13. 제네릭

과목명 : 애플리케이션 개발(JAVA)

Contents

1절. 왜 제네릭을 사용해야 하는가?

❖ 제네릭(Generic) 타입이란?

- '컴파일 단계'에서 '잘못된 타입 사용될 수 있는 문제'제거 가능
- 자바5부터 새로 추가 !
- 컬렉션, 람다식(함수적 인터페이스), 스트림, NIO에서 널리 사용
- 제네릭을 모르면 API 도큐먼트 해석 어려우므로 학습 필요

Class ArrayList<E>

```
default BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after)
```

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
- 타입변환 제거 가능

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

```
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();
```

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, ...> { ... }

```
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<>();
```

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 (p.661~664)

```
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
```

5절. 제한된 타입 파라미터

❖ 타입 파라미터에 지정되는 구체적인 타입 제한할 필요

- 상속 및 구현 관계 이용해 타입 제한

```
public <T extends 상위타입> 리턴타입 메소드(매개변수, ...) { ... }
```

  - 상위 타입은 클래스 뿐만 아니라 인터페이스도 가능

- 타입 파라미터를 대체할 구체적인 타입
  - 상위타입이거나 하위 또는 구현 클래스만 지정 가능
  - 사용시 주의할 점은 p.664~665 참조

6절. 와일드카드 타입

❖ 와일드카드 타입의 세가지 형태

- 제네릭타입<?> : Unbounded Wildcards (제한없음)
  타입 파라미터를 대치하는 구체적인 타입으로 모든 클래스나 인터페이스 타입이 올 수 있다.

- 제네릭타입<? extends 상위타입> : Upper Bounded Wildcards (상위 클래스 제한)
  타입 파라미터를 대치하는 구체적인 타입으로 상위 타입이나 하위 타입만 올 수 있다.

- 제네릭타입<? super 하위타입> : Lower Bounded Wildcards (하위 클래스 제한)
  타입 파라미터를 대치하는 구체적인 타입으로 하위 타입이나 상위 타입이 올 수 있다.

7절. 제네릭 타입의 상속과 구현

❖ 제네릭 타입을 부모 클래스로 사용할 경우

- 타입 파라미터는 자식 클래스에도 기술해야 !!!

```
public class ChildProduct<T, M> extends Product<T, M> { ... }
```

- 추가적인 타입 파라미터 가질 수 있음

```
public class ChildProduct<T, M, C> extends Product<T, M> { ... }
```

❖ 제네릭 인터페이스를 구현할 경우

- 제네릭 인터페이스를 구현한 클래스도 제네릭 타입
  - 예제 (p.670~671)를 통해 개념 확실히 이해할 것