A Tour of Java IV

Sungjoo Ha

March 27th, 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.
- 각각의 객체는 기본형을 사용하는 것과 비슷한 느낌으로 사용할 수 있어야 한다.
- 인스턴스는 개별적인 상태를 갖는다.

Vector

```
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 일부이다.
- *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 class MyLinkedList<T extends Comparable<T>> implements Iterable<T>
```

- 과제 2 뼈대 코드의 일부이다.
- *MyLinkedList*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

```java
@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<QueryResult> result = new MyLinkedList<QueryResult>();

for (QueryResult 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

```
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

```
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

```
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 Command {
    void apply(MovieDatabase db, String args) throws DatabaseException;
}

public abstract class AbstractCommand implements Command {
    @Override
    public void apply(MovieDatabase db, String args) throws DatabaseException {
        String[] arga = parse(args);
        queryDatabase(db, arga);
    }
    private String[] parse(String args) throws CommandParseException {
        if (args.isEmpty()) {
            return new String[] {};
        } else {
            // FIXME implement this
            // Parse the input appropriately.
            // You may need to change the return value.
            return args.split(" ");
        }
    }
    protected abstract void queryDatabase(MovieDatabase db, String[] arga) thro
}
```

Abstract Class

- 과제 2 뼈대 코드의 일부이다.
- *AbstractCommand*는 추상(abstract) 클래스이다.
  - *abstract* 키워드로 이를 표시하고 있다.
- 추상 클래스는 인스턴스를 만들 수 없다.
  - *AbstractCommand a = new AbstractCommand(); // Wrong*
- 추상 클래스는 추상 메소드(abstract method)를 가질 수 있다.
  - 추상 메소드는 구현이 없는 메소드이다.
  - 이는 자식 클래스에서 구현해야 한다.

Abstract Class Inheritance

```
import java.util.Arrays;

public class InsertCmd extends AbstractCommand {
    @Override
    protected void queryDatabase(MovieDatabase db, String[] arga) throws Databa
        checkArga(arga);
        db.insert(arga[0], arga[1]);
    }

    private void checkArga(String[] arga) throws DatabaseException {
        if (arga.length != 2)
            throw new CommandParseException("INSERT", Arrays.toString(arga), "i
    }
}
```

- *InsertCmd* 클래스는 *AbstractCommand* 클래스를 상속한다.
- 추상 메소드인 *queryDatabase()*를 구현하고 있다.

Polymorphism

```
Command command = commands.get(cmd);

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

String arguments = inputs.length > 1 ? inputs[1] : "";

command.apply(db, arguments);
```

- *Command* 타입은 *apply()* 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.