Bounded Types with Generics in Java

There may be times when you want to restrict the types that can be used as type arguments in a parameterized type. For example, a method that operates on numbers might only want to accept instances of Numbers or their subclasses. This is what bounded type parameters are for.

- Sometimes we don't want the whole class to be parameterized. In that case, we can create a Java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too.
- Suppose we want to restrict the type of objects that can be used in the parameterized type. For example, in a method that compares two objects and we want to make sure that the accepted objects are Comparables.
- The invocation of these methods is similar to the unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error.

How to Declare a Bounded Type Parameter in Java?

1. List the type parameter's name,
2. Along with the extends keyword
3. And by its upper bound. (which in the below example c is **A**.)

Syntax

```
<T extends superClassName>
```

Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes). Also, This specifies that T can only be replaced by superClassName or subclasses of superClassName. Thus, a superclass defines an inclusive, upper limit.

}

Let's take an example of how to implement bounded types (extend superclass) with generics.

Java

```
// This class only accepts type parameters as any class
// which extends class A or class A itself.
// Passing any other type will cause compiler time error

class Bound<T extends A>
{

    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}

class A
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}

class B extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class B");
    }
}

class C extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}

public class BoundedClass
{
    public static void main(String a[])
    {

        // Creating object of sub class C and
        // passing it to Bound as a type parameter.
        Bound<C> bec = new Bound<C>(new C());
        bec.doRunTest();

        // Creating object of sub class B and
        // passing it to Bound as a type parameter.
        Bound<B> beb = new Bound<B>(new B());
        beb.doRunTest();

        // similarly passing super class A
        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();

    }
}
```

Output

```
Inside sub class C
Inside sub class B
Inside super class A
```

Now, we are restricted to only type A and its subclasses, So it will throw an error for any other type of subclasses.

Java

```
// This class only accepts type parameters as any class
// which extends class A or class A itself.
// Passing any other type will cause compiler time error
```

```java
class Bound<T extends A>
{

    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}

class A
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}

class B extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class B");
    }
}

class C extends A
{
    public void displayClass()
    {
        System.out.println("Inside sub class C");
    }
}

public class BoundedClass
{
    public static void main(String a[])
    {
        // Creating object of sub class C and
        // passing it to Bound as a type parameter.
        Bound<C> bec = new Bound<C>(new C());
        bec.doRunTest();

        // Creating object of sub class B and
        // passing it to Bound as a type parameter.
        Bound<B> beb = new Bound<B>(new B());
        beb.doRunTest();
        // similarly passing super class A
        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();

        Bound<String> bes = new Bound<String>(new String());
        bea.doRunTest();
    }
}
```

Output :

```
error: type argument String is not within bounds of type-variable T
```

Multiple Bounds

Bounded type parameters can be used with methods as well as classes and interfaces.

Java Generics supports multiple bounds also, i.e., In this case, A can be an interface or class. If A is class, then B and C should be interfaces. We can't have more than one class in multiple bounds.

Syntax:

[illegible]

Java

```
class Bound<T extends A & B>
{

    private T objRef;

    public Bound(T obj){
        this.objRef = obj;
    }

    public void doRunTest(){
        this.objRef.displayClass();
    }
}

interface B
{
    public void displayClass();
}
```

```
class A implements B
{
    public void displayClass()
    {
        System.out.println("Inside super class A");
    }
}

public class BoundedClass
{
    public static void main(String a[])
    {
        //Creating object of sub class A and
        //passing it to Bound as a type parameter.
        Bound<A> bea = new Bound<A>(new A());
        bea.doRunTest();

    }
}
```

Output

```
Inside super class A
```

This article is contributed by **Saket Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Last Updated : 15 Mar, 2022

Similar Reads

1. Generics in Java
2. Templates in C++ vs Generics in Java
3. How to Implement Queue in Java using Array and Generics?
4. Program to convert a Set to Stream in Java using Generics