Built-in Functional Interfaces

java.util.function paketindeki hazır interface'ler: Predicate<T>, Function<T,R>, Consumer<T>, Supplier<T>, UnaryOperator<T>, BinaryOperator<T>. Dört method reference biçimi: Class::staticMethod, object::instanceMethod, Class::instanceMethod, Class::new.

Orta 22 dk

Oluşturulma tarihi: 2 Ekim 2026
Çevrimiçi oku: https://www.learnforgex.com/tr/topics/built-in-functional-interfaces

---

Built-in Functional Interfaces & Method References

`java.util.function` paketi, her ihtiyaç duyulan şekilde bir custom functional interface tanımlamak zorunda kalmayasınız diye hazır gelir. `Predicate<T>`, `Function<T,R>`, `Consumer<T>`, `Supplier<T>` gibi tipler, gündelik lambda kullanımının neredeyse tamamını kapsar; method reference'lar da bu tipleri, zaten var olan bir metoda işaret ederek, lambda'dan bile daha kısa yazmanın yolu.

Built-in Functional Interfaces Nedir?

"Interface" dersinin "Functional Interface ve Lambda" bölümünde ve "Lambda Expressions" dersinde kendi functional interface'inizi nasıl tanımlayacağınızı görmüştünüz: tek bir soyut metodu olan, `@FunctionalInterface` ile işaretlenebilen bir interface. Ancak pratikte "bir değeri kontrol et", "bir değeri dönüştür", "bir değerle bir şey yap" gibi ihtiyaçlar o kadar sık tekrarlanır ki, JDK bunlar için hazır, genel amaçlı interface'ler sağlar. `java.util.function` paketindeki bu interface'lere **built-in functional interface** denir.

Bu konu, kendi interface'inizi yazmayı değil, JDK'nın size zaten verdiği interface'leri doğru yerde kullanmayı öğretir.

Neden Var?

Her yeni ihtiyaç için özel bir interface tanımlamak, kod tabanını gereksiz yere büyütür. Bir metot "bir `String` alıp `boolean` döndüren" bir davranış bekliyorsa, bunun için `StringChecker` diye bir interface yazmaya gerek yoktur; `Predicate<String>` zaten bunu ifade eder. Built-in functional interface'ler:

- **Ortak bir sözlük** sağlar: `Predicate` görünce herkes "koşul kontrolü" anlar, `Function` görünce "dönüşüm" anlar.
- **API'ler arası uyumluluk** sağlar: `Stream.filter()` de, sizin yazacağınız bir metot da aynı `Predicate<T>` tipini kabul edebilir.
- **Kombinasyon metotları** (`and()`, `or()`, `andThen()`, `compose()` gibi default metotlar) sunar; bunları kendi interface'inizde tekrar tekrar yazmanız gerekmez.

Tarihçe

`java.util.function` paketi, Java 8 ile (2014) lambda expression'lar ve Stream API ile birlikte geldi. Amaç, Stream API'nin (ve genel olarak fonksiyonel tarzın) ihtiyaç duyduğu ortak tipleri tek bir yerde toplamaktı: `Stream.filter()` bir `Predicate` bekler, `Stream.map()` bir `Function` bekler — bu tipler olmasaydı her metot kendi özel interface'ini tanımlardı ve Stream API ile üçüncü parti kod arasında uyumsuzluk olurdu.

Predicate<T>: Bir Koşulu Temsil Etmek

`Predicate<T>`, tek soyut metodu `boolean test(T t)` olan bir interface'tir: bir değeri alır, o değer hakkında evet/hayır cevabı verir. `String::isBlank` gibi bir method reference veya `s -> s.length() > 5` gibi bir lambda, `Predicate<String>` tipine atanabilir.

`Predicate`, `negate()`, `and()`, `or()` gibi **default metotlar** sağlar; bunlar mevcut predicate'leri birleştirerek yeni predicate'ler üretir, sıfırdan lambda yazmanıza gerek kalmaz.

```
import java.util.function.Predicate;

// Predicate<T>: a single method, test(T) -> boolean -- represents a yes/no condition
// about a value, without the caller needing to describe HOW to check it.
class PredicateExample {
    public static void main(String[] args) {
        Predicate<String> isBlank = String::isBlank;
        Predicate<String> isLong = s -> s.length() > 5;

        System.out.println(isBlank.test(""));
        System.out.println(isBlank.test("hello"));

        // negate(), and(), or() are default methods -- they combine predicates
        // without writing a new lambda from scratch.
        Predicate<String> isNotBlank = isBlank.negate();
        Predicate<String> isBlankOrLong = isBlank.or(isLong);
        Predicate<String> isNotBlankAndLong = isBlank.negate().and(isLong);

        System.out.println(isNotBlank.test("hi"));
        System.out.println(isBlankOrLong.test("hello world"));
        System.out.println(isNotBlankAndLong.test("hi"));
    }
}
```

Function<T,R>: Bir Dönüşümü Temsil Etmek

`Function<T, R>`, tek soyut metodu `R apply(T t)` olan bir interface'tir: bir `T` tipinde değer alır, bir `R` tipinde değer döndürür — girdi ve çıktı tipleri farklı olabilir. `String::length` (bir `String` alıp bir `Integer` döndürür) tipik bir örnektir.

`Function`, iki tanesini zincirlemek için `andThen()` ve `compose()` default metotlarını sağlar. `f.andThen(g)`, önce `f`'yi, sonucunu `g`'ye vererek çalıştırır. `f.compose(g)`, önce `g`'yi, sonucunu `f`'ye vererek çalıştırır — yani `compose()`, `andThen()`'in ayna görüntüsüdür.

```
import java.util.function.Function;

// Function<T, R>: a single method, apply(T) -> R -- represents a transformation from
// one type to another.
class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> length = String::length;
        Function<Integer, Integer> square = n -> n * n;

        System.out.println(length.apply("hello"));

        // andThen(): run THIS function first, feed its result into the next one.
        Function<String, Integer> lengthThenSquare = length.andThen(square);
        System.out.println(lengthThenSquare.apply("hello"));

        // compose(): run the ARGUMENT first, feed its result into THIS one -- the
        // mirror image of andThen().
        Function<Integer, String> intToLabel = n -> "n=" + n;
        Function<Integer, Integer> labelThenLength = length.compose(intToLabel);
        System.out.println(labelThenLength.apply(5));
    }
}
```

Consumer<T> ve Supplier<T>: Yan Etki ve Üretim

`Consumer<T>`, tek soyut metodu `void accept(T t)` olan bir interface'tir: bir değer alır, onunla bir **yan etki** (ekrana yazdırma, bir listeye ekleme, bir dosyaya yazma) gerçekleştirir, geriye hiçbir şey döndürmez. `System.out::println` en yaygın örnektir.

`Supplier<T>`, tek soyut metodu `T get()` olan bir interface'tir: hiçbir girdi almadan bir değer **üretir**. `Supplier`'ın önemli özelliği, `get()` çağrılana kadar hiçbir şey yapılmamasıdır — pahalı olabilecek, hatta hiç ihtiyaç duyulmayabilecek bir hesaplamayı ertelemek için idealdir (`orElseGet()` gibi metotların `Supplier` beklemesinin sebebi de budur).

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;
import java.util.function.Supplier;

// Consumer<T>: accept(T) -> void -- represents a side effect performed WITH a value,
// no result comes back. Supplier<T>: get() -> T -- represents producing a value with
// NO input at all, evaluated only when get() is actually called.
class ConsumerSupplierExample {
    public static void main(String[] args) {
        Consumer<String> print = System.out::println;
        Consumer<String> printUpper = s -> System.out.println(s.toUpperCase());

        // andThen() chains two consumers -- both run, in order, on the SAME input.
        Consumer<String> printBoth = print.andThen(printUpper);
        printBoth.accept("hi");

        // Supplier defers work until get() is actually called -- useful for values
        // that are expensive to build and might not even be needed.
        Supplier<List<String>> newList = ArrayList::new;
        List<String> list = newList.get();
        list.add("a");
        System.out.println(list);
    }
}
```

UnaryOperator<T> ve BinaryOperator<T>: Function /BiFunction'ın Özel Halleri

`UnaryOperator<T>`, `Function<T, T>`'yi genişletir: girdi ve çıktı tipi **aynıdır**. `BinaryOperator<T>`, `BiFunction<T, T, T>`'yi genişletir: iki girdi de, çıktı da aynı tiptedir.

Bu iki tip, teknik olarak `Function<T, T>` veya `BiFunction<T, T, T>` yazsanız da derlenir; var olma sebepleri saf okunabilirlik ve niyet ifadesidir — "bu fonksiyon bir değeri kendi tipinde bir başka değere dönüştürüyor" demenin daha net yoludur. `BinaryOperator`, ayrıca bir `Comparator`'dan `BinaryOperator` üreten `maxBy()` / `minBy()` static factory metotlarını sağlar.

```
import java.util.function.BinaryOperator;
import java.util.function.Function;
import java.util.function.UnaryOperator;

// UnaryOperator<T> extends Function<T, T> -- input and output are the SAME type.
// BinaryOperator<T> extends BiFunction<T, T, T> -- two inputs of the same type, one
// output of that same type. Both exist purely to make a signature more specific and
// self-documenting; Function<T, T> would compile just as well.
class UnaryBinaryOperatorExample {
    public static void main(String[] args) {
        UnaryOperator<String> shout = s -> s.toUpperCase() + "!";
        System.out.println(shout.apply("hi"));

        // A UnaryOperator IS-A Function -- this assignment compiles because of that.
        Function<String, String> asFunction = shout;
        System.out.println(asFunction.apply("hey"));

        BinaryOperator<Integer> max = (a, b) -> a > b ? a : b;
        System.out.println(max.apply(3, 9));

        // BinaryOperator.maxBy()/minBy() are static factory methods that build a
        // BinaryOperator FROM a Comparator -- a small but common convenience.
        BinaryOperator<String> longer = BinaryOperator.maxBy((a, b) -> a.length() - b.
length());
        System.out.println(longer.apply("hi", "hello"));
    }
}
```

Method References: Lambda'nın Kısayolu

Bir lambda'nın gövdesi tek bir metodu çağırmaktan ibaretse (`s -> s.length()` gibi), bu metodu doğrudan **isimle işaret ederek** aynı şeyi daha kısa yazabilirsiniz: `String::length`. Buna **method reference** denir.

Method reference, lambda'nın alternatifi değil, belirli bir durumda lambda'nın yerini alabilen bir **kısayoldur**. Derleyici, method reference'ı olduğu yerdeki target type'a (hangi functional interface'e atandığına) bakarak bir lambda'ya dönüştürür — bu da "Lambda Expressions" dersinin "Lambda'nın Functional Interface ile Bağlantısı: Target Typing" bölümünde anlatılan target typing mekanizmasının aynısıdır.

Üç Biçim: Class::method, object::method, Class::instanceMethod

Var olan bir metoda işaret eden method reference'ların üç biçimi vardır:

`Class::staticMethod` — bir static metoda işaret eder; functional interface'in parametre listesi, doğrudan static metodun kendi parametre listesine karşılık gelir. Örnek: `Integer::parseInt`.

`object::instanceMethod` ("bound") — belirli, zaten var olan bir nesnenin instance metoduna işaret eder; o nesne, tıpkı bir lambda'nın çevresindeki bir değişkeni yakalaması gibi, method reference tarafından yakalanır (capture edilir). Örnek: `greeting::concat` (burada `greeting` zaten var olan bir `String` nesnesidir).

`Class::instanceMethod` ("unbound") — belirli bir alıcı (receiver) belirtmeden bir instance metoduna işaret eder; functional interface'in **ilk parametresi**, metodun üzerinde çağrılacağı alıcı olur, kalan parametreler metodun kendi argümanları olur. Örnek: `String::startsWith` bir `BiFunction<String, String, Boolean>`'a atandığında, ilk `String` argümanı `startsWith()`'in çağrıldığı nesne, ikinci argüman ise `startsWith()`'e verilen parametre olur.

```
import java.util.function.BiFunction;
import java.util.function.Function;

// The three method reference forms that refer to an EXISTING method (the fourth,
// Class::new, refers to a constructor instead -- see ConstructorReferenceExample).
class MethodReferenceExample {
    public static void main(String[] args) {
        // Class::staticMethod -- refers to a static method; the functional
        // interface's parameter list maps directly onto the static method's own.
        Function<String, Integer> parse = Integer::parseInt;
        System.out.println(parse.apply("42"));

        // object::instanceMethod ("bound") -- refers to an instance method on a
        // SPECIFIC, already-existing object; that object is captured, just like a
        // lambda capturing a variable from its enclosing scope.
        String greeting = "Hello";
        Function<String, String> concatWithGreeting = greeting::concat;
        System.out.println(concatWithGreeting.apply(", world"));

        // Class::instanceMethod ("unbound") -- refers to an instance method with NO
        // specific receiver; the functional interface's FIRST parameter becomes the
        // receiver the method is called ON, the rest become the method's own arguments.
        BiFunction<String, String, Boolean> startsWith = String::startsWith;
        System.out.println(startsWith.apply("hello world", "hello"));
    }
}
```

Class::new: Constructor Reference

Dördüncü ve son method reference biçimi, `Class::new`'dir: bir metoda değil, bir **constructor'a** işaret eder. Hangi overload'un (parametresiz, tek parametreli, iki parametreli...) seçileceği, tıpkı diğer method reference biçimlerinde olduğu gibi, target typing ile belirlenir.

Bu biçim kendi tanımladığınız tipler için de çalışır — bir record'un canonical constructor'ı da, herhangi bir başka constructor gibi, `Class::new` ile işaret edilebilir.

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.BiFunction;
import java.util.function.Function;
import java.util.function.Supplier;

record Point(int x, int y) {
}

// Class::new -- refers to a constructor. Which overload gets picked (no-arg, one-arg,
// two-arg...) is decided by target typing, exactly like any other method reference.
class ConstructorReferenceExample {
    public static void main(String[] args) {
        Supplier<List<String>> newList = ArrayList::new; // matches the no-arg constructor
        System.out.println(newList.get());

        Function<String, StringBuilder> newBuilder = StringBuilder::new; // one-arg
constructor
        System.out.println(newBuilder.apply("hi"));

        // Works for user-defined types too -- a record's canonical constructor is
        // just a constructor, like any other.
        BiFunction<Integer, Integer, Point> newPoint = Point::new;
        System.out.println(newPoint.apply(3, 4));
    }
}
```

Best Practices

- **Anlamlı ismi olan built-in tipi tercih edin.** Bir `Function<T, T>` yerine `UnaryOperator<T>` yazmak, kodu okuyan kişiye niyetinizi daha net anlatır.
- **Method reference'ı, okunabilirliği artırdığı yerde kullanın.** `s -> s.length()` yerine `String::length` yazmak kısa ve nettir; ama method reference kodu daha karmaşık hale getiriyorsa (örneğin hangi biçimin kullanıldığı belirsizleşiyorsa), sade bir lambda daha iyi bir tercih olabilir.
- **`Supplier`'ı gerçekten tembel (lazy) değerlendirme gerektiğinde kullanın.** `orElseGet(Supplier)` ile `orElse(değer)` arasındaki fark tam olarak budur: `orElse()`'e verilen değer her zaman hesaplanır, `orElseGet()`'e verilen `Supplier` sadece gerçekten ihtiyaç duyulursa çağrılır.
- **`andThen()` / `compose()` ile küçük fonksiyonları birleştirin**, tek ve büyük bir lambda yazmak yerine. Bu, her adımı ayrı ayrı test edilebilir ve isimlendirilebilir hale getirir.

Yaygın Hatalar

- **`andThen()` ile `compose()`'u karıştırmak.** `f.andThen(g)` önce `f`'yi çalıştırır; `f.compose(g)` önce `g`'yi çalıştırır. Sırayı karıştırmak, özellikle iki fonksiyon da yan etki içeriyorsa, sessiz ve fark edilmesi zor hatalara yol açar.

- `object::instanceMethod` **(bound) ile** `Class::instanceMethod` **(unbound) biçimlerini karıştırmak.** İkisi de görünüşte "bir noktadan sonra iki nokta üst üste" şeklindedir, ama biri belirli bir nesneyi yakalar, diğeri functional interface'in ilk parametresini alıcı olarak kullanır. Bu farkı gözden kaçırmak, "neden bir parametre fazladan/eksik" tarzı derleme hatalarına yol açar.
- **Her yerde method reference kullanmaya çalışmak.** Bir method reference, karşılık geldiği lambda'dan daha az okunaklıysa (örneğin hangi parametrenin nereye gittiği belirsizse), zorla method reference'a çevirmek kodu kötüleştirir.
- `UnaryOperator` / `BinaryOperator` **yerine her zaman** `Function` / `BiFunction` **yazmak.** Teknik olarak ikisi de çalışır, ama daha spesifik tip kullanmak, kodu okuyana "girdi ve çıktı aynı tipte" bilgisini bedavaya verir.

Özet, Cheat Sheet ve Terimler Sözlüğü

`java.util.function` paketi, en sık ihtiyaç duyulan fonksiyonel şekilleri hazır interface'ler olarak sunar: `Predicate<T>` bir koşulu (`test`), `Function<T,R>` bir dönüşümü (`apply`), `Consumer<T>` bir yan etkiyi (`accept`), `Supplier<T>` tembel bir üretimi (`get`) temsil eder. `UnaryOperator<T>` ve `BinaryOperator<T>` , girdi/çıktı tipi aynı olan özel `Function` / `BiFunction` halleridir. Method reference'lar (`Class::staticMethod` , `object::instanceMethod` , `Class::instanceMethod` , `Class::new`), var olan bir metoda veya constructor'a işaret ederek lambda yazmanın daha kısa bir yoludur; hangi biçimin kullanılacağını ve hangi overload'un seçileceğini target typing belirler.

Hızlı referans:

```
Predicate<T>      boolean test(T t)          // bir koşulu kontrol et
Function<T,R>     R apply(T t)               // bir değeri dönüştür
Consumer<T>       void accept(T t)           // bir değerle yan etki yap
Supplier<T>       T get()                    // girdisiz bir değer üret
UnaryOperator<T>  T apply(T t)                // aynı tipte dönüşüm
BinaryOperator<T> T apply(T t1, T t2)         // aynı tipte iki değeri birleştir

Integer::parseInt        // Class::staticMethod
greeting::concat         // object::instanceMethod (bound)
String::startsWith       // Class::instanceMethod (unbound)
ArrayList::new           // Class::new (constructor reference)
```

Terimler Sözlüğü

Built-in functional interface — JDK'nın `java.util.function` paketinde hazır sunduğu, genel amaçlı functional interface.

Predicate — Bir değeri kontrol edip `boolean` döndüren fonksiyon soyutlaması.

Method reference — Var olan bir metoda veya constructor'a isimle işaret ederek lambda'nın yerini alan, `::` operatörüyle yazılan kısayol.

Bound method reference — Belirli, zaten var olan bir nesnenin instance metoduna işaret eden method reference (`object::method`).

Unbound method reference — Belirli bir nesne belirtmeyen, ilk parametreyi alıcı olarak kullanan method reference (`Class::method`).

Constructor reference — Bir constructor'a işaret eden method reference (`Class::new`).