제네릭(Generic) 타입

Contents

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭(Generic) 이란?

- 결정되지 않은 타입을 파라미터로 처리하고
  실제 사용할 때 파라미터를 구체적인 타입으로 대체시키는 기능

- 자바5부터 새로 추가

- 컬렉션, 람다식(함수적 인터페이스), 스트림 등에서 널리 사용

- 제네릭을 모르면 API 도큐먼트 해석 어려우므로 학습 필요

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭을 사용하는 코드의 이점

- 컴파일 시 강한 타입 체크 가능
  - 실행 시 타입 에러가 나는 것 방지
  - 컴파일 시에 미리 타입을 강하게 체크해서 에러 사전 방지

- 리스트에 데이터 저장

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0);
```

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭을 사용하는 코드의 이점

- 컴파일 시 강한 타입 체크 가능
  - 실행 시 타입 에러가 나는 것 방지
  - 컴파일 시에 미리 타입을 강하게 체크해서 에러 사전 방지

- 리스트에 데이터 저장

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0);
```

제너릭 이용 시 타입 변환 생략

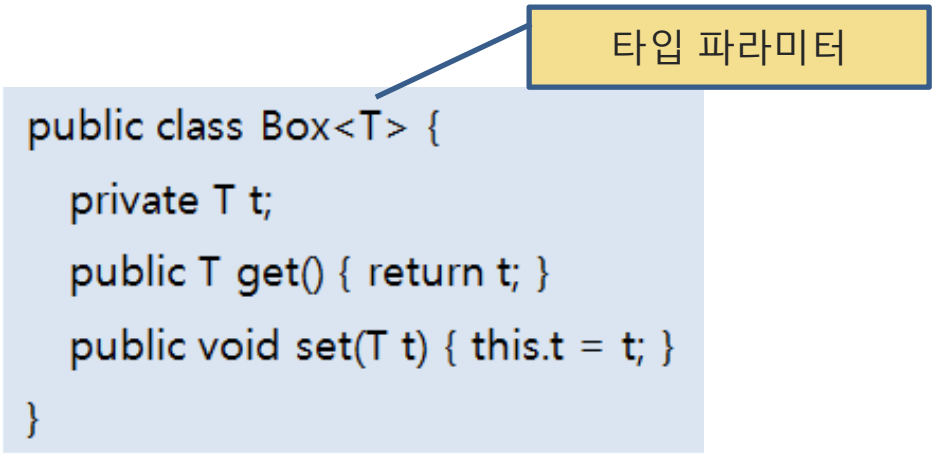
```
List<String> list = new ArrayList<String>();
list.add("hello");
String str = list.get(0);
```

2절. 제네릭 타입

❖ 제네릭 타입이란?

- 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언 시 클래스 또는 인터페이스 이름 뒤에 “<>” 부호 붙임
- “<>” 사이에는 타입 파라미터 위치

- 타입 파라미터
  - 일반적으로 대문자 알파벳 한 문자로 표현
  - 개발 코드에서는 타입 파라미터 자리에 구체적인 타입을 지정해야

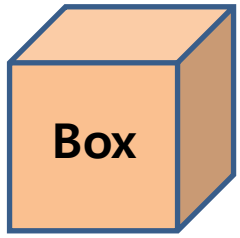


2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입을 사용하지 않은 경우
  - Object 타입 사용 → 빈번한 타입 변환 발생 → 프로그램 성능 저하

```
public class Box {
   private Object object;
   public void set(Object object) { this.object = object; }
   public Object get() { return object; }
}
```




```
Box box = new Box();
box.set("hello");                  //String 타입을 Object 타입으로 자동 타입 변환해서 저장
String str = (String) box.get();   //Object 타입을 String 타입으로 강제 타입 변환해서 얻음
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

▪ 제네릭 타입 사용한 경우

- 클래스 선언할 때 타입 파라미터 사용
- 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

```
public class Box<T> {
    private T t;
    public T get() { return t; }
    public void set(T t) { this.t = t; }
}
```

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

```
public class Box<String> {
    private String t;
    public void set(String t) { this.t = t; }
    public String get() { return t; }
}
```

```
Box<String> box = new Box<String>();
box.set("hello");
String str = box.get();
```

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

```
public class Box<Integer> {
    private Integer t;
    public void set(Integer t) { this.t = t; }
    public Integer get() { return t; }
}
```

```
Box<Integer> box = new Box<Integer>();
box.set(6);
int value = box.get();
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입 사용한 경우
  - 클래스 선언할 때 타입 파라미터 사용
  - 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

```
public class Box<T> {
    private T t;
    public T get() { return t; }
    public void set(T t) { this.t = t; }
}
```

- 타입 파라미터

Byte / Integer / Character
Float / Double / Boolean

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

```
public class Box<String> {
    private String t;
    public void set(String t) { this.t = t; }
    public String get() { return t; }
}
```

```
Box<String> box = new Box<String>();
box.set("hello");
String str = box.get();
```

타입 파라미터 = 클래스, 인터페이스

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

```
public class Box<Integer> {
    private Integer t;
    public void set(Integer t) { this.t = t; }
    public Integer get() { return t; }
}
```

```
Box<Integer> box = new Box<Integer>();
box.set(6);
int value = box.get();
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

▪ 제네릭 타입 사용한 경우

- 클래스
- 컴파일

```
public class BoxEx {
    public static void main(String[] args) {
        Box<String> box = new Box<String>();

        box.setT("Hello");
        String str = box.getT();

        System.out.println(str);

        Box<Integer> box2 = new Box<Integer>();
        box2.setT(100);
        System.out.println(box2.getT());

    }
}
```

```
public class Box<T> {
   private T t;
   public T get() { retur
   public void set(T t) {
}
```

```
v Box<String>();
```

```
   public Integer get() { return t; }
}
```

```
Box<Integer> box = new Box<Integer>();
box.set(6);
int value = box.get();
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 여부에 따른 비교

- 제네릭 타입 사용한 경우
  - 클래스 선언할 때 타입 파라미터 사용
  - 컴파일 시 타입 파라미터가 구체적인 클래스로 변경

제네릭(Generic)이란?
결정되지 않은 타입을 파라미터로 처리하고
실제 사용할 때 파라미터를 구체적인 타입으로 대체시키는 기능

2절. 제네릭 타입

❖ 제네릭 타입 실습

- 데이터를 입력받아 출력하는 메소드 선언
  - 데이터 타입마다 다른 메소드 정의 => 메소드 오버로딩

```
public void printValue(int value) {
    System.out.println("value = " + value);
}

public void printValue(String value) {
    System.out.println("value = " + value);
}

public void printValue(boolean value) {
    System.out.println("value = " + value);
}
```

2절. 제네릭 타입

❖ 제네릭 타입 실습

- 데이터를 입력받아 출력하는 메소드 선언
  - 데이터 타입을 제너릭으로 표현

```
public class MyClass<T> {
    public void printValue(T value) {
        System.out.println("value = " + value);
    }

}
```

2절. 제네릭 타입

❖ 제네릭 타입 실습

- 데이터를 입력받아 출력하는 메소드 선언
  - 데이터 타입을 제너릭으로 표현

```
public class MyClass<T> {
    public void printValue(T value) {
        System.out.println("value = " + value);
    }

}
```

```
public class MyClassEx {
    public static void main(String[] args) {
        MyClass<Integer> intClass = new MyClass<Integer>();
        intClass.printValue(100);

        MyClass<String> strClass = new MyClass<String>();
        strClass.printValue("홍길동");

        MyClass<Boolean> boolClass = new MyClass<Boolean>();
        boolClass.printValue(true);

    }
}
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 – 인터페이스

- Rentable 인터페이스를 제네릭 타입으로 선언
  - 렌트하기 위해 rent() 추상 메소드 선언
  - 다양한 제품을 렌트하기 위해 리턴 타입을 타입 파라미터로 선언

2절. 제네릭 타입

❖ 제네릭 타입 사용 – 인터페이스

- Rentable 인터페이스를 제네릭 타입으로 선언
  - 렌트하기 위해 rent() 추상 메소드 선언
  - 다양한 제품을 렌트하기 위해 리턴 타입을 타입 파라미터로 선언

```
public class Home {
    public void turnOnLight() {
        System.out.println("전등을 켭니다.");
    }
}
```

```
public class Car {
    public void run() {
        System.out.println("자동차가 달립니다.");
    }
}
```

2절. 제네릭 타입

❖ 제네릭 타입 사용 – 인터페이스

- Rentable 인터페이스를 제네릭 타입으로 선언
  - 렌트하기 위해 rent() 추상 메소드 선언
  - 다양한 제품을 렌트하기 위해 리턴 타입을 타입 파라미터로 선언

```
public interface Rentable<P> {
    P rent();
}
```

```
public class HomeAgency implements Rentable<Home>{
    @Override
    public Home rent() {
        return new Home();
    }
}
```

```
public class CarAgency implements Rentable<Car> {
    @Override
    public Car rent() {
        return new Car();
    }
}
```

```
public class RentEx {
    public static void main(String[] args) {
        HomeAgency homeA = new HomeAgency();
        Home home = homeA.rent();
        home.turnOnLight();

        CarAgency carA = new CarAgency();
        Car car = carA.rent();
        car.run();
    }
}
```

3절. 멀티 타입 파라미터

❖ 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능

- 각 타입 파라미터는 콤마로 구분
  - **Ex) class<K, V, …> { … }**
  - **interface<K, V, …> { … }**

- **E** : Element(컬렉션 요소)
- **T** : Type
- **V** : Value
- **K** : Key

```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() { return this.kind; }
    public M getModel() { return this.model; }

    public void setKind(T kind) { this.kind = kind; }
    public void setModel(M model) { this.model = model; }
}
```

```
Product<Tv, String> product = new Product<Tv, String>();
```

- 자바 7부터는 다이아몬드 연산자 사용해 간단히 작성과 사용 가능

```
Product<Tv, String> product = new Product<>();
```

3절. 멀티 타입 파라미터

❖ 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능

```
public class Product<T, M> {
    private T kind;
    private M model;

    public T getKind() {
        return kind;    }
    public M getModel() {
        return model;   }
    public void setModel(M model) {
        this.model = model; }
    public void setKind(T kind) {
        this.kind = kind;   }
}

class Tv{
    public void tvPrn() {
        System.out.println("TV 종류");    }
}
class Car{
    public void carPrn() {
        System.out.println("Car 종류");   }
}
```

```
public class ProductEx {
    public static void main(String[] args) {
        Product<Tv, String> prod1 = new Product<>();
        prod1.setKind(new Tv());
        prod1.setModel("스마트TV");

        Tv tv = prod1.getKind();
        String tvModel = prod1.getModel();
        tv.tvPrn();
        System.out.println("==> " + tvModel);

        Product<Car, String> prod2 = new Product<>();
        prod2.setKind(new Car());
        prod2.setModel("그랜져");

        Car car = prod2.getKind();
        String carModel = prod2.getModel();
        car.carPrn();
        System.out.println("==> " + carModel);
    }
}
```

4절. 제네릭 메소드

❖ 제네릭 메소드

- 매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드
- 제네릭 메소드 선언 방법
  - 리턴 타입 앞에 “<>” 기호를 추가하고 타입 파라미터 기술
  - 타입 파라미터를 리턴 타입과 매개변수에 사용

```
public <타입파라미터,...> 리턴타입 메소드명(매개변수,...) { ... }
```

```
public <T> Box<T> boxing(T t) { ... }
```

- 제네릭 메소드 호출하는 두 가지 방법

```
리턴타입 변수 = <구체적타입> 메소드명(매개값);      //명시적으로 구체적 타입 지정
리턴타입 변수 = 메소드명(매개값);                  //매개값을 보고 구체적 타입을 추정
```

```
Box<Integer> box = <Integer>boxing(100);      //타입 파라미터를 명시적으로 Integer 로 지정
Box<Integer> box = boxing(100);               //타입 파라미터를 Integer 으로 추정
```

4절. 제네릭 메소드

❖ 제네릭 메소드

```
public class Util {
    public static <T> Box<T> boxing(T t){
        Box<T> box = new Box<T>();
        box.setT(t);
        return box;
    }
}
class Box<T>{
    private T t;

    public T getT() { return t; }
    public void setT(T t) { this.t = t; }
}
```

```
public class UtilEx {
    public static void main(String[] args) {
        Box<Integer> box1 = Util.boxing(100);
        int intVal = box1.getT();
        System.out.println("box1 = " + intVal);

        Box<String>  box2 = Util.boxing("홍길동");
        String strVal = box2.getT();
        System.out.println("box2 = " + strVal);
    }
}
```