제네릭의 이해와 기본 문법

제네릭의 이해와 기본 문법

- 제네릭 이전의 코드
  - 제네릭 : 일반화 -> 자바에서 일반화의 대상은 자료형

```
public class Apple {
    public String toString() {
        return "I am an apple";
    }
}
```

```
public class OrangeBox {
    private Orange or;
    public Orange getOr() {return or;}
    public void setOr(Orange or) {this.or = or;}
}
```

```
public class AppleBox {
    private Apple ap;
    // 사과를 꺼낸다.
    public Apple getAp() {return ap;}
    // 사과를 담는다.
    public void setAp(Apple ap) {this.ap = ap;}
}
```

```
public class Orange {
    public String toString() {
        return "I am an orange";
    }
}
```

```
public class FruitAndBox {
    public static void main(String[] args) {
        // 사과 상자와 오렌지 상자 생성
        AppleBox appleBox = new AppleBox();
        OrangeBox orangeBox = new OrangeBox();
        // 사과와 오렌지를 각각의 상자에 담는다.
        appleBox.setAp(new Apple());
        orangeBox.setOr(new Orange());
        // 사과와 오렌지를 각각의 상자에서 꺼낸다.
        Apple ap = appleBox.getAp();
        Orange or = orangeBox.getOr();
        System.out.println(ap);
        System.out.println(or);
    }
}
```

제네릭의 이해와 기본 문법

- 사과 상자와 오렌지상자가 하는 일은 성격과 내용이 같아 하나의 클래스로 대체 가능

```
public class Apple {
    public String toString() {
        return "I am an apple";
    }
}
```

```
public class Orange {
    public String toString() {
        return "I am an orange";
    }
}
```

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

```
public class FruitAndBox2 {
    public static void main(String[] args) {
        // 상자 생성
        Box appleBox = new Box();
        Box orangeBox = new Box();
        // 사과와 오렌지를 각각의 상자에 담는다
        appleBox.setObj(new Apple());
        orangeBox.setObj(new Orange());
        // 상자에서 사과와 오렌지를 꺼낸다.
        Apple ap = (Apple) appleBox.getObj();
        Orange or = (Orange) orangeBox.getObj();
        System.out.println(ap);
        System.out.println(or);
    }
}
```

- Box 클래스를 사과와 오렌지뿐 아니라 무엇이든 담을 수 있는 클래스로 변경
- Box 인스턴스에서 내용물을 꺼낼 때 형 변환 필요 -> 귀찮고 실수로 이어질 가능성 존재

제네릭의 이해와 기본 문법

- Box 인스턴스에서 내용물을 꺼낼 때 형 변환 필요 -> 귀찮고 실수로 이어질 가능성 존재

```
public class Apple {
    public String toString() {
        return "I am an apple";
    }
}
```

```
public class Orange {
    public String toString() {
        return "I am an orange";
    }
}
```

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

```
public class FruitAndBoxFault {
    public static void main(String[] args) {
        Box aBox = new Box();
        Box oBox = new Box();

        // 아래 두 문장에서 사과와 오렌지가 이닌 문자열 저장
        aBox.set("Apple");
        aBox.set("Orange");

        // 상자에 과일이 담기지 않았는데 과일을 꺼내려 시도
        Apple ap = (Apple) aBox.get();
        Orange or = (Orange) oBox.get();
        System.out.println(ap);
        System.out.println(or);
    }
}
```

- 문자열을 저장한 것은 컴파일 과정에서 발견되지 않음
- 인스턴스가 아닌 문자열을 저장한 뒤 다시 인스턴스로 저장하려 하면 ClassCastException 예외 발생
- 예외 발생 보다는 코드 컴파일 시 오류로 발생하면 쉽게 오류를 발견할 수 있음

제네릭의 이해와 기본 문법

- 제네릭 기반의 클래스 정의하기
  - 자료형에 의존적이지 않은 클래스로 정의
  - <T> 키워드로 정의하며, 자료형을 인스턴스 생성 시 결정
  - Box 클래스의 T를 각각 Apple, Orange로 결정하여 인스턴스를 생성
  - 사과 상자에는 Apple또는 Apple을 상속하는 하위 클래스의 인스턴스 저장 가능(오렌지도 동일)

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

- Box<T>에서 T : 타입 매개변수(Type Parameter)
- Box<Apple>에서 Apple : 타입 인자(Type Argument)
- Box<Apple> : 매개변수화 타입(Parameterized Type) 또는 제네릭 타입(Generic Type)

```
public class FruitAndBox {
    public static void main(String[] args) {
        // 상자 생성
        Box<Apple> appleBox = new Box<Apple>();
        Box<Orange> orangeBox = new Box<Orange>();
        // 사과와 오렌지를 각각의 상자에 담는다
        appleBox.setObj(new Apple());
        orangeBox.setObj(new Orange());
        // 상자에서 사과와 오렌지를 꺼낸다.
        Apple ap = (Apple) appleBox.getObj();
        Orange or = (Orange) orangeBox.getObj();
        System.out.println(ap);
        System.out.println(or);
    }
}
```

제네릭의 이해와 기본 문법

- 앞에서 예외가 발생했던 코드를 제네릭 버전으로 수정하면 예외가 아닌 컴파일 과정에서 오류로 발생

```
public class Apple {
    public String toString() {
        return "I am an apple";
    }
}
```

```
public class Orange {
    public String toString() {
        return "I am an orange";
    }
}
```

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

```
public class FruitAndBoxFaultGeneric {
    public static void main(String[] args) {
        Box<Apple> aBox = new Box<Apple>();
        Box<Orange> oBox = new Box<Orange>();

        // 오류 발생
        aBox.set("Apple");
        aBox.set("Orange");

        Apple ap = aBox.get();
        Orange og = oBox.get();

        System.out.println(ap);
        System.out.println(og);
    }
}
```

제네릭의 이해와 기본 문법

- 다중 매개변수 기반 제네릭 클래스의 정의
  - 둘 이상의 타입 매개변수에 대한 제네릭 클래스 정의
  - 타입 매개변수의 이름은 한 문자, 대문자로 명명
  - 자주 사용하는 타입 매개변수 : E(Element), K(Key), N(Number), T(Type), V(value)

```
public class DBox<L, R> {
    private L left;
    private R right;

    public void setData(L left, R right) {
        this.left = left;
        this.right = right;
    }

    @Override
    public String toString() {
        return this.left + " & " + this.right;
    }
}
```

```
public class MultiTypeParam {
    public static void main(String[] args) {
        DBox<String, Integer> box = new DBox<String,
Integer>();
        box.setData("Apple", 25);
        System.out.println(box);
    }
}
```

제네릭의 이해와 기본 문법

- 기본 자료형에 대한 제한 그리고 래퍼 클래스, 타입 인자의 생략
  - 제네릭 클래스에 대해 Box<Apple>과 같이 매개변수화 타입을 구성할 때 기본 자료형의 이름(int 등) 사용 불가
    - Box<int> box = new Box<int>() : 컴파일 오류 발생
  - 기본 자료형에 대한 래퍼 클래스로 사용 가능
  - 제네릭 관련 문장의 참조 변수를 선언하는 과정에서 오른쪽의 타입 인자 생략 가능

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

```
public class PrimitivesAndGeneric {
    public static void main(String[] args) {
        Box<Integer> iBox = new Box<>();
        iBox.setObj(125);   // 오토 박싱 진행
        int num = iBox.getObj();    // 오토 언박싱 진행
        System.out.println(num);
    }
}
```

제네릭의 이해와 기본 문법

- '매개변수화 타입'을 '타입 인자'로 전달하기
  - 상황 : 상자를 하나 더 생성하여 그 안에 문자열을 저장한 다음 다른 상자에 넣은 뒤 한번 더 다른 상자에 넣음
  - 즉, 하나의 문자열을 세 개의 상자로 겹겹이 포장
  - Box<String>과 같은 '매개변수화 타입'이 '타입 인자'로 사용 될 수 있음

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

```
public class BoxInBox {
    public static void main(String[] args) {
        Box<String> sBox = new Box<>();
        sBox.set("I am so Happy.");

        Box<Box<String>> wBox = new Box<>();
        wBox.set(sBox);

        Box<Box<Box<String>>> zBox = new Box<>();
        zBox.set(wBox);

        System.out.println(zBox.get().get().get());
    }
}
```

제네릭의 이해와 기본 문법

- 제네릭 클래스의 타입 인자 제한하기
  - 앞서 정의한 Box<T>에는 무엇이든 저장 가능하지만, 저장할 것을 제한해야 할 경우 존재
  - extents 키워드 사용
  - 만약 Number 클래스를 상속하는 클래스의 인스턴스만 저장하고 싶다면 아래와 같이 클래스 정의
    - class Box<T extens Number> {...}

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

```
public class BoundedBox {
    public static void main(String[] args) {
        Box<Integer> iBox = new Box<>();
        iBox.setObj(24);

        Box<Double> dBox = new Box<>();
        dBox.setObj(5.97);

        System.out.println(iBox.getObj());
        System.out.println(iBox.getObj());

        Box<String> sBox = new Box<>(); // 오류 발생
    }
}
```

제네릭의 이해와 기본 문법

- 제네릭 클래스의 타입 인자를 인터페이스로 제한하기

```
public interface Eatable {
    public String eat();
}
```

```
public class Box<T extends Eatable> {
    private T obj;
    public T getObj() {
        // Eatable 인터페이스로 제한하였기 때문에 가능
        System.out.println(obj.eat());
        return obj;
    }
    public void setObj(T obj) {
        this.obj = obj;
    }
}
```

```
public class Apple implements Eatable {
    @Override
    public String toString() {
        return "나는 사과입니다.";
    }
    @Override
    public String eat() {
        return "정말 맛있어요.";
    }
}
```

```
public class BoundedInterfaceBox {
    public static void main(String[] args) {
        Box<Apple> box = new Box<>();
        box.setObj(new Apple());
        Apple ap = box.getObj();
        System.out.println(ap);
        Box<Integer> box1 = new Box<>();    // 오류
    }
}
```

- 타입 인자를 제한할 때에는 다음과 같이 하나의 클래스와 하나 이상의 인터페이스에 대해 동시에 제한 가능
  - class Box<T extends Number & Eatable> {...}

제네릭의 이해와 기본 문법

- 제네릭 메소드 정의
  - 클래스가 아닌 일부 메소드에 대해서 제네릭으로 정의할 수 있음
    - public static <T> Box<T> makeBox(T o) {...}
    - 메소드의 이름은 makeBox이고 반환형은 Box<T>
    - static과 Box<T> 사이의 <T>는 T가 타입 매개변수임을 알리는 표시

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

```
public class BoxFactory {
    public static <T> Box<T> makeBox(T o) {
        Box<T> box = new Box<>();
        box.setObj(o);
        return box;
    }
}
```

```
public class GenericMethodBoxMaker {
    public static void main(String[] args) {
        Box<String> sBox = BoxFactory.makeBox("Sweet");
        System.out.println(sBox.getObj());
        Box<Double> dBox = BoxFactory.makeBox(3.14);
        System.out.println(dBox.getObj());
    }
}
```

제네릭의 이해와 기본 문법

- 제네릭 메소드의 제한된 타입 매개변수 선언
  - 클래스가 아닌 일부 메소드에 대해서 제네릭 정의 시 전달되는 타입 인자를 제한할 수 있음

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

```
public class Unboxer {
    // 타입 인자를 Number를 상속하는 클래스로 제한
    public static <T extends Number> T openBox(Box<T> box)
{
        System.out.println("Unboxed data : " +
box.get().intValue());
        return box.get();
    }
}
```

```
public class BoxFactory {
    // <T extends Number>는 타입 인자를 Number를 상속하
는 클래스로 제한함을 의미
    public static <T extends Number> Box<T> makeBox(T o) {
        Box<T> box = new Box<T>();
        box.set(o);
        // 타입 인자 제한으로 inValue 호출 가능
        System.out.println("Boxed data : " + o.intValue());
        return box;
    }
}
```

```
public class BoundedGenericMethod {
    public static void main(String[] args) {
        Box<Integer> sBox = BoxFactory.makeBox(new
Integer(5959));
        int n = Unboxer.openBox(sBox);
        System.out.println("Returned data : " + n);
    }
}
```

제네릭의 이해와 기본 문법

1. 다음 코드가 실행되도록 swapBox 메소드를 정의하되, Box<T> 인스턴스를 인자로 전달받을 수 있도록 정의하자. 단 이때 Box<T> 인스턴스의 T는 Number 또는 이를 상속하는 하위 클래스만 올 수 있도록 제한된 매개변수 선언을 하자.

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

```
public class BoxSwapDemo {
    public static void main(String[] args) {
        Box<Integer> box1 = new Box<>();
        box1.set(99);
        Box<Integer> box2 = new Box<>();
        box2.set(55);
        System.out.println(box1.get() + " & " + box2.get());
        swapBox(box1, box2);    // 정의해야 할 swapBox 메소드
        System.out.println(box1.get() + " & " + box2.get());
    }
    // 이 위치에 swapBox 메소드를 정의하자
}
```

그리고 실행 결과는 다음과 같아야 한다. 즉 wswapBox 메소드의 호출 결과로 인자로 전달된 두 상자안에 저장된 내용물이 서로 바뀌어야 한다.

99 & 55

55 & 99

제네릭의 이해와 기본 문법

1. 앞서 작성한 수납공간이 둘로 나누어져 있는 상자를 표현한 제네릭 클래스를 사용한다. 수납 공간이 둘로 나누어져 있는 상자를 표현한 클래스를 DDBox<U, D>라는 이름으로 하나 더 정의하여 DBox<L, R> 인스턴스 둘을 이 상자에 저장하고자 한다. 그럼 다음 main 메소드를 기반으로 컴파일 및 실행이 가능하도록 DDBox<U, D> 제네릭 클래스를 정의해보자.

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

```
public class BoxSwapDemo {
    public static void main(String[] args) {
        Box<Integer> box1 = new Box<>();
        box1.set(99);
        Box<Integer> box2 = new Box<>();
        box2.set(55);
        System.out.println(box1.get() + " & " + box2.get());
        swapBox(box1, box2);    // 정의해야 할 swapBox 메소드
        System.out.println(box1.get() + " & " + box2.get());
    }
    // 이 위치에 swapBox 메소드를 정의하자
}
```

그리고 위 main 메소드의 실행 결과로 다음의 출력을 보이게 하자. "Apple & 24", "Orange & 33"

2. 문제 1의 내용에 해당하는 프로그램은 사실 별도의 클래스를 정의하지 않고 Dbox 하나로 완성할 수 있다. 따라서 이번에는 문제 1의 내용과 결과를 보이는 프로그램을 작성하되 DBox 클래스 하나만 활용하여 작성해보자.