11-03 [Java - Generic]

소유자	종수 김
태그	

제네릭1

제네릭이 필요한 이유

```
package generic.ex1;

public class IntegerBox {
    private Integer value;

    public void set(Integer value) {
        this.value = value;
    }

    public Integer get() {
        return value;
    }
}

package generic.ex1;

public class StringBox {
    private String value;

    public String get() {
        return value;
    }
```

```
    public void set(String value) {
        this.value = value;
    }
}


package generic.ex1;

public class BoxMain1 {
    public static void main(String[] args) {
        IntegerBox integerBox = new IntegerBox();
        integerBox.set(10); //auto boxing
        Integer integer = integerBox.get();
        System.out.println("integer = " + integer);

        StringBox stringBox = new StringBox();
        stringBox.set("hello");
        String str = stringBox.get();
        System.out.println("str = " + str);
    }
}
```

문제

이후에 `Double`, `Boolean`을 포함한 다양한 타입을 담는 박스가 필요하다면 각각의 타입별로 `DoubleBox`, `BooleanBox`와 같이 클래스를 새로 만들어야 한다. 담는 타입이 수십개라면, 수십개의 `XxxBox` 클래스를 만들어야 한다.

이 문제를 어떻게 해결할 수 있을까?

다형성을 통한 중복 해결 시도

Object는 모든 클래스의 부모이므로 Object로 해결할 수 있지 않을까?

```
package generic.ex1;

public class ObjectBox {
    private Object value;
```

```
    public Object get() {
        return value;
    }

    public void set(Object value) {
        this.value = value;
    }
}

package generic.ex1;

public class BoxMain2 {
    public static void main(String[] args) {
        ObjectBox integerBox = new ObjectBox();
        integerBox.set(10);

        Object object = integerBox.get();
        Integer integer = (Integer) object;

        System.out.println("integer = " + integer);

        ObjectBox stringBox = new ObjectBox();
        stringBox.set("hello");
        String str = (String) stringBox.get();
        System.out.println("str = " + str);

        //잘못된 타입의 인수 전달시
        integerBox.set("100");
        Integer result = (Integer) integerBox.get();
        System.out.println("result = " + result); //ClassCastI
    }
}
```

문제점

1. 반환 타입이 맞지 않는 문제
    a. integerBox를 만들어서 숫자 10을 보관했을 때, 이를 꺼내기 위해 다운캐스팅이 필수임.

2. 잘못된 타입의 인수 전달 문제
    a. integerBox.set(”100”)
    b. 개발자의 의도는 integerBox에 숫자가 들어가길 원했지만, Object로 인해 어떤 타입을 넣어도 컴파일에러가 나지 않음.

정리

다형성을 활용한 덕분에 코드의 중복을 제거하고, 기존 코드를 재사용할 수 있게 되었다. 하지만 입력할 때 실수로 원하지 않는 타입이 들어갈 수 있는 타입 안전성 문제가 발생한다. 예를 들어서 `integerBox`에는 숫자만 넣어야 하고, `stringBox`에는 문자열만 입력할 수 있어야 한다. 하지만 박스에 값을 보관하는 `set()`의 매개변수가 `Object`이기 때문에 다른 타입의 값을 입력할 수 있다. 그리고 반환 시점에도 `Object`를 반환하기 때문에 원하는 타입을 정확하게 받을 수 없고, 항상 위험한 다운 캐스팅을 시도해야한다. 결과적으로 이 방식은 타입 안전성이 떨어진다.

지금까지 개발한 프로그램은 코드 재사용과 타입 안전성이라는 2마리 토끼를 한번에 잡을 수 없다. 코드 재사용을 늘리기 위해 `Object`와 다형성을 사용하면 타입 안전성이 떨어지는 문제가 발생한다.

- `BoxMain1` : 각각의 타입별로 `IntegerBox`, `StringBox`와 같은 클래스를 모두 정의
    - 코드 재사용X
    - 타입 안전성O
- `BoxMain2` : `ObjectBox`를 사용해서 다형성으로 하나의 클래스만 정의
    - 코드 재사용O
    - 타입 안전성X

코드 재사용, 타입 안전성 둘을 다 만족할 수 없음

제네릭 적용

코드 재사용, 타입 안전성을 둘 다 만족할 수 있는 기능

```
package generic.ex1;

public class GenericBox<T> {
    private T value;

    public T get() {
        return value;
    }

    public void set(T value) {
        this.value = value;
```

```
    }
}
```

- `<>`를 사용한 클래스를 제네릭 클래스라 한다. 이 기호(`<>`)를 보통 다이아몬드라 한다.
- 제네릭 클래스를 사용할 때는 `Integer`, `String` 같은 타입을 미리 결정하지 않는다.
- 대신에 클래스명 오른쪽에 `<T>`와 같이 선언하면 제네릭 클래스가 된다. 여기서 `T`를 **타입 매개변수**라 한다. 이 타입 매개변수는 이후에 `Integer`, `String`으로 변할 수 있다.
- 그리고 클래스 내부에 `T` 타입이 필요한 곳에 `T value`와 같이 타입 매개변수를 적어두면 된다.

```
package generic.ex1;

public class BoxMain3 {
    public static void main(String[] args) {
        GenericBox<Integer> integerBox = new GenericBox<>();
        integerBox.set(10);
//        integerBox.set("100"); // Integer 타입만 허용, 컴파일 오
        Integer integer = integerBox.get(); // Integer 타입 반환
        System.out.println("integer = " + integer);

        GenericBox<String> stringBox = new GenericBox<>();
        stringBox.set("Hello");
//        stringBox.set(100); // String 타입만 허용, 컴파일 오류
        String string = stringBox.get(); // String 타입 반환(캐
        System.out.println("string = " + string);

        //원하는 모든 타입 사용 가능
        GenericBox<Double> doubleGenericBox = new GenericBox<
        doubleGenericBox.set(3.14);
        Double doubleDouble = doubleGenericBox.get();
        System.out.println("doubleDouble = " + doubleDouble);

    }
}
```

1. 생성 시점에 타입을 결정
2. Set, Get의 매개변수 타입이 고정되므로 타입 안전성 보장

    a. 캐스팅이 필요하지 않음.
3. 원하는 모든 타입 사용 가능
    a. primitve타입은 사용 불가능
    b. Wrapper Class 사용 해야함.
4. 타입 추론

참고로 제네릭을 도입한다고 해서 앞서 설명한 `GenericBox<String>`, `GenericBox<Integer>`와 같은 코드가 실제 만들어지는 것은 아니다. 대신에 자바 컴파일러가 우리가 입력한 타입 정보를 기반으로 이런 코드가 있다고 가정하고 컴파일 과정에 타입 정보를 반영한다. 이 과정에서 타입이 맞지 않으면 컴파일 오류가 발생한다. 더 자세한 내용은 뒤에서 설명한다.

타입 추론

```
GenericBox<Integer> integerBox = new GenericBox<Integer>() // 타입 직접 입력
GenericBox<Integer> integerBox2 = new GenericBox<>() // 타입 추론
```

첫번째 줄의 코드를 보면 변수를 선언할 때와 객체를 생성할 때 `<Integer>`가 두 번 나온다. 자바는 왼쪽에 있는 변수를 선언할 때의 `<Integer>`를 보고 오른쪽에 있는 객체를 생성할 때 필요한 타입 정보를 얻을 수 있다. 따라서 두 번째 줄의 오른쪽 코드 `new GenericBox<>()`와 같이 타입 정보를 생략할 수 있다. 이렇게 자바가 스스로 타입 정보를 추론해서 개발자가 타입 정보를 생략할 수 있는 것을 타입 추론이라 한다.
참고로 타입 추론이 그냥 되는 것은 아니고, 자바 컴파일러가 타입을 추론할 수 있는 상황에만 가능하다. 쉽게 이야기해서 읽을 수 있는 타입 정보가 주변에 있어야 추론할 수 있다.

정리
제네릭을 사용한 덕분에 코드 재사용과 타입 안전성이라는 두 마리 토끼를 모두 잡을 수 있었다.

제네릭 용어와 관례

제네릭의 핵심은 사용할 타입을 미리 결정하지 않는다는 것.
클래스 내부에서 사용하는 타입을 클래스를 정의하는 시점에 결정하는 것이 아니라 실제 사용하는 생성 시점에 타입을 결정하는 것.

메서드와 매개변수랑 같은 관계

```
public String method1(){
    return "hello";
} // hello만 출력 가능 재사용성이 떨어짐.
public String method2(String param){
```

```
    return param;
} // 실행시점에 넘어온 param에 따라 어떤 값이든 출력 가능.
```

제네릭의 타입 매개변수와 타입 인자

메서드의 매개변수는 사용할 값에 대한 결정을 나중으로 미루는 것.
제네릭의 타입 매개변수는 사용할 타입에 대한 결정을 나중으로 미루는 것.

1. 메서드는 매개변수에 인자를 전달해서 사용할 값을 결정
2. 제네릭 클래스는 타입 매개변수에 타입 인자를 전달해서 사용할 타입을 결정

용어 정리

제네릭(Generic)

- 일반적인, 범용적인
- 특정 타입에 속한 것이 아닌 일반적으로, 범용적으로 사용할 수 있음

제네릭 타입(Generic Type)

- 클래스나 인터페이스를 정의할 때 타입 매개변수를 사용하는 것.
- 제네릭 클래스, 제네릭 인터페이스를 모두 합쳐서 제네릭 타입

타입 매개변수(Type Parameter)

- 제네릭 타입이나 메서드에 사용되는 변수, 실제 타입으로 대체
- GenericBox<T>
- T가 타입 매개변수

타입 인자(Type Argument)

- 제네릭 타입을 사용할 때 제공되는 실제 타입
- GenericBox<Integer>
- Integer가 타입 인자

제네릭 명명 관례

타입 매개변수는 일반적인 변수명처럼 소문자로 사용해도 문제는 없다.
하지만 일반적으로 대문자를 사용하고 용도에 맞는 단어의 첫글자를 사용하는 관례를 따른다.

주로 사용하는 키워드는 다음과 같다.

- E - Element
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

로우 타입

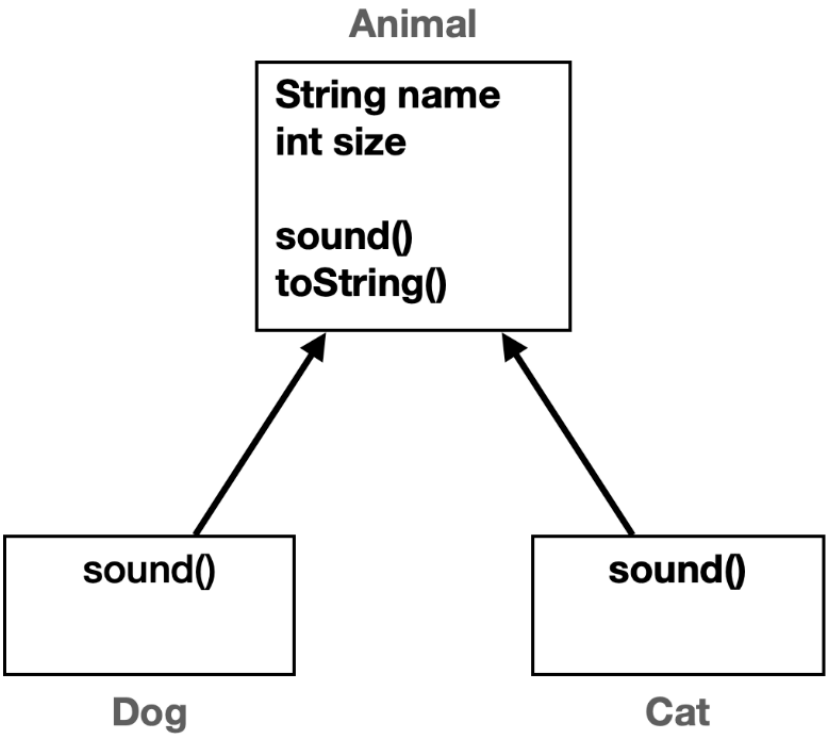
```
package generic.ex1;

public class RowTypeMain {
    public static void main(String[] args) {
        GenericBox<Object> integerBox = new GenericBox<>(); /
//        GenericBox integerBox = new GenericBox(); // 권장하지
    }
}
```

제네릭이 자바 처음시작 때 부터 존재하던 기능이 아닌, 추 후 등장했기에 제네릭이 없던 시절과 호환으로 인해 Row타입 존재
자바 버전문제로 인해, 사용해야 라이브러리는 <>기호를 포함하나, 내 프로젝트가 이보다 하위라 <>를 사용할 수 없는 버전일 수 있기 때문.

제네릭 활용 예제

이번에는 직접 클래스를 만들고, 제네릭도 도입해보자.
지금부터 사용할 `Animal` 관련 클래스들은 이후 예제에서도 사용하므로 `generic.animal`이라는 별도의 패키지에서 관리하겠다.




```
package generic.ex2;

import generic.animal.Animal;
import generic.animal.Cat;
import generic.animal.Dog;

public class AnimalMain1 {
    public static void main(String[] args) {
        Animal animal = new Animal("동물", 0);
        Dog dog = new Dog("멍멍이", 100);
        Cat cat = new Cat("야옹이", 50);

        Box<Dog> dogBox = new Box<>();
        dogBox.set(dog);
//        dogBox.set(cat); // 컴파일 에러
        Dog findDog = dogBox.get();
        System.out.println("findDog = " + findDog);
```

```
        Box<Cat> catBox = new Box<>();
        catBox.set(cat);
//        catBox.set(dog); // 컴파일 에러
        Cat findCat = catBox.get();
        System.out.println("findCat = " + findCat);

        Box<Animal> animalBox = new Box<>();

        animalBox.set(dog); // 자식 타입 모두 가능
        animalBox.get().sound();
        animalBox.set(cat); // 자식 타입 모두 가능
        animalBox.get().sound();

        animalBox.set(animal);
        Animal findAnimal = animalBox.get();
        System.out.println("findAnimal = " + findAnimal);
    }
}
```