A Tour of Java III

Sungjoo Ha

March 20th, 2015

Review

- First principle – 문제가 생기면 침착하게 영어로 구글에서 찾아본다.
- 타입은 가능한 값의 집합과 연산의 집합을 정의한다.
- 기본형이 아니라면 이름표가 메모리에 달라 붙는다.
- 클래스로 사용자 정의 타입을 만든다.
- 프로그래밍은 복잡도 관리가 중요하다.
- OOP는 객체가 서로 메시지를 주고 받는 방식으로 프로그램을 구성해서 복잡도 관리를 꾀한다.

Objects

- 자바에서 사용자 정의 타입을 정의하고 사용하는 방법이 클래스이다.
- 코드가 어떤 “개념”을 대표하도록 하는 것이 클래스이다.
- 자바의 기초적인 타입, 연산자와 명령을 제외한 대부분의 기능이 클래스를 더 잘 만들거나 잘 사용할 수 있기 위해 존재한다.
  - 사고하기 쉽게
  - 읽기 쉽게
  - 사용하기 쉽게
  - 우아하게
  - 효율적으로 동작하게
  - 유지보수가 쉽게
  - 올바로 동작하게

Classes

- 클래스로 만들어진 타입은 기본형과 비슷한 방식으로 동작해야 한다.
- 물론 자체적인 의미와 연산을 갖게 된다.
- *BigInteger*는 마치 *int*를 사용하는 것과 같은 방식으로 사용할 수 있어야 한다.

Complex

```
class Complex {
    private int re, im;
    public Complex(int re, int im) {
        this.re = re;
        this.im = im;
    }
    public Complex(int r) {
        this(r, 0);
    }
    public Complex() {
        this(0, 0);
    }
    public int real() {
        return re;
    }
    public int imag() {
        return im;
    }
    public Complex add(Complex that) {
        return new Complex(this.re + that.real(), this.im + that.imag());
    }
    public Complex sub(Complex that) {
        return new Complex(this.re - that.real(), this.im - that.imag());
    }
}
```

Complex

- *private*과 *public* 키워드를 사용해서 구현과 인터페이스를 분리하였다.
- 여러 종류의 생성자를 정의하였다.
- *this*를 사용해서 생성자를 호출할 수 있다.
- 또 다른 *this*의 사용예로 자기 자신을 가리킬 수 있다.
  - *this.re = re*
- 객체가 개념적으로 같은지 여부는 *equals* 메소드를 만들어서 비교한다.
- *add*와 *sub*가 각각 새 객체를 생성하여 리턴한다.
- *new* 연산자와 함께 생성자를 부르고 이를 통해 클래스의 인스턴스를 생성한다.
  - 즉, *Complex* 타입의 객체를 생성한다.
  - *Complex* 타입의 일종(instance)을 만드는 것이기에 이를 인스턴스라고 부른다.

Complex

```
class Complex {
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + re;
        result = prime * result + im;
        return result;
    }

    public boolean equals(Object obj) {
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Complex other = (Complex) obj;
        return re == other.re && im == other.im;
    }
}
```

Complex – Equality

- 두 객체가 개념적으로 같은지 비교하기 위해 *equals* 메소드를 사용한다.
- *equals* 메소드는 항상 *Object*를 인자로 받아야 한다.
- 자기 자신과의 비교는 참을 리턴해야 한다.
- *equals* 메소드를 만들면 *hashCode* 메소드를 항상 함께 만들어줘야 한다.
- Eclipse에서 자동 생성할 수 있다.
  - Source 메뉴의 Generate hashCode and equals

User-Defined Types

```
public class ComplexTest {
    public static void main(String[] args) {
        Complex a = new Complex(12);
        Complex b = new Complex(1, 2);
        Complex c = new Complex(1, 2);
        Complex d = a.add(b);
        System.out.println(d.real() + " + " + d.imag() + "i");
        System.out.println(b == c);
        System.out.println(b.equals(c));
    }
}
```

- *Complex* 타입을 사용하는 방식이 기본형인 *int*와 크게 다르지 않음을 주시하라.

Equality

```
class Something {
    int member;
    String h;

    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((h == null) ? 0 : h.hashCode());
        result = prime * result + member;
        return result;
    }

    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Something other = (Something) obj;
        if (h == null) {
            if (other.h != null) return false;
        } else if (!h.equals(other.h)) {
            return false;
        }
        if (member != other.member) return false;
        return true;
    }
```

Equality

- 또 다른 *equals* 예제이다.
- 각 멤버에 적합한 방식으로 *equals*와 *hashCode*를 만들어야 한다.
- 멤버로 정수와 문자열이 있다. 논리적으로 두 객체가 같음은 같은 논리적인 내용을 담고 있음을 뜻한다.
- *hashCode*는 논리적으로 같은 객체라면 같은 값을 뱉어야 한다.
  - 보통 *hashCode*는 소수를 곱하고 멤버를 적절히 숫자로 표현한 값을 더하는 것을 반복하여 계산된다.
- 자신이 없다면 Eclipse를 사용해서 자동 생성하기를 권한다.

Container

```
class Vector {
    // construct a Vector
    public Vector(int s) {
        elem = new double[s];
        sz = s;
    }
    // elem access
    public double elementAt(int i) {
        return elem[i];
    }
    public void set(int i, double e) {
        elem[i] = e;
    }
    public int size() {
        return sz;
    }
    private int sz; // number of elements
    private double[] elem; // array of elements
}
```

- Container는 원소 여러 개를 한데 모은 것을 가리킨다.
- *Vector*는 그러므로 container이다.
- 과제 2번에서 구현할 linked list도 container이다.

Container Scope/Lifetime

```
public class VectorConsole {
    public static void main(String[] args) {
        int n = 10;
        Vector v1 = new Vector(n);
        {
            int m = 20;
            Vector v2 = new Vector(2*n);
        } // m and v2 are out of scope
    }
}
```

- 사용자가 *Vector*를 쓸 때에 마치 기본형을 사용하는 것과 비슷한 느낌으로 사용할 수 있어야 한다.
- *Vector*는 기본형과 같은 스코프 및 생명주기를 가진다.

Container Usage

```
public void add(double e) {
    // Append an element at the end of the sequence.
    // Increase the size accordingly.
}

Vector v1 = new Vector(10);
for (int i = 0; i < 20; ++i) {
    v1.add(i);
}
```

- Container는 원소를 들고 있기 위해 존재한다.
- 그러므로 원소를 container에 넣는 편리한 방법이 있어야 한다.
- *add()* 메소드는 원소를 시퀀스의 마지막에 추가하며 필요하다면 메모리 할당을 더 하고 크기를 증가시켜야 한다.
- 클래스를 구현할 때 사용자의 입장에서 어떻게 쉽게 사용할 수 있는지 고민하자.
  - 간결하게 표현할 수 있도록
  - 실수하지 않도록

Class Variables

```
class Block {
    public static String message = "What I cannot create, I do not understand";
}

public class BlockTest {
    public static void main(String[] args) {
        System.out.println(Block.message);
    }
}
```

- 특정 인스턴스가 아닌 클래스에 달린 변수를 만들고 싶을 때가 있다.
- *static* 키워드를 사용하여 달성할 수 있다.
- *message*는 특정 인스턴스가 없이 바로 접근 가능함을 주시하라.

Class Variable Lifetime

```
class Block2 {
    static int count = 0;
    public Block2() {
        count += 1;
        System.out.println("Block #" + count);
    }
}

public class BlockTest2 {
    public static void main(String[] args) {
        new Block2();
        new Block2();
        new Block2();
    }
}
```

- *static* 변수는 특정 인스턴스에 묶여 있지 않고 클래스에 달려 있다.
- 새 인스턴스가 생긴다고 새로 초기화 되는 것이 아니다.

Class Methods

```
class Block3 {
    private static int count = 0;
    private int id = 0;
    public Block3() {
        count += 1;
        id = count;
    }
    public int getID() {
        return id;
    }
    public void printRanking() {
        System.out.println(id + " out of " + getCount() + " blocks");
    }
    public static int getCount() {
        return count;
    }
}
public class BlockTest3 {
    public static void main(String[] args) {
        Block3 b1 = new Block3();
        Block3 b2 = new Block3();
        Block3 b3 = new Block3();
        System.out.println(Block3.getCount());
        b2.printRanking();
    }
```

Class Methods

- *static* 메소드도 특정 인스턴스에 묶여 있지 않고 클래스에 달려 있다.
- 특정 인스턴스에 묶여 있지 않기에 인스턴스 변수에 접근하는 것은 성립하지 않는다.

- 인스턴스 메소드는 인스턴스 변수와 인스턴스 메소드에 직접적으로 접근할 수 있다.
- 인스턴스 메소드는 클래스 변수와 클래스 메소드에 직접적으로 접근할 수 있다.
- 클래스 메소드는 클래스 변수와 클래스 메소드에 직접적으로 접근할 수 있다.
- 클래스 메소드는 인스턴스 변수와 인스턴스 메소드에 직접적으로 접근할 수 **없다**.

Class Methods

```
class Mathematics {
    static int abs(int a) {
        if (a > 0) {
            return a;
        }
        else {
            return -a;
        }
    }
}

public class MathematicsTest {
    public static void main(String[] args) {
        System.out.println(Mathematics.abs(-10));
    }
}
```

- 절대값을 계산하는 메소드는 특정 인스턴스에 묶이는 것이 자연스럽지 않다.
- *static* 메소드를 만들기 전에 특정 인스턴스가 없어도 해당 메소드를 부르는 게 적합한지 질문해보자.

Word of Caution I

```
public class BigIntStatic {
    public static int[] add(String left, String right) {
        // magic happens
        return new int[10];
    }

    public static void main(String[] args) {
        // somehow the input is represented as a string
        String leftStr = "123456789";
        String rightStr = "987654321";
        char op = '+';

        int[] result = add(leftStr, rightStr);
    }
}
```

- 자주 하는 실수 – 논리적 단위로 코드를 나누며 메소드를 추출하지만 *main* 메소드에서 바로 호출하기 위해 *static* 메소드를 다량 만드는 경우.
- OOP는 객체가 서로 메시지를 주고 받는 방식으로 프로그램을 구성해서 복잡도 관리를 꾀한다.
- 클래스 메소드로 모든 것을 만들지 마라.

Word of Caution II

```
class Mathematics {
    public int abs(int a) {
        if (a > 0) {
            return a;
        }
        else {
            return -a;
        }
    }
}
public class InstanceHell {
    public static void main(String[] args) {
        int input = -1234;
        Mathematics math = new Mathematics();
        System.out.println(math.abs(input));
    }
}
```

- 자주 하는 실수 – 클래스 메소드를 피하기 위해 객체를 만들고 인스턴스 메소드만 사용하는 경우.
- 클래스 메소드를 피한다고 모든 것을 인스턴스 메소드로 만들지 마라.

Numbers

```
public class Box {
    public static void main(String[] args) {
        int a = new Integer(10);
        Integer b = 30;
    }
}
```

- 숫자를 다룰 때 기본형에 대응되는 클래스가 있다.
  - *int*와 *Integer*
  - *float*과 *Float*
  - ...
  - `http://docs.oracle.com/javase/tutorial/java/data/numbers.html`
- 이런 클래스는 기본형을 감싸서 (wrap) 객체로 다룰 수 있게 만들어주는 역할을 한다.
- 편의를 위해 컴파일러가 감싸고 푸는 것을 자동으로 해준다. 이를 autoboxing, unboxing이라 부른다.

Wrapper Objects

```java
import java.util.ArrayList;

public class Conversion {
    public static void main(String[] args) {
        String a = "123124";
        int b = Integer.parseInt(a);
        System.out.println(b);
        System.out.println(Integer.MAX_VALUE);
        ArrayList<Integer> l = new ArrayList<>();
    }
}
```

- Wrapper object를 굳이 사용하는 이유
  - 객체를 기대하는 곳에 사용하기 위해 (메소드 호출 인자, 제네릭 타입 인자)
  - 객체가 제공하는 클래스 변수나 클래스 메소드를 사용하기 위해
- `http://docs.oracle.com/javase/tutorial/java/data/numberclasses.html`

String

```
public class StringTest {
    public static void main(String[] args) {
        String a = "Knowledge and productivity are";
        String b = " like compound interest";
        int c = 10;
        System.out.println(a.length());
        System.out.println(a.charAt(0));
        System.out.println(a + b);
        System.out.println(c);
        System.out.println(c + " times");
    }
}
```

- 문자열도 객체로 다양한 메소드를 지원한다.
- 정수형을 바로 *println* 하면 사실 내부적으로 *Integer.toString* 이 호출된다.
- 마찬가지로 *10 + "times"*는 10이 문자열로 변환된 뒤 문자열 간의 '+' 연산이 실행된다.
- `http://docs.oracle.com/javase/tutorial/java/data/strings.html`

String and StringBuilder

- String은 불변형(immutable)이다.
  - 객체의 데이터에 변환이 일어나면 새 객체가 생성된다.
  - *concat()*, *replace()*, . . .
- Mutable한 String의 짝으로 StringBuilder가 있다.
- StringBuilder는 코드가 매우 간결해지거나 성능이 필요할 때만 사용한다.
- String 간의 '+' 연산은 사실 StringBuilder로 변환된 뒤 *append*로 추가된다.
- `http://docs.oracle.com/javase/tutorial/java/data/buffers.html`

Reverse

```
public class Reverse {
    static String stringReverse(String str) {
        int len = str.length();
        char[] tempCharArray = new char[len];
        char[] charArray = new char[len];
        for (int i = 0; i < len; ++i) {
            tempCharArray[i] = str.charAt(i);
        }
        for (int i = 0; i < len; ++i) {
            charArray[i] = tempCharArray[len - 1 - i];
        }
        String reversed = new String(charArray);
        return reversed;
    }
    static String stringBuilderReverse(String str) {
        StringBuilder sb = new StringBuilder(str);
        sb.reverse();
        return sb.toString();
    }
    public static void main(String[] args) {
        String str = "Laziness";
        System.out.println(stringReverse(str));
        System.out.println(stringBuilderReverse(str));
    }
}
```

String vs StringBuilder

```
public class StringConcatComparison {
    private static String s = "1234567890";
    private static int loopCnt = 100000;
    private static String stringBuilderConcat() {
        StringBuilder sb = new StringBuilder();;
        for (int i = 0; i < loopCnt; ++i) sb.append(s);
        return sb.toString();
    }
    private static String stringConcat() {
        String c = "";
        for (int i = 0; i < loopCnt; ++i) c += s;
        return c;
    }

    public static void main(String[] args) {
        System.out.println(stringBuilderConcat().length());
        System.out.println(stringConcat().length()); // slow
    }
}
```

Regex

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;

public class RegexTest {
    public static void main(String[] args) {
        String patternStr = "(?<sign>[[+][-]]?)(?<num>[0-9]+)";
        String str = "12312-321";

        Pattern pattern = Pattern.compile(patternStr);
        Matcher matcher = pattern.matcher(str);

        while (matcher.find()) {
            System.out.println(matcher.group() + " beginning at "
                    + matcher.start() + " and ending at " + matcher.end());
            System.out.println(matcher.group("sign"));
            System.out.println(matcher.group("num"));
        }
    }
}
```

- http://docs.oracle.com/javase/tutorial/essential/regex/index.html

Advice

- 코드로 의도를 분명히 표현하라.
- 가능하면 일련의 데이터 조작이 아닌 객체 간의 메시지 주고 받기로 코드를 작성하라.
- 각각의 객체는 마치 기본형을 사용하는 것과 비슷한 느낌으로 사용할 수 있도록 한다.
- 객체가 제공하는 연산은 상식적으로 동작해야 한다.
- 두 객체가 논리적으로 같음을 표현하기 위해 *equals* 메소드를 사용한다.
- *equals* 메소드와 함께 *hashCode* 메소드를 정의해줘야 한다.

Advice

- 인스턴스 변수는 개별 객체의 상태를 표현한다.
- 인스턴스 메소드는 객체의 내부적인 데이터에 접근해야 하면 사용한다.
- 특정 인스턴스에 묶이지 않은 변수를 만들기 위해 클래스 변수를 사용한다.
- 클래스 변수는 모든 객체가 공유한다.
- 클래스 메소드는 특별히 객체가 없는 상황에서 동작하는 행동을 만들기 위해 사용한다.