# Generics

## 지네릭스(Generics)

- JDK1.5 에 처음 도입.
- Generics add stability to your code by making more of your bugs detectable at compile time. - Oracle Javadoc
- 제네릭(Generic)은 클래스 내부에서 사용할 데이터 타입을 외부에서 지정하는 기법을 의미한다. - 생활코딩
- 지네릭스는 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일 시의 타입체크를 해주는 기능이다. - 자바의 정석
- 데이터 형식에 의존하지 않고, 하나의 값이 여러 다른 데이터 타입들을 가질 수 있도록 하는 방법'

### Generics의 장점

1. 타입 안정성을 제공.
2. 타입체크와 형변환을 생략할 수 있으므로 코드가 보다 간결.
3. 제네릭을 사용하면 잘못된 타입이 들어올 수 있는 것을 컴파일 단계에서 방지 가능.
4. 클래스 외부에서 타입을 지정해주기 때문에 따로 타입을 체크하고 변환해줄 필요가 없다. 즉, 관리하기가 편하다.
5. 비슷한 기능을 지원하는 경우 코드의 재사용성이 높아진다.

# 지네릭스(Generics)

우리가 흔히 쓰는 ArrayList, LinkedList 등을 생성할 때
* 객체〈타입〉 객체명 = new 객체〈타입〉();
* 〈 〉 괄호 안에 들어가는 타입을 지정해준다

```java
public class Generic_ex1 {

    ArrayList<Integer> list1 = new ArrayList<Integer>();
    ArrayList<String> list2 = new ArrayList<>();

    LinkedList<Double> list3 = new LinkedList<Double>();
    LinkedList<Character> list4 = new LinkedList<Character>();

    /*
    * 우리가 흔히 쓰는 ArrayList, LinkedList 등을 생성할 때
    * 객체<타입> 객체명 = new 객체<타입>(); 이렇게 쓴다. 즉, 위와와 같이 여러 생성방식이 있다.
    */
```

## 지네릭스(Generics)

우리가 흔히 쓰는 ArrayList, LinkedList 등을 생성할 때
* 객체〈타입〉 객체명 = new 객체〈타입〉();
* 〈 〉 괄호 안에 들어가는 타입을 지정해준다

```
ArrayList list = new ArrayList(); //제네릭을 사용하지 않을경우
list.add("test");
String temp = (String) list.get(0); //타입변환이 필요함

ArrayList<String> list2 = new ArrayList(); //제네릭을 사용할 경우
list2.add("test");
temp = list2.get(0); //타입변환이 필요없음
```

# 지네릭스(Generics)

사용 이유

1. String 타입도 지원하고 싶고 Integer타입도 지원하고 싶고 많은 타입을 지원하고 싶다.
   그러면 String에 대한 클래스, Integer에 대한 클래스 등 하나하나 타입에 따라 만들 것인가?

```
ArrayList<Integer> list1 = new ArrayList<Integer>();
ArrayList<String> list2 = new ArrayList<>();
```

그건 너무 비효율적이다. 이러한 문제를 해결하기 위해 우리는 제네릭이라는 것을 사용한다.

2. 이렇듯 제네릭(Generic)은 클래스 내부에서 지정하는 것이 아닌 외부에서 사용자에 의해 지정되는 것을 의미한다.
   한마디로 특정(Specific) 타입을 미리 지정해주는 것이 아닌 필요에 의해 지정할 수 있도록 하는 일반(Generic) 타입이라는 것이다.

3. 정확히 말하자면 지정된다는 것 보다는 타입의 경계를 지정하고,
   컴파일 때 해당 타입으로 캐스팅하여 매개변수화 된 유형을 삭제하는 것이다.

# 지네릭스(Generics)

## Generics class의 선언

기호의 종류만 다를 뿐, 모두 **'임의의 참조형 타입'** 을 의미하는 것.

대중적으로 통하는 통상적인 선언

타입변수가 여러 개인 경우에는 , 를 구분자로 나열하면 된다.
Ex) Map(K,V)

| 타입 | 설명 |
|---|---|
| <T> | Type |
| <E> | Element |
| <K> | Key |
| <V> | Value |
| <N> | Number |

# 지네릭스(Generics)

## Generics class의 선언

Generics class의 객체를 선언할 때는 참조변수와 생성자에
타입 T대신에 사용될 실제 타입을 지정해주어야 한다.

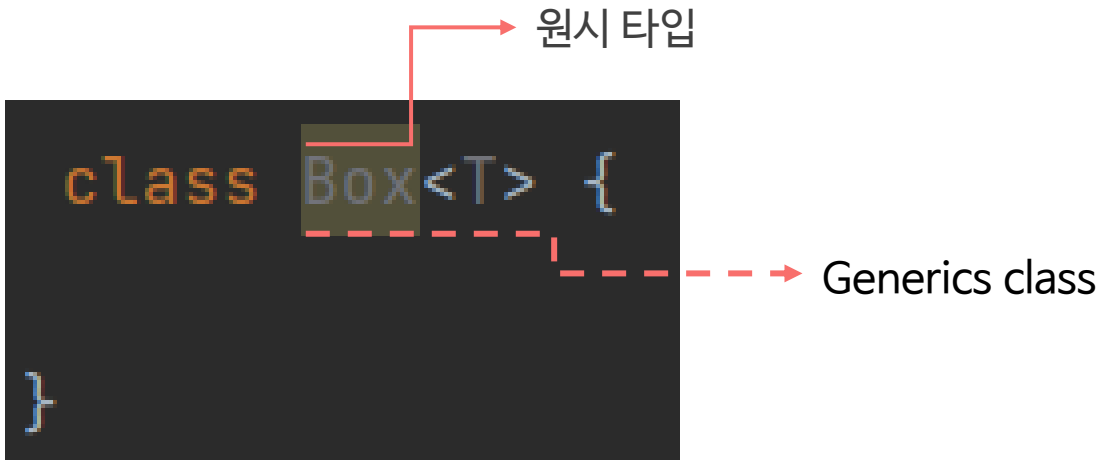
```
public class Generic_ex2 {

    public static class ClassName <T, K> { ... }

        public static void main(String[] args) {
            ClassName<String, Integer> a = new ClassName<String, Integer>();
        }
```

# 지네릭스(Generics)

## Generics 용어




- Box⟨T⟩ : 지네릭 클래스. ‘T의 Box’ 또는 ‘T Box’ 라고 읽는다.
- T : 타입변수 또는 타입 매개변수. (T는 타입 문자)
- Box : 원시타입 (raw type)

## 지네릭스(Generics)의 제한

- 지네릭 클래스의 객체를 생성할 때, 객체별로 다른 타입을 지정하는 것은 적절하다.

- 지네릭스는 인스턴스별로 다르게 동작하도록 하려고 만든 기능!

```
Box<Apple> appleBox = new Box<Apple>();
Box<Grape> grapeBox = new Box<Grape>();
```

- static 멤버에 타입 변수 T는 사용 불가.
  -〉 T는 인스턴스 변수로 간주되기 때문.
  -〉 static 멤버는 모든 객체에 대해 동일하게 동작해야 하기 때문.
  -〉 static 멤버는 인스턴스 변수를 참조할 수 없다.
  -〉 static 멤버는 타입변수에 지정된 타입, -〉 대입된 타입의 종류에 관계없이 동일한 것이어야 한다.

```
class Box<T> {
  static T item;  // 에러
  static int compare(T t1, T t2) { // 에러
    //...
  }
}
```

## 지네릭스(Generics)의 제한

- 제네릭 배열 타입의 참조변수를 선언은 가능하지만 제네릭 타입의 배열의 생성을 허용하지 않는다.
- 제네릭 배열을 생성할 수 없는 것은 new연산자 때문인데, 이 연산자는 컴파일 시점에 타입 T가 뭔지 정확히 알아야한다.
  그런데 아래의 코드에 정의된 Box〈T〉클래스를 컴파일하는 시점에서는 T가 어떤 타입이 될지 전혀 알 수 없다.
- instanceof연산자도 new연산자와 같은 이유로 T를 피연산자로 사용할 수 없다.

```
class Box<T> {
  T[] itemArr; // OK T타입의 배열을 위한 참조변수
  T[] toArray() {
    T[] tempArr = new T[itemArr.length]; // 에러, 제네릭 배열 생성불가
    return tempArr;
  }
}
```

## Generics클래스의 객체 생성과 사용

- ArrayList를 이용해 여러 객체를 저장할 수 있도록 하였다.

```
class Box<T> {
  ArrayList<T> list = new ArrayList<T>();

  void add(T item) {
    list.add(item);
  }

  T get(int i) {
    return list.get(i);
  }

  ArrayList<T> getList() {
    return list;
  }

  int size() {
    return list.size();
  }

  public String toString() {
    return list.toString();
  }
}
```

## Generics클래스의 객체 생성과 사용

- Box<T>의 객체를 생성할 때는 위와 같이 작성하면 된다.
- 참조변수와 생성자에 대입된 타입이 일치해야 한다.
- 만약 일치하지 않으면 에러가 발생한다.

```
Box<Apple> appleBox = new Box<Apple>(); // OK, 타입 일치
Box<Apple> appleBox = new Box<Grape>(); // 에러, 타입 불일치
```

## Generics클래스의 객체 생성과 사용

- Apple이 Fruit에 자손이라는 가정. Fruit 〉 Apple
- 상속관계에 있어도 타입이 일치 해야만 한다.
- Apple이 Fruit에 자손임에도 에러가 발생한다.

```
Box<Fruit> appleBox = new Box<Apple>(); // 에러, 대입된 타입이 다르다
```

## Generics클래스의 객체 생성과 사용

- Box 〉 FruitBox 라는 가정
- 두 제네릭 클래스의 타입이 상속관계에 있고, 대입된 타입이 같을 경우는 허용 된다.
- FruitBox는 Box의 자손이고, 참조변수의 타입과 생성자에 대입된 타입이 Apple과 동일하다

```
Box<Apple> appleBox = new FruitBox<Apple>(); // OK, 타입일치, 두 제네릭 클래스의 타입이 상속관계
```

## Generics클래스의 객체 생성과 사용

- JDK1.7부터 추정이 가능한 경우 타입을 생략 가능.
- 참조변수의 타입으로부터 Box가 Apple타입의 객체만 저장한다는 것을 알 수 있기 때문에,
  생성자에 반복해서 타입을 지정해주지 않아도 된다.

```
Box<Apple> appleBox = new Box<Apple>();
Box<Apple> appleBox = new Box<>(); // OK, JDK1.7부터 생략 가능
```

## Generics클래스의 객체 생성과 사용

- 메서드의 매개변수에 대입된 타입과 다른 타입의 객체를 추가할 수 없다.

```
Box<Apple> appleBox = new Box<Apple>();
appleBox.add(new Apple());  // OK
appleBox.add(new Greape()); // 에러, Box<Apple>에는 Apple객체만 추가 가능
```

- 메서드의 매개변수에 대입된 타입의 자손의 객체는 추가할 수 있다.

```
Box<Fruit> FruitBox = new Box<Fruit>();
fruitBox.add(new Fruit());  // OK
fruitBox.add(new Apple());  // OK, void add(Fruit item);
```

# 제네릭 클래스의 객체 생성과 사용 예제 1

- InteliJ

```
public class FruitBoxEx1 {
    public static void main(String[] args) {
        Box<Fruit> fruitBox = new Box<Fruit>();
        Box<Apple> appleBox = new Box<Apple>();
        Box<Toy> toyBox = new Box<Toy>();
        //Box<Grape> grapeBox = new Box<Fruit>(); // 에러, 타입불일치

        fruitBox.add(new Fruit());
        fruitBox.add(new Apple());

        appleBox.add(new Apple());
        appleBox.add(new Apple());
        //appleBox.add(new Toy()); // 에러 Box<Apple>에는 Apple만 담을 수 있음

        toyBox.add(new Toy());
        //toyBox.add(new Apple()); // 에러 Box<Toy>에는 Apple을 담을 수 없음

        System.out.println(fruitBox);
        System.out.println(appleBox);
        System.out.println(toyBox);
    }
}
```

# 제한된 Generics클래스

```java
public class Generic_ex4 {

    // Fruit의 자손만 타입으로 지정 가능
    // 위와 같이 제네릭 타입에 extends를 사용 하면 특정 타입의 자손들만 대입할 수 있게 제한할 수 있다.
    class FruitBox<T extends Fruit> {
        ArrayList<T> list = new ArrayList<T>();
    }



    // 위와 같이 제네릭 타입에 extends를 사용 하면 특정 타입의 자손들만 대입할 수 있게 제한할 수 있다.
    FruitBox<Apple> appleBox = new FruitBox<Apple>(); // OK, Apple은 Fruit의 자손이기 때문에 가능
    FruitBox<Toy> toyBox = new FruitBox<Toy>(); // 에러, Toy는 Fruit의 자손이 아니기 때문 불가능
```

## 제한된 Generics 클래스 예제 1

- InteliJ

```
public class FruitBoxEx2 {
    public static void main(String[] args) {
        FruitBox2<Fruit2> fruitBox = new FruitBox2<Fruit2>();
        FruitBox2<Apple2> appleBox = new FruitBox2<Apple2>();
        FruitBox2<Grape2> grapeBox = new FruitBox2<Grape2>();
        //FruitBox2<Grape2> grapeBox = new FruitBox2<Apple2>(); // 에러, 타입 불일치
        //FruitBox2<Toy2> toyBox = new FruitBox2<Toy2>(); // 에러

        fruitBox.add(new Fruit2());
        fruitBox.add(new Apple2());
        fruitBox.add(new Grape2());
        appleBox.add(new Apple2());
        //appleBox.add(new Grape2()); // 에러, Grape는 Apple의 자손이 아님
        grapeBox.add(new Grape2());

        System.out.println("fruitBox = " + fruitBox);
        System.out.println("appleBox = " + appleBox);
        System.out.println("grapeBox = " + grapeBox);
    }
}
```

## 와일드카드

- 제네릭 타입을 매개변수나 반환 타입으로 사용할 때 구체적인 타입 대신 와일드 카드를 다음과 같이 세 가지 형태로 사용할 수 있다.
- 〈? extends T〉 : 와일드 카드의 상한제한, T와 그의 자손들만 가능
- 〈? super T〉 : 와일드 카드의 하한제한, T와 그의 조상들만 가능
- 〈?〉 : 제한 없음, 모든 타입이 가능, 〈? extends Object〉와 동일
- 제네릭 클래스와 달리 와일드 카드에는 &를 사용할 수 없음

## 와일드카드

- 제네릭 타입을 매개변수나 반환 타입으로 사용할 때 구체적인 타입 대신 와일드 카드를 다음과 같이 세 가지 형태로 사용할 수 있다.
- 〈? extends T〉 : 와일드 카드의 상한제한, T와 그의 자손들만 가능
- 〈? super T〉 : 와일드 카드의 하한제한, T와 그의 조상들만 가능
- 〈?〉 : 제한 없음, 모든 타입이 가능, 〈? extends Object〉와 동일
- 제네릭 클래스와 달리 와일드 카드에는 &를 사용할 수 없음