계층방법**이라고 한다. 이 방법은 Byte 래퍼 클래스의 경우 상위 클래스인 Short, Integer, Long, Float, Double 클래스의 모든 멤버를 불필요하게 상속받게 되며, 여러 메소드를 오버라이딩하게 된다. 다른 하위 클래스도 역시 여러 상위 클래스로부터 멤버를 상속받게 되며 여러 메소드를 오버라이딩한다. 이로 인하여 객체의 생성이나 메소드 호출시 성능이 저하된다.

3.3 java.lang.Number 클래스를 이용한 래퍼 클래스간의 호환성 제공

본 절에서는 래퍼 클래스 계층방법의 성능상의 문제점을 극복하기 위해 Number 클래스 이용방법을 소개한다. java.lang.Number 클래스는 abstract 클래스로서, 숫자 래퍼 클래스의 공통적 abstract 메소드인 byteValue(), shortValue(), intValue(), longValue(), floatValue(), doubleValue() 등을 선언하고 있다. 이 클래스를 이용하면 간단하게 래퍼 클래스간의 호환성을 제공할 수 있으며, 이를 **Number 클래스 이용방법**이라고 한다. Number 클래스 이용방법은 새로운 래퍼 객체 변환규칙 $TR_{wrap}2$를 사용하며, $TR_{wrap}2$는 다음과 같다.

> **두 번째 래퍼 객체의 변환규칙** $TR_{wrap}2$: **v** = o에서 o가 클래스 타입 C의 값이고 변수 v가 원시 타입 T의 변수이며 T의 대응되는 래퍼 클래스가 C_T이며 클래스 C_T의 객체에서 타입 T의 값을 구하는 메소드가 m()일 때, C가 C_T의 상위 타입이면서 T가 boolean 또는 char이면 **v** = **((**C_T**) o).m()**으로 변환하고, T가 그 외의 타입이면 **v** = **((Number) o).m()**으로 변환한다. 이 규칙은 o가 실매개변수이며 v가 형식매개변수일 때에도 적용된다.

```
import java.util.Vector;
class Example2 {
  public static void main(String a[]) {
   Vector aVector = new Vector();

   aVector.addElement(new Integer(1));
   double d = ((Number) aVector.elementAt(0)).
              doubleValue();
    System.out.println("d = " + d);
  }
}
```

그림 7 TR_{prim}과 $TR_{wrap}2$에 의해 그림 4의 프로그램이 변환된 프로그램

그림 7은 TR_{prim}과 $TR_{wrap}2$를 이용하여 그림 4의 프로그램을 변환한 것이다. 그림 7에서 aVector에서 검색된 Integer 클래스의 객체가 double 변수 d에 저장되기 위해 Number 타입으로 캐스팅된 후 Integer 클래스의 메소드 doubleValue()를 통하여 double 값이 추출된다.

Number 클래스 이용방법은 기존의 래퍼 클래스를 수정하지 않고서 래퍼 객체를 원시값으로 변환시킬 수 있다. 그런데 이 방법은 다음의 두 가지 문제를 가지고 있다. 첫째, float 값이나 double 값을 래퍼 객체로 만든 후 이를 int 변수나 long 변수에 저장할 경우에는 오류가 생기지 않는다는 것이다. 둘째, Java에서 char 타입은 int 변수나 long 변수에 저장할 수 있지만 char 타입의 값을 객체로 만든 후 이를 정수 변수나 실수 변수에 저장할 수 없다. 이는 래퍼 클래스 Character에 intValue(), floatValue(), doubleValue() 메소드가 선언되지 않았기 때문이다. 그림 8에서 이러한 문제가 발생하는 경우를 보여주고 있다.

```
import java.util.Vector;
class Example3 {
  public static void main(String a[]) {
     Vector aVector = new Vector();
     // 실수를 aVector에 저장한 후 정수 변수에 다시 저장
     aVector.addElement(new Double(1.2));
     int n = ((Number) aVector.elementAt(0)).intValue();
     System.out.println("n = " + n);
     // 문자를 aVector에 저장한 후 정수 변수에 다시 저장
     aVector.addElement(new Character('C'));
     int n = ((Number) aVector.elementAt(1)).intValue();
     System.out.println("n = " + n);
  }
}
```

그림 8 $TR_{wrap}2$의 문제점을 보이는, 변환된 프로그램

그림 8에서 double 값 1.2를 aVector에 저장한 후 이를 검색하여 int 변수에 저장하여도 아무런 오류가 발생하지 않는다. 그러나 Java의 원시타입간의 호환성에 의하면 이는 오류이어야 한다. 한편 char 값을 aVector에 저장한 후 다시 이를 검색하여 int 변수에 저장하면 실행시에 오류가 발생하는데, 문자값을 int 변수에 저장하는 것은 Java의 오류가 아니다.

3.4 래퍼 인터페이스 계층을 이용한 래퍼 클래스간의 호환성 제공

Number 클래스를 이용한 호환성 제공은 Java 원시타입간의 호환성과 다른 호환성을 제공하므로 만족스러운 해결책이 되지 못한다. 본 논문에서 제안하는 가장 바람직한 해결책은 그림 9와 같이 원시 타입에 대응되는 인터페이스들을 새로이 선언하고 래퍼 클래스를 이들 인터페이스를 구현하는 클래스로 수정하는 것이다. 이 때, 원시 타입에 대응되는 인터페이스는 원시 타입의 포함 관계에 따라 인터페이스 계층을 형성하도록 한다. 본 논문에서는 이를 **래퍼 인터페이스 계층방법**이라고 한다.

그림 9에서 래퍼 클래스들간의 호환성을 위하여 원시타입에 대응되는 여러 인터페이스가 선언되었다. NumberType에는 여러 멤버가 선언되었으나 나머지 인터페이스에는 멤버가 하나도 없다. 왜냐하면 나머지 인터페이스는 하위 인터페이스 타입이 상위 인터페이스를 갖는 서브타입 다형성(subtype polymorphism)[8]을 제공하기 위해 선언된 인터페이스들이기 때문이다.

```
public interface NumberType {
   // 래퍼 클래스간의 호환성을 위한 abstract 메소드를 정의
   double doubleValue();
   float floatValue();
   long longValue();
   int intValue();
   short shortValue();
   byte byteValue();
   char charValue();
}

// 이들 인터페이스는 원시 타입과 동일한 호환성을 제공한다.
public interface DoubleType extends NumberType { }
public interface FloatType extends DoubleType { }
public interface LongType extends FloatType { }
public interface IntType extends LongType { }
public interface ShortType extends IntType { }
public interface ByteType extends ShortType { }
public interface CharType extends IntType { }
public interface BooleanType {
          boolean booleanValue();
}

// 래퍼 클래스는 대응되는 인터페이스를 구현하도록 수정된다.
public final class Double implements Comparable, DoubleType {...}
public final class Float implements Comparable, FloatType {...}
public final class Long implements Comparable, LongType {...}
public final class Integer implements Comparable, IntType {...}
public final class Short implements Comparable, ShortType {...}
public final class Byte implements Comparable, ByteType {...}
public final class Character
     implements java.io.Serializable, Comparable, CharType {...}
public final class Boolean implements Comparable, BooleanType {...}
```

그림 9 래퍼 인터페이스 계층방법에서의 클래스들

래퍼 클래스는 대응되는 인터페이스를 구현하고 있으므로 이들 클래스에서는 NumberType에 선언된 모든 abstract 메소드를 구현하게 된다. 그림 9의 수정된 래퍼 클래스는 기존의 Number 클래스를 확장하지 않고 대응되는 인터페이스를 구현한다. 또한 Number 클래스에는 charValue() 메소드가 선언되어 있지 않지만 NumberType 인터페이스에는 이 메소드가 선언되어 있으므로 NumberType의 하위 클래스는 반드시 이 메소드를 구현하여야 한다. 이 메소드는 Character 클래스의 객체가 int, long, float, double 타입의 변수에 저장될 수 있기 위해 필요한 메소드이다.

래퍼 인터페이스 계층방법을 이용하여 래퍼 객체를 원시 타입의 변수에 저장하는 규칙 $TR_{wrap}3$은 다음과 같다.

$TR_{wrap}3$에서의 원시타입과 이에 대응 래퍼 클래스 및 인터페이스, 래퍼 객체에서 원시값을 추출하는 메소드 등은 표 2에 나타나 있다.

> **세 번째 래퍼 객체의 변환규칙 $TR_{wrap}3$:** **v = o**에서 o가 클래스 타입 C의 값이고 변수 v가 원시 타입 T의 변수이며 T의 대응되는 래퍼 클래스가 C_T이며 T의 대응되는 인터페이스가 I_T의이며 클래스 C_T의 객체에서 타입 T의 값을 구하는 메소드가 m()일 때, C가 I_T의 상위타입 또는 하위타입이면 **v = ((**I_T**) o).m()**으로 변환한다. 이 규칙은 o가 실매개변수이며 v가 형식매개변수일 때에도 적용된다.

표 2 TR_{prim}과 $TR_{wrap}3$에서의 원시 타입과 대응 래퍼 클래스 및 인터페이스

원시 타입	대응 래퍼 클래스	대응 인터페이스	래퍼 객체의 값 추출 메소드
boolean	Boolean	BooleanType	booleanValue()
char	Character	CharType	charValue()
byte	Integer	ByteType	byteValue()
short	Integer	ShortType	shortValue()
int	Integer	IntType	intValue()
long	Long	LongType	longValue()
float	Float	FloatType	floatValue()
double	Double	DoubleType	doubleValue()

래퍼 인터페이스 계층방법과 $TR_{wrap}3$을 이용할 경우, 그림 8에 해당되는 프로그램은 그림 10과 같다. 그림 10의 프로그램이 수행되면 double 값 1.2를 aVector에 저장한 후 이를 꺼내어 int 변수 n에 저장할 때에 오류가 발생하게 된다. 한편 문자값 'C'에 대해서는 오류가 발생하지 않는다. 그러므로 $TR_{wrap}3$을 이용할 경우 $TR_{wrap}2$에서의 과도한 호환성과 char 값의 문제점이 해결된다.

```
import java.util.Vector;
class Example3 {
   public static void main(String a[]) {
      Vector aVector = new Vector();

      aVector.addElement(new Double(1.2));
      int n = ((IntType) aVector.elementAt(0)).intValue();
      System.out.println("n = " + n);

      aVector.addElement(new Character('C'));
      int n = ((IntType) aVector.elementAt(1)).intValue();
      System.out.println("n = " + n);
   }
}
```

그림 10 $TR_{wrap}3$을 이용하여 변환된 Java 프로그램

4. 비교분석과 구현

4.1 래퍼 클래스간의 호환성 제공방법의 비교

표 3 래퍼 클래스간의 호환성 제공방법의 비교

	래퍼 클래스 계층 방법	java.lang.Number 클래스 이용 방법	래퍼 인터페이스 계층방법
래퍼 클래스간의 호환성 제공 방법	원시 타입의 포함관계와 동일한 래퍼 클래스들의 계층을 형성한다.	래퍼 클래스들을 Number 인터페이스의 하위 클래스가 되게 한다.	원시 타입의 포함관계와 동일한 래퍼 인터페이스 계층을 형성한다.
사용된 변환규칙	TR_{prim}, $TR_{wrap}1$	TR_{prim}, $TR_{wrap}2$	TR_{prim}, $TR_{wrap}3$
래퍼 객체 o를 정수 값으로 변환하는 예	((Integer) o).intValue()	((Number) o).intValue()	((IntType) o).intValue()
호환성 제공을 위한 수정의 정도	래퍼 클래스의 수정이 필요	없음	래퍼 클래스의 수정과 새로운 인터페이스의 정의가 필요
장점	원시 타입간의 호환성과 일치하는 래퍼 클래스 계층이 형성된다.	기존의 래퍼 클래스에 변화가 없다.	원시 타입간의 호환성을 래퍼 클래스에서도 제공하며 성능에서도 우수하다.
단점	클래스 계층으로 인한 불필요한 멤버의 상속과 메소드 오버라이딩되며 래퍼 클래스가 final 클래스가 아님.	원시 타입간의 호환성을 벗어나는 과도한 호환성과 char 값의 처리에 문제가 있다.	새로운 많은 인터페이스가 정의된다.

3장에서 제시된 래퍼 클래스간의 호환성을 위한 3가지 방법을 정성적으로 비교한 결과는 표 3에 나타나있다. 표 3을 살펴보면 java.lang.Number 클래스 이용방법은 원시타입간의 호환성과 다른 래퍼 클래스간의 호환성을 제공하는 문제가 있으므로 래퍼 클래스간의 호환성 제공방법으로 부적당하다. 래퍼 클래스 계층방법과 래퍼 인터페이스 계층방법이 원시타입간의 호환성과 일치하는 래퍼 클래스간 호환성을 제공함을 알 수 있다. 래퍼 클래스 계층방법은 기존의 래퍼 클래스에 대한 수정이 적은 반면에 래퍼 인터페이스 계층방법은 여러 인터페이스가 새로이 정의되는 등 많은 다소 많은 수정이 요구된다.

한편 래퍼 클래스 계층방법은 다소 긴 클래스 계층이 형성되므로 상위 클래스의 멤버가 하위 클래스로 불필요하게 상속되며, 메소드 오버라이딩이 반복되고, 래퍼 클래스가 final 클래스가 되지 못한다. 이로 인하여 래퍼 객체를 생성하고 래퍼 객체에서 값을 꺼낼 때 성능이 다소 떨어질 것으로 예상된다. 이에 반하여 래퍼 인터페이스 계층방법에서는 이러한 성능의 저하가 발생하지 않는다.

4.2 래퍼 클래스 계층방법과 래퍼 인터페이스 계층방법의 성능비교

원시값과 래퍼 객체간의 변환은 기본적인 변환이므로 변환의 성능도 중요할 요인이 될 것이다. 성능적인 면에서는 래퍼 클래스 계층방법보다 래퍼 인터페이스 계층방법이 우수할 것으로 예상되는데, 실제적인 성능측정을 통하여 이를 확인해 보았다.

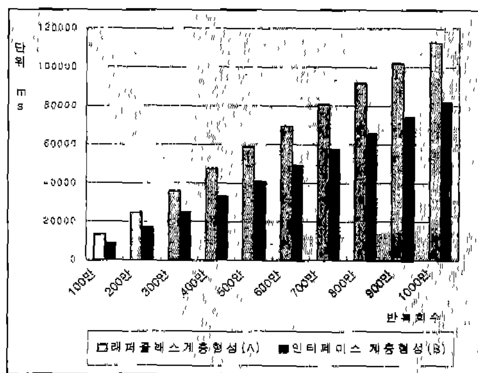
성능비교를 위하여 시뮬레이션 프로그램을 작성한 후, 래퍼 클래스 계층방법(TR_{prim}과 $TR_{wrap}1$ 이용)과 래퍼 인터페이스 계층방법(TR_{prim}과 $TR_{wrap}3$ 이용)의 프로그램으로 변환하고 이에 대한 수행시간을 측정하였다. 그림 11은 변환되기 전의 프로그램이다. 이 프로그램은 1.0, 1.0f, 1, (short)1, (byte) 1 등의 값을 래퍼 객체에 저장하고 다시 원시 타입의 변수에 저장하는 것을 100만 번에서 1000만 번까지 반복하면서 수행되는 시간을 측정한다. 이 프로그램을 통하여 변환된 두 방법의 실행시의 성능을 간접적으로 비교할 수 있다.

그림 11의 프로그램을 변환하기 위하여 그림 11의 프로그램과 같이 간단한 배정문에 대해 자동적인 변환을 수행하는 실험적인 프리프로세서를 Java로 개발하였다. 또한 변환된 프로그램을 수행하기 위하여 그림 6과 그림 9의 클래스들과 인터페이스들이 작성하였다. 성능측정은 Windows 98 OS의 IBM PC에서 실시되었으며, 수행시간에 대한 결과는 그림 12와 같다.

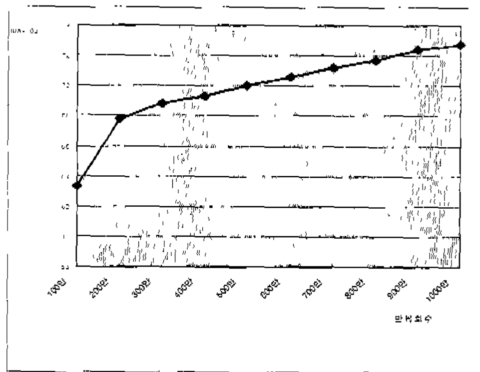
```
import java.util.*;
class CheckTime {
  public static void main(String a[]) {
    Object o; double d; float f; int i; short s; byte b;
    Date  D = new Date();
    for (int i=0; i<10000000; )  { // 1000만 번 변환을 실시
      o = 1.0;             d = o;
      o = 1.0f;            f = o;
       o = 1;               i = o;
       o = (short) 1;       s = o;
       o = (byte) 1;       b = o;
      if (++i % 1000000 == 0) // 100만번마다 소요시간을 출력
        System.out.println(new Date().getTime()-D.getTime());
    }
  }
}
```

그림 11 변환되기 전의 시뮬레이션 프로그램

그림 12에서 래퍼 인터페이스 계층방법의 수행속도가 래퍼 클래스 계층방법보다 약 27%에서 37% 정도 빠름을 알 수 있다. 이것은 2.2절과 4.1절에서 이미 언급된 것처럼 래퍼 객체를 생성할 때 상위 클래스의 객체 생성자가 연속적으로 호출되며, 래퍼 객체를 값으로 변환할 때 오버라이딩 메소드의 실행시간 바인딩으로 인한 오버헤드 때문일 것이다. 이러한 결과를 통하여 래퍼 인터페이스 계층방법이 래퍼 클래스 계층방법보다 다소 많은 수정을 요하지만 성능의 측면에서 보다 우수한 방법임을 확인할 수 있다.



a) 두 방법의 수행시간



b) 두 방법의 수행시간 비율

그림 12 래퍼 클래스 계층과 래퍼 인터페이스 계층에 대한 수행시간 비교

4.3 원시값과 래퍼 객체간 호환성 구현방법

원시 타입과 래퍼 클래스간의 호환성을 구현하는 방법에는 첫째 프리프로세싱을 통한 방법, 둘째 Java 컴파일러만 수정하는 방법, 셋째 java 컴파일러와 클래스 라이브러리를 수정하는 방법, 넷째 Java 가상 기계, Java 컴파일러 그리고 클래스 라이브러리를 모두 수정하는 방법 등이 있다.

프리프로세싱의 경우 기존의 Java 컴파일러, 클래스 라이브러리를 변경하지 않은 상태에서 호환성의 제공이 가능하다. 이를 위해서는 새로운 이름의 래퍼 클래스와 새로운 인터페이스를 정의하고, 프리프로세서를 개발하여야 한다. 이 방법은 기존의 Java 시스템에 영향을 미치지 않으면서 호환성을 제공할 수 있다는 장점이 있지만 프리프로세싱을 한 후 다시 컴파일해야하는 불편함이 있으며 컴파일 과정이 2 패스로 구성되므로 비효율성도 문제가 된다.

재컴파일링의 불편함과 비효율성을 해소하기 위해서는 Java 컴파일러에 호환성을 위한 처리부분을 구현할 필요가 있다. 이때, 기존의 클래스 라이브러리를 변경할 수도 있고 변경 없이 새로운 클래스와 인터페이스를 추가할 수도 있다. 이러한 Java 컴파일러의 변경은 기존의 여러 Java 프로그램에 영향을 미치지 않도록 해야하므로 신중히 고려되어야 한다. 호환성으로 인한 오류 메시지의 처리 등 완벽한 호환성 제공을 위해서는 Java 가상기계의 수정도 필요하다.

5. 결 론

Java의 원시 타입과 클래스 타입간에는 호환성이 없으므로 객체가 필요한 곳에서 원시값을 사용하기 위해서는 그 값을 저장하는 래퍼 객체의 생성이 필요하며, 래퍼 객체에서 원시값을 추출하기 위해 추출 메소드를 명시적으로 호출하여야 한다.

본 논문에서는 원시값을 객체처럼, 래퍼 객체를 원시값처럼 사용할 수 있도록 원시 타입과 래퍼 클래스간의 호환성을 제공하는 여러 방법을 제안하였다. 가장 간단한 방법으로 기존의 Java 클래스 라이브러리나 Java 가상기계를 수정하지 않고서 원시타입의 값을 래퍼 객체로, 래퍼 객체를 원시값으로 변환하는 변환규칙을 제시하였다. 이 방법은 간단한 반면에 래퍼 클래스간의 호환성이 제공되지 않는 단점이 있다.

래퍼 클래스간의 호환성을 제공하기 위해 3가지 방법을 제시하였다. 첫 번째, java.lang 패키지의 래퍼 클래스를 일부 수정하여 래퍼 클래스 계층을 형성하는 래퍼 클래스 계층방법은 연속적인 상속과 오버라이딩으로 인한 성능저하의 문제가 야기된다. 두 번째, java.lang.Number 클래스를 이용하는 Number 클래스 이용방법은 클래스 라이브러리의 수정 없이 간단하게 래퍼 클래스간의 호환성을 제공할 수 있는 반면에 과도한 호환성

과 char 값에서 문제가 발생한다. 최종적인 해결책으로 원시 타입에 대응되는 인터페이스를 정의하여 래퍼 인터페이스 계층을 형성하고 각 래퍼 클래스가 대응되는 인터페이스를 구현하게 하는 래퍼 인터페이스 계층방법을 제안하였다.

본 논문에서 제안한 방법에 대한 비교분석을 하였으며, 래퍼 클래스 계층방법과 래퍼 인터페이스 계층방법에 대한 시뮬레이션을 통하여 래퍼 인터페이스 계층방법이 성능적으로 약 30%정도 우수함을 확인하였다.

현재 간단한 배정문에 대한 변환을 수행하는 프리프로세서를 개발하고 본 논문의 변환방법을 적용시켜 논문에서 제한한 변환방법이 올바르게 작동하는 것을 확인할 수 있었다. 추후의 연구로 보다 완전한 프리프로세서를 개발하고, 이 변환방법을 Java 컴파일러에 반영시키는 것이 필요하다. Java 컴파일러에 반영하기 위한 구체적인 방안은 더 연구되어야 할 것이다.

참 고 문 헌

[1] K. Arnold and J. Gosling, The Java Programming language, Addison-Wesley, 1996.
[2] J. H. Solorzano, S. Alagic, Parametric Polymorphism for Java: A Reflective Solution, Proceedings of OOPSLA '98, pp. 216-225, 1998.
[3] B. Strostrup, The C++ programming Language, Addison-Wesley, 1993.
[4] B. Meyer, Eiffel: The Language, Prentice-Hall, 1992.
[5] O. Agesen, S. N. Freund and J. C. Mitchell, Adding Type Parameterization to the Java Language, Proceedings of OOPSLA '97, pp. 49-65, 1997.
[6] M. Odersky and P. Walder, Pizza into Java with virtual type, ACM Symposium on Principles of Programming Languages, pp 149-159. 1997.
[7] M. Grand, Java Language Reference, O'REILLY, 1997.
[8] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys, Vol. 17, No. 4, Dec. 1985.
[9] T. Bennet, "A Pragmatic Set Operation and its Implementation in C," Proc. of the ACM/SIGAPP Symposium on Applied Computing, 1992.
[10] J. Bergin, "Run-time Design for Object-Oriented Extensions to Pascal," Proc. of the ACM Conference on Computer Science, 1995.

김 성 기

1983년 서울대학교 컴퓨터공학과 졸업(공학사). 1985년 서울대학교 대학원 컴퓨터학과 졸업(공학석사). 1985년 ~ 1986년 삼성전자 연구원. 1992년 서울대학교 대학원 컴퓨터학과 졸업(공학박사). 1993년 ~ 현재 한신대학교 컴퓨터학과 부교수. 관심분야는 데이타베이스 시스템, 객체 지향 시스템.

김 상 철

1983년 서울대학교 컴퓨터공학과 졸업(공학사). 1986년 한국과학기술원 전산학과 졸업(이학석사). 1983년 ~ 1995년 한국전자통신연구소 선임연구원. 1994년 Michigan State Univ. 컴퓨터학과 졸업(공학박사). 1995년 ~ 현재 한국외국어대학 컴퓨터공학과 교수. 관심분야는 멀티미디어 시스템 및 통신, 컴퓨터 하드웨어 및 통신

정 병 수

1983년 서울대학교 컴퓨터공학과 졸업(공학사). 1985년 한국과학기술원 전산학과 졸업(석사). 1995년 Georgia Institute of Technology, College of Computing 졸업(박사). 1985년 3월 ~ 1989년 8월 한국데이타통신(주) 정보통신연구소 주임연구원. 1996년 ~ 현재 경희대학교 전자정보학부(컴퓨터공학전공) 부교수. 관심분야는 실시간 데이타베이스, 병렬 데이타베이스, 분산시스템 등임.