13장 제네릭

1. 제네릭 클래스와 메소드
2. 사용예시
3. 제한된 타입 파라미터
4. 와일드카드 타입 파라미터
5. 예제 풀이

1 Object Class

1. 제네릭이란?

자바에서 클래스, 메서드, 인터페이스를 정의할 때, 타입(type)을 파라미터(parameter)화하여 다양한 타입의 객체를 다룰 수 있도록 하는 기능이다.

제네릭을 사용하면 코드의 재사용성과 유지 보수성을 높일 수 있다.

2. 제네릭 클래스와 제네릭 메서드
2.1 제네릭 클래스

```
public class Box<T> {
    private T value;

    public void setValue(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}
```

위의 예제는 제네릭 클래스 Box를 정의한 것이다. T는 타입 파라미터로, 실제 사용할 때 구체적인 타입으로 대체된다.

Object Class

3. 제네릭 사용 예시

3.1 제네릭 클래스 사용하기

```
public class Main {
    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<>();
        integerBox.setValue(10);
        System.out.println("Integer Value: " + integerBox.getValue());

        Box<String> stringBox = new Box<>();
        stringBox.setValue("Hello, world!");
        System.out.println("String Value: " + stringBox.getValue());
    }
}
```

3.2 제네릭 메서드 사용하기

```
public class Main {
    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        Utils.printArray(intArray);

        String[] stringArray = {"apple", "banana", "orange"};
        Utils.printArray(stringArray);
    }
}
```

3 제한된 타입 파라미터

```
public class NumberBox<T extends Number> {
   private T number;
   public NumberBox(T number) {
      this.number = number;
   }
   public double square() {
      return number.doubleValue() * number.doubleValue();
   } public static void main(String[] args) {
      NumberBox<Integer> intBox = new NumberBox<>(5);
      System.out.println("Square of integer: " + intBox.square());
      NumberBox<Double> doubleBox = new NumberBox<>(3.5);
      System.out.println("Square of double: " + doubleBox.square());

     // String은 Number 클래스를 상속하지 않으므로 컴파일 오류 발생
     // NumberBox<String> stringBox = new NumberBox<>("test");
   }
}
```

- 위의 예제는 제한된 타입 파라미터를 사용하여 **Number** 클래스 또는 그 하위 클래스만을 허용하는 **NumberBox** 클래스를 정의한 것이다.

- 제한된 타입 파라미터를 사용하여 클래스나 메서드를 정의할 때, 타입 파라미터가 특정 클래스의 하위 클래스여야 함을 지정할 수 있다.

4 와일드카드 타입-상위

```
import java.util.List;

public class NumberUtils {
    public static double sumOfList(List<? extends Number> list) {
        double sum = 0.0;
        for (Number number : list) {
            sum += number.doubleValue();
        }
        return sum;
    }
    public static void main(String[] args) {
        List<Integer> intList = List.of(1, 2, 3, 4, 5);
        System.out.println("Sum of integers: " + sumOfList(intList));
        List<Double> doubleList = List.of(1.5, 2.5, 3.5, 4.5, 5.5);
        System.out.println("Sum of doubles: " + sumOfList(doubleList));
    }
}
```

• 위의 예제는 상위 와일드카드를 사용하여 **Number** 클래스나 그 하위 클래스를 포함하는 리스트의 합을 구하는 메서드를 정의한 것이다.

• **List<? extends Number>**는 **Number** 클래스나 그 하위 클래스의 리스트를 나타낸다.

• 따라서 이 메서드는 **List<Integer>** 또는 **List<Double>**과 같은 리스트를 매개변수로 받아서 합을 계산할 수 있다.

4 와일드카드 타입-하위

```
import java.util.List;
public class StringUtils {
    public static void addStrings(List<? super String> list) {
        list.add("Hello");
        list.add("World");
    }
    public static void main(String[] args) {
        List<Object> objList = new ArrayList<>();
        addStrings(objList);
        System.out.println("Objects list: " + objList);
        List<CharSequence> charSeqList = new ArrayList<>();
        addStrings(charSeqList);
        System.out.println("CharSequence list: " + charSeqList);
    }
}
```

- 위의 예제는 하위 와일드카드를 사용하여 **String** 클래스나 그 상위 클래스를 포함하는 리스트에 문자열을 추가하는 메서드를 정의한 것이다.
- **List<? super String>**는 **String** 클래스나 그 상위 클래스의 리스트를 나타낸다.
- 따라서 이 메서드는 **List<Object>** 또는 **List<CharSequence>**과 같은 리스트를 매개변수로 받아서 문자열을 추가할 수 있다.

이렇게 제한된 타입 파라미터와 와일드카드 타입 파라미터를 사용하면 메서드나 클래스의 유연성을 높일 수 있으며, 타입 안정성을 보장할 수 있다.

5 예제풀이

문제 1:
다음은 제네릭 클래스 Pair이다. 아래 코드에서 getFirst()와 getSecond() 메서드를 완성하시오.

```
public class Pair<T, S> {
    private T first;
    private S second;

    // 생성자
    public Pair(T first, S second) {
        this.first = first;
        this.second = second;
    }

    // 첫 번째 요소 반환 메서드
    public T getFirst() {
        return first;
    }

    // 두 번째 요소 반환 메서드
    public S getSecond() {
        return second;
    }
}
```

- **getFirst()** 메서드는 **Pair** 클래스의 첫 번째 요소를 반환합니다. 이 메서드는 제네릭 타입 **T**를 반환하므로, **first** 필드의 타입과 일치하는 값을 반환합니다.

- **getSecond()** 메서드도 마찬가지로 **Pair** 클래스의 두 번째 요소를 반환합니다. 이 역시 제네릭 타입 **S**를 반환하므로, **second** 필드의 타입과 일치하는 값을 반환합니다.

5 예제풀이

문제 2:
다음은 제네릭 메서드 **swap**이다. 아래 코드에서 **swap** 메서드를 완성하고, 주어진 **main** 메서드가 정상적으로 동작하도록 하시오.

```
public class SwapUtils {
   // 제네릭 메서드 swap
   public static <T> void swap(T[] array, int i, int j) {
      T temp = array[i];
      array[i] = array[j];
      array[j] = temp;
   }

   public static void main(String[] args) {
      Integer[] intArray = {1, 2, 3, 4, 5};
      System.out.println("Before swapping:");
      Utils.printArray(intArray);

      swap(intArray, 0, 4);

      System.out.println("After swapping:");
      Utils.printArray(intArray);
   }
}
```

- **swap()** 메서드는 제네릭 메서드로, 어떤 타입의 배열이든 요소를 교환할 수 있도록 합니다.
- **T temp = array[i];**를 통해 배열의 **i**번째 요소를 임시 변수 **temp**에 저장합니다.
  그리고 **array[i]**를 **array[j]**로 대체하고,
  array[j]를 **temp**로 대체하여 요소를 교환합니다.

5 예제풀이

문제 3:
다음은 제네릭 메서드 'findMax'이다. 아래 코드에서 'findMax' 메서드를 완성하고, 주어진 'main'메서드가 정상적으로 동작하도록 하시오

```
public class MaxUtils {
   // 제네릭 메서드 findMax
   public static <T extends Comparable<T>> T findMax(T[] array) {
      if (array == null || array.length == 0) { return null; }

      T max = array[0];
      for (int i = 1; i < array.length; i++) {
         if (array[i].compareTo(max) > 0) {
            max = array[i];
         }
      }
      return max;
   }
   public static void main(String[] args) {
      Integer[] intArray = {3, 7, 1, 9, 5};
      Integer maxInt = findMax(intArray);
      System.out.println("Max Integer: " + maxInt);

      String[] stringArray = {"apple", "banana", "orange"};
      String maxString = findMax(stringArray);
      System.out.println("Max String: " + maxString);
   }
}
```

- **findMax()** 메서드는 제네릭 메서드로, 배열에서 최대값을 찾아 반환합니다.
- **<T extends Comparable<T>>**는 제네릭 타입 **T**가 **Comparable** 인터페이스를 구현한 타입이어야 함을 나타냅니다. 이는 최대값을 찾기 위해 요소들을 비교할 수 있어야 함을 의미합니다.
- 배열의 첫 번째 요소를 **max**로 초기화하고, 배열의 모든 요소를 순회하면서 **max**와 비교하여 더 큰 값이 있으면 **max**를 업데이트합니다.
- 최종적으로 가장 큰 요소를 반환합니다.

5 예제풀이

문제 4:
다음은 제네릭 클래스 'Stack'이다. 아래 코드에서 'pop()'메서드를 구현하시오.

```
import java.util.ArrayList;
import java.util.List;

public class Stack<T> {
    private List<T> elements;

    // 생성자
    public Stack() {
        elements = new ArrayList<>();
    }

    // 스택에 요소를 추가하는 메서드
    public void push(T element) {
        elements.add(element);
    }

    // 스택에서 마지막 요소를 꺼내는 메
    public T pop() {
        if (elements.isEmpty()) {
            return null;
        }
        return elements.remove(elements.size() - 1);
    }
}
```

- **pop()** 메서드는 제네릭 클래스 **Stack**에서 요소를 꺼내는 메서드입니다.
- 먼저 **elements** 리스트가 비어 있는지 확인하고, 비어 있다면 **null**을 반환합니다.
- 비어 있지 않다면 **elements** 리스트에서 마지막 요소를 꺼내어 반환합니다.

5 예제풀이

문제 5:
다음은 제네릭 메서드 'mergeArrays'이다.
아래 코드에서 두 개의 배열을 병합하여 하나의 배열로 변환하는 메서드를 완성하시오.

```
public class ArrayUtils {
   // 제네릭 메서드 mergeArrays
   public static <T> T[] mergeArrays(T[] firstArray, T[] secondArray) {
      int firstLength = firstArray.length;
      int secondLength = secondArray.length;
      T[] mergedArray = Arrays.copyOf(firstArray, firstLength + secondLength);
      System.arraycopy(secondArray, 0, mergedArray, firstLength, secondLength);
      return mergedArray;
   }

   public static void main(String[] args) {
      Integer[] array1 = {1, 2, 3};
      Integer[] array2 = {4, 5, 6};
      Integer[] mergedArray = mergeArrays(array1, array2);
      Utils.printArray(mergedArray);

      String[] strArray1 = {"apple", "banana"};
      String[] strArray2 = {"orange", "grape"};
      String[] mergedStrArray = mergeArrays(strArray1, strArray2);
      Utils.printArray(mergedStrArray);
   }
}
```

- **mergeArrays()** 메서드는 두 개의 제네릭 배열을 병합하여 하나의 배열로 반환합니다.

- 먼저 두 배열의 길이를 가져온 후, 두 배열의 길이를 합친 길이로 새로운 배열을 생성합니다.

- 첫 번째 배열의 내용을 새로운 배열에 복사한 후, 두 번째 배열의 내용을 그 뒤에 이어붙입니다.

- 병합된 배열을 반환합니다.

감사합니다