A Tour of Java II

Sungjoo Ha

March 13th, 2015

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 *v*를 위한 메모리 공간을 할당하게 된다.
- 하지만 현재 구조로는 큰 쓸모가 없다.
  - *v*의 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;
}
```

- *v*의 *elem*은 *double[]* 타입이므로 이에 해당하는 메모리 공간이 *new double[s]*에 의해 할당된다.
- 마찬가지로 *v*의 *sz*는 *int* 타입으로 *s*의 값이 복사된다.
- 메소드 인자로 *Vector v*를 받을 때 *v*가 인자로 넘어온 *Vector*가 위치한 메모리 공간에 이름표로 붙는다.
  - 그러므로 *vector_init()*에 의해 *v*의 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; }
    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

```
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

```
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

```
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

```
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

```
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

```
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.len
                //...
                }
```

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

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

Bare Minimum

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

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

Bare Minimum

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

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

Bare Minimum

```
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

```
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

```
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;
    }
}
```

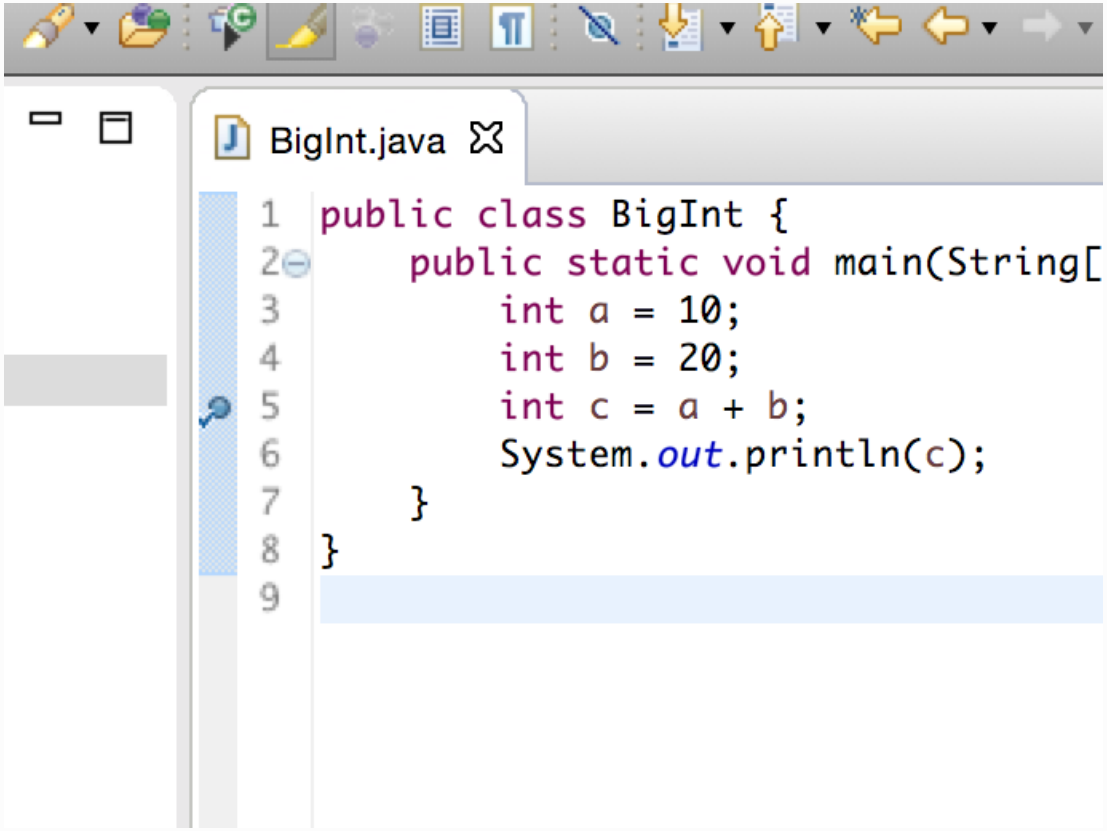
Bare Minimum

```
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

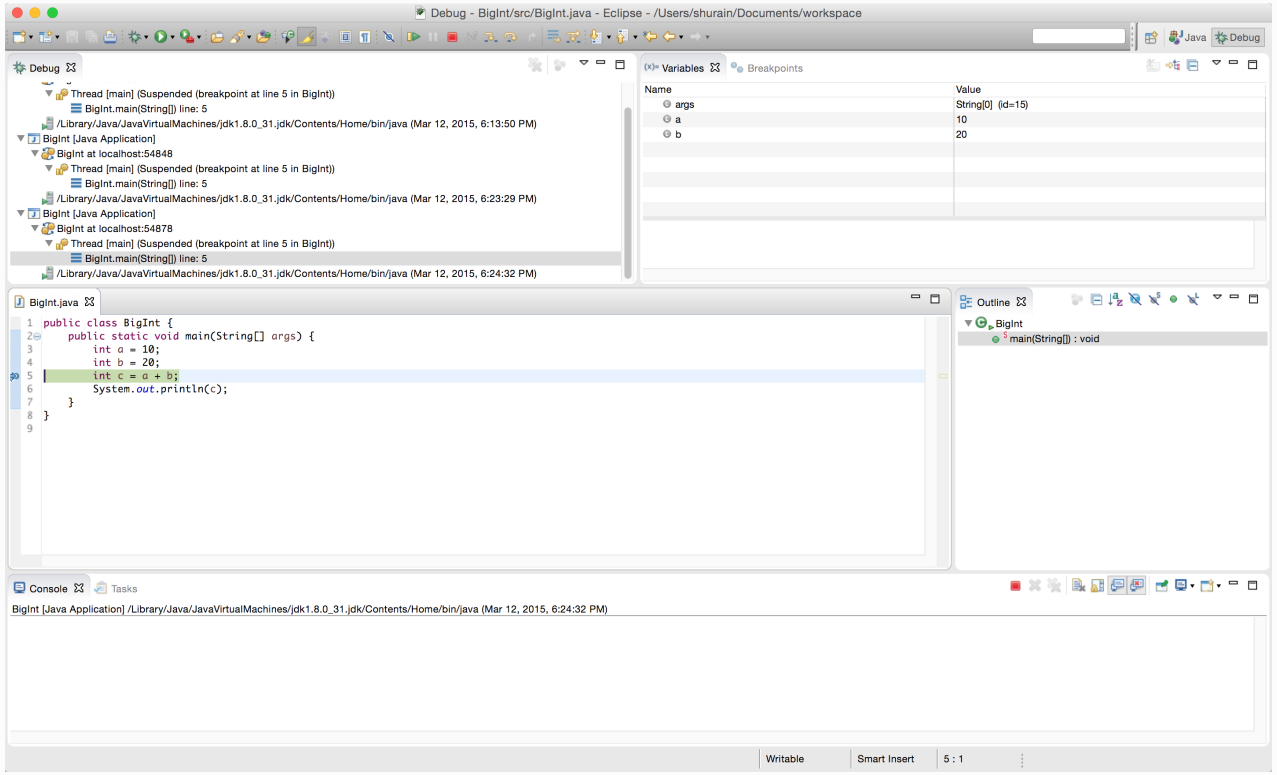
- 프로그램이 예상과 다르게 동작할 때 문제를 발견하고 이를 해결하는 과정을 디버깅이라 한다.
  - System.out.println
  - jdb
  - assert
  - Eclipse의 debug 기능
- Eclipse에서 F11을 누르면 디버그 모드로 프로그램이 실행된다.
  - Run 메뉴 밑에 필요한 기능이 모여있다.

Breakpoint



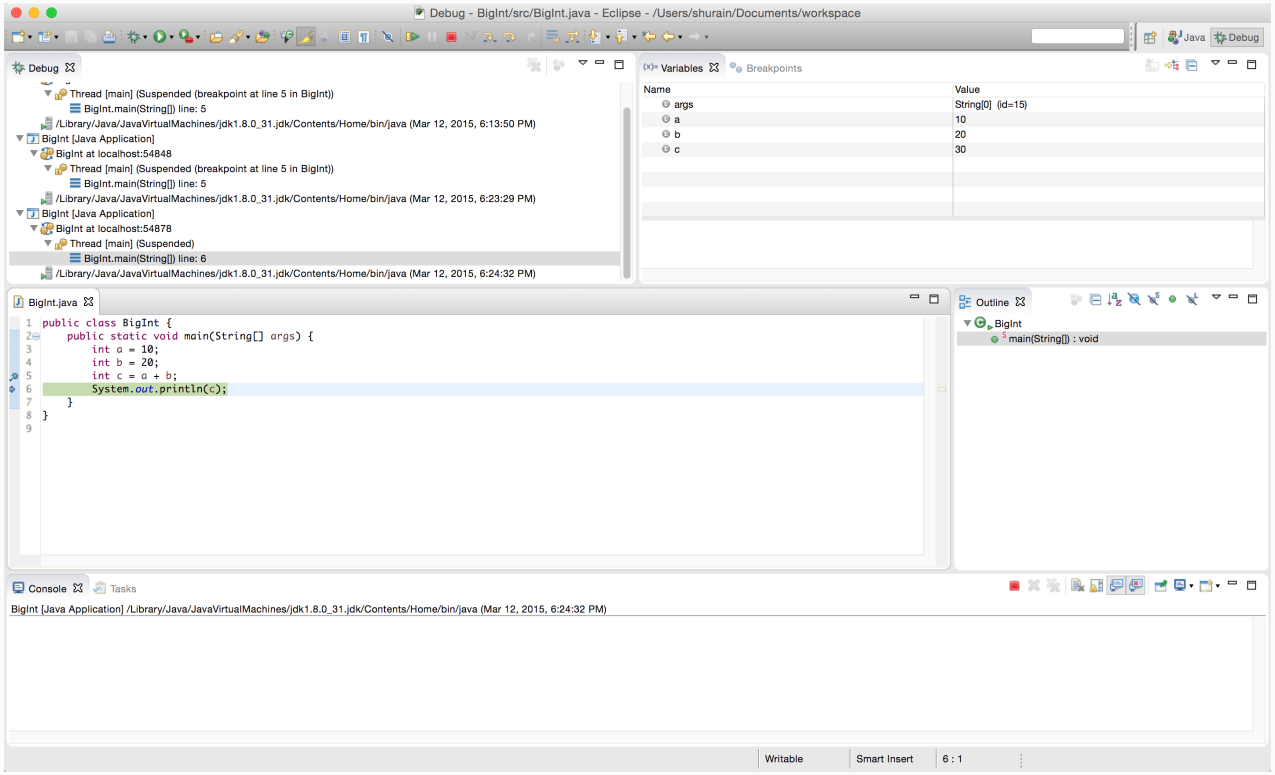
- 디버그 모드로 프로그램을 실행할 때 프로그램의 실행을 멈출 특정 위치를 중단점(breakpoint)라 한다.
- Eclipse에서는 코드의 왼쪽 편을 더블 클릭해서 설정할 수 있다.
- 그 밖에도 특정 메소드 단위로 설정하는 것도 가능하다.

Debug Mode



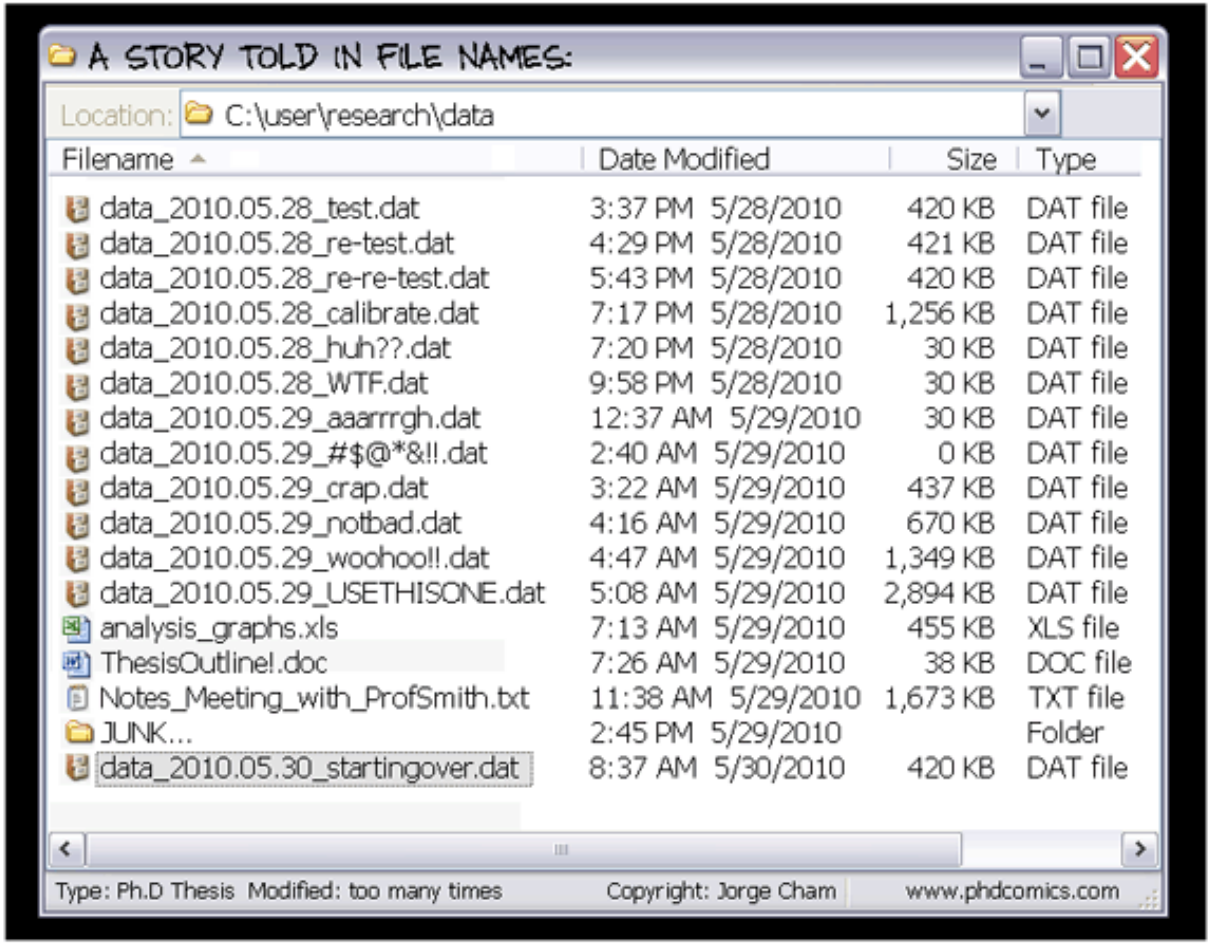
- 디버그 모드에서는 특정 변수에 어떤 값이 들어있는지 살펴볼 수 있다.

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


A STORY TOLD IN FILE NAMES:
Location: C:\user\research\data
Filename
Date Modified
Size
Type
data_2010.05.28_test.dat 3:37 PM 5/28/2010 420 KB DAT file
data_2010.05.28_re-test.dat 4:29 PM 5/28/2010 421 KB DAT file
data_2010.05.28_re-re-test.dat 5:43 PM 5/28/2010 420 KB DAT file
data_2010.05.28_calibrate.dat 7:17 PM 5/28/2010 1,256 KB DAT file
data_2010.05.28_huh??.dat 7:20 PM 5/28/2010 30 KB DAT file
data_2010.05.28_WTF.dat 9:58 PM 5/28/2010 30 KB DAT file
data_2010.05.29_aaarrrgh.dat 12:37 AM 5/29/2010 30 KB DAT file
data_2010.05.29_#$@*&!!.dat 2:40 AM 5/29/2010 0 KB DAT file
data_2010.05.29_crap.dat 3:22 AM 5/29/2010 437 KB DAT file
data_2010.05.29_notbad.dat 4:16 AM 5/29/2010 670 KB DAT file
data_2010.05.29_woohoo!!.dat 4:47 AM 5/29/2010 1,349 KB DAT file
data_2010.05.29_USETHISONE.dat 5:08 AM 5/29/2010 2,894 KB DAT file
analysis_graphs.xls 7:13 AM 5/29/2010 455 KB XLS file
ThesisOutline!.doc 7:26 AM 5/29/2010 38 KB DOC file
Notes_Meeting_with_ProfSmith.txt 11:38 AM 5/29/2010 1,673 KB TXT file
JUNK... 2:45 PM 5/29/2010 Folder
data_2010.05.30_startingover.dat 8:37 AM 5/30/2010 420 KB DAT file
Type: Ph.D Thesis Modified: too many times
Copyright: Jorge Cham
www.phdcomics.com

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.