Java프로그래밍

6강. 제네릭과 람다식 (교재 5장)

컴퓨터과학과 김희천 교수

한국방송통신대학교

오늘의 학습목차

1. 제네릭 타입
2. 제네릭 메소드와 타입 제한
3. 람다식

1. 제네릭 타입

1) 제네릭의 의미

- 제네릭 클래스, 제네릭 인터페이스, 제네릭 메소드
  - ✓ 클래스, 인터페이스, 메소드를 정의할 때 타입 매개변수(타입 파라미터)를 선언하고 사용할 수 있음

장점

- 여러 유형에 걸쳐 동작하는 일반화된 클래스나 메소드를 정의할 수 있음
- 자료형을 한정함으로써 컴파일 시점에 자료형 검사가 가능
  - ✓ 실행 오류를 찾아 고치는 것은 어렵기 때문
- 캐스트(형변환) 연산자의 사용이 불필요

2) 제네릭의 사용

◆ ArrayList 클래스는 List 인터페이스를 구현한 클래스

```
class ArrayList<E> implements List<E> … {
      boolean add(E e) { … }
      E get(int index) { … }
      E remove(int index) { … }
      …
}
```

```
List list1 = new ArrayList( );
list1.add("hello");
String s1 = (String)list1.get(0); //형변환 필요

List<String> list2 = new ArrayList<String>( );
list2.add("hello");
String s2 = list2.get(0);  //형변환이 필요 없음
```

3) 제네릭 클래스

- 클래스 정의에서 타입 파라미터를 선언함
  - ✓ 클래스를 사용할 때는 타입을 명시해야 함
- 타입 파라미터는 참조형만 가능함
  - ✓ 필드의 자료형, 메소드 반환형, 인자의 자료형으로 사용할 수 있음
- 컴파일 할 때, 명확한 타입 검사를 수행할 수 있음
  - ✓ 메소드 호출 시 인자의 유형이 맞는지
  - ✓ 메소드 호출의 결과를 사용할 때 유형이 맞는지
- 자료형을 매개변수로 가지는 클래스와 인터페이스를 제네릭 타입이라고 함

4) 제네릭 클래스의 정의

◆ 문법

- ✓ class 클래스이름<T1, T2, …> { … }
- ✓ 클래스 정의에서 클래스 이름의 오른편, 각 괄호 < > 안에 타입 파라미터를 표시함
- ✓ 컴마(,)로 구분하여 여러 개의 타입 파라미터를 지정할 수 있음
- ✓ 타입 파라미터는 타입을 전달 받기 위한 것
- ✓ 타입 파라미터의 이름은 관례적으로 E, K, V, N, T … 을 사용함

5) 제네릭 클래스의 예

```
class Data {
    private Object object;

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

```
class Data2<T> {
    private T t;

    public void set(T t) { this.t = t; }
    public T get( ) { return t; }
}
```

6) 제네릭 클래스의 필요성

- 제네릭 타입과 자료형 검사
  - ✓ 제네릭 타입을 사용하지 않으면 컴파일 시점에서 오류를 검출하지 못함
- 의미가 명확하면 생성자 호출 시, 괄호만 사용할 수 있음
  - ✓ Data2<String> b3 = new Data2< >( );

```
public class GenericsTest2 {
    public static void main(String args[ ]) {
        Data2<String> data = new Data2<String>( );
        Integer i = new Integer(20);
        data.set(i); //컴파일 오류
        String s = (String) data.get( );
         … …
    }
}
```

7) 제네릭 인터페이스를 구현하는 제네릭 클래스

◆ 2개의 타입 매개변수(K와 V)를 가지는 클래스

```
interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

class OrderedPair <K, V> implements Pair <K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey( ) { return key; }
    public V getValue( ) { return value; }
}
```

```
public class MultipleType {
    public static void main(String args[ ]) {
        Pair <String, Integer> p1;
        p1 = new OrderedPair < > ("Even", 8);
        Pair <String, String> p2;
        p2 = new OrderedPair < > ("hello", "java");
        … …
     }
}
```

8) 제네릭 타입을 상속/구현하는 일반 클래스

◆ 제네릭 인터페이스를 구현하는 일반 클래스

✓ 클래스를 정의할 때 제네릭 인터페이스의 < > 안에 자료형을 지정하면 됨

```
class MyPair implements Pair<String, Integer> {
    private String key;
    private Integer value;
    public MyPair(String key, Integer value) {
        this.key = key;    this.value = value;
    }
    public String getKey( ) {  return key; }
    public Integer getValue( ) { return value; }
}

public class ClassFromGeneric {
    public static void main(String args[ ]) {
        MyPair mp = new MyPair("test", 1);
        … …
   }
}
```

9) Raw 타입

- 제네릭 타입이지만 일반 타입처럼 사용하는 경우, 제네릭 타입을 지칭하는 용어
- 타입 매개변수 없이 사용되는 제네릭 타입
  - ✓ 자료형을 Object로 처리함
- 예
  - ✓ Data2 data = new Data2(“hello”);
  - ✓ 이때 Data2는 제네릭 타입 Data2<T>의 raw 타입

2. 제네릭 메소드와 타입 제한

1) 제네릭 메소드(1)

- 자료형을 매개변수로 가지는 메소드
  - ✓ 하나의 메소드 정의로 여러 유형의 데이터를 처리할 때 유용함
- 메소드 정의에서 반환형 왼편, 각 괄호 < > 안에 타입 매개변수를 표시
  - ✓ 타입 매개변수를 메소드의 반환형이나 메소드 매개변수의 자료형, 지역 변수의 자료형으로 사용할 수 있음

```
public static <T> T getLast(T[ ] a) {
      return a[a.length-1];
}
```

1) 제네릭 메소드(2)

- 인스탄스 메소드와 static 메소드 모두 제네릭 메소드로 정의 가능
- 제네릭 메소드를 호출할 때, 타입을 명시하지 않아도 인자에 의해 추론이 가능함

```
class Util {
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
      return p1.getKey().equals(p2.getKey())&& p1.getValue().equals(p2.getValue());
    }
}

public class GenericsTest5 {
    public static void main(String args[]) {
       Pair<Integer, String> p1 = new OrderedPair<>(1, "apple");
       Pair<Integer, String> p2 = new OrderedPair<>(2, "pear");
       boolean same = Util.<Integer, String>compare(p1, p2);
       System.out.println(same);
    }
}
```

2) 제네릭의 타입 제한

- 자료형을 매개변수화하여 클래스/인터페이스/메소드를 정의할 때, 적용가능한 자료형에 제한을 두는 것
- <T extends Number>와 같이 하면 T를 상한으로 정할 수 있음
  - ✓ T에 주어지는 자료형은 Number의 서브 클래스라야 함

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

```
public class BoundedType {
  public static void main(String args[ ]) {
    Data<Integer> data = new Data< >(20);
    System.out.println(data.get( ));
    }
}
```

3) 제네릭 타입과 형변환

◆ 상속 관계가 있어야 상위/하위 자료형의 관계가 존재함

✓ Integer나 Double은 Number의 자식 클래스

✓ Data <Number>와 Data <Integer>는 상하위 관계가 없음

```
class Data<T> { … }
class FormattedData<T> extends Data<T> { //상속 관계
}

public class GenericTypeConversion1 {
    public static void main(String args[ ]) {
        Data<Number> data = new Data<Number>( );
        data.set(new Integer(10)); //OK
        data.set(new Double(10.1)); //OK
        Data<Number> data1 = new Data<Integer>( ); //컴파일 오류
        Data<Integer> data = new FormattedData<Integer>( );
    }
}
```

4) 제네릭 타입 사용 시 유의 사항

- 기본 자료형은 타입 매개변수로 지정할 수 없음
  - ✓ Data<int> d = new Data<>( ); //오류
- 타입 매개변수로 객체 생성을 할 수 없음
  - ✓ class Data <T> { private T t1 = new T( ); } //오류
- 타입 매개변수의 타입으로 static 데이터 필드를 선언할 수 없음
  - ✓ class Data <T> { private static T t2; } //오류
- 제네릭 타입의 배열을 선언할 수 없음
  - ✓ Data <Integer>[ ] arrayOfData; //오류

3. 람다식

1) 람다식

◆ 인터페이스를 구현하는 익명 클래스의 객체 생성 부분을 수식화 한 것

✓ 구현할 것이 1개의 추상 메소드뿐일 때 간단히 표현할 수 있음

```
Runnable runnable = new Runnable( ) {
      public void run( ) {...} };
```

람다식 구문

◆ 메소드 매개변수의 괄호, 화살표, 메소드 몸체로 표현

✓ 인터페이스 객체변수 = (매개변수목록) -> { 실행문목록 }

```
Runnable runnable = ( ) -> {...} ;
```

2) 람다식 기본 문법(1)

- 익명 구현 클래스의 객체 생성 부분만 람다식으로 표현함
  - ✓ 익명 서브 클래스의 객체 생성은 람다식이 될 수 없음
- 이때 인터페이스에는 추상 메소드가 1개만 있어야 함
  - ✓ 2개 이상의 추상 메소드를 포함하는 인터페이스는 사용 불가
- 람다식의 결과 타입을 타깃 타입이라고 함
- 1개의 추상 메소드를 포함하는 인터페이스를 함수적 인터페이스라 함
  - ✓ 메소드가 1개 뿐이므로 메소드 이름을 생략할 수 있음
  - ✓ 람다식은 이름 없는 메소드 선언과 유사함

2) 람다식 기본 문법(2)

- **인터페이스 객체변수 = (매개변수목록) -> { 실행문목록 } ;**
  - ✓ 매개변수 목록에서 자료형은 인터페이스(타깃 타입) 정의에서 알 수 있으므로 자료형을 생략하고 변수 이름만 사용 가능
  - ✓ 매개변수가 1개면 괄호도 생략 가능하며 변수 이름 하나만 남음
  - ✓ 매개변수를 가지지 않으면 괄호만 남음
  - ✓ 화살표 사용
  - ✓ 실행문 목록에서 실행문이 1개이면 중괄호 생략 가능
  - ✓ 실행문이 return문 뿐이라면 return과 세미콜론, 중괄호를 동시 생략해야 하고 1개의 수식만 남게 됨

3) 람다식 사용 예(1)

```
interface Addable {
   int add(int a, int b);
}

public class LambdaExpressionTest {
   public static void main(String[] args) {

      //익명 클래스
      Addable ad1 = new Addable() {
         public int add(int a, int b) {
            return (a + b);
         }
      };
      System.out.println(ad1.add(100, 200));

   //매개변수 자료형과 return문을 가진 람다식
      Addable ad2 = (int a, int b) -> {
          return (a + b);
      };
      System.out.println(ad2.add(10, 20));

         //간단한 람다식
      Addable ad3 = (a, b) -> (a + b);
      System.out.println(ad3.add(1, 2));
   }
}
```

```
300
30
3
```

3) 람다식 사용 예(2)

```
interface MyInterface1 { public void method(int a, int b); }
interface MyInterface2 { public void method(int a); }

public class LambdaTest1 {
    public static void main(String args[]) {
        MyInterface1 f1, f2, f3;
        MyInterface2 f4, f5;

        f1 = (int a, int b) -> { System.out.println(a + b); };
        f1.method(3, 4);
        f2 = (a, b) -> { System.out.println(a + b); };
        f2.method(5, 6);
        f3 = (a, b) -> System.out.println(a + b);
        f3.method(7, 8);

        f4 = (int a) -> { System.out.println(a); };
        f4.method(9);
        f5 = a -> System.out.println(a);
        f5.method(10);
    }
}
```

```
7
11
15
9
10
```

4) 람다식의 활용

- 함수적 인터페이스(function interface)
  - ✓ 1개의 추상 메소드만 가지는 단순한 인터페이스를 함수적 인터페이스라 함
  - ✓ 패키지 java.util.function 에서 표준 함수적 인터페이스가 제네릭 인터페이스로 제공됨
  - ✓ 함수적 인터페이스를 구현하는 클래스를 정의할 때, 익명 클래스 정의를 활용할 수 있으나 람다식이 효율적

표준 함수적 인터페이스의 예

- Consumer<T>는 void accept(T t)를 가짐
- Supplier<T>는 T get( ) 메소드를 가짐
- Function<T, R>은 R apply(T t)를 가짐

5) 함수적 인터페이스와 람다식

```
public class RunnableTest4 {
    public static void main(String args[]) {
        Thread thd = new Thread(( ) -> System.out.println("my thread"));
         thd.start( );
    }
}
```

```
import java.util.function.*;

public class ConsumerTest{
    public static void main(String args[]) {
         Consumer<String>con = t -> System.out.println("Hello " + t);
         con.accept("Java");
    }
}
```

Java프로그래밍

다음시간안내

7강. 패키지와 예외 처리