09_2 Generics

Object-Oriented Programming

Generics에 대해 강의하겠습니다.

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<>();`

페이지 two

Generics는 무엇일까요?
class를 구성하는 instance variable들
method의 parameter들의 type만이 다르고
하는 일이 같은 class나 method들을 만들어야
할 필요가 많이 있습니다.
예를 들면 class Test { Integer x; } 라는
간단한 class를 고려해 본다면
x가 Integer가 아니라 Double인 class를 만들려면
새로운 class TestD { Double x; } 를 만들어 주어야 합니다.
이렇게 단순히 type만이 달라졌는데도
새로운 class나 method를 만들어야 하는 불편함을 줄이기 위해
Generics mechanism을 이용할 수 있습니다.
Generics에서는 type 자체를 <T> 와 같은 symbol parameter로 받고
실제로 사용할 때에는 T 대신 concrete type을 넣어서 사용합니다.
예를 들면 class Test<T> { T x; } 라는 generic class가 있다면
이것을 사용할 때에는 Test<Integer> t1 = new Test<>();
또는 Test<Double> t2 = new Test<>();
Test<Student> t3 = new Test<>();
와 같은 형태로 사용할 수 있습니다.
이 세가지 경우에 generic class Test<T> 의
instance variable x의 type이 각각
Integer, Double, Student가 되는 것입니다.

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`

페이지 3

Generics에서는 type을 parameter로 표현합니다.
이 parameter를 type parameter라 부릅니다.
type parameter의 이름은 어떤 identifier라도 가능하지만
주로 T, U, S 등의 alphabet 대문자 한 글자를 사용하는 경우가 많습니다.
예를 들면 interface BInterface type parameter U 또는
class AClass type parameter S implements BInterface type parameter U
와 같은 것들입니다.
Generics를 사용할 때에는 type parameter에
실제 class type을 대입하여 사용합니다.
예를 들면 class AClass type parameter S 대신 String을 사용할 수 있습니다.
여기서 한가지 주목할 것은 type parameter로
int, double, boolean 등의 primitive type은 사용할 수 없다는 것입니다.
따라서 primitive type을 대신하여
Integer, Double 등의 wrapper class 등을 사용할 수 있습니다.

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;
    }
}
```


페이지 4

이제 generic class의 첫번째 example인
Box generic class를 살펴보겠습니다
class Box 뒤에 type parameter <T>를 붙였습니다.
type parameter T는 instance variable item의 type으로 사용되었습니다.
Constructor의 parameter인 item도 T type으로 정의하여
this.item에 parameter item을 assign하도록 되어 있습니다
accessor method인 getItem()의 return type에도 T가 사용되어
private instance variable인 T type의 item을 return할 수 있도록 하였습니다.
mutator인 setItem()의 parameter에도 역시 T가 사용되었습니다.
getItem()과 setItem() 과 같이 type parameter가 사용된 method들을
generic methods라 부릅니다.

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
```


페이지 5

driver class인 GenericClassExample의 main method에서는
먼저 Box<Integer> class type인 intBox object를 생성하였습니다.
generic class Box의 type parameter <T> 의 자리에
<Integer> 가 대입되어 concrete class type을 만들고 있는 것을 볼 수 있습니다.
new 뒤에 나오는 constructor call인 Box<>(123); 에서는
type parameter를 빈칸으로 남겨 놓았는데
앞에서 intBox의 type을 Box<Integer> 라고 declare했기 때문에
다시 반복을 할 필요가 없기 때문이기는 하지만
constructor call을 Box<Integer>(123) 으로 해도 문제는 없습니다.
그 다음 line에서는 intBox의 accessor method인 getItem()을 call하여
item의 value인 123을 print하였습니다.
그 아래에는 type parameter를 String으로 하여 Box<String> type의 object인
strBox를 생성하고, str.getItem()을 call하였습니다.
그 다음에는 type parameter를 Double로 하여 Box<Double> class type인
doubleBox object를 생성하고, 역시 doubleBox.getItem()을 call하였습니다.
그 다음 line에는 mutator method인 strBox.setItem을 call하여
item의 String value를 "New String Value" 로 바꾸고
accessor인 strBox.getItem() 으로 바뀐 String 값을 확인하였습니다.

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

페이지 6

Generic method에서는 type parameter가 method parameter의 type이나
return type 또는 local variable의 type 등에 사용될 수 있습니다.
Type parameter T는 compilation동안 generic method call이 되는 부분이 나올때
그에 맞추어 concrete type으로 대체되게 됩니다.

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();
    }
```


페이지 7

generic method의 example을 보겠습니다.
이 프로그램에서 GenericMethodExample class는 generic이 아니고
그 안에 정의된 printArray method가 generic으로 정의되어
서로 다른 type의 array 들을 parameter로 넘겨 받아
그 내용을 for each 문을 이용하여 print해 주는 일을 합니다.
특히 generic method의 경우
access modifier와 static 뒤와 return type (여기서는 void) 사이에
type parameter <T> 를 사용할 것이라는 표시를 해 두어야 하며
이 type parameter를 method 안에서 type으로 사용할 수 있게 되어 있습니다.
이와 같이 class 자체가 generic이 아니더라도
그 안의 method만 generic 일 수도 있으며
심지어 generic class의 type parameter와
그 안의 generic class의 type parameter가
서로 다른 경우도 있을 수 있습니다.

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
```


페이지 8

main method에서는
처음에 Wrapper Integer array인 intArray를
printArray의 parameter로 intArray를 pass하여
그 내용을 print하게 하였습니다.
이와 같이 generic method를 call할 때에는
그냥 형식에만 맞는 서로 다른 type의 parameter를
자연스럽게 pass하기만 하면 됩니다.
Wrapper Integer array를 initialize할 때
= {1, 2, 3, 4, 5} 와 같이 int literal들을 그냥 사용했는데
이 initialization이 가능한 이유는
Wrapper Integer type이 int 에서 Integer로의 automatic boxing을 제공하기 때문입니다.
두번째로는 String의 array인 strArray를
generic method printArray로 pass하였으면
마지막으로 Wrapper Double type의 array를
generic method인 printArray를 사용하여 print하였습니다.

Example: Multiple Type Parameters (1/3)

```java
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;
    }
```


페이지 9

여기에서는 type parameter를 여러개 사용하는 generics의 예를 보여주고 있습니다.
generic class Pair에 두 개의 type parameter K와 V가 사용되었습니다.
instance variable key의 type은 K로 정의되었고
value의 type은 V로 정의 되었습니다.
constructor에도 K와 V, 두개의 type parameter가 사용되었습니다.
instance variable key를 위한 accessor와
mutator에는 type parameter K가 사용되었습니다.

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 +
                '}';
    }
```


페이지 10

instance variable value를 위한
accessor method getValue()와
mutator setValue() 에는
type parameter V가 적절히 사용되었습니다.
마지막으로 toString이 override되어 있는데
여기에는 K와 V type parameter를
사용할 일이 없었습니다.
key와 value가 String과 concatenate될 때
그들 type의 toString이 자동으로 call될 것이기 때문입니다.

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}
    }
}
```


페이지 11

main에서 먼저
Pair<String, Integer> class인 studentGrade object를 생성하였습니다.
이 class는 그 초기화 value들에서 알 수 있듯이
학생 이름과 그의 점수를 key와 value의 pair로 가지고 있는 class가 됩니다.
그 아래에는 Pair<String,String> class object인 countryCapital을 생성했는데
이 class의 key는 나라이름, value는 수도 이름이 됩니다.
두 object를 각각의 toString을 이용하여 print하고
studentGrade object의 student를 accessor로 return 받고
grade를 역시 accessor method로 return 받아
다른 형식으로 print하였습니다.
마지막으로 studentGrade object의 key와 value의 값을
mutator method들을 이용하여
"Bob" 과 85로 바꾸었습니다.

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);
    }
```


페이지 12

이 예제 프로그램에서는 generic class 안에서
type parameter가 class의 type parameter와 다른
generic method를 정의하고 사용하는 예를 보여줍니다.
Container generic class의 type parameter는 T인데
T는 instance variable item의 type이며
constructor의 parameter type이기도 하고
accessor getItem() method의 return value
그리고 mutator setItem() method의 parameter type으로도 사용되었습니다.
맨 마지막의 displayItemWithDetails method는 generic method 인데
Container class의 type parameter T 대신
다른 type parameter U를 사용하였습니다.
U type의 parameter인 detail을 pass받아서
instance variable item과 detail의 값을 print합니다.
이와 같이 특정한 generic class안에서
그 generic class의 type parameter와 다른 type parameter를
generic method에서 정의하여 사용할 수도 있는 것입니다.

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
    }
}
```


페이지 13

main method에서는
generic class Container<String> type인
stringContainer object를 생성하였고
"Apple" 이라는 초기값으로 instance variable item을 초기화 하였습니다.
이 stringContainer object의 displayItemWithDetails method를
parameter 123과 함께 call하였는데
이 때 displayItemWithDetails generic method의 type parameter인 U는
123의 type에 따라 자동으로 Integer type으로 setting 됩니다.
두번째 object인 integerContainer는 Container<Integer> type으로 생성되었고
integerContainer.displayItemWithDetails("Detail about 456") 이 call되는 순간
generic method displayItemWithDetails의 type parameter U가
자동으로 String type으로 setting되게 됩니다.

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, ...

페이지 14

Type parameter를 어떤 특정한 조건에 맞도록 bound (제한) 할 수 있는 기능이 있습니다.
이 때 제한되는 parameter를 bounded type parameter라고 부릅니다.
Type parameter의 제한은 상속에서 사용된 "extends" keyword를 사용합니다.
예를 들면 class SomeClass<T extends SuperClass> { ... } 는
T라는 type parameter를 SuperClass의 descendant들로 제한합니다.
또 다른 예로, class SomeClass<T extends SuperInterface<T>> { ...} 는
T type parameter를 SuperInterface<T>를 implement하는 class들 중 하나로 제한하는 것입니다.
Bounded type parameter를 위해
super class (또는 interface) 로 많이 쓰이는 것들은
먼저 class Number가 있는데
Number는 Integer, Double, Byte, Short, Long, Float의 superclass입니다.
또 interface Comparable<T> 가 있는데
이 interface를 implement하는 class들은
abstract method인 compareTo method가 implement되어 있는 class들입니다.
interface Runnable도 많이 사용되는데
abstract method인 void run() 을 가지고 있습니다.
java.util.Comparator<T> 는 abstract method int compare(a, b) 를 가지고 있습니다.
Comparable과 Comparator interface에 대해서는
07장의 interface 부분을 다시 참고하기 바랍니다.
그 외에도 CharSequence interface가 있는데
이 interface를 implement하고 있는 class들은
String, StringBuilder, StringBuffer 등이 있으며,
주로 String 데이터를 읽기 위한 method들을 장착하고 있는데
charAt(), length(), subSequence() 등의 method들입니다

Example: NumberContainer (1/2)

```java
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;
    }
```


페이지 15

Bounded type parameter의 example을 하나 보도록 하겠습니다.
NumberContainer<T extends Number> 는
generic class로서 class Number의 subclass들인
Integer, Double, Byte, Short, Long, Float 중의 하나로
bound 되어 있습니다.
맨 마지막의 method doubleValue() 는
number.doubleValue() return 값에 2를 곱한 값을 return 하는데
class Number에 구현되어 있는 doubleValue() method는
원래의 값을 double type으로 conversion하여 return하는 method 입니다.

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");
    }
}
```


페이지 16

main에서 먼저 type parameter를 Integer로 하는
NumberContainer<Integer> class의 object인 intContainer를 생성합니다.
초기 값은 5 이고, doubleValue값은 10.0 이 됩니다.
두번째 object인 doubleContainer는
type parameter를 Double을 가지는 NumberContainer<Double> type이고
초기값은 3.14, doubleValue값은 6.28이 됩니다.
그러나 마지막 comment된 부분에서 볼 수 있듯이
NumberContainer<String> 을 정의하려고 시도하면 compile error가 납니다.
NumberContainer의 type parameter는 Number class의 subclass들로 bound되어 있고
String은 Number class의 subclass가 아니기 때문입니다.