09_2 Generics

Object-Oriented Programming

What is Generics?

- Treat an undetermined type as a parameter
- Replace the parameter with a concrete type in actual use
- ex)
  - <T> symbol to indicate that T is a type parameter
  - Indicating T can be used where a type is required
  - ```
    class Test<T> { T x; }
    ```
    - ```
      Test<Integer> t1 = new Test<>();
      ```
    - ```
      Test<Double> t2 = new Test<>();
      ```
    - ```
      Test<Student> t3 = new Test<>();
      ```

Generic Type

- Type parameters: usually represented by a single capitalized letter
  - ex) `interface BInterface<U>`
  - ex) `class AClass<S> implements BInterface<U>`
- To use generic types externally, specify a concrete type in the type parameter
  - ex) `class AClass<String> std = new AClass<>();`
  - Type parameter should not be the primitive type

Example: Generic Class (1/2)

```
class Box<T> { // T: type parameter
    private T item;

    public Box(T item) {  // constructor
        this.item = item;
    }

    public T getItem() { // accessor, generic method
        return item;
    }

    public void setItem(T item) { // mutator, generic method
        this.item = item;
    }
}
```

Example: Generic Class (2/2)

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

        Box<Integer> intBox = new Box<>(123);
        System.out.println("Integer value: " + intBox.getItem());

        Box<String> strBox = new Box<>("Hello, Generics!");
        System.out.println("String value: " + strBox.getItem());

        Box<Double> doubleBox = new Box<>(3.14);
        System.out.println("Double value: " + doubleBox.getItem());

        strBox.setItem("New String Value");
        System.out.println("Updated String value: " + strBox.getItem());
    }
}
```

```
OUTPUT:
Integer value: 123
String value: Hello, Generics!
Double value: 3.14
Updated String value: New String Value
```

Generic Methods

- A method that has a type parameter
- Type parameter is used in the return type and parameter type
  - ex) `public <T> void genericMethod(T param) { ... }`
- Type parameter T is replaced with a concrete type **during compilation**, depending on the type of the parameter

Example: Generic Method (1/2)

```
public class GenericMethodExample {

    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
```

Example: Generic Method (2/2)

```
    public static void main(String[] args) {

        Integer[] intArray = {1, 2, 3, 4, 5};
        System.out.print("Integer Array: ");
        printArray(intArray);

        String[] strArray = {"Hello", "World", "Generics", "in", "Java"};
        System.out.print("String Array: ");
        printArray(strArray);

        Double[] doubleArray = {1.1, 2.2, 3.3, 4.4, 5.5};
        System.out.print("Double Array: ");
        printArray(doubleArray);
    }
}
```

OUTPUT:

```
Integer Array: 1 2 3 4 5
String Array: Hello World Generics in Java
Double Array: 1.1 2.2 3.3 4.4 5.5
```

Example: Multiple Type Parameters (1/3)

```
public class Pair<K, V> { // two type parameters
    private K key;
    private V value;

    public Pair(K key, V value) {  // constructor
        this.key = key;
        this.value = value;
    }

    public K getKey() {  // accessor for key
        return key;
    }

    public void setKey(K key) {  // mutator for key
        this.key = key;
    }
```

Example: Multiple Type Parameters (2/3)

```
    public V getValue() {  // accessor for value
        return value;
    }

    public void setValue(V value) {  // mutator for value
        this.value = value;
    }

    @Override
    public String toString() {
        return "Pair{" +
                "key=" + key +
                ", value=" + value +
                '}';
    }
```

Example: Multiple Type Parameters (3/3)

```
    public static void main(String[] args) {
        Pair<String, Integer> studentGrade = new Pair<>("Alice", 95);
        Pair<String, String> countryCapital = new Pair<>("Germany", "Berlin");

        System.out.println(studentGrade); // Pair{key=Alice, value=95}
        System.out.println(countryCapital); // Pair{key=Germany, value=Berlin}

        String student = studentGrade.getKey();
        int grade = studentGrade.getValue();
        System.out.println("Student: " + student + ", Grade: " + grade);
        // Student: Alice, Grade: 95

        studentGrade.setKey("Bob");
        studentGrade.setValue(85);
        System.out.println("Updated: " + studentGrade);
        // Updated: Pair{key=Bob, value=85}
    }
}
```

Example: Using Different Type Param (1/2)

```
public class Container<T> {
    private T item;
    public Container(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
    public void setItem(T item) {
        this.item = item;
    }
    public <U> void displayItemWithDetails(U detail) { // using U, not T
        System.out.println("Item: " + item);           // in the method
        System.out.println("Detail: " + detail);
    }
```

Example: Using Different Type Param (2/2)

```
    public static void main(String[] args) {
        // Container<String>
        Container<String> stringContainer = new Container<>("Apple");

        // Detail<U> = Integer
        stringContainer.displayItemWithDetails(123); // OUTPUT: Apple 123

        // Container<Integer>
        Container<Integer> integerContainer = new Container<>(456);

        // Item과 추가적인 Detail을 출력 (String 타입 사용)
        integerContainer.displayItemWithDetails("Detail about 456");
        // OUTPUT: 456 Detail about 456
    }
}
```

Bounded Parameters

- Type parameter is limited (bounded) to a descendant of a given specific superclass (super interface) type
  - ex) `class SomeClass<T extends SuperClass> { ... }`
  - ex) `class SomeClass<T extends SuperInterface<T>> { ... }`
  - ...
- Typical Superclass (interface)
  - class Number: superclass of Integer, Double, Byte, Short, Long, Float
  - interface Comparable<T>
  - interface Runnable
  - interface java.util.Comparator<T>
  - interface CharSequence
    - implemented by the classes: String, StringBuilder, StringBuffer, ...

Example: NumberContainer (1/2)

```
public class NumberContainer<T extends Number> { // T is bounded by Number
    private T number;

    public NumberContainer(T number) {
        this.number = number;
    }

    public T getNumber() {
        return number;
    }

    public void setNumber(T number) {
        this.number = number;
    }

    public double doubleValue() {
        return number.doubleValue() * 2;
    }
```

Example: NumberContainer (2/2)

```
    public static void main(String[] args) {

        NumberContainer<Integer> intContainer = new NumberContainer<>(5);
        System.out.println("Integer Value: " + intContainer.getNumber()); // 5
        System.out.println("Doubled Value: " + intContainer.doubleValue());  // 10.0

        NumberContainer<Double> doubleContainer = new NumberContainer<>(3.14);
        System.out.println("Double Value: " + doubleContainer.getNumber()); // 3.14
        System.out.println("Doubled Value: " + doubleContainer.doubleValue());  // 6.28

        // Compile Error: String is not the descendant of Number
        // NumberContainer<String> stringContainer = new NumberContainer<>("Hello");
    }
}
```