

제네릭

제네릭의 이해

제네릭 이전의 코드

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

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

// 다음 상자는 사과도 오렌지도 담을 수 있다.

```
class Box {    // 무엇이든 저장하고 꺼낼 수 있는 상자  
    private Object ob;  
  
    public void set(Object o) { ob = o; }  
    public Object get() {return ob; }  
}
```

제네릭 이전의 코드의 사용의 예

```
public static void main(String[] args) {  
    Box aBox = new Box();    // 상자 생성  
    Box oBox = new Box();    // 상자 생성  
  
    aBox.set(new Apple());    // 상자에 사과를 담는다.  
    oBox.set(new Orange());   // 상자에 오렌지를 담는다.  
  
    Apple ap = (Apple)aBox.get();    // 상자에서 사과를 꺼낸다.  
    Orange og = (Orange)oBox.get();  // 상자에서 오렌지를 꺼낸다.  
  
    System.out.println(ap);  
    System.out.println(og);  
}
```

어쩔 수 없이 형 변환의 과정이 수반된다.

그리고 이는 컴파일러의 오류 발견 가능성을 낮추는 결과로 이어진다.

명령 프롬프트

```
C:\JavaStudy>java FruitAndBox2  
I am an apple.  
I am an orange.  
  
C:\JavaStudy>
```

제네릭 이전 코드가 갖는 문제점 1

프로그래머의 실수가 컴파일 과정에서 발견되지 않는다.

```
public static void main(String[] args) {
```

```
    Box aBox = new Box();
```

```
    Box oBox = new Box();
```

```
    // 아래 두 문장에서는 사과와 오렌지가 아닌 '문자열'을 담았다.
```

```
    aBox.set("Apple");
```

```
    oBox.set("Orange");
```

```
    // 상자에 과일이 담기지 않았는데 과일을 꺼내려 한다.
```

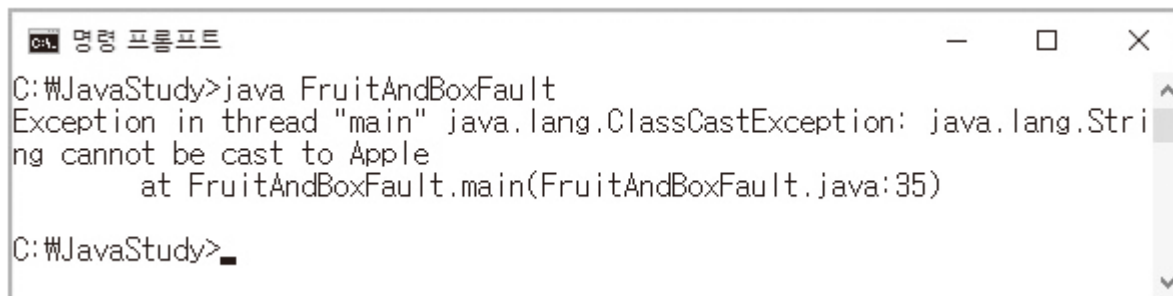
```
    Apple ap = (Apple)aBox.get();
```

```
    Orange og = (Orange)oBox.get();
```

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

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

```
}
```

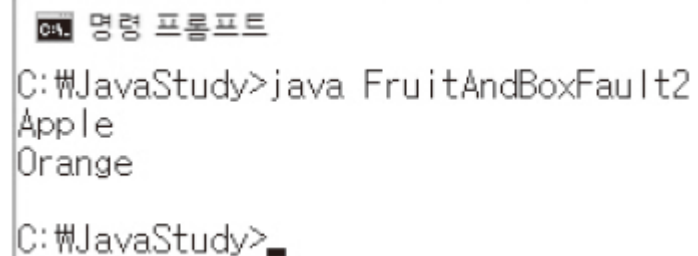


```
명령 프롬프트
C:\JavaStudy>java FruitAndBoxFault
Exception in thread "main" java.lang.ClassCastException: java.lang.String
cannot be cast to Apple
    at FruitAndBoxFault.main(FruitAndBoxFault.java:35)
C:\JavaStudy>
```

제네릭 이전 코드가 갖는 문제점 2

프로그래머의 실수가 실행 과정에서 조차 발견되지 않을 수 있다. 정말 큰 문제!!!

```
public static void main(String[] args) {  
    Box aBox = new Box();  
    Box oBox = new Box();  
  
    // 다음 두 문장은 프로그래머의 실수이다!  
    aBox.set("Apple");  
    oBox.set("Orange");  
  
    System.out.println(aBox.get());  
    System.out.println(oBox.get());  
}
```



```
명령 프롬프트  
C:\JavaStudy>java FruitAndBoxFault2  
Apple  
Orange  
C:\JavaStudy>
```

제네릭 기반의 클래스 정의하기

인스턴스 생성시 결정이 되는 자료형의 정보를 T로 대체한다.

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



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

제네릭 클래스 기반 인스턴스 생성

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

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

```
Box<Apple> aBox = new Box<Apple>();  
→ T를 Apple로 결정하여 인스턴스 생성  
→ 따라서 Apple 또는 Apple을 상속하는 하위 클래스의 인스턴스 저장 가능
```

```
Box<Orange> oBox = new Box<Orange>();  
→ T를 Orange로 결정하여 인스턴스 생성  
→ 따라서 Orange 또는 Orange를 상속하는 하위 클래스의 인스턴스 저장 가능
```


제네릭 이후의 코드: 개선된 결과

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

public static void main(String[] args) {
    Box<Apple> aBox = new Box<Apple>();    // T를 Apple로 결정
    Box<Orange> oBox = new Box<Orange>();  // T를 Orange로 결정

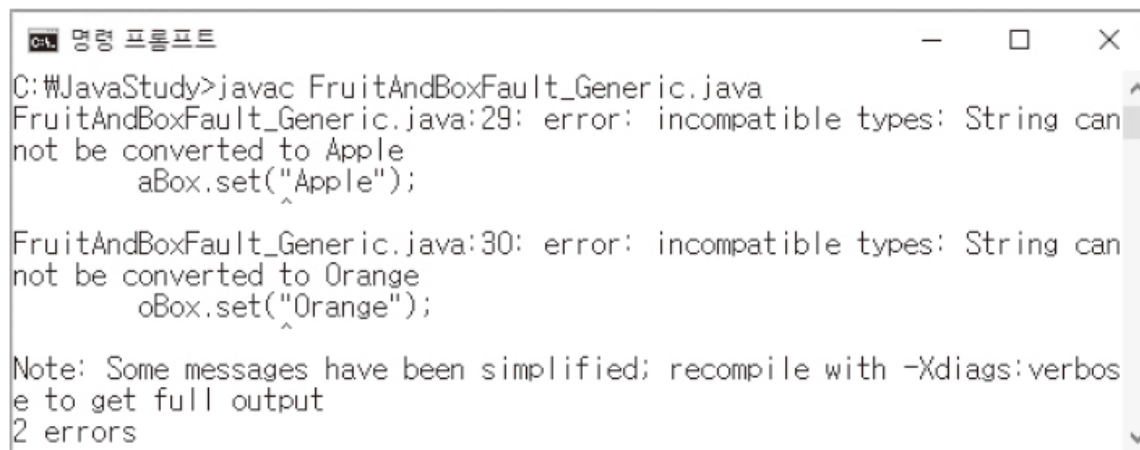
    aBox.set(new Apple());    // 사과를 상자에 담는다.
    oBox.set(new Orange());    // 오렌지를 상자에 담는다.

    Apple ap = aBox.get();    // 사과를 꺼내는데 형 변환 하지 않는다.
    Orange og = oBox.get();    // 오렌지를 꺼내는데 형 변환 하지 않는다.

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

실수가 컴파일 오류로 이어진다.

```
public static void main(String[] args) {  
    Box<Apple> aBox = new Box<Apple>();  
    Box<Orange> oBox = new Box<Orange>();  
  
    aBox.set("Apple");    // 프로그래머의 실수  
    oBox.set("Orange");   // 프로그래머의 실수  
  
    Apple ap = aBox.get();  
    Orange og = oBox.get();  
  
    System.out.println(ap);  
    System.out.println(og);  
}
```



```
C:\JavaStudy>javac FruitAndBoxFault_Generic.java  
FruitAndBoxFault_Generic.java:29: error: incompatible types: String cannot  
be converted to Apple  
    aBox.set("Apple");  
           ^  
FruitAndBoxFault_Generic.java:30: error: incompatible types: String cannot  
be converted to Orange  
    oBox.set("Orange");  
           ^  
Note: Some messages have been simplified; recompile with -Xdiags:verbose  
to get full output  
2 errors
```

제네릭의 기본 문법

다중 매개변수 기반 제네릭 클래스의 정의

```
class DBox<L, R> {  
    private L left;    // 왼쪽 수납 공간  
    private R right;   // 오른쪽 수납 공간  
  
    public void set(L o, R r) {  
        left = o;  
        right = r;  
    }  
  
    @Override  
    public String toString() {  
        return left + " & " + right;  
    }  
}
```

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



타입 매개 변수의 이름 규칙

일반적인 관례

한 문자로 이름을 짓는다.

대문자로 이름을 짓는다.

보편적인 선택

E Element

K Key

N Number

T Type

V Value

기본 자료형에 대한 제한 그리고 래퍼 클래스

```
class Box<T> {  
    private T ob;
```

```
Box<int> box = new Box<int>();
```

```
    public void set(T o) {  
        ob = o;
```

→ 타입 인자로 기본 자료형이 올 수 없으므로 컴파일 오류 발생

```
    }
```

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

```
}
```

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

다이아몬드 기호

따라서 다음 문장을 대신하여,

```
Box<Apple> aBox = new Box<Apple>();
```

다음과 같이 쓸 수 있다.

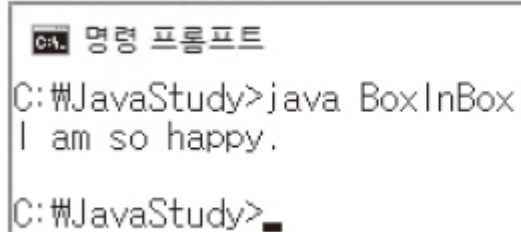
```
Box<Apple> aBox = new Box<>();
```

참조변수 선언을 통해서 컴파일러가 사이에 Apple이 와야 함을 유추한다.

'매개변수화 타입'을 '타입 인자'로 전달

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

```
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());  
}
```



```
C:\JavaStudy>java BoxInBox  
I am so happy.  
C:\JavaStudy>
```


제네릭 클래스의 타입 인자 제한하기

```
class Box<T extends Number> {...}
```

→ 인스턴스 생성 시 타입 인자로 Number 또는 이를 상속하는 클래스만 올 수 있음

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

```
public static void main(String[] args) {  
    Box<Integer> iBox = new Box<>();  
    iBox.set(24);  
  
    Box<Double> dBox = new Box<>();  
    dBox.set(5.97);  
    . . . .  
}
```

타입 인자 제한의 효과

```
class Box<T> {  
    private T ob;  
    ....  
    public int toIntValue() {  
        return ob.intValue(); // ERROR!  
    }  
}
```

```
class Box<T extends Number> {  
    private T ob;  
    ....  
    public int toIntValue() {  
        return ob.intValue(); // OK!  
    }  
}
```

이 효과가 타입 인자를 제한하는 실질적인 이유인 경우가 많다.

이 내용 조금 중요한 편에 속합니다. ^^

제네릭 클래스의 타입 인자를 인터페이스로 제한하기

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

```
class Apple implements Eatable {  
    public String eat() {  
        return "It tastes so good!";  
    }  
    . . .  
}
```

```
class Box<T extends Eatable> {  
    T ob;  
  
    public void set(T o) { ob = o; }  
    public T get() {  
        System.out.println(ob.eat());    // Eatable로 제한하였기에 eat 호출 가능  
        return ob;  
    }  
}
```

하나의 클래스와 하나의 인터페이스에 대해 동시 제한

```
class Box<T extends Number & Eatable> {...}
```

Number는 클래스 이름 Eatable은 인터페이스 이름

제네릭 메소드의 정의

클래스 전부가 아닌 메소드 하나에 대해 제네릭으로 정의

```
class BoxFactory {  
    public static <T> Box<T> makeBox(T o) {  
        Box<T> box = new Box<T>();    // 상자를 생성하고,  
        box.set(o);    // 전달된 인스턴스를 상자에 담아서,  
        return box;    // 상자를 반환한다.  
    }  
}
```

제네릭 메소드의 T는 메소드 호출 시점에 결정한다.

```
Box<String> sBox = BoxFactory.<String>makeBox("Sweet");  
Box<Double> dBox = BoxFactory.<Double>makeBox(7.59);    // 7.59에 대해 오토 박싱 진행됨
```

다음과 같이 타입 인자 생략 가능

```
Box<String> sBox = BoxFactory.makeBox("Sweet");  
Box<Double> dBox = BoxFactory.makeBox(7.59);    // 7.59에 대해 오토 박싱 진행됨
```

제네릭 메소드의 제한된 타입 매개변수 선언

// <T extends Number>는 타입 인자를 Number를 상속하는 클래스로 제한함을 의미

```
public static <T extends Number> Box<T> makeBox(T o) {
```

```
    ....
```

```
    // 타입 인자 제한으로 intValue 호출 가능
```

```
    System.out.println("Boxed data: " + o.intValue());
```

```
    return box;
```

```
}
```

// 타입 인자를 Number를 상속하는 클래스로 제한

```
public static <T extends Number> T openBox(Box<T> box) {
```

```
    // 타입 인자 제한으로 intValue 호출 가능
```

```
    System.out.println("Unboxed data: " + box.get().intValue());
```

```
    return box.get();
```

```
}
```