Wildcard'lar

Sınırsız wildcard (`<?>`), üst sınırlı wildcard (`<? extends T>`), alt sınırlı wildcard (`<? super T>`), üçü arasındaki fark, ve PECS (Producer Extends, Consumer Super) kuralı -- Java generics'in en yaygın yanlış anlaşılan bölümü, gerçek örneklerle adım adım. Generics serisinin 4.'sü.

Orta 30 dk

Son güncelleme: 7 Kasım 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/wildcards

---

"Sınırlı Tür Parametreleri", `T` gibi bir tür parametresinin ne ile doldurulabileceğini kısıtladı. Wildcard'lar ilgili ama farklı bir sorunu çözer: bir KULLANIM NOKTASINDA — bir metot parametresi, bir alan, bir değişken — generic bir türü, hangi tür argümanıyla inşa edildiğine tam olarak bağlanmadan kabul etmek istediğinde ne yapılacağı. Bu, Java generics'inin en yaygın yanlış anlaşılan bölümlerinden biri, bu yüzden bu ders yavaş ilerliyor.

Wildcard Nedir?

`?` olarak yazılan bir wildcard, generic bir türün belirli bir kullanımında BİLİNMEYEN bir tür argümanının yerini tutar — `List<?>`, "bir tür `List`, hangisi olduğunu söylemiyorum" demektir. Bir tür parametresinden (`T`) farklı olarak, bir wildcard hiçbir zaman bir isim almaz ve yeni generic sınıflar ya da metotlar bildirmek için asla kullanılmaz — yalnızca generic bir tür KULLANILIRKEN, bir parametre türü gibi bir yerde görünür.

Neden Var?

Java generics DEĞİŞMEZDİR (invariant): `Integer` bir `Number` OLSA bile, `List<Integer>`, bir `List<Number>` DEĞİLDİR — ikisi tamamen ilgisiz iki tür olarak ele alınır.

```
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.
    }
}
```

Beklenen Çıktı

```
6.5
```

`sumNumbers(List<Number> numbers)`, yalnızca TAM OLARAK `List<Number>` olan bir parametreyi kabul eder — eleman türü ne kadar yakından ilişkili olursa olsun, bir `List<Integer>` doğrudan reddedilir. Başka bir araç olmadan, bir sayı listesini toplamak için bile her olası eleman türü için ayrı bir overload gerekirdi. Wildcard'lar, bir metodun tek, esnek bir parametre türü aracılığıyla ilişkili tür argümanlarının bütün bir AİLESİNİ kabul etmesine izin vermek için var. ("Koleksiyonlarla Generics", aynı değişmezlik kuralına, özellikle `List` ve `Map` gibi koleksiyonlar açısından tekrar bakacak.)

Sınırsız Wildcard: <?>

`List<?>`, herhangi bir eleman türünden bir `List`i kabul eder — bir metodun elemanların ne olduğunu gerçekten umursamadığı ve yalnızca ne olursa olsun çalışan işlemlere ihtiyaç duyduğu durumlarda doğru araçtır.

```
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));
    }
}
```

Beklenen Çıktı

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

`printSize(...)`, bir `List<String>`, bir `List<Integer>` ya da başka herhangi bir şey üzerinde çalışır — eleman türünü asla bilmesi gerekmez, çünkü yalnızca `size()`'ı çağırır ve elemanları `Object` olarak okur.

Üst Sınırlı Wildcard: `<? extends T>`

`List<? extends Number>`, `Number`'dan YA DA onun alt türlerinden birinden bir `List`i kabul eder — `List<Integer>`, `List<Double>`, `List<Number>`'ın kendisi, hepsi uygundur.

```
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>
    }
}
```

Beklenen Çıktı

```
6.0
4.0
6.5
```

`sum(...)`, listeden yalnızca OKUR — her eleman, tam türü ne olursa olsun, en azından bir `Number` olmayı garanti eder, bu yüzden `n.doubleValue()`'yu çağırmak her zaman güvenlidir. GÜVENLİ OLMAYAN şey ona bir şey eklemektir: derleyicinin listenin gerçek eleman türünü bilmesinin bir yolu yoktur (özellikle bir `List<Double>` olabilir), bu yüzden bir `Integer` bile olsa herhangi bir şey eklemene izin vermez.

Alt Sınırlı Wildcard: `<? super T>`

`List<? super Integer>`, `Integer`'dan YA DA onun SÜPER türlerinden birinden bir `List`i kabul eder — `List<Integer>`, `List<Number>`, `List<Object>`, hepsi uygundur.

```
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);
    }
}
```

Beklenen Çıktı

```
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(...)`, listeye yalnızca YAZAR — liste gerçekte `Integer`'ın hangi süper türünü tutuyor olursa olsun, bir `Integer` eklemek her zaman güvenlidir. GÜVENLİ OLMAYAN şey belirli bir türü geri okumaktır: derleyici yalnızca listenin `Integer`'ın BİR süper türünü tuttuğunu garanti eder, bu `Object` kadar geniş olabilir, bu yüzden bir okuma yalnızca `Object` olarak ele alınabilir.

Üçünü Karşılaştırmak: get ve add Gerçekte Neye İzin Verir

Yan yana konulduğunda, üç formun ardındaki desen somutlaşır.

```
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>`, güvenli bir şekilde `get` etmene izin verir ama asla `add` etmene izin vermez (herhangi bir türe uyan `null` hariç). `<? super Integer>`, güvenli bir şekilde bir `Integer` `add` etmene izin verir ama yalnızca `Object` olarak geri `get` etmene izin verir. Düz `<?>`, ne `Object`'in ötesinde anlamlı bir `get`e ne de herhangi bir `add`e izin verir. Bu "get vs. put" davranışı, bir sonraki bölümdeki kuralın işlemesinin tüm nedenidir.

PECS: Producer Extends, Consumer Super

PECS, hangi wildcard formuna başvurulacağını seçmek için akılda kalıcı bir kuraldır: parametrelenmiş bir tür yalnızca senin için değer ÜRETİYORSA (yalnızca ondan okuyorsan), `extends` kullan; yalnızca senden değer TÜKETİYORSA (yalnızca ona yazıyorsan), `super` kullan. Bu, tam olarak önceki iki örneğin zaten gösterdiği şey — `sum(...)` yalnızca okur (`extends`), `addOneToFive(...)` yalnızca yazar (`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);
    }
}
```

Beklenen Çıktı

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

`copy(...)`'nin AYNI ANDA HER İKİ role de ihtiyacı var: `src`'den okunur (bir üretici, dolayısıyla `extends`), ve `dest`'e yazılır (bir tüketici, dolayısıyla `super`). Tek başına hiçbir wildcard formu bu işi yapamazdı — `src`, `List<? super T>` olamazdı (ondan güvenilir biçimde bir `T` geri okuyamazsın), ve `dest`, `List<? extends T>` olamazdı (ona bir `T` ekleyemezsin).

> Bir parametrenin AYNI belirli türün hem okunmasına hem yazılmasına ihtiyacı varsa, wildcard'lar yardımcı olamaz — o parametreye hiç wildcard değil, `List<T>` gibi düz, sınırsız bir tür parametresi gerekir. Wildcard'lar yalnızca bir parametrenin rolü, üretici ya da tüketici olarak, net olduğunda uygulanır.

Best Practices

- PECS'i doğrudan uygula: bir parametre yalnızca üretiyorsa (ondan okuyorsan) `extends`, yalnızca tüketiyorsa (yalnızca ona yazıyorsan) `super`.
- Bir metodun eleman türüne hiç dokunmadığı durumlarda `<?>`'ye başvur — hangi sınırı kullanacağın konusundaki belirsizlikten dolayı varsayılan olarak ona başvurma.
- Bir dönüş türüne asla bir wildcard ekleme — `List<? extends Number>` döndüren bir metot, her çağıranı bilinmeyen bir türle uğraşmaya zorlar, PECS'in hiçbir faydası olmadan, çünkü bir dönüş türünde ne "okuma" ne "yazma" gerçekleşir.

- Bir parametrenin aynı türle hem okunmaya hem yazılmaya ihtiyacı olduğunda, ona zorla bir wildcard koyma — bunun yerine sıradan bir tür parametresi kullan.

Yaygın Hatalar

- Bir `List<? extends T>`'ye `add(...)` yapmaya çalışıp derleyicinin bunu reddetmesine şaşırmak — bu, en yaygın wildcard hatası, ve PECS'in `extends` kuralının tam olarak tasarlandığı gibi çalışması.
- Bir `List<? super T>`'den belirli bir türü (`Object` değil) geri okumaya çalışmak — derleyici yalnızca `T`'nin bir süper türünü garanti eder, asla daha dar bir şeyi.
- Metot aslında yalnızca okurken (`? extends` olmalıyken) ya da yalnızca belirli bir türü yazarken (`? super` olmalıyken) `<?>`'ye başvurmak — bu, derleyicinin hataları yakalamak için kullanabileceği bilgiyi çöpe atar.
- Bir wildcard'ı (`?`, yalnızca generic bir tür KULLANILIRKEN görünür) bir tür parametresiyle (`T`, bir sınıf ya da metotta bildirilir) karıştırmak — bir wildcard asla bildirilmez ve asla bir isim almaz.

Özet, Cheat Sheet ve Terimler Sözlüğü

Özet

- Bir wildcard (`?`), isimlendirilmiş bir tür parametresinden farklı olarak, generic bir türün bir kullanımında bilinmeyen bir tür argümanının yerini tutar.
- Generics değişmezdir, bu yüzden `List<Integer>` bir `List<Number>` değildir — wildcard'lar, bir parametrenin ilişkili tür argümanlarının bütün bir ailesini kabul etmesine izin vermek için var.
- `<?>` (sınırsız), herhangi bir eleman türünü kabul eder ama anlamlı bir okuma ya da yazmaya izin vermez.
- `<? extends T>` (üst sınırlı), `T` olarak güvenli okumaya izin verir ama yazmaya izin vermez (`null` hariç).
- `<? super T>` (alt sınırlı), güvenli `T` yazmaya izin verir ama yalnızca `Object` olarak okumaya izin verir.
- PECS: bir üretici için (okuyorsan) `extends`, bir tüketici için (yazıyorsan) `super` kullan.

Cheat Sheet

```
// Sınırsız: eleman türünü umursama
void printSize(List<?> list) { ... }

// Üst sınırlı: üretici, yalnızca okur -- PECS: extends
double sum(List<? extends Number> numbers) {
    double total = 0;
    for (Number n : numbers) total += n.doubleValue();
    return total;
}

// Alt sınırlı: tüketici, yalnızca yazar -- PECS: super
void addOneToFive(List<? super Integer> list) {
    for (int i = 1; i <= 5; i++) list.add(i);
}

// Aynı anda iki rol -- PECS'in tam hâli
static <T> void copy(List<? extends T> src, List<? super T> dest) {
    for (T item : src) dest.add(item);
}
```

Terimler Sözlüğü

- **Wildcard**: `?`, generic bir türün bir kullanımında bilinmeyen bir tür argümanının yerini tutar.
- **Sınırsız wildcard**: `<?>`, herhangi bir tür argümanını kabul eder.
- **Üst sınırlı wildcard**: `<? extends T>`, `T`'yi ya da onun alt türlerinden birini kabul eder; okumak güvenli, yazmak güvenli değildir.
- **Alt sınırlı wildcard**: `<? super T>`, `T`'yi ya da onun süper türlerinden birini kabul eder; yazmak güvenli, `Object` dışında bir şey olarak okumak güvenli değildir.
- **PECS**: "Producer Extends, Consumer Super" -- bir parametrenin okunup okunmadığına ya da yazılıp yazılmadığına göre `extends` ile `super` arasında seçim yapma kuralı.