13장 제네릭

13.1 왜 제네릭을 사용해야 하는가?

3.1.1 제네릭(Generic)이란?

■ 타입을 파라미터화해서 컴파일시 구체적인 타입이 결정되도록 하는 것
- 자바5부터 새로 추가된 기능이다.
- 컬렉션, 람다식(함수적 인터페이스), 스트림, NIO에서 널리 사용된다.
- 제네릭을 모르면 도큐먼트를 해석할 수 없다.

```
class ArrayList<E>
default <T,U> BiConsumer<T,U> andThen(BiConsumer<? super T,? super U> after) {...}
```

3.1.2 제네릭을 사용하는 코드의 이점

■ 컴파일 시 강한 타입 체크를 할 수 있다.
- 실행 시 타입 에러가 나는 것 방지
- 컴파일 시에 미리 타입을 강하게 체크해서 에러 사전 방지

■ 타입 변환(casting)을 제거한다. -> 프로그램 성능이 향상된다.

```
List list = new ArrayList();
list.add("hello");
String str = (String) list.get(0); //타입 변환을 해야 한다.

List<String> list = new ArraryList<String>();
list.add("hello");
String str = list.get(0); //타입 변환을 하지 않는다.
```

13.2 제네릭 타입(class<T>, interface<T>)

■ 제네릭 타입이란?
- 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언 시 클래스 또는 인터페이스 이름 뒤에 "<>" 부호 붙임
- "<>" 사이에는 타입 파라미터 위치

```
public class 클래스명<T> { ... }
public interface 인터페이스명<T> { ... }
```

■ 제네릭 타입을 사용하지 않을 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 변환.
```

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

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

Box<String> box1 = new Box<String>();
box1.set("hello");
String str = box1.get();

Box<Integer> box2 = new Box<Integer>();
box2.set(6);
int value = box2.get();
```

[Box.java] 제너릭 타입

```
01  package sec02.exam02_generic_type;
02
03  public class Box<T> {
04            private T t;
05            public T get() { return t; }
06            public void set(T t) { this.t = t; }
07  }
```

[BoxExample.java] 제네릭 타입 이용

```
01  package sec02.exam02_generic_type;
02
03  public class BoxExample {
04            public static void main(String[] args) {
05                      Box<String> box1 = new Box<String>();
06                      box1.set("hello");
07                      String str = box1.get();
08
09                      Box<Integer> box2 = new Box<Integer>();
10                      box2.set(6);
11                      int value = box2.get();
12            }
13  }
```

13.3 멀티 타입 파라미터(class<K,V,...>, interface<K,V,...>)

- 제네릭 타입은 두 개 이상의 타입 파라미터 사용 가능
- 자바 7부터는 다이아몬드 연산자 사용해 간단히 작성과 사용 가능

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

[Product.java] 제너릭 클래스

```
01  package sec03.exam01_multi_type_parameter;
02
03  public class Product<T, M> {
04          private T kind;
05          private M model;
06
07          public T getKind() {
08                  return this.kind;
09          }
10
11          public M getModel() {
12                  return this.model;
13          }
14
15          public void setKind(T kind) {
16                  this.kind = kind;
17          }
18
19          public void setModel(M model) {
20                  this.model = model;
21          }
22  }
```

[ProductExample.java] 제너릭 객체 생성

```
01  package sec03.exam01_multi_type_parameter;
02
03  public class ProductExample {
04          public static void main(String[] args) {
05                  // Product<Tv, String> product1 = new Product<Tv, String>();
06                  Product<Tv, String> product1 = new Product<>(); // 자바7 부터는 다이아몬드 연산자를
07  사용
08                  product1.setKind(new Tv());
09                  product1.setModel("스마트Tv");
10                  Tv tv = product1.getKind();
11                  String tvModel = product1.getModel();
12
13                  Product<Car, String> product2 = new Product<Car, String>();
14                  product2.setKind(new Car());
15                  product2.setModel("디젤");
16                  Car car = product2.getKind();
17                  String carModel = product2.getModel();
18          }
19  }
```

13.4 제너릭 메소드(<T, R> R method(T t))

■ 매개변수 타입과 리턴 타입으로 타입 파라미터를 갖는 메소드를 말한다.
■ 선언하는 방법은 리턴 타입 앞에 <> 기호를 추가하고 타입 파라미터를 기술한 다음, 리턴 타입과 매개 타입으로 타입 파라미터를 사용하면 된다.

```
// 제네릭 메소드의 기본형
public <타입파라미터,...> 리턴타입 메소드명(매개변수,...) {...}
public <T> Box<T> boxing(T t) { ... }

// 제네릭 메소드의 호출 방식
리턴타입 변수 = <구체적타입> 메소드명(매개값);
리턴타입 변수 = 메소드명(매개값);
Box<Integer> box = <Integer>boxing(100);  //타입 파라미터를 명시적으로 Integer로 지정
Box<Integer> box = boxing(100);           //타입 파라미터를 Integer로 추정
```

[Util.java] 제너릭 메소드

```
01  package sec04.exam01_generic_method;
02
03  public class Util {
04          public static <T> Box<T> boxing(T t) {
05                  Box<T> box = new Box<T>();
06                  box.set(t);
07                  return box;
08          }
09  }
```

[BoxingMethodExample.java] 제네릭 메소드 호출

```
01  package sec04.exam01_generic_method;
02
03  public class BoxingMethodExample {
04          public static void main(String[] args) {
05                  Box<Integer> box1 = Util.<Integer>boxing(100);
06                  int intValue = box1.get();
07
08                  Box<String> box2 = Util.boxing("홍길동");
09                  String strValue = box2.get();
10          }
11  }
```

13.5 제한된 타입 파라미터(<T extends 최상위타입>)

■ 타입 파라미터에 지정되는 구체적인 타입을 제한할 필요가 종종 있다.

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

[Util.java] 제네릭 메소드

```
01  package sec05.exam01_bounded_type;
02
```

```
03  public class Util {
04          public static <T extends Number> int compare(T t1, T t2) {
05                  double v1 = t1.doubleValue();
06                  //System.out.println(t1.getClass().getName());
07                  double v2 = t2.doubleValue();
08                  //System.out.println(t2.getClass().getName());
09                  return Double.compare(v1, v2);
10          }
11  }
```

[BoundedTypeParameterExample.java] 제네릭 메소드 호출

```
01  package sec05.exam01_bounded_type;
02
03  public class BoundedTypeParameterExample {
04          public static void main(String[] args) {
05                  // String str = Util.compare("a", "b");  // x, String은 Number 타입이 아님
06
07                  int result1 = Util.compare(10, 20);  // int -> Integer, 자동 Boxing
08                  System.out.println(result1);
09
10                  int result2 = Util.compare(4.5, 3);  // dboule -> Double, 자동 Boxing
11                  System.out.println(result2);
12          }
13  }
```

13.6 와일드 카드 타입(<?>, <? extends ...>, <? super ...>)

- 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.

```
// 수강생이 될 수 있는 타입은 다음 4가지 클래스라고 가정하자.
// Person의 하위 클래스로 Worker와 Student가 있고, Student의 하위 클래스로 HighStudent가 있다.

Course<?>  // 수강생은 모든 타입(Person, Worker, Student, HighStudent)이 될 수 있다.
Course<? extends Student>  // 수강생은 Student와 HighStudent만 될 수 있다.
Course<? super Worker>  // 수강생은 Worker와 Person만 될 수 있다.
```

[Course.java] 제네릭 타입

```
01  package sec06.exam01_generic_wildcard;
02
03  public class Course<T> {
04          private String name;
05          private T[] students;
06
07          // 타입 파라미터로 배열을 생성하려면 new T[] 형태로 배열을 생성할 수 없고
08          // (T[]) (new Object[n]) 형태로 생성해야 한다.
09          public Course(String name, int capacity) {
10                  this.name = name;
11                  students = (T[]) (new Object[capacity]);
12          }
```

```
13
14              public String getName() {
15                      return name;
16              }
17
18              public T[] getStudents() {
19                      return students;
20              }
21
22              // 배열에 비어있는 부분을 찾아서 수강생을 추가하는 메소드
23              public void add(T t) {
24                      for (int i = 0; i < students.length; i++) {
25                              if (students[i] == null) {
26                                      students[i] = t;
27                                      break;
28                              }
29                      }
30              }
31      }
```

[Person.java]

```
01      package sec06.exam01_generic_wildcard;
02
03      public class Person {
04              private String name;
05
06              public Person(String name) {
07                      this.name = name;
08              }
09
10              public String getName() { return name; }
11              public String toString() { return name; }
12      }
```

[WildCardExample.java] 와일드카드 타입 매개 변수

```
01      package sec06.exam01_generic_wildcard;
02
03      import java.util.Arrays;
04
05      public class WildCardExample {
06              // 모든 과정
07              public static void registerCourse(Course<?> course) {
08                      System.out.println(course.getName() + " 수강생: " +
09      Arrays.toString(course.getStudents()));
10              }
11
12              // 학생 과정
13              public static void registerCourseStudent(Course<? extends Student> course) {
14                      System.out.println(course.getName() + " 수강생: " +
15      Arrays.toString(course.getStudents()));
16              }
17
18              // 직장인과 일반인 과정
19              public static void registerCourseWorker(Course<? super Worker> course) {
20                      System.out.println(course.getName() + " 수강생: " +
21      Arrays.toString(course.getStudents()));
22              }
23
24              public static void main(String[] args) {
25                      Course<Person> personCourse = new Course<Person>("일반인과정", 5);
```

```
26                      personCourse.add(new Person("일반인"));
27                      personCourse.add(new Worker("직장인"));
28                      personCourse.add(new Student("학생"));
29                      personCourse.add(new HighStudent("고등학생"));
30
31                      Course<Worker> workerCourse = new Course<Worker>("직장인과정", 5);
32                      workerCourse.add(new Worker("직장인"));
33
34                      Course<Student> studentCourse = new Course<Student>("학생과정", 5);
35                      studentCourse.add(new Student("학생"));
36                      studentCourse.add(new HighStudent("고등학생"));
37
38                      Course<HighStudent> highStudentCourse = new Course<HighStudent>("고등학생과정", 5);
39                      highStudentCourse.add(new HighStudent("고등학생"));
40
41                      registerCourse(personCourse);
42                      registerCourse(workerCourse);
43                      registerCourse(studentCourse);
44                      registerCourse(highStudentCourse);
45                      System.out.println();
46
47                      // registerCourseStudent(personCourse); (x)
48                      // registerCourseStudent(workerCourse); (x)
49                      registerCourseStudent(studentCourse);
50                      registerCourseStudent(highStudentCourse);
51                      System.out.println();
52
53                      registerCourseWorker(personCourse);
54                      registerCourseWorker(workerCourse);
55                      // registerCourseWorker(studentCourse); (x)
56                      // registerCourseWorker(highStudentCourse); (x)
57              }
58      }
```

13.7 제네릭 타입의 상속과 구현

■ 제네릭 타입을 부모 클래스로 사용할 경우
- 타입 파라미터는 자식 클래스에도 기술해야 !!!
- 추가적인 타입 파라미터 가질 수 있음

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

[Product.java] 부모 제네릭 클래스

```
01      package sec07.exam01_generic_extends_implements;
02
03      public class Product<T, M> {
04              private T kind;
05              private M model;
06
07              public T getKind() { return this.kind; }
08              public M getModel() { return this.model; }
09
10              public void setKind(T kind) { this.kind = kind; }
11              public void setModel(M model) { this.model = model; }
12      }
13
```

```
14   class Tv {}
```

[ChildProduct.java] 자식 제네릭 클래스

```
01   package sec07.exam01_generic_extends_implements;
02
03   public class ChildProduct<T, M, C> extends Product<T, M> {
04                private C company;
05                public C getCompany() { return this.company; }
06                public void setCompany(C company) { this.company = company; }
07   }
```

[Storage.java] 제네릭 인터페이스

```
01   package sec07.exam01_generic_extends_implements;
02
03   public interface Storage<T> {
04             public void add(T item, int index);
05             public T get(int index);
06   }
```

[StorageImpl.java] 제네릭 구현 클래스

```
01   package sec07.exam01_generic_extends_implements;
02
03   public class StorageImpl<T> implements Storage<T> {
04             private T[] array;
05
06             public StorageImpl(int capacity) {
07                       this.array = (T[]) (new Object[capacity]);
08             }
09
10             @Override
11             public void add(T item, int index) {
12                       array[index] = item;
13             }
14
15             @Override
16             public T get(int index) {
17                       return array[index];
18             }
19   }
```

[ChildProductAndStorageExample.java] 제네릭 타입 사용 클래스

```
01   package sec07.exam01_generic_extends_implements;
02
03   public class ChildProductAndStorageExample {
04             public static void main(String[] args) {
05                       ChildProduct<Tv, String, String> product = new ChildProduct<>();
06                       product.setKind(new Tv());
07                       product.setModel("SmartTV");
08                       product.setCompany("Samsung");
09
10                       Storage<Tv> storage = new StorageImpl<Tv>(100);
11                       storage.add(new Tv(), 0);
12                       Tv tv = storage.get(0);
13             }
14   }
```

[과제] 확인문제

1. 제네릭에 대한 설명으로 틀린 것은 무엇입니까?

(1) 컴파일 시 강한 타입 체크를 할 수 있다.
(2) 타입 변환(casting)을 제거한다.
(3) 제네릭 타입은 타입 파라미터를 가지는 제네릭 클래스와 인터페이스를 말한다.
(4) 제네릭 메소드는 리턴 타입으로 타입 파라미터를 가질 수 없다.

2. ContainerExample 클래스의 main() 메소드는 Container 제네릭 타입을 사용하고 있습니다. main() 메소드에서 사용하는 방법을 참고해서 Container 제네릭 타입을 선언해보세요.

[ContainerExample.java] 제네릭 타입 이용

```
01  package verify.exam02;
02
03  public class ContainerExample {
04            public static void main(String[] args) {
05                      Container<String> container1 = new Container<String>();
06                      container1.set("홍길동");
07                      String str = container1.get();
08
09                      Container<Integer> container2 = new Container<Integer>();
10                      container2.set(6);
11                      int value = container2.get();
12            }
13  }
```

3. ContainerExample 클래스의 main() 메소드는 Container 제네릭 타입을 사용하고 있습니다. main() 메소드에서 사용하는 방법을 참고해서 Container 제너릭 타입을 선언해보세요.

[ContainerExample.java] 제네릭 타입 이용

```
01  package verify.exam03;
02
03  public class ContainerExample {
04            public static void main(String[] args) {
05                      Container<String, String> container1 = new Container<String, String>();
06                      container1.set("홍길동", "도적");
07                      String name1 = container1.getKey();
08                      String job = container1.getValue();
09
10                      Container<String, Integer> container2 = new Container<String, Integer>();
11                      container2.set("홍길동", 35);
12                      String name2 = container2.getKey();
13                      int age = container2.getValue();
14            }
15  }
```

4. Util.getValue() 메소드는 첫 번째 매개값으로 Pair 타입의 하위 타입만 받고, 두 번째 매개값으로 키값을 받습니다. 리턴값은 키값이 일치할 경우 Pair에 저장된 값을 리턴하고, 일치하지

않으면 null을 리턴하도록 getValue() 제네릭 메소드를 정의해보세요.

[UtilExample.java] 제네릭 메소드 호출

```
01  package verify.exam04;
02
03  public class UtilExample {
04          public static void main(String[] args) {
05                  Pair<String, Integer> pair = new Pair<>("홍길동", 35);
06                  Integer age = Util.getValue(pair, "홍길동");
07                  System.out.println(age);
08
09                  ChildPair<String, Integer> childPair = new ChildPair<>("홍삼원", 20);
10                  Integer childAge = Util.getValue(childPair, "홍삼순");
11                  System.out.println(childAge);
12
13                  /*OtherPair<String, Integer> otherPair = new OtherPair<>("홍삼원", 20);
14                  //OtherPair는 Pair를 상속하지 않으므로 예외가 발생해야 합니다.
15                  int otherAge = Util.getValue(otherPair, "홍삼원");
16                  System.out.println(otherAge);*/
17          }
18  }
19
20  // 실행 결과
21  // 35
22  // null
```

[Pair.java] 제네릭 타입

```
01  package verify.exam04;
02
03  public class Pair<K, V> {
04          private K key;
05          private V value;
06
07          public Pair(K key, V value) {
08                  this.key = key;
09                  this.value = value;
10          }
11
12          public K getKey()   { return key; }
13          public V getValue() { return value; }
14  }
```

[ChildPair.java] 제네릭 타입 상속

```
01  package verify.exam04;
02
03  public class ChildPair<K, V> extends Pair<K,V> {
04          public ChildPair(K k, V v) {
05                  super(k, v);
06          }
07  }
```

[OtherPair.java] 제네릭 타입

```
01  package verify.exam04;
02
03  public class OtherPair<K, V> {
04          private K key;
```

```
05            private V value;
06
07            public OtherPair(K key, V value) {
08                      this.key = key;
09                      this.value = value;
10            }
11
12            public K getKey()   { return key; }
13            public V getValue() { return value; }
14    }
```

[Util.java] 제네릭 메소드 정의

```
01    package verify.exam04;
02
03    public class Util {
04      // 작성 위치
05
06    }
```