제네릭의 심화 문법

제네릭의 심화 문법

- 제네릭 클래스와 상속
  - 제네릭 클래스도 상속 가능
  - SteelBox클래스는 Box클래스를 상속하기 때문에 "Box<Integer> iBox = new SteelBox<>(7959)" 와 같이 인스턴스 구성 가능
    - "SteelBox<Integer> 클래스(제네릭 타입)는 Box<Integer> 클래스(제네릭 타입)를 상속한다."
  - 단, 타입인자의 상속은 불가
  - 즉, Number를 Integer가 상속하지만 Box<Number>와 Box<Integer>는 상속 관계를 형성하지 않음

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

```
public class SteelBox<T> extends Box<T> {
    public SteelBox(T o) {
        this.ob = o;
    }
}
```

```
public class GenericExtends {
    public static void main(String[] args) {
        Box<Integer> iBox = new SteelBox<>(7959);
        Box<String> sBox = new SteelBox<>("Simple");
        System.out.println(iBox.get());
        System.out.println(sBox.get());
    }
}
```

제네릭의 심화 문법

- 타겟 타입(Targer Type)
  - 자바 컴파일러는 생략된 자료형 정보에 대해 유추할 수 있음
  - 컴파일러가 자료형 유추를 진행하는 상황은 다양함
  - makeBook 메소드 : 제네릭 메소드로 상자를 생성해서 반환
  - 이전에 구현했던 makeBox 메소드와 달리 인자를 전달 받지 않기 때문에 메소드 호출 시 T에 대한 타입 인자 전달해야 함
  - 자바 7부터 메소드 호출 시 T에 대한 타입 인자 전달하지 않아도 호출 가능 -> 자료형 유추
  - T의 유추에 사용된 정보 Box<Integer>를 타겟 타입이라 함

```
public class Box<T> {
    private T obj;
    public T getObj() { return obj; }
    public void setObj(T obj) { this.obj = obj; }
}
```

```
public class EmptyBoxFactory {
    public static <T> Box<T> makeBook() {
        Box<T> box = new Box<T>();  // 상자 생성
        return box; // 생성한 상자 반환
    }
}
```

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

제네릭의 심화 문법

- 와일드카드(Wildcard)
  - 제네릭에서 어렵다고 알려진 개념
  - 앞서 정의했던 UnBoxer 클래스에 무엇이 들어있나를 확인하는 기능의 제네릭 메소드 추가

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

```
public class Unboxer {
    public static <T> T openBox(Box<T> box) {
        return box.get();
    }
    // 상자 안의 내용물을 확인하는 기능
    public static <T> void peekBox(Box<T> box) {
        System.out.println(box);
    }
}
```

```
public class WildcardUnboxer {
    public static void main(String[] args) {
        Box<String> box = new Box<>();
        box.set("So simple String");
        Unboxer.peekBox(box);   // 상자 안의 내용물 확인
    }
}
```

  - peekBox 메소드를 제네릭으로 정의한 이유가 Box<Integer>, Box<String>의 인스턴스를 인자로 전달 받기 위함이므로 peekBox 메소드를 "public static void peekBox(Box<Object> box)"로 정의하면 되겠지만 오류 발생
  - String 클래스나 Integer 클래스가 Object 클래스를 상속하기는 하지만 타겟 타입은 상속할 수 없음 -> 와일드카드 사용

제네릭의 심화 문법

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

```
public class Unboxer {
    public static <T> T openBox(Box<T> box) {
        return box.get();
    }
    // 와일드 카드 사용
    public static void peekBox(Box<?> box) {
        System.out.println(box);
    }
}
```

```
public class WildcardUnboxer {
    public static void main(String[] args) {
        Box<String> box = new Box<>();
        box.set("So simple String");
        Unboxer.peekBox(box);   // 상자 안의 내용물 확인
    }
}
```

- ? 기호를 사용하여 와일드 카드 선언 -> Box<T>를 기반으로 생성된 Box<Integer>와 Box<String> 인스턴스를 인자로 받음
- 기능적인 측면에서는 제네릭으로 선언하거나 와일드 카드를 사용하는 것에 차이 없음
- 코드 간결을 이유로 보편적인 선호도가 더 높음

제네릭의 심화 문법

- 와일드카드의 상한과 하한의 제한 : Bounded Wildcards
  - peekbox 메소드에 Number또는 Number의 하위 클래스인 제네릭 타입의 인스턴스만 제한 -> 상한 제한된 와일드카드 사용
    - Box<? extends Number> box : box는 Box<T> 인스턴스를 참조하며, Box<T> 인스턴스의 T는 Number또는 하위 클래스

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

```
public class Unboxer {
    // 와일드 카드 사용
    public static void peekBox(Box<? extends Number> box) {
        System.out.println(box);
    }
}
```

```
public class UpperBoundedWildcard {
    public static void main(String[] args) {
        Box<Integer> iBox = new Box<>();
        iBox.set(1234);
        Box<Double> dBox = new Box<>();
        dBox.set(10.009);
        Unboxer.peekBox(iBox);
        Unboxer.peekBox(dBox);
    }
}
```

제네릭의 심화 문법

- 하한 제한된 와일드카드 : Box<? super Integer> box
  - box는 Box<T> 인스턴스를 참조하는 참조변수
  - 이때 Box<T> 인스턴스의 T는 Integer 또는 Integer가 상속하는 클래스
  - 따라서 인자로 전달될 수 있는 인스턴스의 타입 종류는 Box<Integer>, Box<Number>, Box<Object>

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

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

```
public class UpperBoundedWildcard {
    public static void main(String[] args) {
        Box<Integer> iBox = new Box<>();
        iBox.set(1234);
        Box<Number> nBox = new Box<>();
        nBox.set(new Integer(9955));
        Box<Object> oBox = new Box<>();
        oBox.set("My simple Instance");
        Box<Double> dBox = new Box<>();
        dBox.set(10.009);
        Unboxer.peekBox(iBox);
        Unboxer.peekBox(nBox);
        Unboxer.peekBox(oBox);
        Unboxer.peekBox(dBox);  // 오류
    }
}
```

제네릭의 심화 문법

- 언제 와일드카드에 제한을 걸어야 하는가? : (1) 도입
  - public static void peekBox(Box<? extends Number> box) : Box<T>의 T를 Number 또는 Number를 상속하는 클래스로 제한
  - public static void peekBox(Box<? super Integer> box) : Box<T>의 T를 Integer 또는 Integer가 상속하는 클래스로 제한
  - 위 두 메소드를 이해해야 다음과 같은 메소드를 이해할 수 있음
    - Collections 클래스의 복사 메소드 : public static <T> void copy(List<? super T> dest, List<? extends T> src)

제네릭의 심화 문법

- 언제 와일드카드에 제한을 걸어야 하는가? : (2) 상한 제한의 목적

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

```
public class BoundedWildcardBase {
    public static void main(String[] args) {
        Box<Toy> box = new Box<>();
        BoxHandler.inBox(box, new Toy());
        BoxHandler.outBox(box);
    }
}
```

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

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

- outBox 메소드는 잘 동작하지만 "필요한 만큼만 기능을 허용하여, 코드의 오류가 컴파일 과정에서 최대한 발견되도록 한다. " 라는 조건을 만족하지 않음
- outBox 메소드는 상자에서 내용물을 꺼내는 기능이며 매개변수 box를 대상으로 get은 물론 set의 호출도 가능
- 위 상황에서 outBox 메소드 내에서 실수로 "box.set(new Toy())"와 같이 set 메소드를 호출하며 임의의 인스턴스를 전달할 수도 있으며 이때 컴파일에서 오류 발생하지 않음
- 이러한 실수를 방지하기 위해 outBox 메소드를 정의할 때 매개변수 box를 대상으로 get은 가능하지만 set은 불가능하도록 제한

제네릭의 심화 문법

- 위 과정 진행 시 컴파일 과정에서 오류 발견 가능
- public static void outBox(Box<Toy> box) -> public static void outBox(Box<? extends Toy> box) 로 변경
- 위와 같이 변경 시 set 메소드 호출이 불가능한 이유
  - 매개변수로 Toy 인스턴스를 저장할 수 있는 상자만(즉, Box<T> 인스턴스만) 전달된다는 사실을 보장할 수 없음
  - 만약 Toy를 상속하는 클래스가 타입인자로 전달 된다면 Toy 인스턴스를 상자에 담을 수 없음

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

```
public class BoundedWildcardBase {
    public static void main(String[] args) {
        Box<Toy> box = new Box<>();
        BoxHandler.inBox(box, new Toy());
        BoxHandler.outBox(box);
    }
}
```

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

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

제네릭의 심화 문법

- 언제 와일드카드에 제한을 걸어야 하는가? : (3) 하한 제한의 목적
  - inBox 메소드는 잘 동작하지만 "필요한 만큼만 기능을 허용하여, 코드의 오류가 컴파일 과정에서 최대한 발견되도록 한다. " 라는 조건을 만족하지 않음
  - inBox 메소드는 인스턴스를 저장하는 것이 목적인데 "Toy myToy = box.get();" 코드를 넣을 수 있다면 실수이며, 이는 컴파일 과정에서 발견되지 않음
  - 이러한 실수를 방지하기 위해 inBox 메소드를 정의할 때 매개변수 box를 대상으로 set은 가능하지만 get은 불가능하도록 제한
  - public static void inBox(Box<Toy> box, Toy n) -> public static void inBox(Box<? super Toy> box, Toy n)
  - 반환형을 Toy로 결정할 수 없기 때문에 get 메소드 사용 시 오류 발생
  - 만약 Toy 클래스가 Plastic 클래스로부터 상속받았다면 Box<plastic> 인스턴스로 생성하였을 경우 get 메소드의 반환형이 Box<Toy>가 아니기 때문에 오류 발생

제네릭의 심화 문법

- 언제 와일드카드에 제한을 걸어야 하는가? : (4) 정리하기
  - 와일드카드의 상한과 하한 제한이 필요한 이유의 본질은 그 자체로 이해하기 난해함
  - 아래 규칙으로 정리하는 것을 추천
    - Box<? extends Toy> box -> box가 참조하는 인스턴스를 대상으로 get 작업만 허용
    - Box<? super Toy> box -> box가 참조하는 인스턴스를 대상으로 set 작업만 허용