Sri SAI RAM ENGINEERING COLLEGE
INSTITUTE OF TECHNOLOGY
West Tambaram, Chennai - 44

SAIRAM DIGITAL RESOURCES

YEAR II

SEM III

CS8392

OBJECT ORIENTED PROGRAMMING
(Common to CSE, EEE, EIE, ICE, IT)

UNIT NO 4

Mulithreading and Generic Programming

4.9 Bounded types-Restrictions and Limitation

COMPUTER SCIENCE & ENGINEERING

Bounded Type Parameters:

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 Number or its subclasses

Bounded type parameters

- Sometimes we don't want whole class to be parameterized, in that case we can create java generics method.
- Since 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 unbounded method except that if we will try to use any class that is not Comparable, it will throw compile time error

Declare a bounded type parameter

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

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its *upper bound*, which in this example is Number.

Syntax

<T extends **superClassName**>

In the above syntax, 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, superclass defines an inclusive, upper limit.

Example on how to implement bounded types (extend superclass) with generics

```
// This class only accepts type parametes 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 restricted to only of type A and its sub classes, So it will throw an error for any other type or sub classes.

```
 // This class only accepts type parametes 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[])
        {       Bound<C> bec = new Bound<C>(new C());
                bec.doRunTest();
                Bound<B> beb = new Bound<B>(new B());
                beb.doRunTest();
                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
```

ANOTHER EXAMPLE PROGRAM

```
public class Box<T> {

    private T t;

    public void set(T t) {
    this.t = t;
    }

    public T get() {
    return t;
    }
```

By modifying our generic method to include this bounded type parameter, compilation will now fail, since our invocation of inspect still includes a String:

```
Box.java:21: <U>inspect(U) in Box<java.lang.Integer> cannot be applied to (java.lang.String)
integerBox.inspect("10");
                  ^
1 error
```

In addition to limiting the types you can use to instantiate a generic type, bounded type parameters allow you to invoke methods defined in the bounds:

```
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 Integer class through n.

Multiple Bounds

The preceding example illustrates the use of a type parameter with a single bound, but a type parameter can have *multiple bounds:*

<T extends B1 & B2 & B3>

A type variable with multiple bounds is a subtype of all the types listed in the bound. If one of the bounds is a class, it must be specified first. For example:

```
Class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C> { /* ... */ }
```

If bound A is not specified first, you get a compile-time error:

```
class D <T extends B & A & C> { /* ... */ }  // compile-time error
```

Bounded type parameters can be used with methods as well as classes andinterfaces.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 : <T extends **superClassName& Interface**>

```
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[])
        {           Bound<A> bea = new Bound<A>(new A());
                    bea.doRunTest();
         }       }
```

OUTPUT:

```
Inside super class A
```

Restrictions

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or instanceof With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Cannot Instantiate Generic Types with Primitive Types

```
class Pair<K, V> {

    private K key;
    private V value;
        public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }
   }
```

When creating a Pair object, you cannot substitute a primitive type for the type parameter K or V:

Pair<`int, char`> p = new Pair<>(8, 'a'); // compile-time error

You can substitute only non-primitive types for the type parameters K and V:

Pair<`Integer, Character`> p = new Pair<>(8, 'a');

Note that the Java compiler autoboxes 8 to Integer.valueOf(8) and 'a' to Character('a'):

Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

Cannot Create Instances of Type Parameters

You cannot create an instance of a type parameter. For example, the following code causes a compile-time error:

```
public static <E> void append(List<E> list) {
        E elem = new E();  // compile-time error
        list.add(elem);
}
```

As a workaround, you can create an object of a type parameter through reflection:

```
public static <E> void append(List<E> list, Class<E> cls) throws Exception {
        E elem = cls.newInstance();   // OK
        list.add(elem);
}
```

You can invoke the append method as follows:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Cannot Declare Static Fields Whose Types are Type Parameters

A class's static field is a class-level variable shared by all non-static objects of the class. Hence, static fields of type parameters are not allowed. Consider the following class:

```
public class MobileDevice<T> {
        private static T os;


        // …}
```

If static fields of type parameters were allowed, then the following code would be confused:

```
MobileDevice<Smartphone> phone = new MobileDevice<>();
MobileDevice<Pager> pager = new MobileDevice<>();
MobileDevice<TabletPC> pc = new MobileDevice<>();
```

Because the static field os is shared by phone, pager, and pc, what is the actual type of os? It cannot be Smartphone, Pager, and TabletPC at the same time. You cannot, therefore, create static fields of type parameters.

Cannot Use Casts or instanceof with Parameterized Types

Because the Java compiler erases all type parameters in generic code, you cannot verify which parameterized type for a generic type is being used at runtime:

```
public static <E> void rtti(List<E> list) {
        if (list instanceof ArrayList<Integer>) {  // compile-time error
        // ...
        }
  }
```

The set of parameterized types passed to the rtti method is:

S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

The runtime does not keep track of type parameters, so it cannot tell the difference between an ArrayList<Integer> and an ArrayList<String>. The most you can do is to use an unbounded wildcard to verify that the list is an ArrayList:

```
public static void rtti(List<?> list) {
        if (list instanceof ArrayList<?>) {  // OK; instanceof requires a reifiable type
        // ...
        }
  }     }
```

Typically, you cannot cast to a parameterized type unless it is parameterized by unbounded wildcards. For example:

```
List<Integer> li = new ArrayList<>();
List<Number>  ln = (List<Number>) li;  // compile-time error
```

However, in some cases the compiler knows that a type parameter is always valid and allows the cast. For example:

```
List<String> l1 = ...;
ArrayList<String> l2 = (ArrayList<String>)l1;  // OK
```

Cannot Create Arrays of Parameterized Types

You cannot create arrays of parameterized types. For example, the following code does not compile:

```
List<Integer>[] arrayOfLists = new List<Integer>[2];  // compile-time error
```

The following code illustrates what happens when different types are inserted into an array:

```
Object[] strings = new String[2];
strings[0] = "hi";   // OK
strings[1] = 100;       // An ArrayStoreException is thrown.
```

If you try the same thing with a generic list, there would be a problem:

```
Object[] stringLists = new List<String>[];  // compiler error, but pretend it's allowed
stringLists[0] = new ArrayList<String>();   // OK
stringLists[1] = new ArrayList<Integer>();  // An ArrayStoreException should be thrown,
                                            // but the runtime can't detect it.
```

If arrays of parameterized lists were allowed, the previous code would fail to throw the desired ArrayStoreException.

Cannot Create, Catch, or Throw Objects of Parameterized Types

generic class cannot extend the Throwable class directly or indirectly. For example, the following classes will not compile:

```
// Extends Throwable indirectly
   class MathException<T> extends Exception { /* ... */ }     // compile-time error
    // Extends Throwable directly
   class QueueFullException<T> extends Throwable { /* ... */ // compile-time error
```

A method cannot catch an instance of a type parameter:

```
public static <T extends Exception, J> void execute(List<J> jobs) {
         try {
         for (J job : jobs)
                  // ...
         } catch (T e) {   // compile-time error
         // ...
         }}
```

You can, however, use a type parameter in a throws clause:

```
class Parser<T extends Exception>
 {      public void parse(File file) throws T {      // OK


   // ...} }
```

Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw

TypeA class cannot have two overloaded methods that will have the same signature after type erasure.

```
public class Example {      public void print(Set<String> strSet) { }


      public void print(Set<Integer> intSet) { }        }
```

The overloads would all share the same classfile representation and will generate a compile-time error.