Bounded Type Parameters

…

What is a bounded Type Parameter?

- Restrict the types that may be used as type arguments for a parameterized type
- E.g. a generic methods will work only with Integer-like types

Using bounded types with Generic Methods

Please Note
This means “subtype”
not inheritance

```
public class Box<T> {

    private T t;

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

    public <U extends Number> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }
}
```

Let's check that Java does its job

```
public static void main(String[] args) {

    Box<Integer> integerBox = new Box<Integer>();
    integerBox.set(new Integer(10));

    integerBox.inspect("some text");
    // error: this actual parameter is 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
```

Extra Free Feature!
Bounded Types Parameters also allow 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;
    }

    // ...
}
```

isEven(…) able to call intValue(…) Since n is of data type T

n is of data type T means…

- T happens to extend Integer
- thus T features intValue() method

New Requirement!
What about introducing Multiple Bounds?

Definition

```
<T extends B1 & B2 & B3>
```

Note:
This will be referred to as the leftmost bounded type

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

Example featuring a class

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>     { /* ... */ }
```

Debug this Example

```
public static <T> int countGreaterThan(T[] anArray, T elem) {

    // Return the # of elements greater than elem

    int count = 0;

    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;

    return count;
}
```

What do you think is the problem here?

The > operator works only w/ primitive data types…

… not objects

Debug this Example

```
public static <T> int countGreaterThan(T[] anArray, T elem) {

    // Return the # of elements greater than elem

    int count = 0;

    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;

    return count;
}
```

How do we fix this problem?

T must extend the Comparable <T> interface

Now we may rewrite our original attempt, using this new Comparable<T> interface

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

NOTE

We use “extends” not “implements”

```
public static <T extends Comparable<T>>
int countGreaterThan(T[] anArray, T elem){
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0)
            ++count;
    return count;
}
```

Remember…

This is about subtyping not inheritance or interface implementations