제네릭 2

22-1. 제네릭의 심화 문법

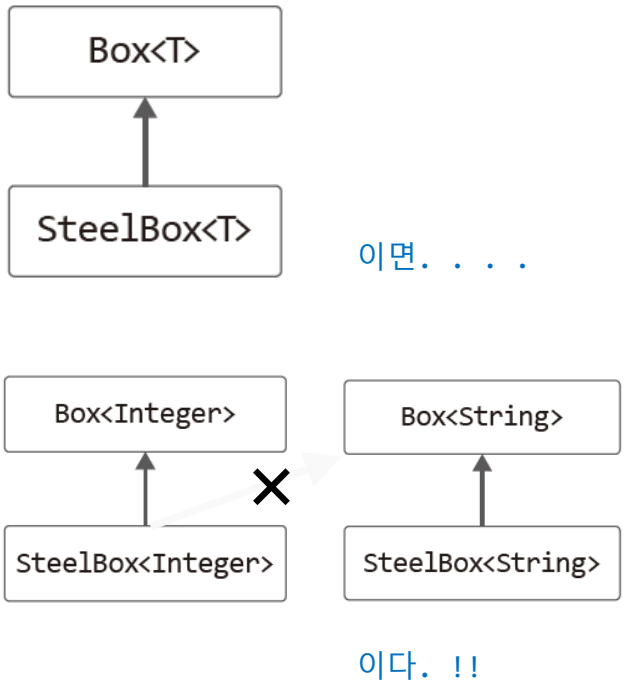
제네릭 클래스의 상속

```
class Box<T> {
   protected T ob;
   public void set(T o) { ob = o; }
   public T get() { return ob; }
}

class SteelBox<T> extends Box<T> {
   public SteelBox(T o) {    // 제네릭 클래스의 생성자
      ob = o;
   }
}

Box<Integer> iBox = new SteelBox<>(7959);
   ↔ Box<Integer> iBox = new SteelBox<Integer>(7959);

Box<String> sBox = new SteelBox<>("Simple");
   ↔ Box<String> sBox = new SteelBox<String>("Simple");
```



타켓 타입

```
class Box<T> {
   private T ob;
   public void set(T o) { ob = o; }
   public T get() { return ob; }
}

class EmptyBoxFactory {
   public static <T> Box<T> makeBox() {
      Box<T> box = new Box<T>();
      return box;
   }
}
```

```
public static void main(String[] args) {
   Box<Integer> iBox = EmptyBoxFactory.<Integer>makeBox();
      // Box<Integer> iBox = EmptyBoxFactory.makeBox();
   iBox.set(25);
   System.out.println(iBox.get());
}
```

```
Box<Integer> iBox = EmptyBoxFactory.makeBox();
```

참조변수의 형 Box<Integer>를 기반으로 makeBox 메소드의 T를 결정하게 된다.
따라서 이를 가리켜 타겟 타입이라 한다.

와이드카드의 설명에 앞서(제네릭 메소드 vs 일반 메소드

Box<Integer>의 인스턴스, Box<String>의 인스턴스를 인자로 전달 가능

```
public static <T> void peekBox(Box<T> box) {
   System.out.println(box);
}
```

Box<Integer>의 인스턴스, Box<String>의 인스턴스를 인자로 전달 가능할 것 같지만 불가능

```
public static void peekBox(Box<Object> box) {
   System.out.println(box);
}
```

"Box<Object>와 Box<String>은 상속 관계를 형성하지 않는다."

"Box<Object>와 Box<Integer>은 상속 관계를 형성하지 않는다."

그러나 와일드카드를 사용하면 일반 메소드도 이 두 인스턴스를 인자로 받을 수 있다.

와이드 카드

Box<Integer>의 인스턴스, Box<String>의 인스턴스를 인자로 전달 가능

```
public static void peekBox(Box<?> box) {
   System.out.println(box);
}
```

기능적으로는 두 메소드 완전 동일

```
public static <T> void peekBox(Box<T> box) {
   System.out.println(box);
} // 제네릭 메소드의 정의

public static void peekBox(Box<?> box) {
   System.out.println(box);
} // 와일드카드 기반 메소드 정의
```

그러나 와일드카드 기반 메소드 정의가 더 간결하므로 우선시 해야 한다고 권고하고 있다.
메소드의 정의가 복잡해질수록 와일드카드 기반 메소드 정의가 더 간결해 보인다.

와이드 카드의 상한과 하한의 제한: Bounded Wildcards

기본적인 문법적 이해는 어렵지 않다.

그러나 목적과 이유에 대한 이해에서 어렵게 느껴질 수 있다.

일단 이해하자. 그리고 이해한 후에는 암기할 정도로 숙달이 되어야 한다.

상한 제한된 와이드카드(Upper-Bounded Wildcards

```
public static void peekBox(Box<?> box) {
   System.out.println(box);
}
```

```
public static void peekBox(Box<? extends Number> box) {
   System.out.println(box);
}
```

box는 Box<T> 인스턴스의 참조 값을 전달받는 매개변수이다.

→ 단 전달되는 인스턴스의 T는 Number 또는 이를 상속하는 하위 클래스이어야 함

하한 제한된 와이드카드(Lower-Bounded Wildcards)

```
public static void peekBox(Box<? super Integer> box) {
   System.out.println(box);
}
```

box는 Box<T> 인스턴스의 참조 값을 전달받는 매개변수이다.

→ 단 전달되는 인스턴스의 T는 Integer 또는 Integer가 상속하는 클래스이어야 함

즉 위 메소드의 인자로 전달 가능한 인스턴스는 Box<Integer>, Box<Number>, Box<Object>으로 제한됨

와일드카드 제한의 이유 설명을 위한 도입

```
class Box<T> {
   private T ob;
   public void set(T o) { ob = o; }
   public T get() { return ob; }
}

class Toy {
   public String toString() { return "I am a Toy"; }
}

class BoxHandler {
   public static void outBox(Box<Toy> box) {
      Toy t = box.get();    // 상자에서 꺼내기
      System.out.println(t);
   }

   public static void inBox(Box<Toy> box, Toy n) {
      box.set(n);    // 상자에 넣기
   }
}
```

아래의 오류 상황에서 컴파일 오류가 발생하지 않는다!

```
public static void outBox(Box<Toy> box) {
   box.get();   // 꺼내는 것! OK!
   box.set(new Toy());   // 넣는 것! 이것도 OK!
}

public static void inBox(Box<Toy> box, Toy n) {
   box.set(n);   // 넣는 것! OK!
   Toy myToy = box.get();   // 꺼내는 것! 이것도 OK!
}
```

상한 제한의 목적

```
class Box<T> {
   private T ob;
   public void set(T o) { ob = o; }
   public T get() { return ob; }
}
```

```
class Car extends Toy {...}    // 자동차 장난감
class Robot extends Toy {...}    // 로봇 장난감
```

. . .

Box<Car> 또는 Box<Robot> 인스턴스가 인자로 전달될 수 있다.

```
public static void outBox(Box<? extends Toy> box) {
   box.get();   // 꺼내는 것! OK!
   box.set(new Toy());   // 넣는 것! ERROR!
}
```

다음과 같이 정리하자!

Box<? extends Toy> box 대상으로 넣는 것 불가!

상한 제한의 결과

```
class BoxHandler {
   public static void outBox(Box<Toy> box) {
      Toy t = box.get();    // 상자에서 꺼내기
      System.out.println(t);
   }

   public static void inBox(Box<Toy> box, Toy n) {
      box.set(n);    // 상자에 넣기
   }
}
```

다음 수준으로 높아졌다.

```
class BoxHandler {
   public static void outBox(Box<? extends Toy> box) {
      Toy t = box.get();    // 상자에서 꺼내기
      System.out.println(t);
   }

   public static void inBox(Box<Toy> box, Toy n) {
      box.set(n);    // 상자에 넣기
   }
}
```

하한 제한의 목적

```
class Box<T> {
   private T ob;
   public void set(T o) { ob = o; }
   public T get() { return ob; }
}
```

```
class Plastic {...}
class Toy extends Plastic {...}
```

. . .

Box<Toy> 또는 Box<Plastic> 인스턴스가 인자로 전달될 수 있다.

```
public static void inBox(Box<? super Toy> box, Toy n) {
   box.set(n);   // 넣는 것! OK!
   Toy myToy = box.get();   // 꺼내는 것! Error!
}
```

다음과 같이 정리하자!

Box<? super Toy> box 대상으로 꺼내는 것 불가!

하한 제한의 결과

```
class BoxHandler {
   public static void outBox(Box<? extends Toy> box) {
      Toy t = box.get();    // 상자에서 꺼내기
      System.out.println(t);
   }

   public static void inBox(Box<Toy> box, Toy n) {
      box.set(n);    // 상자에 넣기
   }
}
```

다음 수준으로 높아졌다.

```
class BoxHandler {
   public static void outBox(Box<? extends Toy> box) {
      Toy t = box.get();    // 상자에서 꺼내기
      System.out.println(t);
   }

   public static void inBox(Box<? super Toy> box, Toy n) {
      box.set(n);    // 상자에 넣기
   }
}
```

상한 제한과 하한 제한의 좋은 예 하나!

```
class BoxContentsMover {
   // from에 저장된 내용물을 to로 이동
   public static void moveBox(Box<? super Toy> to, Box<? extends Toy> from) {
      to.set(from.get());
   }
}
```

제한된 와일드카드 선언을 갖는 제네릭 메소드: 도입

```
class BoxHandler {
   public static void outBox(Box<? extends Toy> box) {
      Toy t = box.get();   // 상자에서 꺼내기
      System.out.println(t);
   }
   public static void inBox(Box<? super Toy> box, Toy n) {
      box.set(n);   // 상자에 넣기
   }
}
```

위 클래스의 두 메소드는 사실상 Box<Toy> 인스턴스를 대상으로 정의된 메소드이다!

따라서 Toy와 전혀 관계 없는 Robot 클래스가 존재하는 상황에서 Box<Robt>을 대상으로는 동작하지 않는다.

그렇다면 이 상황에서 메소드 오버로딩이 가능할까?

다음 형태로 메소드 오버로딩 불가능하다!

```
class BoxHandler {
   // 다음 두 메소드는 오버로딩 인정 안됨.
   public static void outBox(Box<? extends Toy> box) {...}
   public static void outBox(Box<? extends Robot> box) {...}

   // 다음 두 메소드는 두 번째 매개변수로 인해 오버로딩 인정 됨.
   public static void inBox(Box<? super Toy> box, Toy n) {...}
   public static void inBox(Box<? super Robot> box, Robot n) {...}
}
```

왜 이러한 형태의 오버로딩을 허용하지 않을까? 이유는 Type Erasure!
컴파일 과정에서 < . . . > 내용이 모두 지워진다. 따라서 컴파일러가 이러한 형태의 메소드 오버로딩을 허용하지 않음.

그래서 와이드 카드 선언을 갖는 메소드를 제네릭으로

```
public static void outBox(Box<? extends Toy> box) {...}
public static void outBox(Box<? extends Robot> box) {...}
```

이것이 대안!

```
public static <T> void outBox(Box<? extends T> box) {...}
```

와일드 카드 선언을 갖는 제네릭 메소드가 등장했을 때, 이 설명의 흐름을 기억해야 당황하지 않는다.

여기까지 잘 이해했다면 한 고비 넘긴거예요. ☺

제네릭 인터페이스의 정의와 구현

인터페이스 역시 클래스와 마찬가지로 제네릭으로 정의할 수 있다.

```
interface Getable<T> {
   public T get();
}

// 인터페이스 Getable<T>를 구현하는 Box<T> 클래스
class Box<T> implements Getable<T> {
   private T ob;
   public void set(T o) { ob = o; }

   @Override
   public T get() { return ob; }
}

class Toy {
   @Override
   public String toString() {
      return "I am a Toy";
   }
}
```

```
public static void main(String[] args) {
   Box<Toy> box = new Box<>();
   box.set(new Toy());

   // Box<T>가 Getable<T>를 구현하므로 참조 가능
   Getable<Toy> gt = box;
   System.out.println(gt.get());
}
```