Built-in Functional Interfaces

The ready-made interfaces in java.util.function: Predicate<T>, Function<T,R>, Consumer<T>, Supplier<T>, UnaryOperator<T>, BinaryOperator<T>. The four method reference forms: Class::staticMethod, object::instanceMethod, Class::instanceMethod, Class::new.

Intermediate 22 min

Generated on 2 October 2026
Read online: https://www.learnforgex.com/en/topics/built-in-functional-interfaces

---

Built-in Functional Interfaces & Method References

The `java.util.function` package exists so you don't have to define a custom functional interface for every shape a lambda might need. Types like `Predicate<T>`, `Function<T,R>`, `Consumer<T>`, and `Supplier<T>` cover nearly all everyday lambda use; method references are a way to write those same types even shorter than a lambda, by pointing directly at a method that already exists.

What Are Built-in Functional Interfaces?

In the "Interface" lesson's "Functional Interfaces and Lambdas" section and in the "Lambda Expressions" lesson, you saw how to define your own functional interface: an interface with exactly one abstract method, optionally marked with `@FunctionalInterface`. In practice, though, needs like "check a condition on a value," "transform a value," and "do something with a value" repeat so often that the JDK ships ready-made, general-purpose interfaces for them. These interfaces in the `java.util.function` package are called **built-in functional interfaces**.

This lesson isn't about writing your own interface -- it's about using the ones the JDK already gives you, in the right place.

Why Does It Exist?

Defining a brand-new interface for every need bloats a codebase for no reason. If a method just needs "a behavior that takes a `String` and returns a `boolean`," there's no need to write a `StringChecker` interface for it -- `Predicate<String>` already expresses exactly that. Built-in functional interfaces provide:

- A **shared vocabulary**: seeing `Predicate` tells everyone "condition check," seeing `Function` tells everyone "transformation."
- **Interoperability across APIs**: `Stream.filter()` and a method you write yourself can both accept the same `Predicate<T>` type.
- **Combinator methods** (default methods like `and()`, `or()`, `andThen()`, `compose()`) that you don't have to reinvent in your own interface every time.

History

The `java.util.function` package arrived with Java 8 (2014), alongside lambda expressions and the Stream API. The goal was to collect the common types the Stream API (and the functional style more broadly) needed into one place: `Stream.filter()` expects a `Predicate`, `Stream.map()` expects a `Function` -- without these shared types, every method would define its own interface, and the Stream API would be incompatible with third-party code.

Predicate<T>: Representing a Condition

`Predicate<T>` is an interface whose single abstract method is `boolean test(T t)`: it takes a value and answers a yes/no question about it. A method reference like `String::isBlank`, or a lambda like `s -> s.length() > 5`, can both be assigned to `Predicate<String>`.

`Predicate` provides default methods -- `negate()`, `and()`, `or()` -- that combine existing predicates into new ones, so you don't have to write a fresh lambda from scratch every time.

```
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>: Representing a Transformation

`Function<T, R>` is an interface whose single abstract method is `R apply(T t)`: it takes a value of type `T` and returns a value of type `R` -- the input and output types can differ. `String::length` (takes a `String`, returns an `Integer`) is a typical example.

`Function` provides `andThen()` and `compose()` default methods for chaining two of them together. `f.andThen(g)` runs `f` first, then feeds its result into `g`. `f.compose(g)` runs `g` first, then feeds its result into `f` -- `compose()` is the mirror image of `andThen()`.

```
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> and Supplier<T>: Side Effects and Production

`Consumer<T>` is an interface whose single abstract method is `void accept(T t)`: it takes a value and performs a **side effect** with it (printing, adding to a list, writing to a file), returning nothing. `System.out::println` is the most common example.

`Supplier<T>` is an interface whose single abstract method is `T get()`: it **produces** a value with no input at all. The key property of `Supplier` is that nothing happens until `get()` is actually called -- ideal for deferring a computation that might be expensive, or might not even be needed (this is exactly why methods like `orElseGet()` expect a `Supplier`).

```
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> and BinaryOperator<T>: Special Cases of Function/BiFunction

`UnaryOperator<T>` extends `Function<T, T>`: input and output are the **same** type. `BinaryOperator<T>` extends `BiFunction<T, T, T>`: both inputs and the output are the same type.

Technically, `Function<T, T>` or `BiFunction<T, T, T>` would compile just as well as these two types -- they exist purely for readability and to express intent: a clearer way of saying "this function transforms a value into another value of its own type." `BinaryOperator` also provides `maxBy()` / `minBy()` static factory methods that build a `BinaryOperator` from a `Comparator`.

```
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: A Shortcut for Lambdas

When a lambda's body does nothing but call a single method (like `s -> s.length()`), you can write the same thing more concisely by pointing directly at that method by name: `String::length`. This is called a **method reference**.

A method reference isn't an alternative to a lambda -- it's a **shortcut** that can stand in for one in certain situations. The compiler converts a method reference into a lambda by looking at its target type (which functional interface it's being assigned to) at that point in the code, exactly the same target-typing mechanism described in the "Lambda Expressions" lesson's "The Lambda-to-Functional-Interface Connection: Target Typing" section.

Three Forms: Class::method, object::method, Class::instanceMethod

There are three forms of method reference that point at a method that already exists:

`Class::staticMethod` -- points at a static method; the functional interface's parameter list maps directly onto the static method's own parameter list. Example: `Integer::parseInt`.

`object::instanceMethod` ("bound") -- points at an instance method on a specific, already-existing object; that object is captured by the method reference, just like a lambda

capturing a variable from its enclosing scope. Example: `greeting::concat` (where `greeting` is an already-existing `String` object).

`Class::instanceMethod` ("unbound") -- points at an instance method with no specific receiver; the functional interface's **first parameter** becomes the receiver the method is called on, and the remaining parameters become the method's own arguments. Example: when `String::startsWith` is assigned to a `BiFunction<String, String, Boolean>`, the first `String` argument becomes the object `startsWith()` is called on, and the second argument becomes the parameter passed to `startsWith()`.

```
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

The fourth and final method reference form is `Class::new`: it points not at a method, but at a **constructor**. Which overload gets picked (no-arg, one-arg, two-arg...) is decided by target typing, exactly like the other method reference forms.

This form works for your own types too -- a record's canonical constructor can be referenced with `Class::new` just like any other constructor.

```
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

- **Prefer the built-in type with the more meaningful name.** Writing `UnaryOperator<T>` instead of `Function<T, T>` tells the reader your intent more clearly.
- **Use a method reference where it improves readability.** `String::length` is short and clear compared to `s -> s.length()`; but if a method reference makes the code harder to parse (for example, when it's unclear which form is being used), a plain lambda can be the better choice.
- **Reach for `Supplier` when you genuinely need lazy evaluation.** That's exactly the difference between `orElseGet(Supplier)` and `orElse(value)`: the value passed to `orElse()` is always computed, while the `Supplier` passed to `orElseGet()` is only invoked if it's actually needed.
- **Compose small functions with `andThen()` / `compose()`** rather than writing one large lambda. This makes each step independently testable and nameable.

Common Mistakes

- **Mixing up `andThen()` and `compose()`.** `f.andThen(g)` runs `f` first; `f.compose(g)` runs `g` first. Getting the order backwards -- especially when both functions have side effects -- leads to silent, hard-to-spot bugs.

- **Confusing the bound (`object::instanceMethod`) and unbound (`Class::instanceMethod`) forms.** Both look like "a type or object followed by `::`," but one captures a specific object while the other uses the functional interface's first parameter as the receiver. Missing this distinction leads to confusing "why is there an extra/missing parameter" compile errors.
- **Forcing a method reference everywhere.** If a method reference is less readable than the lambda it replaces (for example, when it's unclear where each parameter goes), forcing the conversion makes the code worse, not better.
- **Always writing `Function`/`BiFunction` instead of `UnaryOperator`/`BinaryOperator`.** Both compile fine, but the more specific type gives the reader the "input and output are the same type" information for free.

Summary, Cheat Sheet, and Glossary

The `java.util.function` package offers ready-made interfaces for the most common functional shapes: `Predicate<T>` represents a condition (`test`), `Function<T,R>` represents a transformation (`apply`), `Consumer<T>` represents a side effect (`accept`), and `Supplier<T>` represents lazy production (`get`). `UnaryOperator<T>` and `BinaryOperator<T>` are specialized forms of `Function`/`BiFunction` where input and output share a type. Method references (`Class::staticMethod`, `object::instanceMethod`, `Class::instanceMethod`, `Class::new`) are a shorter way to write a lambda by pointing at a method or constructor that already exists; target typing decides which form applies and which overload gets picked.

Quick reference:

```
Predicate<T>      boolean test(T t)          // check a condition
Function<T,R>     R apply(T t)               // transform a value
Consumer<T>       void accept(T t)           // side effect with a value
Supplier<T>       T get()                    // produce a value, no input
UnaryOperator<T>  T apply(T t)                // same-type transformation
BinaryOperator<T> T apply(T t1, T t2)         // combine two same-type values

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

Glossary

Built-in functional interface — A general-purpose functional interface the JDK provides ready-made in the `java.util.function` package.

Predicate — A function abstraction that checks a value and returns a `boolean`.

Method reference — A shortcut, written with the `::` operator, that stands in for a lambda by pointing by name at a method or constructor that already exists.

Bound method reference — A method reference that points at an instance method on a specific, already-existing object (`object::method`).

Unbound method reference — A method reference with no specific object, where the first parameter is used as the receiver (`Class::method`).

Constructor reference — A method reference that points at a constructor (`Class::new`).