# A Tour of Java V

Sungjoo Ha

April 3rd, 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.

# 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 100 가지라면 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 *T*는 타입 변수를 나타낸다.
- 기존 *Node* 클래스와 비교해보면 *Object*가 *T*로 변한 것을 알 수 있다.
- *T*는 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 *K*는 *String*, *V*는 *Integer*로 초기화 된다.
- Autonodeing에 의해 *Integer* 타입을 기대하는 곳에 *int*를 바로 넣어도 된다.
- 타입을 기대하는 곳에 인자화된 타입(parameterized type)을 넣어도 된다.
  - *V*에 $Node<Integer>$ 타입을 넣어줬다.

# Generic Example

```
class Node<T> {
    // FIXME implement this
    final T item;

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

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

# Generic Example

```
class Genre 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 *T*의 위치에 *Genre*를 넣은 것이다.
  - 그러므로 *int compareTo(Genre other)*를 구현해야 한다.
- 의미상 *Genre*는 다른 *Genre*와 비교할 수 있는 타입임을 뜻한다.

# Bounded Type Parameters

- 제네릭에서 사용할 수 있는 타입의 종류를 한정하고 싶을 때가 있다.
  - 가령 숫자에 대한 작업을 하는 클래스는 숫자 타입만 받았으면 한다.
- 이를 위해 bounded type parameter를 사용한다.

# Bounded Type Parameters

```
class MyNumber<T extends Number> {
    private T t;
    public void set(T t) {
        this.t = t;
    }
    public T get() {
        return t;
    }
    public int intValue() {
        return t.intValue();
    }
}
```

- Bounded type parameter를 선언하기 위해 타입 인자의 이름 뒤에 *extends*와 함께 상한(upper bound)을 표시한다.
- 이 맥락에서 *extends*는 클래스의 확장(extends)과 인터페이스의 구현(implements)을 모두 포함한다.
  - 즉, 특정 클래스만 받거나 특정 인터페이스만 받는 것을 전부 *extends*로 나타낸다.
- *T*는 *Number* 클래스를 상속받은 클래스여야만 한다.
- 그러므로 *Number* 클래스가 제공하는 *intValue()* 메소드를 호출할 수 있다.

# Bounded Type Parameters

```
public class BoundedType {
    public static void main(String[] args) {
        MyNumber<Double> a = new MyNumber<>();
        a.set(10.2);
        System.out.println(a.intValue());
        //MyNumber<String> b = new MyNumber<>();  // compile error
    }
}
```

- *MyNumber* 클래스는 타입 인자가 *Number* 혹은 그 후손이어야 한다.
- 그러므로 *String*을 타입 인자로 받으면 컴파일 오류가 난다.

# Bounded Type Parameters Example

```
class ArrayBox<T> {
    private T[] array;
    public ArrayBox(T[] array) {
        this.array = array;
    }
    public int countGreaterThan(T elem) {
        int count = 0;
        for (T e: array) {
            if (e > elem) {
                count++;
            }
        }
        return count;
    }
}
public class Compare {
    public static void main(String[] args) {
        Integer[] array = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        ArrayBox<Integer> a = new ArrayBox<>(array);
        int count = a.countGreaterThan(5);
        System.out.println(count);
    }
}
```

이 코드의 문제는 무엇일까?

# Bounded Type Parameters Example

```
class ArrayBox<T extends Comparable<T>> {
    private T[] array;
    public ArrayBox(T[] array) {
        this.array = array;
    }
    public int countGreaterThan(T elem) {
        int count = 0;
        for (T e: array) {
            if (e.compareTo(elem) > 0) {
                count++;
            }
        }
        return count;
    }
}
```

- $>$ 연산은 기본형에서만 가능하다.
- 임의의 타입 *T*를 비교하기 위해서는 *T*가 비교 가능해야 한다.
- 그러므로 *T*는 *Comparable*$<T>$ 인터페이스를 구현해야만 한다.
  - 비교는 *compareTo(T)* 메소드를 사용한다.
- 이를 위해 *T*는 bounded type parameter이어야 하며 상한 (upper bound)이 *Comparable*$<T>$이다.

# Bounded Type Parameters Example

```
public class MyLinkedList<T extends Comparable<T>> implements Iterable<T>
```

- 과제 2 뼈대 코드의 일부이다.
- *MyLinkedList*는 타입 인자 *T*를 받는다.
- *T*는 *Comparable*$<T>$ 타입이어야 한다. 즉, 다른 *T*와 비교 가능해야 한다.
  - *T*는 *int compareTo(T o)* 메소드를 구현해야 한다.
- *MyLinkedList*는 *Iterable*$<T>$를 구현한다.
  - *Iterator*$<T>$*iterator()* 메소드를 제공해야 한다.

# Bounded Type Parameters Example

```
MyLinkedList<String> myList = new MyLinkedList<>();
MyLinkedList<Genre> myList2 = new MyLinkedList<>();
```

- *String*은 *Comparable*<*String* >을 구현한다.
- 그러므로 *MyLinkedList*의 타입 인자로 사용할 수 있다.
- 마찬가지로 *Genre*도 *Comparable*<*Genre* >를 구현하므로 타입 인자로 사용할 수 있다.
  - *class Genre implements Comparable*<*Genre* >

# Generic

- 제네릭은 방대한 주제이다. 스스로 공부하기 바란다.
  - 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.