11-04[Java - Generic - 심화]

소유자	종수 김
태그	

제네릭 - 심화

타입 매개변수 제한 - 시작

이번에는 동물 병원을 만들어보자.

요구사항: 개 병원은 개만 받을 수 있고, 고양이 병원은 고양이만 받을 수 있어야 한다.

동물 병원 예제를 통해 타입 매개변수를 제한해야하는 이유

```
package generic.ex3;

import generic.animal.Dog;

public class DogHospital {
    private Dog animal;

    public Dog get() {
        return animal;
    }

    public void set(Dog animal) {
        this.animal = animal;
    }

    public void checkUp() {
        System.out.println("동물 이름 : " + animal.getName()
        System.out.println("동물 크기 : " + animal.getSize()
        animal.sound();
```

```
    }

    public Dog bigger(Dog target) {
        return animal.getSize() > target.getSize() ? animal
    }
}


package generic.ex3;

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

public class CatHospital {
    private Cat animal;

    public Cat get() {
        return animal;
    }

    public void set(Cat animal) {
        this.animal = animal;
    }

    public void checkUp() {
        System.out.println("동물 이름 : " + animal.getName()
        System.out.println("동물 크기 : " + animal.getSize()
        animal.sound();
    }

    public Cat bigger(Cat target) {
        return animal.getSize() > target.getSize() ? animal
    }
}


package generic.ex3;
```

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

public class AnimalHospitalMain {
    public static void main(String[] args) {
        DogHospital dogHospital = new DogHospital();
        CatHospital catHospital = new CatHospital();

        Dog dog = new Dog("멍멍이1", 100);
        Cat cat = new Cat("야옹이1", 300);

        // 개 병원
        dogHospital.set(dog);
        dogHospital.checkUp();

        // 고양이 병원
        catHospital.set(cat);
        catHospital.checkUp();

        // 문제 1. 개 병원에 고양이 전달
//        dogHospital.set(cat); // 다른 타입 입력 : 컴파일 오류

        // 문제 2. 개 타입 반환
        dogHospital.set(dog);
        Dog 멍멍이2 = dogHospital.bigger(new Dog("멍멍이2", 2
        System.out.println("멍멍이2 = " + 멍멍이2);
    }
}
```

이번에 만든 코드는 처음에 제시한 다음 요구사항을 명확히 잘 지킨다.
요구사항: 개 병원은 개만 받을 수 있고, 고양이 병원은 고양이만 받을 수 있어야 한다.

여기서는 개 병원과 고양이 병원을 각각 별도의 클래스로 만들었다.
각 클래스 별로 타입이 명확하기 때문에 개 병원은 개만 받을 수 있고, 고양이 병원은 고양이만 받을 수 있다. 따라서 개 병원에 고양이를 전달하면 컴파일 오류가 발생한다.
그리고 개 병원에서 `bigger()`로 다른 개를 비교하는 경우 더 큰 개를 `Dog` 타입으로 반환한다.

문제

- 코드 재사용X: 개 병원과 고양이 병원은 중복이 많이 보인다.
- 타입 안전성O: 타입 안전성이 명확하게 지켜진다.

타입 매개변수 제한2 - 다형성

Dog, Cat은 Animal이라는 명확한 부모 타입이 있으므로 다형성을 이용하여 시도.

```
package generic.ex3;

import generic.animal.Animal;

public class AnimalHospitalV1 {
    private Animal animal;

    public Animal getAnimal() {
        return animal;
    }

    public void setAnimal(Animal animal) {
        this.animal = animal;
    }

    public void checkUp() {
        System.out.println("동물 이름 : " + animal.getName());
        System.out.println("동물 크기 : " + animal.getSize());
        animal.sound();
    }

    public Animal bigger(Animal target) {
```

```
        return animal.getSize() > target.getSize() ? animal :
    }
}

package generic.ex3;

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

public class AnimalHospitalMainV1 {
    public static void main(String[] args) {
        AnimalHospitalV1 dogHospital = new AnimalHospitalV1()
        AnimalHospitalV1 catHospital = new AnimalHospitalV1()

        Dog dog = new Dog("멍멍이1", 100);
        Cat cat = new Cat("야옹이1", 300);

        // 개 병원
        dogHospital.setAnimal(dog);
        dogHospital.checkUp();

        // 고양이 병원
        catHospital.setAnimal(cat);
        catHospital.checkUp();

        // 문제 1. 개 병원에 고양이 전달
        dogHospital.setAnimal(cat); // 매개변수 체크 실패 : 컴파일

        // 문제 2. 개 타입 반환
        dogHospital.setAnimal(dog);
//        Dog 멍멍이2 = dogHospital.bigger(new Dog("멍멍이2", 20
//        System.out.println("멍멍이2 = " + 멍멍이2);
    }
}
```

- `Animal` 타입을 받아서 처리한다.
- `checkup()`, `getBigger()`에서 사용하는 `animal.getName()`, `animal.getSize()`, `animal.sound()` 메서드는 모두 `Animal` 타입이 제공하는 메서드이다. 따라서 아무 문제없이 모두 호출할 수 있다.

문제

- 코드 재사용O: 다형성을 통해 `AnimalHospitalV1` 하나로 개와 고양이를 모두 처리한다.
- 타입 안전성X
  - 개 병원에 고양이를 전달하는 문제가 발생한다.
  - `Animal` 타입을 반환하기 때문에 다운 캐스팅을 해야 한다.
  - 실수로 고양이를 입력했는데, 개를 반환하는 상황이라면 캐스팅 예외가 발생한다.

타입 매개변수 제한 3 - 제네릭 도입과 실패

제네릭을 도입해서 코드 재사용, 타입 안전성 모두 체크

```
package generic.ex3;


public class AnimalHospitalV2<T> {
    private T animal;

    public T getAnimal() {
        return animal;
    }

    public void setAnimal(T animal) {
        this.animal = animal;
    }

    public void checkUp() {
        // T의 타입을 메서드를 정의하는 시점에는 알 수 없음. Object의 
        System.out.println("동물 이름 : " + animal.getName());
        System.out.println("동물 크기 : " + animal.getSize());
        animal.sound(); // 컴파일 오류
    }
```

```
    public T bigger(T target) {
        return animal.getSize() > target.getSize() ? animal :
    }
}
```

제네릭 타입을 선언하면 자바 컴파일러 입장에서 `T`에 어떤 값이 들어올지 예측할 수 없다. 우리는 `Animal` 타입의 자식이 들어오기를 기대했지만, 여기 코드 어디에도 `Animal`에 대한 정보는 없다. `T`에는 타입 인자로 `Integer`가 들어올 수 도 있고, `Dog`가 들어올 수도 있다. 물론 `Object`가 들어올 수도 있다.

다양한 타입 인자

```
AnimalHospitalV2<Dog> dogHospital = new AnimalHospitalV2<>();
AnimalHospitalV2<Cat> catHospital = new AnimalHospitalV2<>();
AnimalHospitalV2<Integer> integerHospital = new AnimalHospitalV2<>();
AnimalHospitalV2<Object> objectHospital = new AnimalHospitalV2<>();
```

자바 컴파일러는 어떤 타입이 들어올 지 알 수 없기 때문에 `T`를 어떤 타입이든 받을 수 있는 모든 객체의 최종 부모인 `Object` 타입으로 가정한다. 따라서 `Object`가 제공하는 메서드만 호출할 수 있다.
원하는 기능을 사용하려면 `Animal` 타입이 제공하는 기능들이 필요한데, 이 기능을 모두 사용할 수 없다.

여기에 추가로 한가지 문제가 더 있다. 바로 동물 병원에 `Integer`, `Object` 같은 동물과 전혀 관계 없는 타입을 타입 인자로 전달 할 수 있다는 점이다. 우리는 최소한 `Animal`이나 그 자식을 타입 인자로 제한하고 싶다.

문제

1. 제네릭에서 타입 매개변수를 사용하면 어떤 타입이든 들어올 수 있다.
2. 타입 매개변수를 어떤 타입이든 수용할 수 있는 Object로 가정하고, Object의 기능만 사용할 수 있음.

- 타입 매개변수를 Animal, Animal의 자식으로 제한할 수 있으면 해결

타입 매개변수 제한 4 - 타입 매개변수 제한

타입 매개변수를 특정 타입으로 제한할 수 있음.

```
package generic.ex3;
```

```
import generic.animal.Animal;

public class AnimalHospitalV3<T extends Animal> { // extends
// 이를 통해 Animal의 기능을 사용 가능
    private T animal;

    public T getAnimal() {
        return animal;
    }

    public void setAnimal(T animal) {
        this.animal = animal;
    }

    public void checkUp() {
        System.out.println("동물 이름 : " + animal.getName());
        System.out.println("동물 크기 : " + animal.getSize());
        animal.sound(); // 컴파일 오류가 나지 않음
    }

    public T bigger(T target) {
        return animal.getSize() > target.getSize() ? animal :
    }
}

package generic.ex3;

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

public class AnimalHospitalMainV3 {
    public static void main(String[] args) {
        AnimalHospitalV3<Dog> dogHospital = new AnimalHospita
        AnimalHospitalV3<Cat> catHospital = new AnimalHospita

        Dog dog = new Dog("멍멍이1", 100);
        Cat cat = new Cat("야옹이1", 300);
```

```
        // 개 병원
        dogHospital.setAnimal(dog);
        dogHospital.checkUp();

        // 고양이 병원
        catHospital.setAnimal(cat);
        catHospital.checkUp();

        // 문제 1. 개 병원에 고양이 전달
//        dogHospital.setAnimal(cat); // 다른 타입 입력으로 인해 컴

        // 문제 2. 개 타입 반환
        dogHospital.setAnimal(dog);
        Dog 멍멍이2 = dogHospital.bigger(new Dog("멍멍이2", 200)
        System.out.println("멍멍이2 = " + 멍멍이2);
    }
}
```

여기서 핵심은 `<T extends Animal>` 이다.
타입 매개변수 `T`를 `Animal`과 그 자식만 받을 수 있도록 제한을 두는 것이다. 즉 `T`의 상한이 `Animal`이 되는 것이다.

이렇게 하면 타입 인자로 들어올 수 있는 값이 `Animal`과 그 자식으로 제한된다.

```
AnimalHospitalV3<Animal>
AnimalHospitalV3<Dog>
AnimalHospitalV3<Cat>
```

이제 자바 컴파일러는 `T`에 입력될 수 있는 값의 범위를 예측할 수 있다.
타입 매개변수 `T`에는 타입 인자로 `Animal`, `Dog`, `Cat`만 들어올 수 있다. 따라서 이를 모두 수용할 수 있는 `Animal`을 `T`의 타입으로 가정해도 문제가 없다.
따라서 `Animal`이 제공하는 `getName()`, `getSize()` 같은 기능을 사용할 수 있다.

타입 매개변수에 입력될 수 있는 상한을 지정해서 문제를 해결했다.

- `AnimalHospitalV3<Integer>`와 같이 동물과 전혀 관계없는 타입 인자를 컴파일 시점에 막는다.
- 제네릭 클래스 안에서 `Animal`의 기능을 사용할 수 있다.

기존 문제와 해결

1. 타입 안전성 문제
    a. 개 병원에서는 고양이를 받을 수 없음 → 해결
    b. Animal 타입을 반환하기 때문에 다운 캐스팅을 해야함 → 해결
    c. 실수로 고양이를 입력했는데, 개를 반환하는 상황이라면 캐스팅 예외 발생 → 해결
2. 제네릭 도입 문제
    a. 제네릭에서 타입 매개변수를 사용하면 어떤 타입이든 들어올 수 있다. → 해결
    b. 어떤 타입이든 수용할 수 있는 Object로 가정하고 Object의 기능만 사용할 수 있다 → 해결

정리

제네릭에 타입 매개변수 상한을 사용해서 타입 안전성을 지키면서 상위 타입의 원하는 기능까지 사용할 수 있음.
코드 재사용, 타입 안전성 둘 다 해결 가능.

제네릭 메서드

특정 메서드에 제네릭을 적용하는 제네릭 메서드
제네릭 타입과 제네릭 메서드는 둘다 제네릭을 사용하는 서로 다른 기능 제공

```
package generic.ex4;

public class GenericMethod {
    public static Object objMethod(Object obj){
        System.out.println("Object print : " + obj);
        return obj;
    }

    public static <T> T genericMethod(T obj){ // <T>를 통해 제ㄴ
        System.out.println("Object print : " + obj);
        return obj;
    }

    public static <T extends Number> T numberMethod(T t){ //
        System.out.println("bound print" + t);
        return t;
```

```
    }

}


package generic.ex4;

public class MethodMain1 {
    public static void main(String[] args) {
        Integer i = 10;
        GenericMethod.objMethod(i);
        Integer o = (Integer) GenericMethod.objMethod(i); // (

        // 타입 인자 (Type Argument) 명시적 전달
        System.out.println("명시적 타입 인자 전달");
        Integer integer = GenericMethod.<Integer>genericMetho
        Integer integer1 = GenericMethod.<Integer>numberMetho
        Double v = GenericMethod.<Double>genericMethod(10.0);

        System.out.println("o = " + o);
        System.out.println("integer = " + integer);
        System.out.println("integer1 = " + integer1);
        System.out.println("v = " + v);

    }
}
```

- 제네릭 메서드는 클래스 전체가 아니라 특정 메서드 단위로 제네릭을 도입할 때 사용한다.
- 제네릭 메서드를 정의할 때는 메서드의 반환 타입 왼쪽에 다이아몬드를 사용해서 `<T>`와 같이 타입 매개변수를 적어준다.
- 제네릭 메서드는 메서드를 실제 호출하는 시점에 다이아몬드를 사용해서 `<Integer>`와 같이 타입을 정하고 호출한다.

제네릭 메서드의 핵심은 메서드를 호출하는 시점에 타입 인자를 전달해서 타입을 지정하는 것이다. 따라서 타입을 지정하면서 메서드를 호출한다.

- 제네릭 타입

    - 정의 : GenericClass<T>
    - 타입 인자 전달 : 객체를 생성하는 시점
- 제네릭 메서드
    - 정의 : <T> T genericMethod(T t)
    - 타입 인자 전달 : 메서드를 호출하는 시점

제네릭 메서드의 핵심은 메서드를 호출하는 시점에 타입 인자를 전달해서 타입을 지정하는 것.
따라서 타입을 지정하면서 메서드를 호출

인스턴스 메서드, static 메서드

제네릭 메서드는 인스턴스 메서드와 static 메서드에 모두 적용할 수 있다.

```
class Box<T> { //제네릭 타입
    static <V> V staticMethod2(V t) {} //static 메서드에 제네릭 메서드 도입
    <Z> Z instanceMethod2(Z z) {} //인스턴스 메서드에 제네릭 메서드 도입 가능
}
```

참고

제네릭 타입은 static 메서드에 타입 매개변수를 사용할 수 없다. 제네릭 타입은 객체를 생성하는 시점에 타입이 정해진다. 그런데 static 메서드는 인스턴스 단위가 아니라 클래스 단위로 작동하기 때문에 제네릭 타입과는 무관하다.
따라서 static 메서드에 제네릭을 도입하려면 제네릭 메서드를 사용해야 한다.

```
class Box<T> {
    T instanceMethod(T t) {} //가능
    static T staticMethod1(T t) {} //제네릭 타입의 T 사용 불가능
}
```

타입 매개변수 제한

제네릭 메서드도 제네릭 타입과 마찬가지로 타입 매개변수를 제한 가능.

```
public static <T extends Number> T numberMethod(T t) {}
```

```
//GenericMethod.numberMethod("Hello"); // 컴파일 오류 Number의 자식만 입력 가능
```

제네릭 메서드 타입 추론

제네릭 메서드를 호출할 때 `<Integer>`와 같이 타입 인자를 계속 전달하는 것은 매우 불편하다.

```
Integer i = 10;
Integer result = GenericMethod.<Integer>genericMethod(i);
```

자바 컴파일러는 `genericMethod()`에 전달되는 인자 `i`의 타입이 `Integer`라는 것을 알 수 있다.
또한 반환 타입이 `Integer result`라는 것도 알 수 있다. 이런 정보를 통해 자바 컴파일러는 타입 인자를 추론할 수 있다.

앞서 만든 `MethodMain1`에 다음 코드를 추가해서 실행해보자.

```
//타입 추론, 타입 인자 생략
System.out.println("타입 추론");
Integer result2 = GenericMethod.genericMethod(i);
Integer integerValue2 = GenericMethod.numberMethod(10);
Double doubleValue2 = GenericMethod.numberMethod(20.0);
```

제네릭 메서드 활용

제네릭 클래스였던 AnimalHospital을 제네릭 메서드로 리팩토링

```
package generic.ex4;

import generic.animal.Animal;

public class AnimalMethod {
    public static <T extends Animal> void checkUp(T animal) {
        System.out.println("동물 이름 : " + animal.getName());
        System.out.println("동물 크기 : " + animal.getSize());
        animal.sound(); // 컴파일 오류
    }

    public static <T extends Animal> T bigger(T animal, T tar
        return animal.getSize() > target.getSize() ? animal :
    }
}

package generic.ex4;
```

```
import generic.animal.Cat;
import generic.animal.Dog;

public class MethodMain2 {
    public static void main(String[] args) {
        Dog dog = new Dog("멍멍이", 100);
        Cat cat = new Cat("야옹이", 100);

        AnimalMethod.<Dog>checkUp(dog); // 타입 매개변수 명시
        AnimalMethod.checkUp(cat); // 타입 매개변수 추론

        Dog targetDog = new Dog("큰 멍멍이", 200);
        Dog bigger = AnimalMethod.bigger(dog, targetDog);
        System.out.println("bigger = " + bigger);

    }
}
```

제네릭 타입과 제네릭 메서드의 우선순위

```
package generic.ex4;

import generic.animal.Animal;

public class ComplexBox <T extends Animal> {
    private T animal;

    public void set(T animal) {
        this.animal = animal;
    }

    public <T> T printAndReturn(T z) {
        System.out.println("animal.className" + animal.getCla
        System.out.println("z.className" + z.getClass().getNa
        //        z.sound(); // Object이므로 Animal의 sound()메
        return z;
    }
}
```

```
package generic.ex4;

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

public class MethodMain3 {
    public static void main(String[] args) {
        Dog dog = new Dog("멍멍이", 100);
        Cat cat = new Cat("야옹이", 50);

        ComplexBox<Dog> hospital = new ComplexBox<>();
        hospital.set(dog);

        Cat returnCat = hospital.printAndReturn(cat);
        System.out.println("returnCat = " + returnCat);
    }
}
```

제네릭 타입 설정

`class ComplexBox<T extends Animal>`

제네릭 메서드 설정

`<T> T printAndReturn(T t)`

제네릭 타입보다 제네릭 메서드가 높은 우선순위를 가진다.
따라서 `printAndReturn()`은 제네릭 타입과는 무관하고 제네릭 메서드가 적용된다.

여기서 적용된 제네릭 메서드의 타입 매개변수 `T`는 상한이 없다. 따라서 `Object`로 취급된다.
`Object`로 취급되기 때문에 `t.getName()`과 같은 `Animal`에 존재하는 메서드는 호출할 수 없다.

참고로 프로그래밍에서 이렇게 모호한 것은 좋지 않다.
둘의 이름이 겹치면 다음과 같이 둘 중 하나를 다른 이름으로 변경하는 것이 좋다.

와일드카드 - 1

제네릭 타입을 조금 더 편리하게 사용할 수 있는 방법
컴퓨터 프로그래밍에서 *, ? 와 같이 하나 이상의 문자들을 상징하는 특수 문자.

```
package generic.ex5;

import generic.animal.Animal;

public class WildcardEx {
    static <T> void printGenericV1(Box<T> box) {
        System.out.println("T = " + box.get());
    }

    // Box<Dog>, Box<Cat>, Box<Object> 같은 모든 타입이 들어갈 수
    static void printWildcardV1(Box<?> box) {
        System.out.println(" ? = " + box.get());
    }

    static <T extends Animal> void printGenericV2(Box<T> box)
        T t = box.get();
        System.out.println("이름 = " + t.getName());
    }

    static void printWildcardV2(Box<? extends Animal> box) {
        Animal animal = box.get();
        System.out.println("이름 = " + animal.getName());
    }

    static <T extends Animal> T printAndReturn(Box<T> box) {
        T t = box.get();
        System.out.println("이름 = " + t.getName());
        return t;
    }

    static Animal printAndReturnWildcard(Box<? extends Animal
        Animal t = box.get();
        System.out.println("이름 = " + t.getName());
        return t;
    }
```

```
}

package generic.ex5;

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

public class WildcardMain1 {
    public static void main(String[] args) {
        Box<Object> objBox = new Box<>();
        Box<Dog> dogBox = new Box<>();
        Box<Cat> catBox = new Box<>();

        dogBox.set(new Dog("멍멍이", 100));
        WildcardEx.printGenericV1(dogBox);
        WildcardEx.printWildcardV1(dogBox);

        WildcardEx.printGenericV2(dogBox);
        WildcardEx.printWildcardV2(dogBox);

        Dog dog = WildcardEx.printAndReturn(dogBox); // Dog타입
        Animal animal = WildcardEx.printAndReturnWildcard(dogB
    }
}
```

- 제네릭 메서드와 와일드 카드를 비교할 수 있게 같은 기능을 각각 하나씩 배치
- 와일드카드는 ? 를 사용해서 정의

와일드 카드는 제네릭 타입이나, 제네릭 메서드를 선언하는 것이 아닌 이미 만들어진 제네릭 타입을 활용할 때 사용

```java
//이것은 제네릭 메서드이다.
//Box<Dog> dogBox를 전달한다. 타입 추론에 의해 타입 T가 Dog가 된다.
static <T> void printGenericV1(Box<T> box) {
    System.out.println("T = " + box.get());
}

//이것은 제네릭 메서드가 아니다. 일반적인 메서드이다.
//Box<Dog> dogBox를 전달한다. 와일드카드 ?는 모든 타입을 받을 수 있다.
static void printWildcardV1(Box<?> box) {
    System.out.println("? = " + box.get());
}
```

- 두 메서드는 비슷한 기능을 하는 코드이다. 하나는 제네릭 메서드를 사용하고 하나는 일반적인 메서드에 와일드카드를 사용했다.
- 와일드카드는 제네릭 타입이나 제네릭 메서드를 정의할 때 사용하는 것이 아니다. `Box<Dog>`, `Box<Cat>`처럼 타입 인자가 정해진 제네릭 타입을 전달 받아서 활용할 때 사용한다.
- 와일드카드인 `?`는 모든 타입을 다 받을 수 있다는 뜻이다.
  - 다음과 같이 해석할 수 있다. `? == <? extends Object>`
- 이렇게 `?`만 사용해서 제한 없이 모든 타입을 다 받을 수 있는 와일드카드를 비제한 와일드카드라 한다.
  - 여기에는 `Box<Dog> dogBox`, `Box<Cat> catBox`, `Box<Object> objBox`가 모두 입력될 수 있

제네릭 메서드와 와일드카드 실행 예시

제네릭 메서드 실행 예시

```
//1. 전달
printGenericV1(dogBox)

//2. 제네릭 타입 결정 dogBox는 Box<Dog> 타입, 타입 추론 -> T의 타입은 Dog
static <T> void printGenericV1(Box<T> box) {
    System.out.println("T = " + box.get());
}

//3. 타입 인자 결정
static <Dog> void printGenericV1(Box<Dog> box) {
    System.out.println("T = " + box.get());
}

//4. 최종 실행 메서드
static void printGenericV1(Box<Dog> box) {
    System.out.println("T = " + box.get());
}
```

와일드 카드 실행 예시

```
//1. 전달
printWildcardV1(dogBox)

//이것은 제네릭 메서드가 아니다. 일반적인 메서드이다.
//2. 최종 실행 메서드, 와일드카드 ?는 모든 타입을 받을 수 있다.
static void printWildcardV1(Box<?> box) {
    System.out.println("? = " + box.get());
}
```

제네릭 메서드 vs 와일드 카드

printGenricV1()은 타입 매개변수가 존재하므로 특정 시점에 매개변수에 타입 인자를 전달해서 타입을 결정해야함. 이런 과정은 매우 복잡

반면에 printWildcardV1()은 일반적인 메서드에 사용할 수 있고, 단순히 매개변수로 제네릭 타입을 받을 수 있는것 뿐임. 제네릭 메서드처럼 타입을 결정하거나 복잡하게 작동하지 않고 단순히 일반 메서드에 제네릭 타입을 받을 수 있는 매개변수가 하나 있는 것 뿐

- 제네릭 타입이나 제네릭 메서드를 정의하는게 꼭 필요한 상황이 아니라면 단순한 와일드카드 사용 권장

와일드카드 2

상한 와일드카드

```
static <T extends Animal> void printGenericV2(Box<T> box) {
    T t = box.get();
    System.out.println("이름 = " + t.getName());
}

static void printWildcardV2(Box<? extends Animal> box) {
    Animal animal = box.get();
    System.out.println("이름 = " + animal.getName());
}
```

- 제네릭 메서드와 마찬가지로 와일드카드에도 상한 제한을 둘 수 있다.
- 여기서는 `? extends Animal`을 지정했다.
- `Animal`과 그 하위 타입만 입력 받는다. 만약 다른 타입을 입력하면 컴파일 오류가 발생한다.
- `box.get()`을 통해서 꺼낼 수 있는 타입의 최대 부모는 `Animal`이 된다. 따라서 `Animal` 타입으로 조회할 수 있다.
- 결과적으로 `Animal` 타입의 기능을 호출할 수 있다.

타입 매개변수가 꼭 필요한 경우

위의 예시만 봤을 때는 제네릭 메서드가 필요한지 의문이 듦.

와일드카드는 제네릭을 정의할 때 사용하는 것이 아님. Box<Dog>, Box<Cat> 처럼 타입 인자가 전달된 제네릭 타입을 활용할 때 사용

타입 매개변수가 꼭 필요한 경우

와일드카드는 제네릭을 정의할 때 사용하는 것이 아니다. `Box<Dog>`, `Box<Cat>` 처럼 타입 인자가 전달된 제네릭 타입을 활용할 때 사용한다. 따라서 다음과 같은 경우에는 제네릭 타입이나 제네릭 메서드를 사용해야 문제를 해결할 수 있다.

```
static <T extends Animal> T printAndReturnGeneric(Box<T> box) {
    T t = box.get();
    System.out.println("이름 = " + t.getName());
    return t;
}
```

```
static Animal printAndReturnWildcard(Box<? extends Animal> box) {
    Animal animal = box.get();
    System.out.println("이름 = " + animal.getName());
    return animal;
}
```

`printAndReturnGeneric()` 은 다음과 같이 전달한 타입을 명확하게 반환할 수 있다.

```
Dog dog = WildcardEx.printAndReturnGeneric(dogBox)
```

반면에 `printAndReturnWildcard()` 의 경우 전달한 타입을 명확하게 반환할 수 없다. 여기서는 `Animal` 타입으로 반환한다.

```
Animal animal = WildcardEx.printAndReturnWildcard(dogBox)
```

메서드의 타입들을 특정 시점에 변경하려면 제네릭 타입이나, 제네릭 메서드를 사용해야 한다.
와일드카드는 이미 만들어진 제네릭 타입을 전달 받아서 활용할 때 사용한다. 따라서 메서드의 타입들을 타입 인자를 통해 변경할 수 없다. 쉽게 이야기해서 일반적인 메서드에 사용한다고 생각하면 된다.

정리하면 제네릭 타입이나 제네릭 메서드가 꼭 필요한 상황이면 `<T>` 를 사용하고, 그렇지 않은 상황이면 와일드카드를 사용하는 것을 권장한다.

와일드카드는 이미 만들어진 제네릭을 단순 사용하는 것이기 때문에 반환타입을 동적으로 바꾸거나, 타입 매개변수를 정하는 등의 작업이 불가능한 것.

이럴 때는 제네릭 메서드를 사용해야함.

- 제네릭 타입을 리턴받거나 해야한다면 제네릭 <T>를 사용하고 그렇지 않으면 와일드카드 사용 권장

하한 와일드 카드

최소한 super를 기준으로 상위 클래스를 넣어야함.

```
package generic.ex5;

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

public class WildcardMain2 {
    public static void main(String[] args) {
        Box<Object> objBox = new Box<>();
        Box<Animal> animalBox = new Box<>();
        Box<Dog> dogBox = new Box<>();
        Box<Cat> catBox = new Box<>();

        // Animal 포함 상위 타입 전달 가능
        writeBox(objBox);
        writeBox(animalBox);
//        writeBox(dogBox); // Animal 타입보다 하위(자식)이므로 컴
//        writeBox(catBox); // Animal 타입보다 하위(자식)이므로 컴
    }

    static void writeBox(Box<? super Animal> box) {
        box.set(new Dog("멍멍이100", 100));
    }
}
```

```
Box<? super Animal> box
```

이 코드는 `?`가 `Animal` 타입을 포함한 `Animal` 타입의 상위 타입만 입력 받을 수 있다는 뜻이다.

정리하면 다음과 같다.
`Box<Object> objBox`: 허용
`Box<Animal> animalBox`: 허용
`Box<Dog> dogBox`: 불가
`Box<Cat> catBox`: 불가

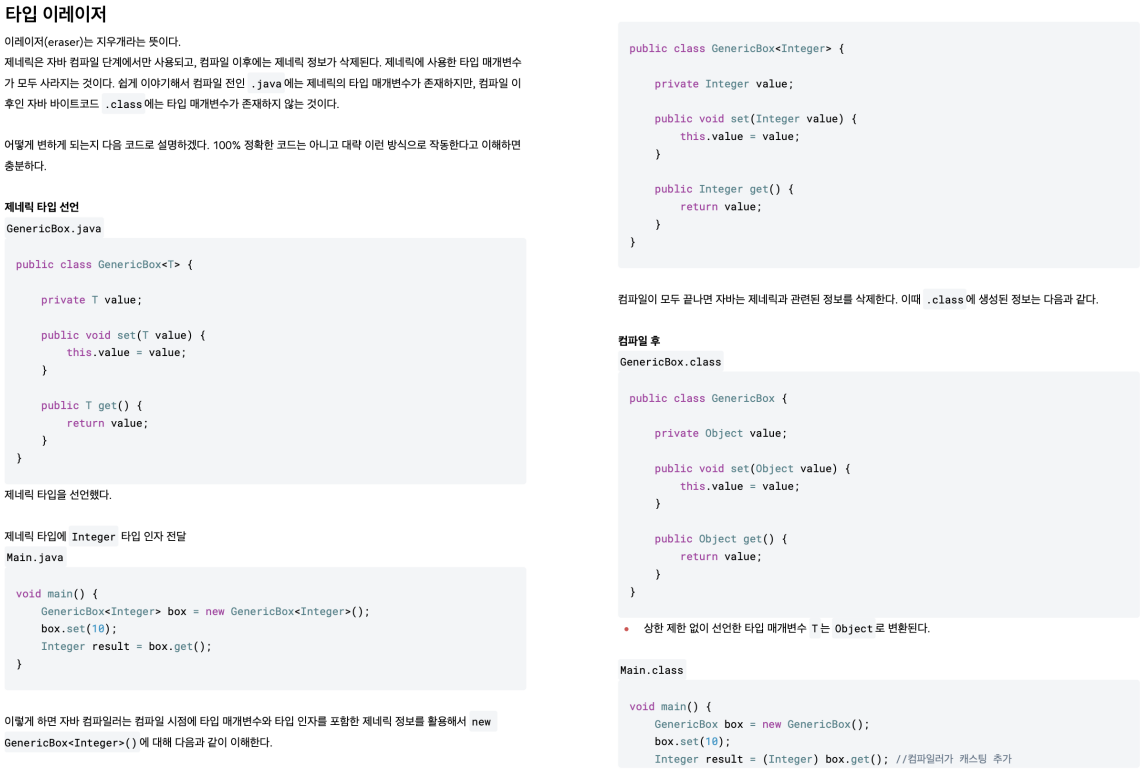
하한을 `Animal`로 제한했기 때문에 `Animal` 타입의 하위 타입인 `Box<Dog>`는 전달할 수 없다.

- 제네릭 타입, 제네릭 메서드에는 사용 불가능하고 와일드카드에만 사용 가능.

타입 이레이저

제네릭은 자바 컴파일 단계에서만 사용되고, 컴파일 이후에는 제네릭 정보가 삭제.
제네릭에 사용한 타입 매개변수가 모두 사라지는 것.

.java에는 제네릭 타입 매개변수가 존재하지만,
컴파일 이후인 자바 바이트코드 .class에는 타입 매개변수가 존재하지 않는 것.



타입 이레이저

이레이저(eraser)는 지우개라는 뜻이다.
제네릭은 자바 컴파일 단계에서만 사용되고, 컴파일 이후에는 제네릭 정보가 삭제된다. 제네릭에 사용한 타입 매개변수가 모두 사라지는 것이다. 쉽게 이야기해서 컴파일 전인 `.java`에는 제네릭의 타입 매개변수가 존재하지만, 컴파일 이후인 자바 바이트코드 `.class`에는 타입 매개변수가 존재하지 않는 것이다.

어떻게 변하게 되는지 다음 코드로 설명하겠다. 100% 정확한 코드는 아니고 대략 이런 방식으로 작동한다고 이해하면 충분하다.

제네릭 타입 선언
`GenericBox.java`

```
public class GenericBox<T> {

    private T value;

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

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

제네릭 타입을 선언했다.

제네릭 타입에 `Integer` 타입 인자 전달
`Main.java`

```
void main() {
    GenericBox<Integer> box = new GenericBox<Integer>();
    box.set(10);
    Integer result = box.get();
}
```

이렇게 하면 자바 컴파일러는 컴파일 시점에 타입 매개변수와 타입 인자를 포함한 제네릭 정보를 활용해서 `new GenericBox<Integer>()`에 대해 다음과 같이 이해한다.

```
public class GenericBox<Integer> {

    private Integer value;

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

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

컴파일이 모두 끝나면 자바는 제네릭과 관련된 정보를 삭제한다. 이때 `.class`에 생성된 정보는 다음과 같다.

컴파일 후
`GenericBox.class`

```
public class GenericBox {

    private Object value;

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

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

- 상한 제한 없이 선언한 타입 매개변수 `T`는 `Object`로 변환된다.

`Main.class`

```
void main() {
    GenericBox box = new GenericBox();
    box.set(10);
    Integer result = (Integer) box.get(); //컴파일러가 캐스팅 추가
```

- 컴파일 전에는 <Integer>라는 타입 매개변수로 넘어온 코드를 통해 컴파일을 진행하고
- 컴파일 후에는 <T>라는 상한 매개변수가 없는 일반 제네릭은 Object로 변환.
- 그 후 컴파일러가(Integer) 캐스팅 추가
- 컴파일 단계에서 Integer로 검증을 한 후 컴파일 후 Casting을 추가하므로 안정성이 검증.

타입 매개변수 제한

```
}
```

- 값을 반환 받는 부분을 `Object`로 받으면 안된다. 자바 컴파일러는 제네릭에서 타입 인자로 지정한 `Integer`로 캐스팅하는 코드를 추가해준다.
- 이렇게 추가된 코드는 자바 컴파일러가 이미 검증하고 추가했기 때문에 문제가 발생하지 않는다.

타입 매개변수 제한의 경우

다음과 같이 타입 매개변수를 제한하면 제한한 타입으로 코드를 변경한다.

컴파일 전

`AnimalHospitalV3.java`

```
public class AnimalHospitalV3<T extends Animal> {

    private T animal;

    public void set(T animal) {
        this.animal = animal;
    }

    public void checkup() {
        System.out.println("동물 이름: " + animal.getName());
        System.out.println("동물 크기: " + animal.getSize());
        animal.sound();
    }

    public T getBigger(T target) {
        return animal.getSize() > target.getSize() ? animal : target;
    }

}
```

```
//사용 코드 예시
AnimalHospitalV3<Dog> hospital = new AnimalHospitalV3<>();
...
Dog dog = animalHospitalV3.getBigger(new Dog());
```

컴파일 후

`AnimalHospitalV3.class`

```
public class AnimalHospitalV3 {

    private Animal animal;

    public void set(Animal animal) {
        this.animal = animal;
    }

    public void checkup() {
        System.out.println("동물 이름: " + animal.getName());
        System.out.println("동물 크기: " + animal.getSize());
        animal.sound();
    }

    public Animal getBigger(Animal target) {
        return animal.getSize() > target.getSize() ? animal : target;
    }

}
```

- `T`의 타입 정보가 제거되어도 상한으로 지정한 `Animal` 타입으로 대체되기 때문에 `Animal` 타입의 메서드를 사용하는데는 아무런 문제가 없다.

```
//사용 코드 예시
AnimalHospitalV3 hospital = new AnimalHospitalV3();
...
Dog dog = (Dog) animalHospitalV3.getBigger(new Dog());
```

- 반환 받는 부분을 `Animal`로 받으면 안되기 때문에 자바 컴파일러가 타입 인자로 지정한 `Dog`로 캐스팅하는 코드를 넣어준다.

자바의 제네릭은 단순하게 생각하면 개발자가 직접 캐스팅 하는 코드를 컴파일러가 대신 처리해주는 것이다. 자바는 컴파일 시점에 제네릭을 사용한 코드에 문제가 없는지 완벽하게 검증하기 때문에 자바 컴파일러가 추가하는 다운 캐스팅에는 문제가 발생하지 않는다.

자바의 제네릭 타입은 컴파일 시점에만 존재하고, 런타임 시에는 제네릭 정보가 지워지는데, 이것을 타입 이레이저라 한다.

- 컴파일 전에는 제한된 경우 상한이였던 부모 클래스로 변경해서 컴파일러가 검증을 진행.
- 컴파일 후에는 자바 컴파일러가 타입 인자로 지정한 구체적인 클래스로 캐스팅한 후 반환 값을 전달.

자바의 제네릭은 단순하게 생각하면 개발자가 직접 캐스팅 하는 코드를 컴파일러가 대신 처리해주는 것.
자바는 컴파일 시점에 제네릭을 사용한 코드가 문제가 없는지 완벽하게 검증하기에 컴파일러가 추가하는 다운 캐스팅에는 문제가 발생하지 않음.

자바의 제네릭 타입은 컴파일 시점에만 존재하고 런타임 시에는 제네릭 정보가 지워지는데 이것이 타입 이레이저

타입 이레이저 방식의 한계

컴파일 이후에는 제네릭의 타입 정보가 존재하지 않음.
.class로 자바를 실행하는 런타임에는 우리가 지정한 Box<Integer>, Box<String>의 타입 정보가 모두 제거

따라서 아래와 같은 코드는 작성이 불가능

```
package generic.ex5;

public class EraserBox<T> {
    public boolean instanceCheck(Object param) {
        return param instanceof T;
    }
    public void create () {
        return new T(); //
    }
}
```

- 여기서 `T`는 런타임에 모두 `Object`가 되어버린다.
- `instanceof`는 항상 `Object`와 비교하게 된다. 이렇게 되면 항상 참이반환되는 문제가 발생한다. 자바는 이런 문제 때문에 타입 매개변수에 `instanceof`를 허용하지 않는다.
- `new T`는 항상 `new Object`가 되어버린다. 개발자가 의도한 것과는 다르다. 따라서 자바는 타입 매개변수에

`new`를 허용하지 않는다.