A Tour of Java

Sungjoo Ha

March 4th, 2016

Introduction

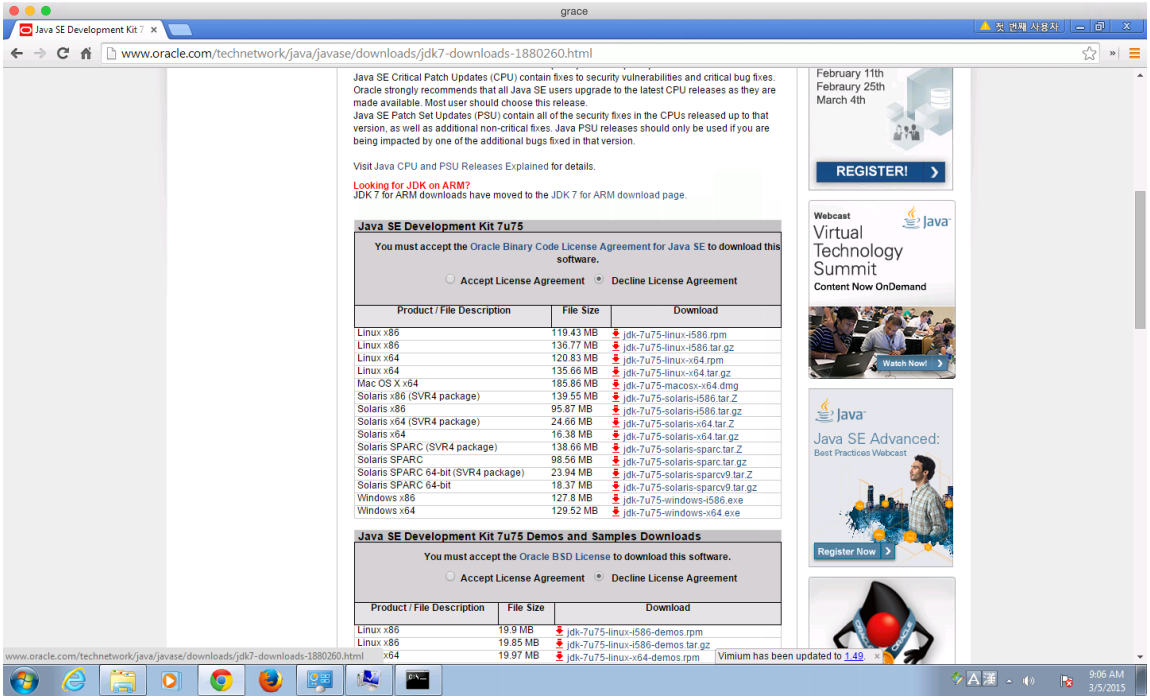
- 앞으로의 계획 (바뀔 수 있음)
  - 환경 구축 및 자바의 기초 소개
  - 사용자 정의 타입 및 모듈화, 프로그래밍 일반론, 디버깅, 등
  - 클래스
  - 인터페이스
  - 제네릭
  - 과제2 안내
- 수업 후 실습실에서 실습 예정

First Principle

- 문제가 생기면 침착하게 (영어로) 구글에서 찾아본다.
  - 오류 메시지를 입력한다.
  - 증상을 설명한다.
- 대체로 stackoverflow 등의 사이트에 답변이 있다.

Java

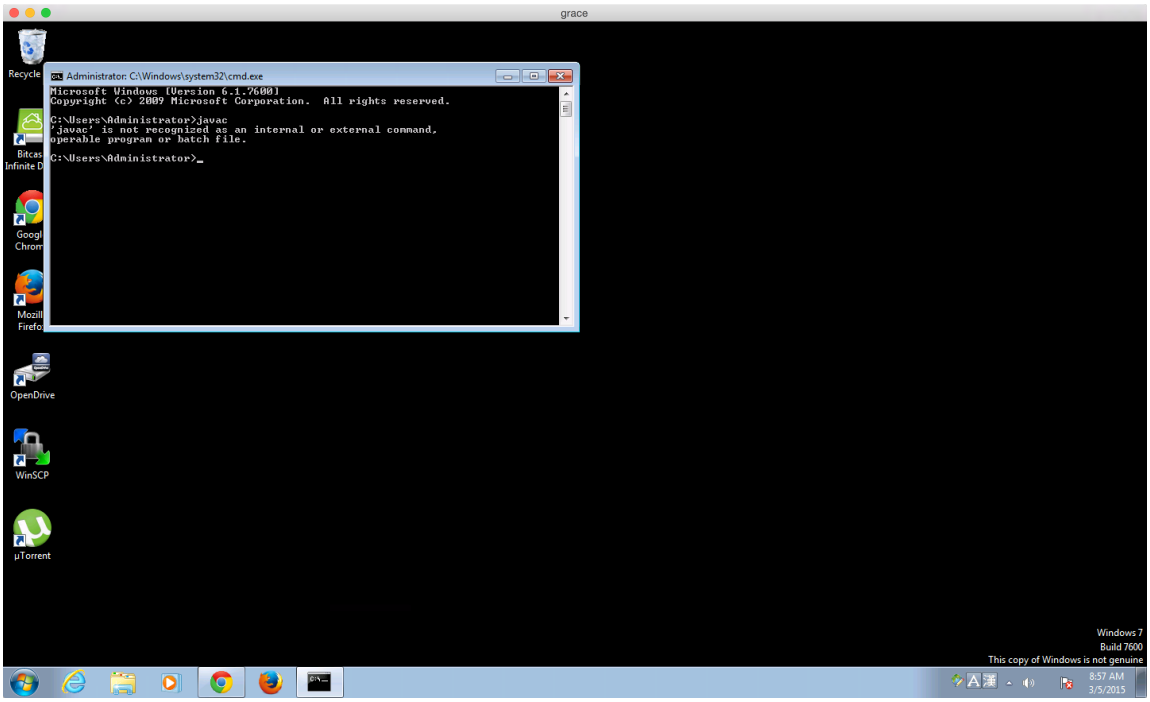
- 최신 버전은 8이지만 채점 서버가 7u75로 되어 있다.
- 자바7과 자바8은 차이가 있으니 염두에 두기 바란다.
- 소프트웨어 실습실에는 자바8이 설치되어 있다.
- `http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`
- JRE가 아니라 JDK를 받아야 한다.

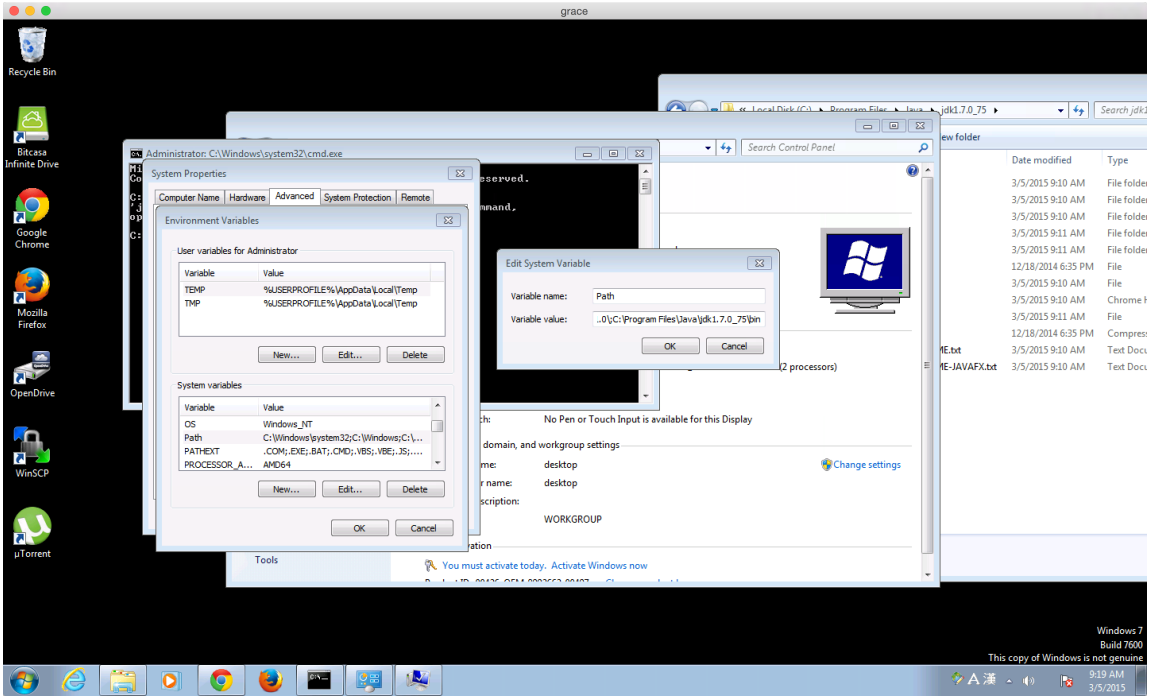


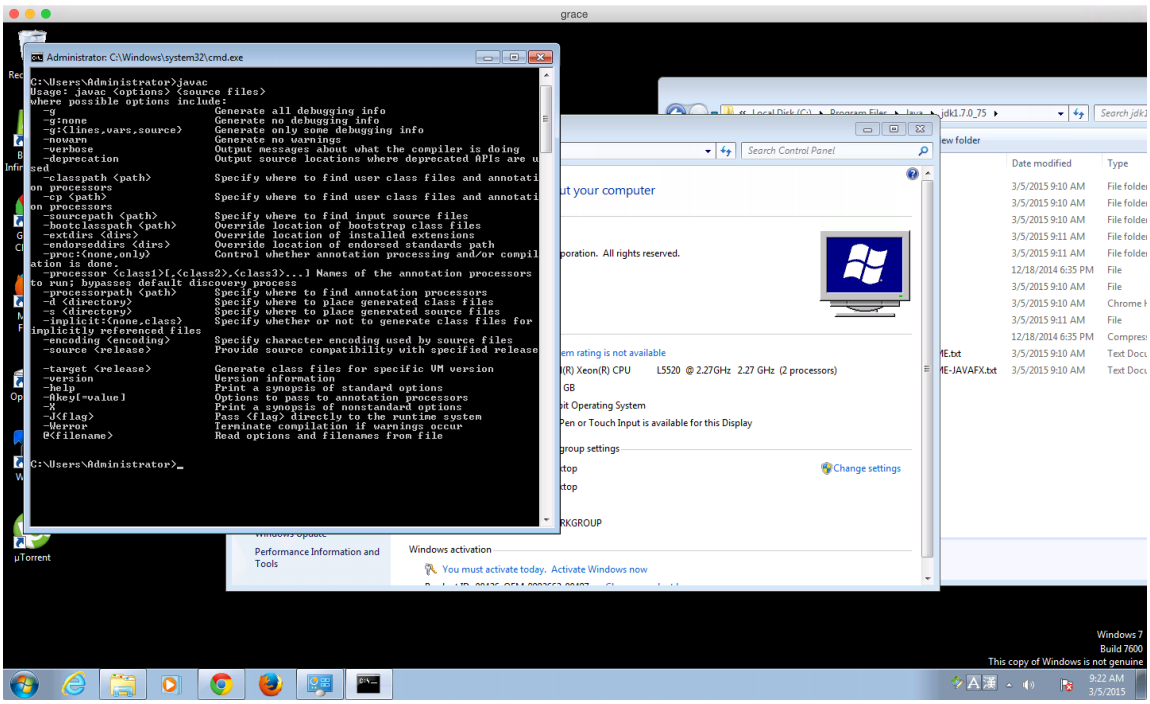
윈도우는 [내 컴퓨터 - 속성]에서 x86인지 x64인지 확인 가능하다.

Path

- 자바 실행을 위해 PATH 설정을 해줘야 한다.
- 제어판 - 시스템 - 고급 설정 - 환경 변수







Eclipse

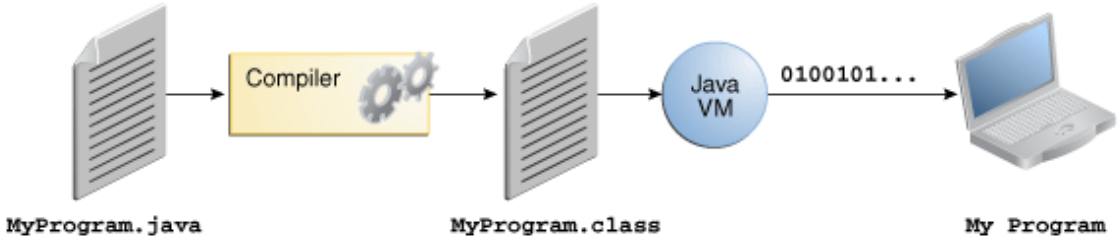
- 통합 개발 환경 (IDE)
- `https://eclipse.org/`

Online IDE

- 이번 학기에는 온라인 IDE를 사용할 예정이며 자세한 내용은 추후 공지될 것이다.

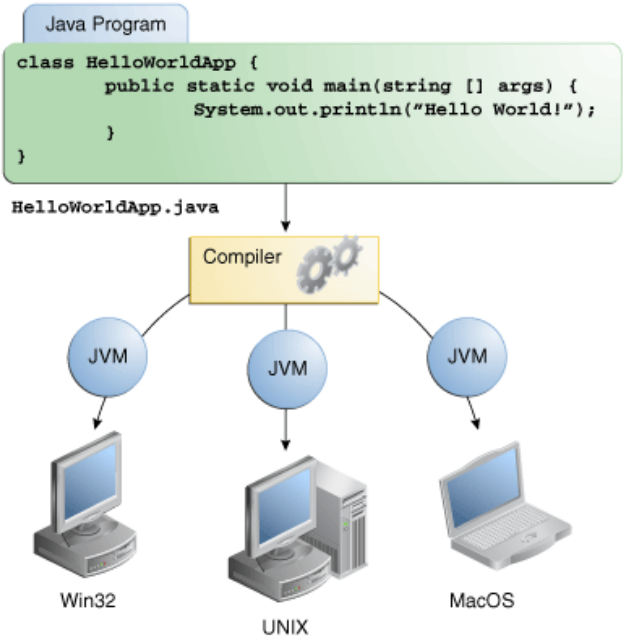
JVM

- 자바는 컴파일 되어야 하는 언어이다.
- 프로그램이 수행되기 위해 소스 텍스트가 컴파일러를 거쳐서 바이트코드가 생성된다.
- 이 바이트코드는 JVM이 이해할 수 있는 언어로 되어 있고, JVM이 이를 실행한다.



JVM

- JVM만 설치 되어 있으면 .class 파일을 어떤 환경에서나 실행할 수 있다.



Java Platform

- 플랫폼은 프로그램이 수행되는 환경이다.
- 자바 플랫폼은 두 개로 나뉜다. JVM과 Java API이다.
  - JVM은 이미 소개하였다.
  - API는 이미 만들어진 프로그램 집합이다.
  - API를 내가 만든 프로그램에서 사용할 수 있다.

Hello, World!

```java
public class HelloWorld {
    // 최소한의 자바 프로그램
    public static void main(String[] args) {}
}
```

- *main* 메소드를 정의하고 있다. *args* 라는 커맨드라인 인자를 받는다.
- 메소드는 다른 언어의 함수와 비슷한 위상이라고 생각해도 좋다.
- *public*, *static*, *class*는 무시한다.
- 클래스 이름 *HelloWorld* 는 파일 이름 *HelloWorld.java*와 같아야 한다.

Hello, World!

```java
public class HelloWorld {
    // 최소한의 자바 프로그램
    public static void main(String[] args) {}
}
```

- 중괄호 {, }는 그룹을 나누는 것을 의미한다. 클래스와 메소드의 시작과 끝을 표현한다.
- "//" 는 주석을 의미한다. 시작부터 줄 마지막까지 적용된다. 주석은 사람이 읽기 위해 존재한다.
- 모든 자바 프로그램은 *main* 메소드를 딱 하나 갖고 있어야 한다. 여기부터 프로그램이 시작한다.
- *void*는 *main* 프로그램의 리턴 타입을 알려주는데, 아무것도 리턴하지 않음을 뜻한다.

Hello, World!

```java
import java.lang.System;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.print("Hello, World!\n");
    }
}
```

- 보통 프로그램은 결과물을 내놓게 마련이다.
- 위의 프로그램은 'Hello, World!'를 출력하는 프로그램이다.

Hello, World!

```java
import java.lang.System;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.print("Hello, World!\n");
    }
}
```

- *import java.lang.System;*은 컴파일러에게 *java.lang.System*의 내용을 포함하라고 지시한다.
- 여기에서는 화면에 출력하기 위한 standard output stream *out*을 활용하기 위해 포함되었다.
- 하지만 기본적으로 *java.lang*의 모든 패키지가 *import*(추가)된다. 그러니 위의 *import*는 필요 없다.

Hello, World!

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.print("Hello, World!\n");
    }
}
```

- 위의 프로그램은 *out*의 *print* 메소드를 호출한다.
- 이 때 인자는 *"Hello, World!\n"* 이다.
- 따옴표로 묶인 문자들의 나열을 문자열 (string) 이라 부른다.
- 역슬래시 \ 와 하나의 문자는 특수 문자이다. *n*은 newline 문자이며 그래서 *Hello, World!*를 출력한 뒤 다음 줄로 넘어가게 된다.

Hello, World!

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

- 새 줄로 넘어가는 기능은 무척 많이 사용하므로 따로 메소드가 있다.

Method Call

```java
public class Square {
    static double square(double x) {
        return x*x;
    }
    static void print_square(double x) {
        System.out.println("the square of " + x
                + " is " + square(x));
    }
    public static void main(String[] args) {
        print_square(1.234);
    }
}
```

- 사실상 실행되는 모든 코드는 어딘가의 메소드로 존재하며, *main* 메소드에서 직간접적으로 호출되게 된다.
- *square* 메소드의 앞 쪽 *double*은 리턴 타입이 *double*형임을 알려준다.

Method

- 자바에서 무언가 하는 주된 방법은 클래스에 속한 메소드를 호출하는 것이다.
- 메소드를 정의해서 하고 싶은 일을 설명하게 된다.
- 메소드의 정의는 메소드에 이름을 달아주고, 호출 시 필요한 인자의 개수 및 타입을 알려주며 리턴값의 타입도 알려준다.

Method

- 우리는 대체로 코드가 이해하기 쉽길 원한다. 그래야 유지보수가 쉽다.
  - 이를 위한 첫 번째 단계는 각종 계산을 이해 가능한 수준의 작은 조각으로 자르는 것이다.
  - 이런 작은 조각은 메소드로 (및 클래스) 표현된다.
- "버그"는 프로그램의 복잡도와 길이에 비례하게 된다.
  - 프로그램을 작은 조각으로 나누면 두 문제 다 피할 수 있다.
  - 특정 코드를 메소드로 만들면 해당 코드가 어떤 일을 하는지 이름 붙이게 된다. 이해에 도움이 된다.
  - 같은 코드를 여러 번 작성하지 않아도 되게 한다.

Overloading

```java
public class Print {
    static void print(double num) {
        System.out.println("double");
        System.out.println(num);
    }
    static void print(String str) {
        System.out.println("string");
        System.out.println(str);
    }
    public static void main(String[] args) {
        print(3.23);
        print("한글");
    }
}
```

- 두 메소드가 같은 이름을 갖지만 다른 종류의 인자를 가지면 컴파일러가 적당한 메소드를 선택해서 호출해준다.

Overloading

```
public class Print2 {
    static void print(int num1, double num2) {
        System.out.println(num1);
        System.out.println(num2);
    }
    static void print(double num1, int num2) {
        System.out.println(num1);
        System.out.println(num2);
    }
    public static void main(String[] args) {
        //print(0, 0); // compile error
    }
}
```

- 만약 두 메소드가 동등하게 적합하다면 컴파일 오류가 난다.

Overloading

- 이렇게 같은 메소드 이름을 사용하는 기법을 오버로딩(overloading)이라고 한다. 이후에 다시 다루게 될 것이다.
- 같은 이름을 갖는 메소드는 같은 "의미"를 구현하여야 한다.
- 가령 *print* 함수라면 "출력" 하는 의미를 갖도록 구현하자.

Type

```
int cm;
```

- 모든 종류의 이름과 표현식(expression)은 타입이 달려 있다.
- 타입에 의해 우리가 할 수 있는 행동이 결정된다.
- *cm*은 *int* 타입을 가짐을 뜻한다. *cm*은 정수형(int) 변수이다.
- 선언(declaration)은 프로그램서 사용할 이름을 선언하는 행위이다.
  - 이는 이름의 타입을 지정하게 된다.
  - 타입은 가능한 값의 집합과 연산의 집합을 정의한다.
  - 객체(object)는 어떤 타입의 값을 쥐고 있는 메모리이다.
  - 값(value)은 타입에 따라 해석되는 비트이다.
  - 변수(variable)는 이름이 달린 객체이다.

Primitive Data Type

```
public class Types {
    public static void main(String[] args) {
        boolean B = true;
        char 문자 = '가';
        int answer = 42;
        double pi = 3.141592;
    }
}
```

- 자바는 다양한 기본 데이터 타입(primitive data type)을 제공한다.
- 가령 *char* 변수는 문자 하나를 저장하는 변수가 된다.
- 자세한 내용은 `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html` 을 참고한다.
- 변수의 초기화(initialization)는 등호(=)를 사용한다.
- 변수는 선언과 동시에 초기화하기를 강력히 권고한다.

Arithmetic Operator

```
public class Arithmetic {
    public static void main(String[] args) {
        int x = 10;
        int y = -3;

        System.out.println(x + y);
        System.out.println(+y);
        System.out.println(x - y);
        System.out.println(-x);
        System.out.println(x * y);
        System.out.println(x / y);
        System.out.println(x % y);
    }
}
```

- 산술 연산자를 적합한 타입의 조합과 함께 사용할 수 있다.

Comparison Operator

```
public class Comparison {
    public static void main(String[] args) {
        int x = 10;
        int y = -3;

        System.out.println(x == y);
        System.out.println(x != y);
        System.out.println(x < y);
        System.out.println(x > x);
        System.out.println(x <= y);
        System.out.println(x >= y);
    }
}
```

- 비교 연산자도 사용할 수 있다.

Bitwise Logical Operator

```
public class Logical {
    public static void main(String[] args) {
        int x = 10;
        int y = -3;

        System.out.println(x & y);
        System.out.println(x | y);
        System.out.println(x ^ y);
        System.out.println(~x);
    }
}
```

- 비트 단위 논리 연산자(bitwise logical operator)도 있다.
- 각 인자의 비트 단위로 연산을 적용한다.

Logical Operator

```
public class Logical {
    public static void main(String[] args) {
        boolean a = true;
        boolean b = false;

        System.out.println(a && b);
        System.out.println(a || b);
    }
}
```

- 논리 연산은 참(*true*) 혹은 거짓(*false*)를 리턴한다.

Assignments

```
public class Arithmetic2 {
    public static void main(String[] args) {
        int x = 10;
        int y = -3;
        x += y;
        ++x;
        x-=y;
        --x;
        x*=y;
        x/=y;
        x%=y;
        System.out.println(x);
        System.out.println(y);
    }
}
```

- 기본적인 산술 연산 및 논리 연산 외에 변수를 변경하는 연산자도 제공한다.

Local Scope

```
public class Scope {
    public static void main(String[] args) {
        int x = 10;
        System.out.println(x);
        {
            System.out.println(x);
            int a = 10;
            System.out.println(a);
        }
        //System.out.println(a); // compile error
    }
}
```

- 메소드 내부에서 선언된 변수는 선언 시점부터 블럭의 끝까지를 스코프로 가진다.
- 블럭은 중괄호로 표현된다.
- 함수 인자는 로컬 스코프로 판정된다.

Class Scope

```
public class Scope2 {
    String name;

    public static void main(String[] args) {
        Scope2 s = new Scope2();
        s.name = "이름";

        System.out.println(s.name);
    }
}
```

- 클래스 내부에서 선언되면 클래스 스코프를 갖는다.
- 객체(object)는 생성(construct)되어야 사용할 수 있다.
- 이렇게 생성된 객체는 스코프가 끝나면 파괴된다.
- 맴버는 객체가 파괴될 때 함께 파괴된다.
- 나중에 클래스를 다룰 때 다시 논하기로 한다.

Constants

```
public class Final {
    static int sum(final int a, final int b) {
        // a = 3; // compile error
        return a + b;
    }

    public static void main(String[] args) {
        final int a = 10;
```

```
        //a = 2; // compile error

        int b = sum(3, 4);
        System.out.println(b);
    }
}
```

- 불변성을 표현하기 위해 *final* 키워드를 사용한다.
- 대략 "이 값을 변화하지 않겠음" 정도의 의미로 사용된다. .

Array

```
public class Array {
    public static void main(String[] args) {
        char [] v = new char[5];
        v[0] = '한';
        v[1] = '글';
        v[2] = '문';
        v[3] = '자';
        v[4] = '열';
        char [] v2 = {'a', 'b', 'c'};
    }
}
```

- *[]* 는 배열을 의미한다.
- *char [] v* 는 문자형 배열 타입을 갖는 변수 *v*를 의미하게 된다.
- 배열의 인덱스는 0부터 시작한다.
- 초기화는 *new* 키워드를 사용하거나 암묵적으로 초기화할 수 있다.

Array

```
public class Array2 {
    public static void main(String[] args) {
        int[] v1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int[] v2 = new int[10];
        for (int i=0; i < 10; ++i) {
            v2[i] = v1[i];
        }
        System.out.println(v2[3]);
    }
}
```

- *for*는 루프문을 의미한다.

Array

```
public class Array3 {
    public static void main(String[] args) {
        int[] v1 = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        for (int i: v1) {
            System.out.println(i);
        }
    }
}
```

- *for each*문도 있다.

Pass By Value

```
public class Pass {
    static void swap(int a, int b) {
        // does not work as intended
        int temp = a;
        a = b;
        b = temp;
    }
    public static void main(String[] args) {
        int x = 1;
        int y = 2;
        swap(x, y);
        System.out.println(x);
        System.out.println(y);
    }
}
```

Pass By Value

기본형은 값의 복사가 일어난다.

- *swap(x, y)*는 x와 y의 값을 각각 a와 b에 복사한다.
- 그러므로 a와 b가 변해도 x와 y는 변하지 않는다.

Pass By Value

```
public class Ref {
    static void setZero(int[] v) {
        v[0] = 10;
        return;
    }
    public static void main(String[] args) {
        int[] v1 = {4, 3, 2, 1, 0};
        for (int i: v1) {
            System.out.println(i);
        }
        setZero(v1);
        for (int i: v1) {
            System.out.println(i);
        }
    }
}
```

Pass By Value

기본형이 아니면 이름표가 걸린다.

- 기본형이 아니라면 변수명은 이름표처럼 동작한다.
- *v1*은 *[4, 3, 2, 1, 0]*의 데이터가 저장된 메모리 공간에 이름표로 걸린다.
- *setZero* 메소드 호출의 인자로 *v1*을 넘기면 이 이름표가 걸린 메모리 공간에 v 이름표도 걸린다.
- *v[0] = 10* 은 해당 메모리 공간을 수정하게 된다.
- 그러므로 값이 변경된다.

Pass By Value

```
public class Ref {
    static void newArray(int[] v) {
        v = new int[5];
        v[0] = 3;
        for (int i: v) {
            System.out.println(i);
        }
    }
    public static void main(String[] args) {
        int[] v1 = {4, 3, 2, 1, 0};
        newArray(v1);
        for (int i: v1) {
            System.out.println(i);
        }
    }
}
```

Pass By Value

기본형이 아니면 이름표가 걸린다.

- *newArray* 메소드는 v 이름표가 *v1* 이름표가 걸려 있던 메모리 공간에 같이 걸린다.
- 다시 v는 *new int[5]*로 생성된 메모리 공간에 걸린다.
- 새로운 공간의 0번과 1번 위치의 값을 3으로 고쳐 쓴다.
- *v1*은 여전히 *[4, 3, 2, 1, 0]*을 가리키므로 변화는 없다.

Conditional
IF

```java
import java.util.Scanner;
public class Input {
    static boolean accept() {
        System.out.println("진행할까요? (y/n)");
        Scanner reader = new Scanner(System.in);
        String answer  = reader.nextLine();
        if (answer.charAt(0) == 'y' || answer.charAt(0) == 'Y') {
            return true;
        }
        else {
            return false;
        }
    }
    public static void main(String[] args) {
        boolean answer = accept();
        System.out.println(answer);
    }
}
```

Conditional
SWTICH

```java
import java.util.Scanner;
public class Input2 {
    static boolean accept() {
        System.out.println("진행할까요? (y/n)");
        Scanner reader = new Scanner(System.in);
        String answer  = reader.nextLine();
        switch(answer.charAt(0)) {
            case 'y':
            case 'Y':
                return true;
            case 'n':
            case 'N':
                return false;
            default:
                System.out.println("아닌걸로 간주할게요.");
                return false;
        }
    }
    public static void main(String[] args) {
        boolean answer = accept();
        System.out.println(answer);
    }
}
```

Loop and Conditional

```java
import java.util.Scanner;
public class Action {
    public static void main(String[] args) {
        int x = 0;
        int y = 0;
        Scanner reader = new Scanner(System.in);
        while (true) {
            String action = reader.nextLine();
            for (char ch: action.toCharArray()) {
                switch (ch) {
                    case 'u': // up
                    case 'n': // north
                        ++y;
                        break;
                    case 'r': // right
                    case 'e': // east
                        ++x;
                        break;
                    // something more
                    default:
                        System.out.println("그대로 멈춰라!");
                        break;
                }
            }
        }
    }
}
```

Advice

- 모르는 것이 있어도 너무 걱정하지 않아도 좋다. 나중에 다시 다루게 된다.
- 좋은 프로그램을 작성하기 위해 언어의 모든 측면을 알아야 하는 것은 아니다.
- 언어의 세세한 특징이 아니라 프로그래밍 테크닉에 집중하라.
- 언어의 자세한 면은 자바 공식 문서를 참고한다. `http://docs.oracle.com/javase/tutorial/index.html`
- 의미 있는 단위의 계산을 하나의 메소드로 묶고 알맞은 메소드 이름을 정해준다.
- 하나의 메소드는 하나의 논리적인 계산을 수행해야 한다.
- 메소드의 길이는 짧게 유지한다.

Advice

- 오버로딩은 개념적으로 같은 일을 다른 타입에 대해 수행하는 메소드에 사용한다.
- 상수에는 이름을 붙여서 사용한다.
- 자주 사용하는 지역 이름은 짧게, 가끔 사용하는 전역 이름은 길게 짓는다.
- 비슷한 이름은 피한다.
- 초기화하지 않은 선언은 피한다.
- 하나의 스코프는 가능하면 작게 유지한다.
- 코드에서 명확한 내용을 주석으로 남기지 않는다.
- 주석에는 의도를 표현한다.
- 인덴트 등을 일관성 있게 유지한다.
- 너무 복잡한 표현식(expression)은 피한다.

A Tour of Java II

Sungjoo Ha

March 11th, 2016

Review

- First principle – 문제가 생기면 침착하게 영어로 구글에서 찾아본다.
- 타입은 가능한 값의 집합과 연산의 집합을 정의한다.
- 기본형이 아니라면 이름표가 메모리에 달라 붙는다.

User-Defined Types

```
int a = 10;
a++;
//int x = "정수형";
//a.charAt(0);
String str = "문자열이다.";
str.charAt(0);
//String newStr = 10;
//str++;
```

- 타입은 가능한 값의 집합과 연산의 집합을 정의한다.

Abstraction

- 자바는 추상화 방법을 제공한다.
- 기본형 및 각종 연산자와 함께 추상화 방법을 활용하여 고차원적인 기능을 만들 수 있다.
- 자바의 추상화 방법은 프로그래머가 스스로 타입을 고안하고 구현하는 방법을 주로 삼는다.
  - 각 타입은 적절한 표현 및 연산을 갖추도록 한다.
  - 각 타입은 쉽고 우아하게 사용할 수 있어야 한다.
- 이렇게 사용자가 정의할 수 있는 타입을 클래스라 부른다.

Class

```
class Vector {
    int sz; // number of elements
    double[] elem; // array of elements
}
```

- 새 타입을 만드는 것은 많은 경우 타입이 갖춰야 하는 데이터를 구성하는 것으로 시작된다.
- *class* 키워드를 사용해서 *Vector*라는 사용자 정의 타입을 만들었다.
- *Vector*는 *int*와 *double []*로 구성된다.

Class Initialization

```
Vector v = new Vector();
```

- *Vector* 타입의 변수는 다음과 같이 만들 수 있다.
- 선언과 함께 초기화를 권장한다.
- *Vector* 타입의 변수 v를 위한 메모리 공간을 할당하게 된다.
- 하지만 현재 구조로는 큰 쓸모가 없다.
  - v의 구성 요소인 *elem*이 메모리 공간조차 할당되지 않았다.
  - 쓸모 있기 위해서는 메모리 공간이 할당되고, 데이터가 차있어야 한다.

Class Member Initialization

```
static void vector_init(Vector v, int s) {
    v.elem = new double[s];
    v.sz = s;
}
```

- v의 *elem*은 *double[]* 타입이므로 이에 해당하는 메모리 공간이 *new double[s]*에 의해 할당된다.
- 마찬가지로 v의 *sz*는 *int* 타입으로 s의 값이 복사된다.
- 메소드 인자로 *Vector* v를 받을 때 v가 인자로 넘어온 *Vector*가 위치한 메모리 공간에 이름표로 붙는다.
  - 그러므로 *vector_init()*에 의해 v의 내부를 수정할 수 있다.
- *new* 연산자는 heap이라 불리는 공간에 메모리를 할당한다.
  - 자바는 가비지 컬레터를 제공하므로 따로 메모리 공간을 회수하지 않아도 된다.
  - 더 이상 사용되지 않는 메모리 공간은 자동으로 회수된다.

User-Defined Type Usage

```
static double read_and_sum(int s) {
    Vector v = new Vector();
    vector_init(v, s);
    Scanner reader = new Scanner(System.in);
    for (int i = 0; i != s; ++i) {
        v.elem[i] = reader.nextDouble();
    }
    double sum = 0.0;
    for (int i = 0; i != s; ++i) {
        sum += v.elem[i];
    }
    return sum;
}
```

- *Vector*의 사용예이다.
  - 자바 API에 *Vector* 클래스가 이미 있다. 위의 예시는 말 그대로 예시일 뿐이다.
  - C++의 vector와 동등한 위치의 자료구조가 필요하다면 *ArrayList*를 사용해라. *Vector*는 더 이상 사용하지 않는다고 봐도 좋다.
- 클래스의 멤버에 접근할 때에는 .을 사용한다.

Class as User-Defined Type

- 데이터만 따로 분리하는 것이 유용하지만 데이터와 이에 적용할 수 있는 연산이 긴밀하게 붙어 있는 때가 자주 있다.
- 사용자 정의 타입이 정말 "타입"으로 동작하기 위해서는 연산이 함께 붙어 있어야 한다.
  - 보통 데이터 자체는 직접적인 접근이 불가능하게 만들고 연산자를 활용해서 이를 변환하거나 살펴볼 수 있게 한다.
  - 이를 통해 데이터의 정합성, 사용의 용이함, 추후 개선 가능성 등을 보장하게 된다.
    - 이를 위해 외부에서 타입과 상호작용하는 방식의 약속(interface)과 내부적인 구현(implementation)을 구분해서 이해할 필요가 있다.
- 이를 위해 사용되는 메커니즘을 클래스라 부른다.

Class as an Interface

```
class Vector {
    public Vector(int s) {
        elem = new double[s];
        sz = s;
    }
    public double get(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
}
```

- 클래스는 맴버를 가지고, 이는 데이터나 메소드가 될 수 있다.
- 클래스가 노출하는 인터페이스는 *public* 맴버로 결정된다.
- 노출된 인터페이스를 통해서만 *private* 맴버에 접근할 수 있다.

Class

```
static double read_and_sum(int s) {
    Vector v = new Vector(s);

    Scanner reader = new Scanner(System.in);

    for (int i = 0; i != v.size(); ++i) {
        v.set(i, reader.nextDouble());
    }

    double sum = 0.0;

    for (int i = 0; i != v.size(); ++i) {
        sum += v.get(i);
    }

    return sum;
}
```

- *read_and_sum()* 메소드는 위와 같이 변한다.
- *Vector*의 데이터(*elem*과 *sz*)는 *public* 맴버로 제공되는 인터페이스를 통해서만 접근 가능하다.
  - *Vector()*, *get()*, *set()*, *size()*

Class

- 클래스와 이름이 같은 "메소드"는 생성자(constructor)라 부른다.
  - *Vector* 클래스의 생성자가 *vector_init()* 메소드를 대신하게 된다.
  - 생성자는 객체를 생성하는 순간 호출되는 것이 보장된다.
- *Vector(int)*는 *Vector* 타입의 객체가 어떻게 생성되는지 정의한다.
  - *int*는 객체의 생성을 위해 정수형 타입의 값이 하나 주어져야 함을 뜻한다.
  - 이 경우에는 정수형 변수로 총 원소의 개수가 결정된다.
- 클래스 내부의 데이터에 접근하기 위해 *get()* 및 *set()* 메소드를 정의했다.
  - 이를 getter/setter 라 부르며 데이터에 직접적으로 접근하는 대신 이렇게 정의된 인터페이스를 통해 간접적으로 데이터에 접근할 수 있다.

Enumeration

```
enum Color {
    red, blue, green
}
```

- 클래스와 함께 사용자 정의 타입을 구성할 수 있는 방법으로 *enum*이 있다.
- *enum*은 값의 나열을 나타내는 타입이다.

Enumeration

```
enum Color {
    red, blue, green
}
enum TrafficLight {
    green, yellow, red
}
public class Enumeration {
    public static void main(String[] args) {
        Color col = Color.red;
        Color col2 = Color.red;
        if (col == col2) {
            System.out.println("Same color.");
        }
        TrafficLight light = TrafficLight.red;
        // Color y = TrafficLight.red; // compile error
        //int i = Color.red; // compile error
    }
}
```

- *red* 및 *green*은 각각 해당하는 *enum* 클래스의 스코프에 존재하므로 같은 이름을 사용할 수 있다.
- 각 *enum* 클래스는 다른 타입이다. 같은 이름을 써도 서로 다른 *enum*이면 비교나 대입 등의 연산이 불가능하다.

Enumeration

```
enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7);
    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double getMass() { return mass; }
```

```java
    private double getRadius() { return radius; }
    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Enumeration

```java
public class Enumeration2 {
    public static void main(String[] args) {
        double earthWeight = 50.0;
        double mass = earthWeight / Planet.EARTH.surfaceGravity();

        for (Planet p: Planet.values()) {
            System.out.printf("Weight on %s is %f%n",
                              p, p.surfaceWeight(mass));
        }
    }
}
```

- *enum* 클래스는 각종 맴버 변수나 메소드를 가질 수 있다.
- *printf()* 메소드는 포매팅 된 스트링을 출력하기 위한 메소드이다.
  - `http://docs.oracle.com/javase/tutorial/java/data/numberformat.html`
- *this*는 메소드 인자가 아닌 클래스 맴버를 가리키기 위해 사용한다.

Error Handling

- 오류 처리는 방대한 주제이다.
- 자바의 기본적인 오류 처리 도구는 타입 시스템 자체이다.
  - 타입은 가능한 값의 집합과 연산의 집합을 정의한다.
  - 기본 타입만 활용해서 모든 프로그램을 만들기보다는 사용자 정의 타입을 만들고 이를 활용한다.
    * 추상화된 개념을 사용하면 프로그래밍이 간명해지고
    * 실수할 여지를 줄여준다.
    * 컴파일러가 실수를 발견할 확률을 높여준다.
- 이렇게 추상화되고 구조화된 기법을 활용하다 보면 런타임 오류가 발견되는 시점과 처리할 수 있는 시점이 달라질 수 있다.

Exception

```java
public class ExceptionExample {
    public static void main(String[] args) {
        Vector v = new Vector(5);
        System.out.println(v.get(10));
    }
}
```

- *Vector* 예제에서 원소 범위 밖의 데이터에 접근하면 어떻게 해야할까?
- *Vector* 코드를 작성한 사람은 사용자가 무엇을 원하는지 알 수 없다.
  - 어떤 프로그램이 이 코드를 사용할지도 미리 알기 어렵다.
- 사용자 입장에서는 항상 문제를 예상하고 이를 막을 수 없다.
  - 그럴 수 있었다면 애초에 범위 밖의 원소에 접근하려 하지 않았을 것이다.
- 이런 상황을 위해 자바에서는 예외(exception) 처리를 위한 메커니즘을 제공한다.

Exception

```java
public double get(int i) {
    if (i < 0 || size() <= i) {
        throw new ArrayIndexOutOfBoundsException("안댕");
    }
    return elem[i];
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 안댕
    at Vector.get(ExceptionExample.java:11)
    at ExceptionExample.main(ExceptionExample.java:31)
```

- *throw* 는 *ArrayIndexOutOfBoundsException* 타입의 예외를 처리하는 처리자(handler)에게 제어권을 넘기게 된다.
- 이 핸들러는 직접적 혹은 간접적으로 *Vector*의 *get* 메소드를 호출한 메소드가 알고 있어야 한다.
- 내부적으로 메소드 호출의 순서를 역행하며 이 타입의 예외를 처리하기로 한 핸들러를 찾아가게 된다.
- 익셉션의 인자로 처음 익셉션이 발생한 환경에 대한 최소한의 정보를 담아준다.

Exception

```java
public double get(int i) {
    return elem[i];
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10
    at Vector.get(ExceptionExample.java:10)
    at ExceptionExample.main(ExceptionExample.java:28)
```

- 자바는 기본적으로 다양한 익셉션을 지원한다.
  - 즉 위의 예에서는 따로 *throw*를 부르지 않아도 알아서 *ArrayIndexOutOfBoundsException* 타입의 예외가 던져진다.

Try-Catch

```java
static void f(Vector v) {
    try {
        v.set(v.size(), 7);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("안된다고: " + e.getMessage());
    }
}
```

- 예외가 혹시 났을 때 이를 직접 처리했으면 하는 부분을 *try* 블럭을 사용해서 표시한다.
  - *set* 메소드를 호출 할 때, *v.size()*를 인덱스로 넣어주면 *Vector* 가 처리할 수 있는 범위를 벗어나게 된다.
- *catch* 블럭은 처리할 예외의 타입과 처리하는 방식을 담게 된다.
  - *ArrayIndexOutOfBoundsException*은 이미 자바 API가 제공하는 익셉션 타입이다.
- 익셉션 처리 메커니즘은 오류를 처리하는 방식을 체계적이게 해준다.
- 이를 위해 *try* 블럭을 남용하는 것은 좋지 않다.

Programming in General

- 프로그램의 큰 목표 중 하나는 복잡도의 제어이다.
  - 학부에서 겪는 과제는 길어 봤자 10,000 라인 넘어가기 힘들다.
  - 리눅스 커널은 주석 제외 10,000,000 라인 정도 된다.
- 많은 수의 버그가 복잡한 프로그램을 다루다 프로그래머가 실수해서 생긴다.
- 좋은 프로그램 제작을 위한 트릭과 방법론은 매우 많다.
  - 방어적 프로그래밍 (defensive programming)
  - 짝 프로그래밍 (pair programming)
  - 리팩토링 (refactoring)
  - 테스트 주도 개발 (test-driven development)
  - ...
- 더 좋은 프로그래머가 되는 것에 관심이 있다면 스스로 찾아보고 공부하기 바란다.

Alan Kay on Smalltalk Design

- Everything is an object.
- Objects communicate by sending and receiving messages (in terms of objects).
- Objects have their own memory (in terms of objects).
- Every object is an instance of a class (which must be an object).
- The class holds the shared behavior for its instances (in the form of objects in a program list).
- To eval a program list, control is passed to the first object and the remainder is treated as its message.

Big Integer – Demo

DEMO

Big Integer – Procedural

```java
public class BigIntProc {
    public static void main(String[] args) {
        // Somehow parse the user input and split it into multiple strings
        String leftStr = "12345";
        String rightStr = "54321";
        char operator = '+';
        // Soemhow express operands in arrays
        int[] leftArray = new int[leftStr.length()];
        int[] rightArray = new int[rightStr.length()];
        int[] resultArray = new int[(leftStr.length() > rightStr.length())
                ? leftStr.length() : rightStr.length()];
        for (int i = 0; i < leftStr.length(); ++i) {
            leftArray[i] = Character.getNumericValue(leftStr.charAt(i));
            System.out.println(leftArray[i]);
        }
        for (int i = 0; i < rightStr.length(); ++i) {
            rightArray[i] = Character.getNumericValue(rightStr.charAt(i));
            System.out.println(rightArray[i]);
        }
        switch (operator) {
            case '+': {
                resultArray[leftStr.length() - 1]
                    = leftArray[leftStr.length() - 1] + rightArray[rightStr.length() - 1];
                //...
                }
        }
    }
}
```

Big Integer – OO

```
class BigInt {
    BigInt(String str) {
        // implement this
    }
    BigInt add(BigInt rhs) {
        // implement this
        return this;
    }
}
public class BigIntObj {
    static void parseInput(String argument) {
        // somehow...
    }
    public static void main(String[] args) {
        // somehow parse the input
        // parseInput();
        String leftStr = "1234";
        String rightStr = "4321";
        BigInt bigLeft = new BigInt(leftStr);
        BigInt bigRight = new BigInt(rightStr);
        BigInt result = bigLeft.add(bigRight);
    }
}
```

Object Oriented Programming

- 객체 지향 프로그래밍(OOP)은 언어의 수준에서 프로그램의 복잡도 제어를 어떻게 할지 제시하는 한 가지 패러다임이다.
- 객체가 서로 메시지를 주고 받는 식으로 프로그램을 구성한다.

Bare Minimum

- 빠른 피드백 루프를 만들어라.
- 코드 작성 - 컴파일 - 실행 루프를 합리적인 수준에서 최대한 짧게 만든다.
  - 프로그램을 작성해서 한 번에 컴파일 되고 정상적으로 실행될 것이라는 기대는 누구도 하지 않는다.
- 합리적인 수준에서 최소한의 코드를 작성한 뒤 생각한 것과 같게 제대로 실행되는지 확인한다.
  - 이를 위해 테스트 코드를 만든다.

Bare Minimum – Demo

DEMO

Bare Minimum

```
public class BigIntTest {
    public static void main(String[] args) {
        String leftStr = "1234";
        String rightStr = "4321";
    }
}
```

Bare Minimum

```
public class BigIntTest {
    public static void main(String[] args) {
        String leftStr = "1234";
        String rightStr = "4321";

        BigInt leftBig = new BigInt(leftStr);
    }
}
```

Bare Minimum

```java
class BigInt {
    public BigInt(String str) {
    }
}
```

Bare Minimum

```java
class BigInt {
    //FIXME this has to change to private
    public int[] data;

    public BigInt(String str) {
    }
}
```

Bare Minimum

```java
class BigInt {
    //FIXME this has to change to private
    public int[] data;

    public BigInt(String str) {
        data = new int[str.length()];
    }
}
```

Bare Minimum

```java
public class BigIntTest {
    public static void main(String[] args) {
        String leftStr = "1234";
        String rightStr = "4321";

        BigInt leftBig = new BigInt(leftStr);
        for (int i = 0; i < leftBig.data.length; ++i) {
            System.out.println(leftBig.data[i]);
        }
    }
}
```

Bare Minimum

```java
class BigInt {
    //FIXME this has to change to private
    public int[] data;

    public BigInt(String str) {
        data = new int[str.length()];
        for (int i = 0; i < str.length(); ++i) {
            data[i] = str.charAt(i);
        }
    }
}
```

Bare Minimum

```java
class BigInt {
    private int[] data;

    public BigInt(String str) {
        data = new int[str.length()];
        for (int i = 0; i < str.length(); ++i) {
            data[i] = Character.getNumericValue(str.charAt(i));
        }
    }

    public int length() {
        return data.length;
    }
}
```

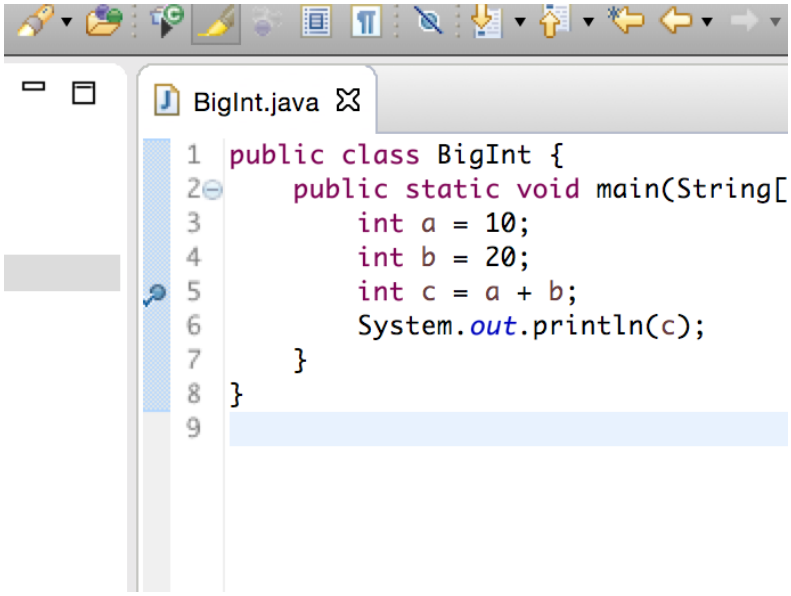
Bare Minimum

```java
class BigInt {
    private int[] data;
    public BigInt(String str) {
        data = new int[str.length()];
        for (int i = 0; i < str.length(); ++i) {
            data[i] = Character.getNumericValue(str.charAt(i));
        }
    }
    public int length() {
        return data.length;
    }
    public int get(int i) {
        return data[i];
    }
}
public class BigIntTest {
    public static void main(String[] args) {
        String leftStr = "1234";
        String rightStr = "4321";
        BigInt leftBig = new BigInt(leftStr);
        for (int i = 0; i < leftBig.length(); ++i) {
            System.out.println(leftBig.get(i));
        }
    }
}
```

Debugging

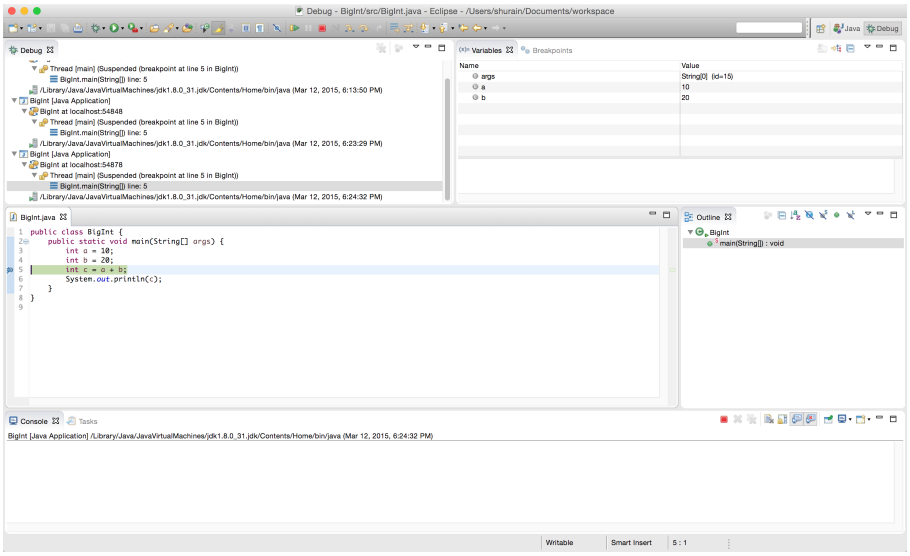
- 프로그램이 예상과 다르게 동작할 때 문제를 발견하고 이를 해결하는 과정을 디버깅이라 한다.
  - System.out.println
  - jdb
  - assert
  - Eclipse의 debug 기능
- Eclipse에서 F11을 누르면 디버그 모드로 프로그램이 실행된다.
  - Run 메뉴 밑에 필요한 기능이 모여있다.

Breakpoint



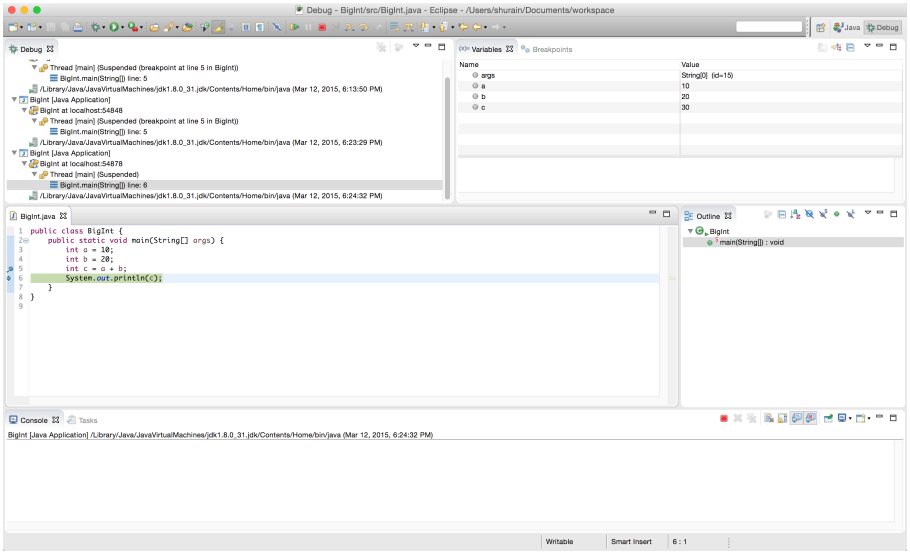
- 디버그 모드로 프로그램을 실행할 때 프로그램의 실행을 멈출 특정 위치를 중단점(breakpoint)라 한다.
- Eclipse에서는 코드의 왼쪽 편을 더블 클릭해서 설정할 수 있다.
- 그 밖에도 특정 메소드 단위로 설정하는 것도 가능하다.

Debug Mode



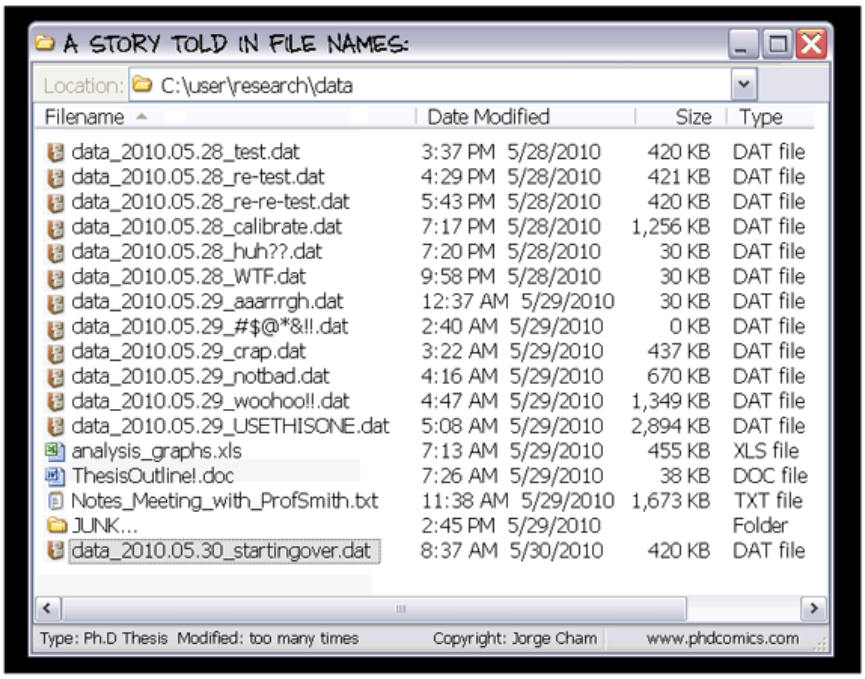
- 디버그 모드에서는 특정 변수에 어떤 값이 들어있는지 살펴볼 수 있다.

Debug Mode



- 중지된 프로그램을 다양한 방식으로 실행 재개할 수 있다.
  - (Step Into) 프로그램을 한 줄 진행하되 메소드를 만나면 메소드 내부로 진입하기.
  - (Steo Over) 프로그램을 한 줄 진행하되 메소드를 만나면 이를 수행한 뒤 결과를 받은 상황으로 진행하기.
  - (Resume) 다음 중단점까지 진행하기.

Code Backup



Code Backup

- 버전 관리 시스템을 사용한다.
  - Git
  - Mercurial
  - SVN
- `https://try.github.io/`

Advice

- 서로 관련있는 데이터는 클래스에 한데 모은다.
- 외부와의 상호작용 약속인 인터페이스와 그 구현을 클래스를 사용해서 분리한다.
- 생성자를 사용해서 클래스의 초기화를 보장한다.
- 이름 달린 상수를 정의하는 데에는 *enum*을 사용한다.
- *enum*에 각종 연산자를 정의해서 안전하고 쉽게 사용할 수 있게 한다.

Advice

- 클래스의 인터페이스와 실제 구현을 구분해서 생각한다.
- 할당된 일을 할 수 없음을 알리기 위해 익셉션을 던진다.
- 익셉션을 사용해서 오류를 처리한다.
- 프로그램 디자인 단계에서 일찍 에러 처리 방법을 고민한다.
- 모든 종류의 익셉션을 모든 메소드에서 처리하려고 하지 않는다.

Advice

- 프로그래밍은 복잡도를 관리하는 것이 아주 큰 부분을 차지한다.
- 객체 지향 프로그래밍은 객체 간의 메시지 전달로 복잡도를 관리한다.
- 추천 도서
  - 프로그래밍 수련법 – 브라이언 W. 커니헌, 롭 파이크
  - 실용주의 프로그래머 – 앤드류 헌트, 데이비드 토머스
  - 코드 컴플릿 – 스티브 맥코넬
  - 리팩토링 – 마틴 파울러
- 코드 버전 관리 및 백업은 버전 관리 시스템을 활용한다.

A Tour of Java III

Sungjoo Ha

March 18th, 2016

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

```java
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

```java
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

```java
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

```java
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

```java
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

```
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

```
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";
```

```java
        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.

A Tour of Java IV

Sungjoo Ha

March 25th, 2016

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.
- 각각의 객체는 기본형을 사용하는 것과 비슷한 느낌으로 사용할 수 있어야 한다.
- 인스턴스는 개별적인 상태를 갖는다.

Vector

```java
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
}
```

Abstract Type

- *Vector*와 같은 타입을 concrete type이라 부른다.
  - 외부 세계와의 계약(interface)과 내부적 구현(implementation)을 모두 포함하고 있다.
- Abstract type은 내부 구현에 대해 전혀 공개하지 않는 타입을 말한다.
  - 이를 위해 인터페이스와 구현을 완전히 분리한다.

Interface

```
interface Container {
    double elementAt(int i);
    void set(int i, double e);
    int size();
}
```

- 타입은 가능한 값의 집합과 연산의 집합을 정의한다.
- *interface*는 abstract type을 정의한다.
  - 내부 구현에 대한 정보는 전혀 없다.
  - {} 부분이 없고 곧바로 ;로 선언을 마치고 있다.
- *interface*는 상수, 메소드 시그너처 등만 가질 수 있다.
  - `http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html`
- *interface*에 정의된 모든 메소드는 *public abstract* 메소드이다.
- *interface*는 인스턴스를 바로 만들 수 없다.
  - *Container c = new Container(); // Wrong*

Container

```
static void use(Container c) {
    final int sz = c.size();
    for (int i = 0; i < sz; ++i) {
        System.out.println(c.elementAt(i));
    }
}
```

- *Container*가 어떤 방식으로 구현될지는 전혀 모르지만 앞서 *interface*를 통해 약속한 연산을 지원해야 한다.
- 우리는 *Container* 타입이 약속한 방법대로 사용할 수 있다.
  - 특정 조건을 만족하는 입력을 넣어주고 메소드를 호출하면 특정 타입의 결과를 얻는다.

VectorContainer

```
class VectorContainer implements Container {
    private Vector v;
    public VectorContainer(int s) {
        v = new Vector(s);
    }
    public double elementAt(int i) {
        return v.elementAt(i);
    }
    public void set(int i, double e) {
        v.set(i, e);
    }
    public int size() {
        return v.size();
    }
}

public static void main(String[] args) {
    Container c = new VectorContainer(10);
    use(c);
}
```

- *implements* 키워드를 통해 *VectorContainer* 클래스가 *Container* 인터페이스를 따름을 명시한다.
- *Container* 타입이 지원하기로 한 연산을 모두 구현해야 한다.

ArrayListContainer

```
import java.util.ArrayList;
class ArrayListContainer implements Container {
    private ArrayList<Double> list;
    public ArrayListContainer(int s) {
        list = new ArrayList<>(s);
        for (int i = 0; i < s; ++i) {
            list.add(0.0);
        }
    }
    public double elementAt(int i) {
        return list.get(i);
    }
    public void set(int i, double e) {
        list.add(i, e);
    }
    public int size() {
        return list.size();
    }
}
```

- *Container* 타입을 따르는 다른 클래스 *ArrayListContainer*를 만들었다.
- *VectorContainer*와 *ArrayListContainer*는 내부 구현이 다르지만 같은 인터페이스를 갖는다.

Container Comparison

```
public static void main(String[] args) {
    Container c = new VectorContainer(10);
    use(c);
}

public static void main(String[] args) {
    Container d = new ArrayListContainer(10);
    use(d);
}
```

- *use* 메소드는 어떤 종류의 *Container*가 입력으로 들어오는지 신경쓰지 않는다.
- *Container*에 의해 정의된 약속만 지켜주면 된다.

Comparable Interface

```
class Genre implements Comparable<Genre>
```

- 과제 뼈대 코드의 일부의 변형이다.
- *Genre*는 *Comparable* $<$*Genre*$>$ 인터페이스에서 정해진 연산을 정의하기로 하였다.
- *Comparable*$<T>$ 타입은 *int compareTo(T)* 연산을 정의하면 된다.
  - `http://docs.oracle.com/javase/7/docs/api/java/lang/Comparable.html`

compareTo

```
@Override
public int compareTo(Genre other) {
    // TODO implement this
    throw new UnsupportedOperationException();
}
```

- *@Override*는 annotation으로 여기에서는 컴파일러를 도와주는 역할을 한다.
- 해당 메소드가 상위 클래스나 인터페이스에 정의된 메소드를 override하였음을 알려준다.
- 이를 바탕으로 컴파일러가 우리가 의도한대로 잘 override 되었는지를 검사한다.

Comparable Example

```
class SmallInt implements Comparable<SmallInt> {
    public int d;
    public SmallInt(int a) {
        d = a;
    }

    @Override
    public int compareTo(SmallInt that) {
        if (this.d < that.d) {
            return -1;
        } else if (this.d > that.d) {
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

- *SmallInt*는 *Comparable* 인터페이스의 연산자를 지원하기로 약속했다.
  - 이를 위해 *compareTo()* 메소드를 구현하였다.

Comparable Example

```
import java.util.Arrays;
public class ComparableExample {
    public static void main(String[] args) {
        SmallInt a = new SmallInt(10);
        SmallInt b = new SmallInt(5);
        SmallInt c = new SmallInt(50);
        SmallInt[] list = new SmallInt[3];
        list[0] = a; list[1] = b; list[2] = c;
        for (int i = 0; i < 3; ++i) {
            System.out.println(list[i].d);
        }
        Arrays.sort(list);
        for (int i = 0; i < 3; ++i) {
            System.out.println(list[i].d);
        }
    }
}
```

- *Comparable*은 객체 사이에 total ordering을 부여하는 역할을 한다.
  - 즉, 객체를 서로 비교할 수 있다는 의미를 갖는 인터페이스이다.
- 비교 가능한 객체라면 정렬을 하거나 정렬된 상태를 유지하는 자료구조에 담을 수 있다.

Iterable Interface

```
public interface ListInterface<T> extends Iterable<T> {
```

- 과제 2 뼈대 코드의 일부이다.
- *ListInterface*는 *Iterable*$< T >$ 인터페이스에서 정해진 연산을 정의하기로 하였다.
- *Iterable*$< T >$ 타입은 *Iterator*$< T >$*iterator()* 연산을 정의하면 된다.
  - `http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html`

Iterator

```
@Override
public Iterator<T> iterator() {
    // This code does not have to be modified.
    // Implement MyLinkedListIterator instead.
    return new MyLinkedListIterator<T>(this);
}
```

- *Iterator*$< T >$는 다음의 세 메소드를 정의하면 된다.

- *boolean hasNext()*
- *T next()*
- *void remove()*
- `http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html`

Iterator

```
MyLinkedList<MovieDBItem> result = db.items();

for (MovieDBItem item : result) {
    System.out.printf("(%s, %s)\n", item.getGenre(), item.getTitle());
}
```

- *Iterator*는 컨테이너를 순회할 수 있게 해주는 객체이다.
- 컨테이너의 사용자가 직접 어디까지 살펴봤는지 기억하는 것이 아니라 컨테이너의 iterator가 이를 기억하고 관리한다.
- 직접적인 인덱스 접근에 비해 몇 가지 장점이 있다.
  - 임의의 위치에 한 번에 도달하는 것이 불가능한 자료구조를 사용할 때
  - 임의의 컨테이너를 순회하는 통일된 방법을 제공
    - *Iterable*는 *foreach* 문을 사용할 수 있게 해준다.

Iterator Example

```
import java.util.Iterator;
public class ArrayIterator implements Iterator<Object> {
    private Object[] array; private int cur;
    public ArrayIterator(Object[] a) { array = a; cur = 0; }
    public boolean hasNext() { return cur < array.length; }
    public Object next() { return array[cur++];    }
    public void remove() { throw new UnsupportedOperationException(); }
    public static void main(String[] args) {
        Object[] a = new Object[] { "A", "B", "C" };
        ArrayIterator it = new ArrayIterator(a);
        while (it.hasNext())
            System.out.println(it.next().toString());
    }
}
```

- *Iterator*의 구현 예이다.
- 자바의 모든 클래스는 *Object* 클래스를 암묵적으로 상속받는다.
- 그러므로 *Object*는 임의의 타입을 담을 수 있다.
- *Iterator* 인터페이스가 요구하는 세 메소드를 구현하고 있다.
- 내부적으로 커서를 두고 이를 옮겨가며 현재 위치를 기억한다.

Interface Usage

```
import java.util.*;

public class InterfaceSample {
    public static void main(String[] args) {
        List<String> s = Arrays.asList("C", "B", "A");
        List<Iterable<String>> cs = new ArrayList<Iterable<String>>();
        cs.add(new ArrayList<String>(s));
        cs.add(new LinkedList<String>(s));
        cs.add(new TreeSet<String>(s));
        cs.add(new HashSet<String>(s));
        for (Iterable<String> c: cs) {
            System.out.println(c.getClass().getName());
            for (String item: c) System.out.println(item);
        }
    }
}
```

- *Iterable* 인터페이스의 사용예이다.
- 다양한 구현체를 다루지만 같은 방식으로 사용할 수 있도록 약속하였기에 사용하는 입장에서 큰 고민 없이 사용할 수 있다.

Multiple Interfaces

```
class MultipleInterfaces implements InterfaceOne, InterfaceTwo
```

- *interface*를 통해 약속을 정한다.
- *interface*를 구현하는 클래스는 약속을 지키기로 한다.
- 한 클래스는 여러 *interface*의 약속을 전부 지키기로 할 수 있다.

Class Hierarchy

- 계층적 구조를 가진 개념을 표현하는 방법으로 상속(inheritance)이 있다.
- 상속을 통해 기존 클래스가 가진 코드를 재사용하는 새 클래스를 만들 수 있다.
- 자식 클래스(subclass, derived class, child class)는 부모의 모든 멤버를 물려 받는다.
- 생성자는 물려받지 않지만 명시적으로 자식 클래스에서 부모 클래스의 생성자를 호출할 수 있다.

Starcraft

```
class Marine {
    public void move() {}
    public void attackWithGun() {}
    public void draw() {}
    public void stimpack() {}
}
class Zergling {
    public void move() {}
    public void attackWithClaw() {}
    public void draw() {}
    public void burrow() {}
}
```

- *Marine*은 움직일 수 있고, 화면에 그려질 수 있어야 한다. 그 외에 총도 쏘고 스팀팩도 맞을 수 있다.
- *Zergling*은 움직일 수 있고, 화면에 그려질 수 있어야 한다. 그 외에 발톱 공격도 하고 땅 속으로 숨을 수도 있다.

Starcraft

```
public class Starcraft {
    public static void main(String[] args) {
        Marine marine = new Marine();
        Zergling zergling = new Zergling();

        marine.move();
    }
}
```

Inheritance

```
class Unit {
    public void move() {}
    public void draw() {}
}

class Marine extends Unit {
    public void attackWithGun() {
    }
    @Override
    public void draw() {
    }
    public void stimpack() {
    }
}

class Zergling extends Unit {
    public void attackWithClaw() {
    }
    @Override
    public void draw() {
    }
    public void burrow() {
    }
}
```

Inheritance

```java
public class Starcraft2 {
    public static void main(String[] args) {
        Marine marine = new Marine();
        Zergling zergling = new Zergling();

        marine.move();
    }
}
```

- *Marine*이나 *Zergling*이나 움직이는 것은 (*move*) 똑같다.
- *Marine*이나 *Zergling*이나 화면에 그려져야 한다는 것은 같으나 각자 다른 그림을 그리게 (*draw*) 될 것이다.
- 보다 상위 개념의 클래스로부터 기능을 빌려오거나 (move) 더 구체적인 기능을 제공하기 위해 (draw) 상속을 한다.
  - *extends* 키워드를 통해 이를 표시한다. 즉, 자식 클래스는 부모 클래스를 확장하는 것이다.
- 더 구체적인 기능을 제공하는 것을 overriding이라고 한다.
  - Overriding - 부모가 제공하는 것을 재정의 하는 것
  - Overloading - 이름은 같지만 시그너쳐가 다른 메소드를 정의하는 것
- 자바는 클래스 다중 상속을 지원하지 않는다.

Polymorphism vs if-else

```java
public class Starcraft {
    public static void main(String[] args) {
        List<Object> group = new ArrayList<Object>();
        group.add(new Marine());
        group.add(new Zergling());
        for (Object unit: group) {
            if (unit instanceof Marine)
                ((Marine) unit).move();
            else if (unit instanceof Zergling)
                ((Zergling) unit).move()
        }
    }
}
public class Starcraft2 {
    public static void main(String[] args) {
        List<Unit> group = new ArrayList<Unit>();
        group.add(new Marine());
        group.add(new Zergling());
        for (Unit unit: group) unit.move(); // OK
    }
}
```

- *if else*문에 비해 더 예쁜 코드를 만들 수 있다.
- Beautiful is better than ugly.

Object

```java
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;
}
```

- 자바의 모든 클래스는 *Object* 클래스를 암묵적으로 상속받는다.

  - *equals*
  - *hashCode*
  - *toString*
  - ...

- http://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html

Inheritance Example

```
public class CommandNotFoundException extends Exception {
    private String command;

    public CommandNotFoundException(String command) {
        super(String.format("input command: %s", command));
        this.command = command;
    }

    private static final long serialVersionUID = 1L;

    public String getCommand() {
        return command;
    }
}
```

- 과제 2 뼈대 코드의 일부이다.
- *super*를 사용해서 부모 클래스의 생성자를 호출한다.
- http://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html

Inheritance Example

```
if (command == null)
    throw new CommandNotFoundException(cmd);

catch (CommandNotFoundException e) {
    System.err.printf("command not found: %s\n", e.getCommand());
    e.printStackTrace(System.err);
}
```

- 새로 정의된 *CommandNotFoundException*은 *Exception*처럼 동작하고 추가적인 기능을 갖게 된다 (확장).

Abstract Class

```
public interface ConsoleCommand {
    void parse(String input) throws CommandParseException;
}

public abstract class AbstractConsoleCommand implements ConsoleCommand {
@Override
public void parse(String input) throws CommandParseException {
    String[] args = input.split(" *% *%? *");
    if (input.isEmpty())
        args = new String[0];
    parseArguments(args);
}
```

Abstract Class

- 과제 2 뼈대 코드의 일부이다.
- *AbstractConsoleCommand*는 추상(abstract) 클래스이다.
  - *abstract* 키워드로 이를 표시하고 있다.
- 추상 클래스는 인스턴스를 만들 수 없다.
  - *AbstractConsoleCommand a = new AbstractConsoleCommand(); // Wrong*

- 추상 클래스는 추상 메소드(abstract method)를 가질 수 있다.
  - 추상 메소드는 구현이 없는 메소드이다.
  - 이는 자식 클래스에서 구현해야 한다.

Abstract Class Inheritance

```
class InsertCmd extends AbstractConsoleCommand {
    @Override
    protected void parseArguments(String[] args) throws CommandParseException {
        if (args.length != 3)
            throw new CommandParseException(
                    "INSERT", Arrays.toString(args), "insufficient argument");
        this.genre = args[1];
        this.movie = args[2];
    }
}
```

- *InsertCmd* 클래스는 *AbstractConsoleCommand* 클래스를 상속한다.
- 추상 메소드인 *parseArguments()*를 구현하고 있다.

Polymorphism

```
ConsoleCommand command = null;
command.parse(input);
```

- *ConsoleCommand* 타입은 *parse()* 기능을 제공하기로 하였다.
- 런타임에 *command*가 실제로 어떤 타입인지 JVM이 살펴보고 적합한 메소드를 호출해준다.
- 다양한 타입을 다룰 수 있는 하나의 인터페이스를 제공하는 것을 폴리모피즘(polymorphism)이라 한다.

Abstract Class vs Interface

- 추상 클래스(abstract class)와 인터페이스(interface)는 약간의 차이가 있으나 유사하다.
  - 추상 클래스는 *static*이나 *final*이 아닌 멤버를 선언할 수 있고 구현을 채워넣은 메소드를 만들 수도 있다.
  - 인터페이스는 여러 개를 받을 수 있다.
  - ...
- 추상 클래스는 이럴 때 사용한다.
  - 서로 관련 깊은 클래스들이 코드를 공유하고 싶을 때
  - 자식 클래스가 부모 클래스의 *public* 아닌 멤버에 접근하고 싶을 때
- 인터페이스는 이럴 때 사용한다.
  - 서로 관계가 깊지 않은 클래스가 모두 구현해야 하는 약속을 정의할 때
    - 가령 *Comparable*
- `http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html`

Interface and Inheritance

- 인터페이스와 상속은 방대한 주제이다. 자세한 내용은 스스로 공부하기 바란다.
- `http://docs.oracle.com/javase/tutorial/java/IandI/index.html`

Advice

- 객체를 사용하는 약속과 구현을 분리하기 위해 *interface*를 사용한다.
- 특정 타입이 보장하는 약속을 사용해서 다양한 타입이 같은 인터페이스를 사용하는 폴리모피즘을 구현할 수 있다.
- 계층적 구조를 지닌 개념을 표현하기 위해 상속을 사용한다.
- *@Override*를 사용해서 메소드를 *override* 함을 명확하게 표현한다.
- 가능하면 *interface*를 상속에 비해 선호하도록 한다.
  - 하지만 스스로 잘 판단하고 필요하다면 상속을 사용하는 것에 두려움을 느끼지 않아도 된다.
- 수업 시간에 배우는 자료구조는 타입에 불과하다. 겁먹지 말자.

A Tour of Java V

Sungjoo Ha

April 1st, 2016

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.

Review

- 각각의 객체는 기본형을 사용하는 것과 비슷한 느낌으로 사용할 수 있어야 한다.
- 인스턴스는 개별적인 상태를 가진다.
- 객체를 사용하는 약속과 구현을 분리하기 위해 *interface*를 사용한다.
- 특정 타입이 보장하는 약속을 사용해서 다양한 타입이 같은 인터페이스를 사용하는 폴리모피즘을 구현할 수 있다.
- 계층적 구조를 지닌 개념을 표현하기 위해 상속을 사용한다.
- 가능하면 *interface*를 상속에 비해 선호하도록 한다.

Generic

- 컨테이너는 일반적인 개념이고 특정 타입에 묶이지 않는다.
- 컨테이너가 특정 타입만 받길 원하지 않는다.
  - 가령 링크드 리스트는 임의의 타입을 지닌 데이터를 받게 하고 싶다.

MovieDatabase Example 1

```
class MovieNode {
    String name;
    MovieNode next;
}
class MovieList {
    MovieNode head;
}
class GenreNode {
    String name;
    GenreNode next;
    MovieList movieList;
}
class GenreList {
    GenreNode head;
}
public class MovieDatabase {
    public static void main(String[] args) {
        GenreList genreList = new GenreList();
    }
}
```

- 과제 2의 수도코드이다.
- 만약 구현해야 하는 것이 Movie와 Genre 두 개가 아니라 100가지라면 어떻게 해야 할까?

Object-Based Node

```
class Node {
    private Object object;
    public void set(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
public class NodeTest {
    public static void main(String[] args) {
        Node node = new Node();
        node.set(1);
        System.out.println(node.get());
        Node node2 = new Node();
        node2.set("NODE");
        System.out.println(node2.get());
    }
}
```

- 임의의 타입을 받을 수 있는 링크드 리스트의 노드 예제이다.
- 자바의 모든 클래스는 *Object* 클래스를 상속 받으므로 이런 방식으로 작성하는 것이 가능하다.

Object-Based Node

```
public class NodeTest {
    public static void main(String[] args) {
        Node node = new Node();
        node.set(1);
        System.out.println(node.get());
        Integer i = (Integer) node.get();
        System.out.println(i);
        String str = (String) node.get();
        System.out.println(str);
    }
}
```

- 위의 코드는 컴파일 오류가 발생하지 않는다.
- 하지만 런타임에 오류가 발생한다.
- *Object*와 타입 캐스팅을 사용하는 접근은 넣는 데이터와 나오는 데이터의 타입을 보장할 수 없다.

Generic Class

- 프로그램에 버그가 발생하는 것은 당연하다.
- 이를 프로그램 실행 중에 발견하는 것보다 컴파일 타임에 발견하길 원한다.
- 제네릭(Generic)을 사용해서 컴파일 타임에 버그를 발견할 확률을 높인다.

Generic Node

```
class NodeGeneric<T> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
}

public class NodeTest2 {
    public static void main(String[] args) {
        NodeGeneric<Integer> node = new NodeGeneric<Integer>();
```

```
        node.set(1);
        System.out.println(node.get());
        NodeGeneric<String> node2 = new NodeGeneric<String>();
        node2.set("NODE");
        System.out.println(node2.get());
        Integer i = node.get();
        System.out.println(i);
        // String str = node.get();  // compile error
        // System.out.println(str);
    }
}
```

Generic Node

- 제네릭 타입은 타입을 인자로 받는 클래스나 인터페이스를 지칭한다.
- $<T>$가 $NodeGeneric$이 제네릭 클래스임을 표시한다.
- <>는 타입 인자를 표시하며 T는 타입 변수를 나타낸다.
- 기존 $Node$ 클래스와 비교해보면 $Object$가 T로 변한 것을 알 수 있다.
- T는 기본형이 아닌 임의의 타입이 들어올 수 있다.
  - 클래스, 인터페이스, 등등
- 암묵적인 약속으로 타입 인자는 대문자 하나로 표시한다.
  - 프로그래머의 원활한 이해를 위해

Object-Based MovieDatabase

```
class Movie {
    String name;
}
class Genre {
    String name;
    LinkedList movieList;
}
class Node {
    Object obj;
    Node next;
}
class LinkedList {
    Node head;
}
public class MovieDatabase2 {
    public static void main(String[] args) {
        LinkedList genreList = new LinkedList();
    }
}
```

Generic MovieDatabase

```
class Movie {
    String name;
}
class Genre {
    String name;
    LinkedList<Movie> movieList;
}
class Node<T> {
    T obj;
    Node<T> next;
}
class LinkedList<T> {
    Node<T> head;
}
public class MovieDatabase3 {
    public static void main(String[] args) {
        LinkedList<Genre> genreList = new LinkedList<Genre>();
    }
}
```

Object vs Generic

```
class Node {
    final Object item;
    public Node(Object obj) {
        this.item = obj;
    }
}

class Node<T> {
    final T item;
    public Node(T obj) {
        this.item = obj;
    }
}
```

- 코드 상 차이는 적다.
- 제네릭이 가져다주는 이득을 음미해보자.

Generic

- 제네릭을 통해 컴파일 타임에 더 강력한 타입 체크를 할 수 있다.
- 제네릭을 통해 타입 캐스팅을 최소화할 수 있다.
- 제네릭을 통해 다양한 타입에 적용할 수 있는 알고리즘을 작성한다.
- 제네릭은 컴파일 시간에 다양한 작업을 하며 추가적인 런타임 부하가 없다.

Diamond

```
NodeGeneric<Integer> node = new NodeGeneric<>();
node.set(1);
System.out.println(node.get());
```

- 자바7부터 컨스트럭터 호출에 필요한 타입 인자를 생략할 수 있다.
  - 컴파일러가 추론할 수 있다면
- 이를 흔히 다이아몬드라 부른다.

Multiple Type Parameters

```
interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

class MyPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;
    public MyPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
}
```

- 제네릭 클래스는 여러 타입 인자를 받을 수 있다.
- 제네릭 인터페이스도 제네릭 클래스와 같은 규칙을 따른다.
- *MyPair* 클래스는 *Pair* 인터페이스를 구현하고 있다.

Multiple Type Parameters

```
Pair<String, Integer> p1 = new MyPair<String, Integer>("Dice", 6);
Pair<String, String> p2 = new MyPair<>("Hello", "World");
Pair<String, NodeGeneric<Integer>> p
    = new MyPair<>("Data", new NodeGeneric<Integer>(10));
```

- 처음 예제에서 K는 *String*, V는 *Integer*로 초기화 된다.
- Autoboxing에 의해 *Integer* 타입을 기대하는 곳에 *int*를 바로 넣어도 된다.
- 타입을 기대하는 곳에 인자화된 타입(parameterized type)을 넣어도 된다.
  - V에 $NodeGeneric<Integer>$ 타입을 넣어줬다.

Generic Example

```
class Node<T> {
    private T item;

    public Node(T obj) {
        this.item = obj;
    }
}
```

- 과제 2 뼈대 코드의 일부이다.
- *Node* 클래스는 타입 인자 T를 받는다.
- 그러므로 *Node*는 임의의 타입을 받아서 이를 *item*에 저장하는 역할을 한다.

Generic Example

```
    class Genre extends Node<String> implements Comparable<Genre> {
    @Override
    public int compareTo(Genre other) {
        // TODO implement this
        throw new UnsupportedOperationException();
    }
}
```

- 과제 2 뼈대 코드의 일부이다.
- *Genre* 클래스는 *Comparable* 인터페이스를 구현하고 있다.
- *Comparable* 인터페이스는 제네릭 인터페이스이다.
  - $public\ interface\ Comparable<T>$
  - $int\ compareTo(T\ o)$
- *Genre*는 *Comparable*의 T의 위치에 *Genre*를 넣은 것이다.
  - 그러므로 *int compareTo(Genre other)*를 구현해야 한다.
- 의미상 *Genre*는 다른 *Genre*와 비교할 수 있는 타입임을 뜻한다.

Generic

- 제네릭은 방대한 주제이다. 스스로 공부하기 바란다.
  - 한정된 타입 인자(bounded type parameter)
  - 제네릭 메소드(generic method)
  - 와일드카드(wildcard)
  - 타입 이레이져(type erasure)
- `http://docs.oracle.com/javase/tutorial/java/generics/index.html`

Advice

- 제네릭을 통해 다양한 타입에 적용할 수 있는 알고리즘을 작성한다.
- 코드의 중복은 피하고 코드 재활용성을 높인다.
- 특정 타입에 묶이지 않은 일반적인 개념/컨테이너를 표현하기 위해 제네릭을 사용한다.
- 제네릭을 사용해서 컴파일 타임에 버그를 발견할 확률을 높인다.
- 제네릭을 사용할 때 미리 타입 인자가 만족해야 하는 요구 사항을 따져본다.