Bounded Generics in Java Programming

Ritwik Banerjee

rbanerjee@cs.stonybrook.edu
Type Parameters

- Generics allow us to use “types” (i.e., classes and interfaces) as parameters when defining new classes, interfaces or methods.
- When we talk of parameters, there are two broad categories:
  - **Formal parameters** used in method declaration.
  - **Type parameters** so that we can re-use code with different types of inputs.
- Inputs to formal parameters are **values**.
- Inputs to type parameters are **types**.
Bounded Type Parameters

- Sometimes, we may want to restrict the types that can be used as type arguments in a parameterized type.
  - An operation on numbers should only accept inputs that are instances of `java.lang.Number` or its subclasses.
- This is where a **bounded type parameter** comes in.
  - A type parameter that is restricted to a certain part of the Java class hierarchy.

Expressing Restrictions ... *in terms of the Java class hierarchy*

- A descendant (i.e., subclass) of `Number`
  - `C` extends `java.lang.Number`
- An ancestor (i.e., superclass) of `Number`
  - `C` super `java.lang.Number`
An example of type parameter restriction

```java
public class Box<T> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }

    // Number is the upper bound of the parameter U
    public <U extends Number> void inspect(U u) {
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.set(new Integer(10));
        integerBox.inspect("some text"); // Error!
    }
}
```
public class NaturalNumber<T extends Integer> {
    private T n;
    public NaturalNumber(T n) { this.n = n; }
    public boolean isEven() { return n.intValue() % 2 == 0; }
}

The `isEven` method invokes the `intValue` method defined in the superclass `Integer`.

Methods defined within bounds
Multiple Bounds

- A type parameter can have multiple bounds
- In such cases, the type variable is a subtype of all the types listed in the bound.

```java
Class A { ... }
interface B { ... }
interface C { ... }
```

- Then, we may write a bounded type of the form
  ```java
class D <T extends A & B & C>
  ```
  - If one of the bounds is a class, it must be specified first. Otherwise there will be a compile-time error.
  ```java
  class D <T extends B & A> ← Compile-time error
  ```
Generic Methods with Bounded Type Parameters

A generic method that counts the number of elements in a set that are greater than a specified element:

```java
static <T> int countGreater(Set<T> aSet, T t) {
    int count = 0;
    for (T item : aSet)
        if (item > t)
            ++count;
    return count;
}
```

This code will fail to compile. Why?

- The operator `>` applies only to primitive types (`short`, `int`, `double`, `long`, `float`, `byte`, `char`). It cannot be used to compare objects.

- To fix this, use a type parameter bounded by the `Comparable` interface.
A generic method that counts the number of elements in a set that are greater than a specified element:

```java
static <T> int countGreater(Set<T> aSet, T t) {
    int count = 0;
    for (T item : aSet)
        if (item > t)
            ++count;
    return count;
}
```

This code will fail to compile. **WHY?**

The operator `>` applies only to primitive types (`short`, `int`, `double`, `long`, `float`, `byte`, `char`). It cannot be used to compare objects. To fix this, use a type parameter bounded by the `Comparable` interface.
Generic Methods with Bounded Type Parameters

A generic method that counts the number of elements in a set that are greater than a specified element:

```java
static <T> int countGreater(Set<T> aSet, T t) {
    int count = 0;
    for (T item : aSet)
        if (item > t)
            ++count;
    return count;
}
```

This code will fail to compile. **WHY?**

- The operator `>` applies only to primitive types (`short, int, double, long, float, byte, char`). It cannot be used to compare objects.
- To fix this, use a type parameter **bounded by** the `Comparable` interface.
A generic method that counts the number of elements in a set that are greater than a specified element:

```java
static <T extends Comparable<T>>
    int countGreater(Set<T> aSet, T t) {
        int count = 0;
        for (T item : aSet)
            if (item.compareTo(t) > 0)
                ++count;
        return count;
    }
```
In Java, if a type parameter is unknown, it can represented by the wildcard “?”. We can use wildcards to represent

- Unbounded type parameters,
- Type parameters with an upper bound, and
- Type parameters with a lower bound.
Upper Bounded Wildcards

Say you want to write a method that works on `List<Integer>`, `List<Double>`, and `List<Number>`.

Clearly, you want to write a piece of code that works for `Number` and its subclasses.

This is where a wildcard should be used, but with an upper bound:

```java
<? extends Number>

static double sum(List<? extends Number> numbers) {
    double sum = 0d;
    for (Number aNumber : numbers)
        sum += aNumber.doubleValue();
    return sum;
}
```
Lower Bounded Wildcards

- In a different scenario, suppose you want to add integers to a list.
- And you want to write your code such that you can add your integers to a list of `Integers`, or a list of `Doubles`, or a list of `Numbers`.
- In this case, you want to work with anything that is a “supertype” of `Integer`.
  - Here, we want a wildcard with a lower bound:
    ```java
    <? super Number>
    ```

    ```java
    static void addTo(int i,
                      List<? super Integer> numbers) {
        numbers.add(i);
    }
    ```