Lecture 2

ADVANCED JAVA PROGRAMMING

Level (3) - SW & IT

mohmd798380@gmail.com

Eng. Mohammed Muthanna

Lecture 2

ADVANCE OBJECT-ORINTEND PROGRAMMING IN JAVA

OOP

mohmd798380@gmail.com

Lecture Outline

- Generics
- Why Generics?
- Types of Java Generics
  - Generic Classes
  - Generic Methods
- Type Parameters
- Multiple Type Parameters
- Bounded type parameters
- Advantages of Generics
- Disadvantages of Generics

Generics

Generics means parameterized types. The idea is to allow a type (like Integer, String, etc., or user-defined types) to be a parameter to methods, classes, and interfaces.

An entity such as a class, interface, or method that operates on a parameterized type is **a generic entity**.

Why Generics?

The Object is the superclass of all other classes, and Object reference can refer to any object. These features lack type safety. **Generics add that type of safety feature.**

❑ Generics

What's Type Safety?

Type safety : is a programming concept that ensures operations on variables or objects are performed on compatible types, preventing type-related runtime errors.

Types of Java Generics:

- Generic Classes
- Generic Methods

Generic Classes

- Classes that can operate on multiple types using type parameters.
- A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section.
- There can be more than one type of parameter, separated by a comma.
- <> is used to specify parameter types in the creation of a generic class.
- Some examples of classes that use generics:
  - HashSet
  - ArrayList
  - HashMap

Note: In Parameter type we can not use primitives like int, char or double.

❑ Generic Classes

❖ Syntax:

```
class ClassName<T> {
    // T represents the type parameter
}
```

❖ Creating Object:

```
// To create an object of generic class
className <Type> obj = new className <Type>();
```

Notes: - Cannot create an array of generic type,
- Cannot use type parameter in static methods.

❑ Example

```java
// Generic Class
class Box<T> {
    private T item;
    public void setItem(T item) {
        this.item = item;
    }
    public T getItem() {
        return item;
    }
}
public class Main {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setItem("Hello");
        System.out.println("String value: " + stringBox.getItem());

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

❑ Generic Methods

- It is exactly like a normal function; however, a generic method has type parameters that are cited by actual type.
- This allows the generic method to be used in a more general way.
- **Key characteristics:**
  - A type parameter is declared within angle brackets (<>) before the return type.
  - **Flexible and Reusable**: Allows the method to handle different types without duplicating code.
  - **Type Safety**: Ensures the correctness of the type at compile time, reducing runtime errors.
- Example: Collections.sort(List<T>).

❑ Generic Methods

❖ Syntax:

```
public <T> returnType methodName(T parameter) {
    // method body
}
```

- **Calling the method:**
  - **methodName("text");**
  - **methodName(20);**
  - **methodName(78.9);**

❑ Example

```java
public class GenericMethods {
    public static <T> void printItem(T item) {
        System.out.println("Item: " + item);
    }
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
    public static <T> T getFirstElement(T[] array) {
        return array.length > 0 ? array[0] : null;
    }
    public static void main(String[] args) {
        printItem("Hello");       // With String
        printItem(100);          // With Integer

        Integer[] intArray = {1, 2, 3, 4};
        String[] strArray = {"A", "B", "C"};
        printArray(intArray);
        printArray(strArray);

      System.out.println("First item: " + getFirstElement(strArray));
    }
}
```

❑ The Common Type Parameters in Java Generics:

- **T – Type**
- **E – Element**
- **K – Key**
- **N – Number**
- **V – Value**

❑ Using Multiple Type Parameters

- You can use multiple type parameters to handle more than one type simultaneously.

❖ **Syntax:**

```
class ClassName<T1, T2, T3> {
    // T represents the type parameter
}
```

❑ Example

```
class MultiTypeClass<T1, T2, T3> {
    private T1 value1;
    private T2 value2;
    private T3 value3;
    public MultiTypeClass(T1 value1, T2 value2, T3 value3) {
        this.value1 = value1;
        this.value2 = value2;
        this.value3 = value3;
    }
    public void displayValues() {
        System.out.println("Value 1: " + value1);
        System.out.println("Value 2: " + value2);
        System.out.println("Value 3: " + value3);
    }
}
public class Main {
    public static void main(String[] args) {
        MultiTypeClass<Integer, String, Double> multiTypeObj = new
                        MultiTypeClass<>(123, "Hello", 45.67);
        multiTypeObj.displayValues();
    }
}
```

❑ Bounded type parameters

- Bounded type parameters in Java allow you to restrict the types that can be used as arguments for a generic type, ensuring type safety.
- **Types of Bounds:**
  - **Upper Bound** *<T extends Type>*:
    - Restricts the type to a specific class or its subclasses using the extends keyword.
    - Example:

```
class Example<T extends Number> {
    // T must be Number or a subclass (e.g., Integer, Double)
}
```

❑ Bounded type parameters

- **Types of Bounds:**
  - **Multiple Bounds** *<T extends Type1 & Type2>*:
    - Specifies multiple interfaces a type must implement using &.
    - Example:

```
class Example<T extends ClassA & InterfaceB & InterfaceC> {
    // T must extend ClassA and implement InterfaceB, InterfaceC
}
```

❑ Example of Upper Bound

```java
class Calculator<T extends Number> {
    public double square(T number) {
        return number.doubleValue() * number.doubleValue();
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator<Integer> intCalc = new Calculator<>();
        System.out.println("Square of 5: " + intCalc.square(5));

        Calculator<Double> doubleCalc = new Calculator<>();
        System.out.println("Square of 2.5: " + doubleCalc.square(2.5));

        // Calculator<String> stringCalc = new Calculator<>(); // Compile-time error
    }
}
```

❑ Example of Multiple Bounds

```
interface Shape {
    double area();
}

class ColoredShape {
    private String color;
    public ColoredShape(String color) {
        this.color = color;
    }
    public String getColor() {
        return color;
    }
}

class Rectangle extends ColoredShape implements Shape {
    private double length, width;
    public Rectangle(double length, double width, String color) {
        super(color);
        this.length = length;
        this.width = width;
    }
    @Override
    public double area() {
        return length * width;
    }
}
```

❑ Example of Multiple Bounds

```
class ShapePrinter<T extends ColoredShape & Shape> {
    public void printDetails(T shape) {
        System.out.println("Color: " + shape.getColor());
        System.out.println("Area: " + shape.area());
    }
}

public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5, 3, "Red");
        ShapePrinter<Rectangle> printer = new ShapePrinter<>();
        printer.printDetails(rect);
    }
}
```

❑ Advantages of Generics:

- **Code Reusability**: You can write a method, class, or interface once and use it with any type.
- **Type Safety**: Generics ensure that errors are detected at compile time rather than runtime, promoting safer code.
- **No Need for Type Casting**: The compiler automatically handles casting, removing the need for explicit type casting when retrieving data.
- **Code Readability and Maintenance**: By specifying types, code becomes easier to read and maintain.
- **Generic Algorithms**: Generics allow for the implementation of algorithms that work across various types, promoting efficient coding practices.

❑ Disadvantages of Generics:

- **Complexity**: For beginners, understanding concepts like wildcards (? extends, ? Super) can be difficult.
- **Performance Overhead**: Type erasure causes some overhead as generic types are converted to Object during runtime.
- **No Support for Primitive Types**: Generics only work with reference types, requiring the use of wrapper classes like Integer, or Double for primitives.

THE END

mohmd798380@gmail.com