# 2. 제네릭 - Generic2

#1.인강/0.자바/4.자바-중급2편



## 타입 매개변수 제한1 - 시작

이번에는 동물 병원을 만들어보자.

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

> **강의 영상 수정 사항**
>
> 강의 영상에서는 실수로 패키지를 잘못 지정했습니다.
>
> - 영상의 패키지: `generic.test.ex3;`
> - 강의 메뉴얼의 패키지: `generic.ex3;`
> - **강의 메뉴얼의 패키지를 사용해주세요.**

```
package generic.ex3;

import generic.animal.Dog;

public class DogHospital {

    private Dog 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 : target;
    }
}
```

- 개 병원은 내부에 `Dog` 타입을 가진다.
- `checkup()` : 개의 이름과 크기를 출력하고, 개의 `sound()` 메서드를 호출한다.
- `bigger()` : 다른 개와 크기를 비교한다. 둘 중에 큰 개를 반환한다.

```
package generic.ex3;

import generic.animal.Cat;

public class CatHospital {

    private Cat 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 getBigger(Cat target) {
        return animal.getSize() > target.getSize() ? animal : target;
    }
}
```

- 고양이 병원은 내부에 `Cat` 타입을 가진다.

- `checkup()` : 고양이의 이름과 크기를 출력하고, 고양이의 `sound()` 메서드를 호출한다.
- `bigger()` : 다른 고양이와 크기를 비교한다. 둘 중에 큰 고양이를 반환한다.

```
package generic.ex3;

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

public class AnimalHospitalMainV0 {

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

        Dog dog = new Dog("멍멍이1", 100);
        Cat cat = new Cat("냐옹이1", 300);

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

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

        // 문제1: 개 병원에 고양이 전달
        // dogHospital.set(cat); // 다른 타입 입력: 컴파일 오류

        // 문제2: 개 타입 반환
        dogHospital.set(dog);
        Dog biggerDog = dogHospital.bigger(new Dog("멍멍이2", 200));
        System.out.println("biggerDog = " + biggerDog);
    }
}
```

**실행 결과**

```
동물 이름: 멍멍이1
동물 크기: 100
멍멍
```

```
동물 이름: 냐옹이1
동물 크기: 300
냐옹
biggerDog = Animal{name='멍멍이2', size=200}
```

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 시도

`Dog`, `Cat`은 `Animal`이라는 명확한 부모 타입이 있다. 다형성을 사용해서 중복을 제거해보자.

```
package generic.ex3;

import generic.animal.Animal;

public class AnimalHospitalV1 {

    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;
    }
}
```

- `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.

```
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("냐옹이1", 300);

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

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

        // 문제1: 개 병원에 고양이 전달
        dogHospital.set(cat); // 매개변수 체크 실패: 컴파일 오류가 발생하지 않음

        // 문제2: 개 타입 반환, 캐스팅 필요
        dogHospital.set(dog);
```

```
        Dog biggerDog = (Dog) dogHospital.getBigger(new Dog("멍멍이2", 200));
        System.out.println("biggerDog = " + biggerDog);
    }
}
```

**실행 결과**

```
동물 이름: 멍멍이1
동물 크기: 100
멍멍
동물 이름: 냐옹이1
동물 크기: 300
냐옹
biggerDog = Animal{name='멍멍이2', size=200}
```

**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 제한3 - 제네릭 도입과 실패

이제 앞서 배운 제네릭을 도입해서 코드 재사용은 늘리고, 타입 안전성 문제도 해결해보자.

```
package generic.ex3;

public class AnimalHospitalV2<T> {

    private T animal;

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

```
    public void checkup() {
        // T의 타입을 메서드를 정의하는 시점에는 알 수 없다. Object의 기능만 사용 가능
        animal.toString();
        animal.equals(null);

        // 컴파일 오류
        //System.out.println("동물 이름: " + animal.getName());
        //animal.sound();
    }

    public T getBigger(T target) {
        // 컴파일 오류
        //return animal.getSize() > target.getSize() ? animal : target;
        return null;
    }
}
```

- `<T>`를 사용해서 제네릭 타입을 선언했다.

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.

```
package generic.ex3;
```

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

public class AnimalHospitalMainV2 {

    public static void main(String[] args) {
        AnimalHospitalV2<Dog> dogHospital = new AnimalHospitalV2<>();
        AnimalHospitalV2<Cat> catHospital = new AnimalHospitalV2<>();
        AnimalHospitalV2<Integer> integerHospital = new AnimalHospitalV2<>();
        AnimalHospitalV2<Object> objectHospital = new AnimalHospitalV2<>();
    }
}
```

**문제**

- 제네릭에서 타입 매개변수를 사용하면 어떤 타입이든 들어올 수 있다.
- 따라서 타입 매개변수를 어떤 타입이든 수용할 수 있는 `Object`로 가정하고, `Object`의 기능만 사용할 수 있다.

발생한 문제들을 생각해보면 타입 매개변수를 `Animal`로 제한하지 않았기 때문이다. 만약 타입 인자가 모두 `Animal`과 그 자식만 들어올 수 있게 제한한다면 어떨까?

## 타입 매개변수 제한4 - 타입 매개변수 제한

타입 매개변수를 특정 타입으로 제한할 수 있다. 우선 코드를 보자.

```
package generic.ex3;

import generic.animal.Animal;

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;
    }

}
```

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.

```
package generic.ex3;

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

public class AnimalHospitalMainV3 {

    public static void main(String[] args) {
        AnimalHospitalV3<Dog> dogHospital = new AnimalHospitalV3<>();
```

```
        AnimalHospitalV3<Cat> catHospital = new AnimalHospitalV3<>();

        Dog dog = new Dog("멍멍이1", 100);
        Cat cat = new Cat("냐옹이1", 300);

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

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

        // 문제1 해결: 개 병원에 고양이 전달
        // dogHospital.set(cat); // 다른 타입 입력: 컴파일 오류

        // 문제2 해결: 개 타입 반환
        dogHospital.set(dog);
        Dog biggerDog = dogHospital.getBigger(new Dog("멍멍이2", 200));
        System.out.println("biggerDog = " + biggerDog);
    }
}
```

**실행 결과**

```
동물 이름: 멍멍이1
동물 크기: 100
멍멍
동물 이름: 냐옹이1
동물 크기: 300
냐옹
biggerDog = Animal{name='멍멍이2', size=200}
```

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

**타입 안전성X 문제**

- 개 병원에 고양이를 전달하는 문제가 발생한다. ➜ 해결
- `Animal` 타입을 반환하기 때문에 다운 캐스팅을 해야 한다. ➜ 해결
- 실수로 고양이를 입력했는데, 개를 반환하는 상황이라면 캐스팅 예외가 발생한다. ➜ 해결

**제네릭 도입 문제**

- 제네릭에서 타입 매개변수를 사용하면 어떤 타입이든 들어올 수 있다. ➜ 해결
- 그리고 어떤 타입이든 수용할 수 있는 `Object`로 가정하고, `Object`의 기능만 사용할 수 있다. ➜ 해결
  - 여기서는 `Animal`을 상한으로 두어서 `Animal`의 기능을 사용할 수 있다.

**정리**

제네릭에 **타입 매개변수 상한**을 사용해서 타입 안전성을 지키면서 상위 타입의 원하는 기능까지 사용할 수 있었다. 덕분에 코드 재사용과 타입 안전성이라는 두 마리 토끼를 동시에 잡을 수 있었다.

# 제네릭 메서드

이번에는 특정 메서드에 제네릭을 적용하는 제네릭 메서드에 대해 알아보자.

참고로 앞서 살펴본 제네릭 타입과 지금부터 살펴볼 제네릭 메서드는 둘다 제네릭을 사용하기는 하지만 서로 다른 기능을 제공한다.

```
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 t) {
        System.out.println("generic print: " + t);
        return t;
    }

    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;
        Object object = GenericMethod.objMethod(i);

        // 타입 인자(Type Argument) 명시적 전달
        System.out.println("명시적 타입 인자 전달");
        Integer result = GenericMethod.<Integer>genericMethod(i);
        Integer integerValue = GenericMethod.<Integer>numberMethod(10);
        Double doubleValue = GenericMethod.<Double>numberMethod(20.0);
    }
}
```

**실행 결과**

```
object print: 10
명시적 타입 인자 전달
generic print: 10
bound print: 10
bound print: 20.0
```

**제네릭 타입**

- 정의: `GenericClass<T>`
- 타입 인자 전달: 객체를 생성하는 시점
  - 예) `new GenericClass<String>`

**제네릭 메서드**

- 정의: `<T> T genericMethod(T t)`
- 타입 인자 전달: 메서드를 호출하는 시점

    - 예) `GenericMethod.<Integer>genericMethod(i)`

- 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.

**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 제한할 수 있다.
다음 코드는 타입 매개변수를 `Number`로 제한했다. 따라서 `Number`와 그 자식만 받을 수 있다.
참고로 `Integer`, `Double`, `Long`과 같은 숫자 타입이 `Number`의 자식이다.

```
```

```
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);
```

**실행 결과**

```
//추가한 내용만 출력
타입 추론
generic print: 10
bound print: 10
bound print: 20.0
```

타입 추론 덕분에 타입 인자를 직접 전달하는 불편함이 줄어든다. 이 경우 타입을 추론해서 컴파일러가 대신 처리하기 때문에 타입을 전달하지 않는 것 처럼 보인다. 하지만 실제로는 타입 인자가 전달된다는 것을 기억하자.

# 제네릭 메서드 활용

앞서 **제네릭 타입**으로 만들었던 `AnimalHospitalV3`의 주요 기능을 **제네릭 메서드**로 다시 만들어보자.

## 제네릭 메서드 활용

```
package generic.ex4;

import generic.animal.Animal;

public class AnimalMethod {

    public static <T extends Animal> void checkup(T t) {
        System.out.println("동물 이름: " + t.getName());
        System.out.println("동물 크기: " + t.getSize());
        t.sound();
    }

    public static <T extends Animal> T getBigger(T t1, T t2) {
        return t1.getSize() > t2.getSize() ? t1 : t2;
    }

}
```

- `checkup()`, `getBigger()`라는 두 개의 제네릭 메서드를 정의했다. 둘 다 `Animal`을 상한으로 제한한다.

```
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("냐옹이", 100);
```

```
        AnimalMethod.checkup(dog);
        AnimalMethod.checkup(cat);

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

**실행 결과**

```
동물 이름: 멍멍이
동물 크기: 100
멍멍
동물 이름: 냐옹이
동물 크기: 100
냐옹
bigger = Animal{name='큰 멍멍이', size=200}
```

기존 코드와 같이 작동하는 것을 확인할 수 있다.
참고로 제네릭 메서드를 호출할 때 타입 추론을 사용했다.

## 제네릭 타입과 제네릭 메서드의 우선순위

정적 메서드는 제네릭 메서드만 적용할 수 있지만, 인스턴스 메서드는 제네릭 타입도 제네릭 메서드도 둘다 적용할 수 있다.
여기에 제네릭 타입과 제네릭 메서드의 타입 매개변수를 같은 이름으로 사용하면 어떻게 될까?

```
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 t) {
        System.out.println("animal.className: " +
animal.getClass().getName());
        System.out.println("t.className: " + t.getClass().getName());
        // t.getName(); // 호출 불가 메서드는 <T> 타입이다. <T extends Animal> 타입이
아니다.
        return t;
    }

}
```

```
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("냐옹이", 50);

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

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

**실행 결과**

```
animal.className: generic.animal.Dog
t.className: generic.animal.Cat
returnCat = Animal{name='냐옹이', size=50}
```

**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.

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

    private T animal;

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

    public <Z> Z printAndReturn(Z z) {
        //...
    }
}
```

# 와일드카드1

이번에는 제네릭 타입을 조금 더 편리하게 사용할 수 있는 와일드카드(wildcard)에 대해 알아보자.
참고로 와일드카드라는 뜻은 컴퓨터 프로그래밍에서 `*`, `?`와 같이 하나 이상의 문자들을 상징하는 특수 문자를 뜻한다.
쉽게 이야기해서 여러 타입이 들어올 수 있다는 뜻이다.

```java
package generic.ex5;

public class Box<T> {

    private T value;

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

    public T get() {
        return value;
    }

}
```

- 단순히 데이터를 넣고 반환할 수 있는 제네릭 타입을 하나 만들었다.

```java
package generic.ex5;

import generic.animal.Animal;

public class WildcardEx {

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

    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 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;
    }
}
```

- 제네릭 메서드와 와일드 카드를 비교할 수 있게 같은 기능을 각각 하나씩 배치해두었다.
- 와일드카드는 `?`를 사용해서 정의한다.
- 코드는 뒤에서 하나씩 설명하겠다.

```
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.printAndReturnGeneric(dogBox);
        Animal animal = WildcardEx.printAndReturnWildcard(dogBox);
    }

}
```

**실행 결과**

```
T = Animal{name='멍멍이', size=100}
? = Animal{name='멍멍이', size=100}
이름 = 멍멍이
이름 = 멍멍이
이름 = 멍멍이
이름 = 멍멍이
```

코드를 하나씩 설명하겠다.

**참고!!!**

와일드카드는 제네릭 타입이나, 제네릭 메서드를 선언하는 것이 아니다. 와일드카드는 이미 만들어진 제네릭 타입을 활용할 때 사용한다.

## 비제한 와일드카드

```
//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

다.

**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 와일드카드**

`printGenericV1()` 제네릭 메서드를 보자. 제네릭 메서드에는 타입 매개변수가 존재한다. 그리고 특정 시점에 타입 매개변수에 타입 인자를 전달해서 타입을 결정해야 한다. 이런 과정은 매우 복잡하다.

반면에 `printWildcardV1()` 메서드를 보자. 와일드카드는 일반적인 메서드에 사용할 수 있고, 단순히 매개변수로 제네릭 타입을 받을 수 있는 것 뿐이다. 제네릭 메서드처럼 타입을 결정하거나 복잡하게 작동하지 않는다. 단순히 일반

메서드에 제네릭 타입을 받을 수 있는 매개변수가 하나 있는 것 뿐이다.
제네릭 타입이나 제네릭 메서드를 정의하는게 꼭 필요한 상황이 아니라면, 더 단순한 와일드카드 사용을 권장한다.

# 와일드카드2

## 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

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.

## 하한 와일드 카드

와일드카드는 상한 뿐만 아니라 하한도 지정할 수 있다.

```
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); // 하한이 Animal
        //writeBox(catBox); // 하한이 Animal

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

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

**실행 결과**

```
animal = Animal{name='멍멍이', size=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.

# 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
```

```
}
```

- 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`

```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;
    }

}
```

```java
//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.

## 타입 이레이저 방식의 한계

컴파일 이후에는 제네릭의 타입 정보가 존재하지 않는다. `.class`로 자바를 실행하는 런타임에는 우리가 지정한 `Box<Integer>`, `Box<String>`의 타입 정보가 모두 제거된다.

따라서 런타임에 타입을 활용하는 다음과 같은 코드는 작성할 수 없다.

**소스 코드**

```
class EraserBox<T> {

    public boolean instanceCheck(Object param) {
        return param instanceof T; // 오류
    }

    public void create() {
        return new T(); // 오류
    }

}
```

**런타임**

```
class EraserBox {

    public boolean instanceCheck(Object param) {
        return param instanceof Object; // 오류
    }

    public void create() {
        return new Object(); // 오류
    }

}
```

- 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.

## 문제와 풀이2

### 준비

- 여러분은 게임속 케릭터를 클래스로 만들어야 한다.
- `BioUnit`은 유닛들의 부모 클래스이다.
- `BioUnit`의 자식 클래스로 `Marine`, `Zealot`, `Zergling`이 있다.
- 문제를 풀기 전에 우선 다음 코드를 완성하자.

```
package generic.test.ex3.unit;

public class BioUnit {

    private String name;
    private int hp;

    public BioUnit(String name, int hp) {
        this.name = name;
        this.hp = hp;
    }

    public String getName() {
        return name;
    }

    public int getHp() {
        return hp;
    }

    @Override
    public String toString() {
        return "BioUnit{" +
                "name='" + name + '\'' +
                ", hp=" + hp +
                '}';
    }
```

```
}
```

```
package generic.test.ex3.unit;

public class Marine extends BioUnit {

    public Marine(String name, int hp) {
        super(name, hp);
    }
}
```

```
package generic.test.ex3.unit;

public class Zealot extends BioUnit {

    public Zealot(String name, int hp) {
        super(name, hp);
    }
}
```

```
package generic.test.ex3.unit;

public class Zergling extends BioUnit {

    public Zergling(String name, int hp) {
        super(name, hp);
    }
}
```

## 문제와 풀이1 - 제네릭 메서드와 상한

**문제 설명**

- 다음 코드와 실행 결과를 참고해서 `UnitUtil` 클래스를 만들어라.
- `UnitUtil.maxHp()` 메서드의 조건은 다음과 같다.

- 두 유닛을 입력 받아서 체력이 높은 유닛을 반환한다. 체력이 같은 경우 둘 중 아무나 반환해도 된다.
- 제네릭 메서드를 사용해야 한다.
- 입력하는 유닛의 타입과 반환하는 유닛의 타입이 같아야 한다.

```
package generic.test.ex3;

import generic.test.ex3.unit.Marine;
import generic.test.ex3.unit.Zealot;

public class UnitUtilTest {

    public static void main(String[] args) {
        Marine m1 = new Marine("마린1", 40);
        Marine m2 = new Marine("마린2", 50);
        Marine resultMarine = UnitUtil.maxHp(m1, m2);
        System.out.println("resultMarine = " + resultMarine);

        Zealot z1 = new Zealot("질럿1", 100);
        Zealot z2 = new Zealot("질럿2", 150);
        Zealot resultZealot = UnitUtil.maxHp(z1, z2);
        System.out.println("resultZealot = " + resultZealot);
    }
}
```

**실행 결과**

```
resultMarine = BioUnit{name='마린2', hp=50}
resultZealot = BioUnit{name='질럿2', hp=150}
```

**정답 - UnitUtil 클래스**

```
package generic.test.ex3;

import generic.test.ex3.unit.BioUnit;

public class UnitUtil {

    public static <T extends BioUnit> T maxHp(T t1, T t2) {
```

```
        if (t1.getHp() > t2.getHp()) {
            return t1;
        } else {
            return t2;
        }
    }
}
```

## 문제와 풀이2 - 제네릭 타입과 상한

**문제 설명**

- 다음 코드와 실행 결과를 참고해서 `Shuttle` 클래스를 만들어라.
- `Shuttle` 클래스의 조건은 다음과 같다.
  - 제네릭 타입을 사용해야 한다.
  - `showInfo()` 메서드를 통해 탑승한 유닛의 정보를 출력한다.

```
package generic.test.ex3;

import generic.test.ex3.unit.Marine;
import generic.test.ex3.unit.Zealot;
import generic.test.ex3.unit.Zergling;

public class ShuttleTest {

    public static void main(String[] args) {
        Shuttle<Marine> shuttle1 = new Shuttle<>();
        shuttle1.in(new Marine("마린", 40));
        shuttle1.showInfo();

        Shuttle<Zergling> shuttle2 = new Shuttle<>();
        shuttle2.in(new Zergling("저글링", 35));
        shuttle2.showInfo();

        Shuttle<Zealot> shuttle3 = new Shuttle<>();
        shuttle3.in(new Zealot("질럿", 100));
        shuttle3.showInfo();
    }
}
```

**실행 결과**

```
이름: 마린, HP: 40
이름: 저글링, HP: 35
이름: 질럿, HP: 100
```

**정답 - Shuttle 클래스**

```
package generic.test.ex3;

import generic.test.ex3.unit.BioUnit;

public class Shuttle<T extends BioUnit> {

    private T unit;

    public void in(T t) {
        unit = t;
    }

    public T out() {
        return unit;
    }

    public void showInfo() {
        System.out.println("이름: " + unit.getName() + ", HP: " + unit.getHp());
    }

}
```

## 문제와 풀이3 - 제네릭 메서드와 와일드카드

**문제 설명**

- 앞서 문제에서 만든 `Shuttle`을 활용한다.
- 다음 코드와 실행 결과를 참고해서 `UnitPrinter` 클래스를 만들어라.
- `UnitPrinter` 클래스의 조건은 다음과 같다.

- `UnitPrinter.printV1()`은 제네릭 메서드로 구현해야 한다.
- `UnitPrinter.printV2()`는 와일드카드로 구현해야 한다.
- 이 두 메서드는 셔틀에 들어있는 유닛의 정보를 출력한다.

```
package generic.test.ex3;

import generic.test.ex3.unit.Marine;
import generic.test.ex3.unit.Zealot;
import generic.test.ex3.unit.Zergling;

public class UnitPrinterTest {

    public static void main(String[] args) {
        Shuttle<Marine> shuttle1 = new Shuttle<>();
        shuttle1.in(new Marine("마린", 40));

        Shuttle<Zergling> shuttle2 = new Shuttle<>();
        shuttle2.in(new Zergling("저글링", 35));

        Shuttle<Zealot> shuttle3 = new Shuttle<>();
        shuttle3.in(new Zealot("질럿", 100));

        UnitPrinter.printV1(shuttle1);
        UnitPrinter.printV2(shuttle1);
    }
}
```

**실행 결과**

```
이름: 마린, HP: 40
이름: 마린, HP: 40
```

**정답 - UnitPrinter 클래스**

```
package generic.test.ex3;

import generic.test.ex3.unit.BioUnit;
```

```
public class UnitPrinter {

    public static <T extends BioUnit> void printV1(Shuttle<T> t1) {
        T unit = t1.out();
        System.out.println("이름: " + unit.getName() + ", HP: " + unit.getHp());
    }

    public static void printV2(Shuttle<? extends BioUnit> t1) {
        BioUnit unit = t1.out();
        System.out.println("이름: " + unit.getName() + ", HP: " + unit.getHp());
    }
}
```

## 정리

실무에서 직접 제네릭을 사용해서 무언가를 설계하거나 만드는 일은 드물다. 그것보다는 대부분 이미 제네릭을 통해 만들어진 프레임워크나 라이브러리들을 가져다 사용하는 경우가 훨씬 많다. 그래서 이미 만들어진 코드의 제네릭을 읽고 이해하는 정도면 충분하다. 실무에서 직접 제네릭을 사용하더라도 어렵고 복잡하게 사용하기 보다는 보통 단순하게 사용한다.
지금까지 학습한 정도면 실무에 필요한 제네릭은 충분히 이해했다고 볼 수 있다.

제네릭은 지금까지 설명한 내용보다 더 복잡하고 어려운 개념들도 있다. 특히 공변(covariant), 반공변(contravariant)과 같은 개념들이 그러하다. 이런 개념들을 이해하면 와일드카드가 존재하는 이유도 더 깊이있게 알 수 있다.
하지만 제네릭을 사용해서 매우 복잡한 라이브러나 프레임워크를 직접 설계하지 않는 이상 이런 개념들을 꼭 이해할 필요는 없다. 이런 부분은 실무에서 많은 경험을 쌓고 본인이 필요하다고 느껴질 때 따로 공부하는 것을 권장한다.

제네릭은 이후에 설명하는 컬렉션 프레임워크에서 가장 많이 사용된다. 따라서 컬렉션 프레임워크를 통해서 제네릭이 어떻게 활용되는지 자연스럽게 학습할 수 있다.