Wildcards

The unbounded wildcard (`<?>`), the upper bounded wildcard (`<? extends T>`), the lower bounded wildcard (`<? super T>`), the difference between the three, and the PECS (Producer Extends, Consumer Super) rule -- the most commonly misunderstood part of Java generics, walked through step by step with real examples. The 4th lesson in the Generics series.

Intermediate 30 min

Last updated: November 7, 2026
Read online: https://www.learnforgex.com/en/topics/wildcards

"Bounded Type Parameters" restricted what a type parameter like `T` could be filled in with. Wildcards solve a related but different problem: what to do at a USE SITE — a method parameter, a field, a variable — when you want to accept a generic type without committing to exactly which type argument it was built with. This is one of the most commonly misunderstood parts of Java generics, so this lesson takes it slowly.

What Is a Wildcard?

A wildcard, written `?`, stands in for an UNKNOWN type argument at a specific use of a generic type — `List<?>` means "a `List` of some type, I'm not saying which." Unlike a type parameter (`T`), a wildcard never gets a name and is never used to declare new generic classes or methods — it only appears where a generic type is being USED, like a parameter type.

Why Do They Exist?

Java generics are INVARIANT: even though `Integer` IS-A `Number`, `List<Integer>` is NOT a `List<Number>` — they're treated as two completely unrelated types.

```
import java.util.List;

public class WildcardMotivationExample {

    // Generics are INVARIANT: even though Integer IS-A Number, List<Integer>
    // is NOT a List<Number>. A parameter typed List<Number> only accepts
    // exactly that -- a List<Number> -- nothing else, no matter how closely
    // related the element types are.
    static double sumNumbers(List<Number> numbers) {
        double total = 0;
        for (Number n : numbers) {
            total += n.doubleValue();
        }
        return total;
    }

    public static void main(String[] args) {
        List<Number> mixed = List.of(1, 2.5, 3L);
        System.out.println(sumNumbers(mixed)); // fine -- this really is a List<Number>

        List<Integer> integers = List.of(1, 2, 3);
        // System.out.println(sumNumbers(integers)); // would NOT compile --
        //     List<Integer> is not a List<Number>, even though every
        //     Integer is a Number. This is exactly the problem wildcards
        //     exist to solve.
    }
}
```

Expected Output

```
6.5
```

`sumNumbers(List<Number> numbers)` only accepts a parameter that is EXACTLY `List<Number>` — a `List<Integer>`, however closely related its element type is, is rejected outright. Without some other tool, you'd need a separate overload for every possible element type just to sum a list of numbers. Wildcards exist to let a method accept a whole FAMILY of related type arguments through one single, flexible parameter type. ("Generics with Collections" looks at this same invariance rule again, specifically from the angle of collections like `List` and `Map`.)

Unbounded Wildcard: <?>

`List<?>` accepts a `List` of any element type whatsoever — the right tool when a method genuinely doesn't care what the elements are, and only needs operations that work no matter what.

```
import java.util.List;

public class UnboundedWildcardExample {

    // List<?> -- an UNBOUNDED wildcard -- accepts a List of ANY element
    // type. It's the right choice when the method genuinely doesn't care
    // what the element type is, and only needs operations every List
    // supports regardless of what it holds.
    static void printSize(List<?> list) {
        System.out.println("size: " + list.size());
        for (Object item : list) { // elements can only be read as Object
            System.out.println(" - " + item);
        }
    }

    public static void main(String[] args) {
        printSize(List.of("a", "b", "c"));
        printSize(List.of(1, 2, 3));
        printSize(List.of(true, false));
    }
}
```

Expected Output

```
size: 3
 - a
 - b
 - c
size: 3
 - 1
 - 2
 - 3
size: 2
 - true
 - false
```

`printSize(...)` works on a `List<String>`, a `List<Integer>`, or anything else — it never needs to know the element type, since it only calls `size()` and reads elements as `Object`.

Upper Bounded Wildcard: `<? extends T>`

`List<? extends Number>` accepts a `List` of `Number` OR any of its subtypes — `List<Integer>`, `List<Double>`, `List<Number>` itself, all qualify.

```
import java.util.List;

public class UpperBoundedWildcardProducerExample {

    // List<? extends Number> accepts a List of Number OR any of its
    // subtypes -- List<Integer>, List<Double>, List<Number> itself, all
    // qualify. This method only ever READS from the list -- it PRODUCES
    // values for the caller -- which is exactly what "extends" is for.
    static double sum(List<? extends Number> numbers) {
        double total = 0;
        for (Number n : numbers) { // reading is always safe: every element IS a Number
            total += n.doubleValue();
        }
        return total;

        // numbers.add(42); // would NOT compile -- the compiler doesn't
        //     know the list's REAL type (it could be a List<Double>), so
        //     it can't verify an Integer is safe to insert.
    }

    public static void main(String[] args) {
        System.out.println(sum(List.of(1, 2, 3)));      // List<Integer>
        System.out.println(sum(List.of(1.5, 2.5)));      // List<Double>
        System.out.println(sum(List.of(1, 2.5, 3L)));     // List<Number>
    }
}
```

Expected Output

```
6.0
4.0
6.5
```

`sum(...)` only ever READS from the list — every element, whatever its exact type, is guaranteed to be at least a `Number`, so `n.doubleValue()` is always safe to call. What ISN'T safe is adding to it: the compiler has no way to know the list's real element type (it could specifically be a `List<Double>`), so it refuses to let you insert anything, even an `Integer`.

Lower Bounded Wildcard: `<? super T>`

`List<? super Integer>` accepts a `List` of `Integer` OR any of its SUPERTYPES — `List<Integer>`, `List<Number>`, `List<Object>` all qualify.

```
import java.util.ArrayList;
import java.util.List;

public class LowerBoundedWildcardConsumerExample {

    // List<? super Integer> accepts a List of Integer OR any of its
    // SUPERTYPES -- List<Integer>, List<Number>, List<Object> all qualify.
    // This method only ever WRITES into the list -- it CONSUMES values
    // from the caller -- which is exactly what "super" is for.
    static void addOneToFive(List<? super Integer> list) {
        for (int i = 1; i <= 5; i++) {
            list.add(i); // always safe: whatever the real type is, it can hold an Integer
        }

        // Integer first = list.get(0); // would NOT compile -- the
        //     compiler only knows the list holds SOME supertype of
        //     Integer, which could be as broad as Object.
        Object first = list.get(0); // reading back is only safe as Object
        System.out.println("first element read back as Object: " + first);
    }

    public static void main(String[] args) {
        List<Number> numbers = new ArrayList<>();
        addOneToFive(numbers); // List<Number> is a valid "supertype of Integer" list

        List<Object> objects = new ArrayList<>();
        addOneToFive(objects); // List<Object> works too

        System.out.println(numbers);
        System.out.println(objects);
    }
}
```

Expected Output

```
first element read back as Object: 1
first element read back as Object: 1
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5]
```

`addOneToFive(...)` only ever WRITES into the list — an `Integer` is always safe to insert, no matter which supertype of `Integer` the list actually holds. What ISN'T safe is reading a specific type back out: the compiler only guarantees the list holds SOME supertype of `Integer`, which could be as broad as `Object`, so a read can only be treated as `Object`.

Comparing the Three: What get and add Actually Allow

Placed side by side, the pattern behind all three forms becomes concrete.

```
import java.util.List;

public class WildcardGetPutRestrictionsExample {

    // A side-by-side look at exactly what each wildcard form allows,
    // using the SAME two operations (read an element, add an element) on
    // each -- this is the "get and put" rule the rest of this lesson
    // builds on.
    static void demonstrate(List<? extends Number> producer,
                            List<? super Integer> consumer,
                            List<?> unknown) {

        // GET is safe on "? extends" -- every element really is at least a Number.
        Number value = producer.get(0);
        // producer.add(1); // NOT allowed -- the real type could be narrower than Integer

        // PUT is safe on "? super" -- Integer fits into any supertype of Integer.
        consumer.add(99);
        // Integer fromConsumer = consumer.get(0); // NOT allowed -- could only be read as
Object

        // Neither is reliably safe on "?" -- only Object-level reads, and
        // no inserts at all (except the literal null, which fits every type).
        Object anything = unknown.get(0);
        unknown.add(null); // the one and only value that fits ANY element type
        // unknown.add("x"); // NOT allowed -- the real element type is unknown
    }

    public static void main(String[] args) {
        List<Integer> ints = new java.util.ArrayList<>(List.of(1, 2, 3));
        List<Number> nums = new java.util.ArrayList<>(List.of(1, 2, 3));
        demonstrate(ints, nums, ints);
        System.out.println(ints);
        System.out.println(nums);
    }
}
```

`<? extends Number>` lets you `get` safely but never `add` (except `null`, which fits any type). `<? super Integer>` lets you `add` an `Integer` safely but only `get` back an `Object`. Plain `<?>` allows neither a meaningful `get` beyond `Object` nor any `add` at all. This "get vs. put" behavior is the entire reason the next section's rule works.

PECS: Producer Extends, Consumer Super

PECS is a memorable rule of thumb for choosing which wildcard form to reach for: if a parameterized type only PRODUCES values for you (you only read from it), use `extends`; if it only CONSUMES values from you (you only write into it), use `super`. This is exactly what the previous two examples already showed — `sum(...)` only reads (`extends`), `addOneToFive(...)` only writes (`super`).

```
import java.util.ArrayList;
import java.util.List;

public class PecsCopyExample {

    // The textbook PECS method: copying reads from "src" (a PRODUCER of T,
    // so "extends") and writes into "dest" (a CONSUMER of T, so "super").
    // Neither wildcard could do this job alone -- "src" must be readable
    // as T, and "dest" must be writable with a T.
    static <T> void copy(List<? extends T> src, List<? super T> dest) {
        for (T item : src) {
            dest.add(item);
        }
    }

    public static void main(String[] args) {
        List<Integer> integers = List.of(1, 2, 3);
        List<Number> destination = new ArrayList<>();

        copy(integers, destination); // T is inferred as Number here

        System.out.println(destination);
    }
}
```

Expected Output

```
[1, 2, 3]
```

`copy(...)` needs BOTH roles at once: `src` is read from (a producer, so `extends`), and `dest` is written into (a consumer, so `super`). Neither wildcard form alone could do this job — `src` couldn't be `List<? super T>` (you can't reliably read a `T` back out of it), and `dest` couldn't be `List<? extends T>` (you can't add a `T` into it).

> If a parameter needs BOTH reading and writing of the same specific type, wildcards can't help — that parameter needs a plain, unbounded type parameter like `List<T>` instead, not a wildcard at all. Wildcards only apply when a parameter's role, as either a producer or a consumer, is clear-cut.

Best Practices

- Apply PECS directly: `extends` when a parameter only produces (you read from it), `super` when it only consumes (you only write into it).
- Reach for `<?>` when a method genuinely doesn't touch the element type at all — don't default to it out of uncertainty about which bound to use.
- Never add a wildcard to a return type — a method returning `List<? extends Number>` forces every caller to deal with an unknown type, with none of PECS's benefit, since there's no "reading" or "writing" happening at a return type.

- When a parameter needs to be both read from and written to with the same type, don't force a wildcard onto it — use an ordinary type parameter instead.

Common Mistakes

- Trying to `add(...)` to a `List<? extends T>` and being surprised the compiler rejects it — this is the single most common wildcard mistake, and it's PECS's `extends` rule working exactly as designed.
- Trying to read a specific type (not `Object`) back out of a `List<? super T>` — the compiler only guarantees a supertype of `T`, never anything narrower.
- Reaching for `<?>` when the method actually only reads (should be `? extends`) or only writes (should be `? super`) a specific type — this throws away information the compiler could otherwise use to catch mistakes.
- Confusing a wildcard (`?`, used only where a generic type is USED) with a type parameter (`T`, declared on a class or method) — a wildcard is never declared and never gets a name.

Summary, Cheat Sheet, and Glossary

Summary

- A wildcard (`?`) stands for an unknown type argument at a use of a generic type, unlike a named type parameter.
- Generics are invariant, so `List<Integer>` is not a `List<Number>` — wildcards exist to let a parameter accept a whole family of related type arguments.
- `<?>` (unbounded) accepts any element type but allows no meaningful reads or writes.
- `<? extends T>` (upper bounded) allows safe reads as `T` but no writes (except `null`).
- `<? super T>` (lower bounded) allows safe writes of `T` but only reads as `Object`.
- PECS: use `extends` for a producer (you read), `super` for a consumer (you write).

Cheat Sheet

```
// Unbounded: don't care about the element type
void printSize(List<?> list) { ... }

// Upper bounded: producer, only reads -- PECS: extends
double sum(List<? extends Number> numbers) {
    double total = 0;
    for (Number n : numbers) total += n.doubleValue();
    return total;
}

// Lower bounded: consumer, only writes -- PECS: super
void addOneToFive(List<? super Integer> list) {
    for (int i = 1; i <= 5; i++) list.add(i);
}

// Both roles at once -- PECS in full
static <T> void copy(List<? extends T> src, List<? super T> dest) {
    for (T item : src) dest.add(item);
}
```

Glossary

- **Wildcard**: `?`, standing for an unknown type argument at a use of a generic type.
- **Unbounded wildcard**: `<?>`, accepting any type argument at all.
- **Upper bounded wildcard**: `<? extends T>`, accepting `T` or any of its subtypes; safe to read, unsafe to write.
- **Lower bounded wildcard**: `<? super T>`, accepting `T` or any of its supertypes; safe to write, unsafe to read as anything but `Object`.
- **PECS**: "Producer Extends, Consumer Super" — the rule for choosing between `extends` and `super` based on whether a parameter is read from or written to.