Bounded Type Parameters

Upper bounds (`<T extends ...>`), multiple bounds joined with `&`, bounding with classes and with interfaces, and how bounding a type parameter expands which methods can be called on it. The 3rd lesson in the Generics series.

Intermediate 20 min

Last updated: November 7, 2026
Read online: https://www.learnforgex.com/en/topics/bounded-type-parameters

---

Every type parameter you've written so far — `T` in `Box<T>`, `T` in `firstElement(...)` — has been UNBOUNDED: it could be filled in with literally any type at all. That flexibility has a real cost, though, as you're about to see: an unbounded `T` is also the LEAST you can know about it. This lesson covers narrowing that down on purpose.

What Is a Bounded Type Parameter?

A bounded type parameter restricts which types are allowed to fill it in — instead of accepting anything, it only accepts a type (or one of its subtypes) that satisfies a stated requirement. In exchange for that restriction, the compiler now knows more about what a value of that type parameter can actually do, and lets you call methods on it that an unbounded `T` would never allow.

Why Do They Exist?

An unbounded `T` could be absolutely anything, so the compiler can only assume it has the methods every `Object` has — `toString()`, `equals(...)`, and nothing more specific.

```
public class UnboundedMethodCallLimitationExample {

    // With an UNBOUNDED type parameter, the compiler only knows T could be
    // literally anything -- so the only methods it will let you call on a
    // T value are the ones every single Object has (toString, equals,
    // hashCode, ...). Nothing more specific is available.
    static <T> String describe(T value) {
        return value.toString(); // fine -- toString() belongs to Object

        // return value.doubleValue(); // would NOT compile -- the compiler
        //                                has no idea T even has a
        //                                doubleValue() method, because an
        //                                unbounded T might not.
    }

    public static void main(String[] args) {
        System.out.println(describe(42));
        System.out.println(describe("hello"));
        System.out.println(describe(3.14));
    }
}
```

Expected Output

```
42
hello
3.14
```

`describe(...)` can call `value.toString()` because every `Object` has one, but nothing beyond that is available — there's no way to call a method specific to numbers, or to comparison, or to anything else, because an unbounded `T` gives the compiler no such guarantee. Bounded type parameters exist to make that guarantee possible.

Upper Bounds with extends

Writing `<T extends SomeType>` declares an UPPER BOUND: `T` must be `SomeType` itself or one of its subtypes — nothing outside that family is allowed. The keyword is `extends` even when the bound is an interface, not just a class.

```
import java.util.List;

public class UpperBoundedSumExample {

    // "T extends Number" is an UPPER BOUND: T can be Number itself or any
    // subclass of it (Integer, Double, Long, ...) -- nothing else is
    // allowed. In exchange, the compiler now knows every T value has every
    // method Number declares, like doubleValue().
    static <T extends Number> double sum(List<T> numbers) {
        double total = 0;
        for (T number : numbers) {
            total += number.doubleValue(); // only legal because of the bound
        }
        return total;
    }

    public static void main(String[] args) {
        System.out.println(sum(List.of(1, 2, 3)));       // T = Integer
        System.out.println(sum(List.of(1.5, 2.5)));       // T = Double

        // sum(List.of("a", "b")); // would NOT compile -- String isn't a Number
    }
}
```

Expected Output

```
6.0
4.0
```

`sum(List<T> numbers)` with `T extends Number` can call `number.doubleValue()` on every element, because the bound guarantees every possible `T` — `Integer`, `Double`, `Long`, or any other `Number` subtype — has that method. Calling `sum(...)` with a `List<String>` simply doesn't compile, since `String` isn't a `Number`.

Multiple Bounds

A type parameter can be bound by more than one requirement at once, joined with `&`. At most one of the bounds may be a class, and if there is one, it must come first; the rest must be interfaces.

```
import java.util.List;

public class MultipleBoundsExample {

    // Multiple bounds are joined with "&": T must satisfy ALL of them at
    // once. At most one bound may be a class (and it must come first if
    // present); the rest must be interfaces. Here T must be both a Number
    // AND Comparable to itself, so the method can use doubleValue() from
    // Number and compareTo(...) from Comparable in the same method.
    static <T extends Number & Comparable<T>> T max(List<T> values) {
        T largest = values.get(0);
        for (T value : values) {
            if (value.compareTo(largest) > 0) {
                largest = value;
            }
        }
        return largest;
    }

    public static void main(String[] args) {
        System.out.println(max(List.of(3, 7, 2, 9, 4)));      // T = Integer
        System.out.println(max(List.of(1.5, 3.2, 0.8)));       // T = Double
    }
}
```

Expected Output

```
9
3.2
```

`<T extends Number & Comparable<T>>` requires `T` to be both a `Number` AND comparable to itself — the method body can freely call both `doubleValue()` (from the `Number` bound) and `compareTo(...)` (from the `Comparable` bound) on the same value.

Bounding with a Class

The bound doesn't have to appear only on a method — a generic CLASS's type parameter can be bounded too, restricting every use of that class the same way.

```
public class BoundedGenericClassExample {

    // The bound can be declared on a CLASS's type parameter too, not just
    // a method's -- every use of NumericBox is now restricted to Number
    // subtypes, and every method inside the class can rely on that.
    static class NumericBox<T extends Number> {
        private final T value;

        NumericBox(T value) {
            this.value = value;
        }

        boolean isPositive() {
            return value.doubleValue() > 0; // legal, thanks to the class-level bound
        }

        T getValue() {
            return value;
        }
    }

    public static void main(String[] args) {
        NumericBox<Integer> intBox = new NumericBox<>(42);
        System.out.println(intBox.isPositive());

        NumericBox<Double> doubleBox = new NumericBox<>(-3.5);
        System.out.println(doubleBox.isPositive());

        // NumericBox<String> stringBox = new NumericBox<>("hi"); // would NOT
        //     compile -- String does not extend Number, so it fails the bound.
    }
}
```

Expected Output

```
true
false
```

`NumericBox<T extends Number>` means `NumericBox<String>` simply cannot be written — it fails to compile, because `String` doesn't satisfy the bound. Every method inside `NumericBox` can rely on `value` having `Number`'s methods, exactly as `sum(...)` could above.

Bounding with an Interface

A bound doesn't need a class at all — bounding purely by an interface is just as common, and often more general, since it isn't tied to any particular type hierarchy.

```
import java.util.List;

public class PracticalMaxFinderExample {

    // Bounding by an INTERFACE alone (no class involved) is just as
    // common as bounding by a class -- here T is only required to be
    // Comparable to itself, which is enough for a general-purpose "find
    // the largest element" utility that works for String, Integer, or any
    // other Comparable type, not just numbers.
    static <T extends Comparable<T>> T max(List<T> items) {
        T largest = items.get(0);
        for (T item : items) {
            if (item.compareTo(largest) > 0) {
                largest = item;
            }
        }
        return largest;
    }

    public static void main(String[] args) {
        System.out.println(max(List.of("banana", "apple", "cherry"))); // T = String
        System.out.println(max(List.of(5, 1, 9, 3)));                  // T = Integer
    }
}
```

Expected Output

```
cherry
9
```

`<T extends Comparable<T>>` accepts any type that can compare itself to another of the same type — `String`, `Integer`, and plenty of your own classes all qualify, with no relationship to `Number` required at all. This is the same shape you'll see used heavily once "Wildcards" introduces `<? extends T>` for a related but different purpose — bounding a type parameter and bounding a wildcard use the same `extends` keyword, but answer different questions.

> `<T extends Comparable<T>>` is one of the most common bounds you'll see in real Java code — it's exactly what lets a single generic method compute a maximum, a minimum, or a sort order for any comparable type, not just numbers.

Best Practices

- Add a bound the moment your generic code needs to call a method beyond what `Object` offers — an unbounded type parameter that quietly needs more is a sign the bound was forgotten, not a sign it's unnecessary.
- Prefer bounding by an interface (like `Comparable<T>`) over a concrete class whenever the requirement is really "can do this operation," not "must literally be this type or a subtype of it."

- When combining bounds, remember the class (if any) must come first, followed by interfaces, all joined with `&`.
- Keep a bound as narrow as the method or class genuinely requires — bounding by `Number` when you only ever call `toString()` gains nothing and needlessly restricts callers.

Common Mistakes

- Forgetting the bound entirely and then being surprised the compiler rejects a call to a method you know every realistic argument will have.
- Writing `<T extends Comparable & Number>` with the interface first — this doesn't compile; a class bound, if present, must always come first.
- Assuming a bound restricts what the type parameter's OWN class can do, rather than restricting which types are allowed to be substituted in for it — the bound describes the argument, not the generic class or method itself.
- Reaching for `Object` as a workaround instead of a proper bound, losing all of the specific-method access a bound would have provided.

Summary, Cheat Sheet, and Glossary

Summary

- An unbounded type parameter only guarantees `Object`'s methods; a bounded one guarantees more, in exchange for restricting which types qualify.
- `<T extends SomeType>` declares an upper bound, using `extends` for both classes and interfaces.
- Multiple bounds are joined with `&`; at most one may be a class, and it must come first.
- A bound can appear on a class's type parameter, restricting every use of that class, not just a single method.
- Bounding by an interface (like `Comparable<T>`) is common and general-purpose, independent of any specific class hierarchy.

Cheat Sheet

```
// Upper bound with a class
static <T extends Number> double sum(List<T> numbers) {
    double total = 0;
    for (T n : numbers) total += n.doubleValue();
    return total;
}

// Multiple bounds: class first, then interfaces, joined with &
static <T extends Number & Comparable<T>> T max(List<T> values) { ... }

// Bound on a class's own type parameter
class NumericBox<T extends Number> { ... }

// Bound by an interface alone
static <T extends Comparable<T>> T max(List<T> items) { ... }
```

Glossary

- **Bounded type parameter**: a type parameter restricted to a specific type (and its subtypes) rather than accepting any type.
- **Upper bound**: the restriction declared with `extends`, allowing the bound type itself or any of its subtypes.
- **Multiple bounds**: two or more requirements joined with `&`, all of which a type must satisfy.
- **Bound**: the type (class or interface) a type parameter is restricted to extend or implement.