Java Generics

ADVANCED PROGRAMMING PRACTICES

INSTRUCTORS: DR. JOEY PAQUET
EMAIL: JOEY.PAQUET@CONCORDIA.CA

DR. AMIN RANJ BAR
EMAIL: AMIN.RANJBAR@CONCORDIA.CA

TA: SHREYAS PATEL

EMAIL: SHREYASKISHORPATEL@GMAIL.COM

Agenda

- Introduction to Java Generics
- Syntax and Basics
- Wildcards and Bounded Types

Introduction to Java Generics

- In Java, Generics is a feature that allows you to write classes, interfaces, and methods that can work with any data type. Instead of specifying a concrete type, you can use type parameters, which are placeholders for actual types.
- This enables you to create more flexible and reusable code by writing algorithms and data structures that can operate on a variety of data types without sacrificing type safety.

Why use Generics in Java?

- Type Safety: Generics enhance compile-time type safety, minimizing the risk of runtime errors by ensuring correct data types in your code.
- Code Reusability: Generics enable you to write code that can be reused with different data types.
- Readability and Maintainability: Generic code tends to be more readable as it abstracts away the details of specific data types.

Syntax and Basics

Declaring Generic Classes: The syntax for declaring a generic class involves introducing a type parameter inside angle brackets (‘<>’) after the class name. This type parameter represents a placeholder for the actual type that will be provided when an instance of the class is created.

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

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

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

In this example, Box is a generic class with a type parameter ‘T’. It can be used to create a box that can hold values of any type.

Syntax and Basics

Using Generic Methods: Similar to generic classes, you can also create generic methods in Java. A generic method is a method that introduces its own type parameters, independent of the class to which it belongs.

```
public class Utils {
    public static <T> T genericMethod(T[] array) {
        if (array != null && array.length > 0) {
            return array[0];
        }
        return null;
    }
}
```

'genericMethod' is a generic method that works with arrays of any type. The type parameter <T> is used to represent the element type of the array.

Syntax and Basics

Implementing Generic Interfaces: Similar to classes, interfaces in Java can also be generic. When implementing a generic interface, you need to provide the actual type argument that the interface will use.

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

public 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;
    }
```

```
    @Override
    public K getKey() {
        return key;
    }

    @Override
    public V getValue() {
        return value;
    }
}
```

Wildcards and Bounded Types

Unbounded Wildcards: An unbounded wildcard is denoted by the question mark (?) and is used to represent an unknown type. It allows for flexibility when dealing with generic types without specifying a concrete type. Unbounded wildcards are often used when the code doesn't need to know the actual type but can operate on any type.

```
public void processList(List<?> list) {
    // Process the list without knowing its actual element type
    for (Object element : list) {
        // ...
    }
}
```

In this example, List<?> indicates that the method ‘processList’ can accept a list of any type, but the method itself doesn't need to know the specific type of elements in the list.

Wildcards and Bounded Types

Upper Bounded Wildcards: An upper-bounded wildcard is denoted by '<? extends T>', where 'T' is a specific type or a class. It allows the generic type to be any type that is a subtype of 'T'. This is useful when you want to accept a range of types that inherit from a common base type.

```
public double sumOfList(List<? extends Number> list) {
    double sum = 0.0;
    for (Number number : list) {
        sum += number.doubleValue();
    }
    return sum;
}
```

In this example, 'List<? extends Number>' means that the method 'sumOfList' can accept a list of any type that extends Number (e.g., Integer, Double, etc.).

Wildcards and Bounded Types

Lower Bounded Wildcards: A lower-bounded wildcard is denoted by '<? super T>', where 'T' is a specific type or a class. It allows the generic type to be any type that is a supertype of 'T'. This is useful when you want to accept a range of types that are superclass of a specific type.

```
public void addIntegers(List<? super Integer> list) {
    // Add integers to the list
    list.add(1);
    list.add(2);
}
```

In this example, 'List<? super Integer>' means that the method 'addIntegers' can accept a list of any type that is a superclass of 'Integer'. This could be a 'List<Object>' or a 'List<Number>', for example.

Wildcards and Bounded Types

When to Use Wildcards and Bounded Types:

- **Unbounded Wildcards:** Use when the code doesn't need to know the specific type and can operate on objects of any type.
- **Upper Bounded Wildcards**: Use when the code requires a range of types that share a common supertype.
- **Lower Bounded Wildcards**: Use when the code requires a range of types that are superclasses of a specific type.

Wildcards and bounded types provide flexibility in designing generic code, allowing methods and classes to work with a variety of types while maintaining type safety. They are powerful tools for creating generic and reusable components in Java.

References

The Java Tutorial on Generics
https://docs.oracle.com/javase/tutorial/java/generics/index.html

Another Tutorial on the same subject, using different examples
https://docs.oracle.com/javase/tutorial/extra/generics/index.html

THANK YOU