Introduction to Lambda Expressions

Welcome to the Lambda Expressions section of the course.

While I considered introducing this topic earlier, it's important to have a solid understanding of core Java concepts before tackling lambda expressions. These expressions, introduced in Java with JDK 8, are now an essential feature and are supported by various methods within Java's interfaces and classes.

Lambda expressions can be a bit challenging to grasp initially, especially when trying to unlock their full potential. That's why it made sense to first build a strong foundation before exploring these more advanced features.

Think of a lambda expression as a shorthand for an anonymous class that implements a functional interface—an interface that contains only a single abstract method. This makes your code more concise and easier to understand.

Introduction to Lambda Expressions

Lambda expressions allow you to pass blocks of code as parameters, offering a powerful and flexible way to write cleaner and more functional code with minimal effort. These compact and expressive constructs can greatly simplify your programming tasks.

Method references build on this concept, providing an even more concise way to refer to existing methods using lambda expression syntax.

By the end of this section, I hope you'll be excited about the new possibilities that lambda expressions bring and feel ready to use them effectively in your projects.

In the final part of this section, I'll introduce you to some handy default and static methods available on many interfaces. These methods can help you reduce repetitive coding tasks, so be sure to explore this part in detail.

Let's get started!

The Lambda Expression

The generated Lambda Expression	Comparator's Abstract Method
`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`	`int compare(T o1, T o2)`

The syntax of this lambda expression is on the left below.

This was passed directly as a method argument, for a parameter type that was a Comparator.

The Comparator's abstract method, compare, is shown here on the right side.

The lambda expression parameters are determined by the associated interface's method, the functional method. More on the functional method, shortly.

In the case of a Comparator, and it's compare method, there are two arguments.

This is why we get o1, and o2 in parentheses, in the generated lambda expression.

The Lambda Expression

The generated Lambda Expression	Comparator's Abstract Method
`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`	`int compare(T o1, T o2)`

These arguments can be used in the expression, which is on the right of the arrow token.

Technically arrow, rather than arrow token is the official term used by Oracle, when using it in a lambda expression, so I'll use that moving forward.

The Syntax of a Lambda Expression

A lambda expression consists of a formal parameter list, usually but not always declared in parentheses; the arrow; and either an expression or a code block after the arrow.

Because lambda expressions are usually simple expressions, it's more common to see them written as shown on this slide

```
(parameter1, parameter2, ... ) -> expression;
```

The expression should return a value, if the associated interface's method returns a value.

In the case of our generated expression, it returns an int, which is the result of the compare method called

```
(o1, o2) -> o1.lastName().compareTo(o2.lastName())
```

Comparing the anonymous class and the lambda expressi

Perhaps you are asking, where's the link between the compare method, and this lambda expression?

It's obvious in the anonymous class, because we override the compare method, and return the result of that expression.

Anonymous Class	Lambda Expression

`new Comparator<Person>() {`
 	
`    @Override`
`    public int compare(Person o1, Person o2) {`
`        return o1.lastName().compareTo(o2.lastName());`
`    }`
`};`
	`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`

Comparing the anonymous class and the lambda expressi

We can see the two parameters and their types, and what the return value should be, in the anonymous class.

But the lambda expression has no reference to an enclosing method, as far as we can see from this code.

Anonymous Class	Lambda Expression

`new Comparator<Person>() {`
 	
`    @Override`
`    public int compare(Person o1, Person o2) {`
`        return o1.lastName().compareTo(o2.lastName());`
`    }`
`};`
	`(o1, o2) -> o1.lastName().compareTo(o2.lastName())`

Where's the method in the lambda expression?

For a lambda expression, **the method is inferred** by Java!

How can Java infer the method?

Java takes its clue from the reference type, in the context of the lambda expression usage.

Here, I show a simplified view, of the sort method on List.

```
void sort(Comparator c)
```

And here is the call to that method passing the lambda expression

```
people.sort((o1, o2) -> o1.lastName().compareTo(o2.lastName()));
```

From this, Java can infer that this lambda expression, resolves to a Comparator type, because of the method declaration.

This means the lambda expression passed, should represent code for a specific method on the Comparator interface.

How can Java infer the method?

But which method?

Well, there's only one the lambda expression cares about, and that's **the abstract method** on Comparator.

Java requires types which support lambda expressions, to be something called a functional interface.

What's a functional interface?

A functional interface is an interface that has **one, and only one, abstract method.**

This is how Java can infer the method to derive the parameters and return type, for the lambda expression.

You may also see this referred to as SAM, which is short for **Single Abstract Method**, which is called the functional method.

A functional interface is the **target type for a lambda expression.**

The Lambda Expression

The functional interface is the framework that lets a lambda expression be used.

Lambda expressions are also called lambdas for short.

Many of Java's classes use functional interfaces in their method signatures, which allows you to pass lambdas as arguments to them.

You'll soon discover that lambdas will reduce the amount of code you write.

In an upcoming video, I'm going to cover many of Java's built-in functional interfaces.

The Consumer Interface

The Consumer interface is in the java.util.function package.

It has one abstract method that takes a single argument and doesn't return anything

```
void accept(T t)
```

This doesn't seem like a very interesting interface at first, but let's see what this means in practice, as far as the lambda expressions we can use with it.

lambda expression variations, for a single paramete

This slide shows the different ways to declare a single parameter in a lambda expression.

Lambda Expression	Description
`element -> System.out.println(element);`	A single parameter without a type can omit the parentheses.
`(element) -> System.out.println(element);`	Parentheses are optional.
`(String element) -> System.out.println(element);`	Parentheses required if a reference type is specified.
`(var element) -> System.out.println(element);`	A reference type can be var.

lambda expression variations, the lambda body

The lambda body can be a single expression or can contain a lambda body in opening and closing curly braces.

Lambda Expression	Description
`(element) -> System.out.println(element);`	An expression can be a single statement. Like a switch expression, that does not require yield for a single statement result, the use of return is not needed and would result in a compiler error.

`(var element) -> {`
`  char first = element.charAt(0);`
`  System.out.println(element + " means " + first);`
`};`
	An expression can be a code block.
	
	Like a switch expression, that requires yield, a lambda that returns a value, would require a final return statement.
	
	All statements in the block must end with semi-colons.

Functional Programming

Lambdas are Java's first step into functional programming.

This is a programming paradigm that focuses on computing and returning results.

For more information about functional programming, there's a good wikipedia article here.

https://en.wikipedia.org/wiki/Functional_programming

Check that out to find out a bit more about functional programming.

Streams

Another feature of Java, makes extensive use of lambda expressions, and that's streams.

Streams are exciting, because they create a pipeline of work that can be formed into a chain.

Many stream operations take functional interfaces as parameters, meaning you can code them with lambda expressions.

I have a section dedicated to streams in this course.

Lambda expressions with multiple parameters

The rules for multiple parameters used in a lambda expression are shown here.

Lambda Expression	Description
`(a, b) -> a + b;`	Parentheses are always required. Explicit types are not.
`(Integer a, Integer b) -> a + b;`	If you use an explicit type for one parameter, you must use explicit types for all the parameters.
`(var a, var b) -> a + b;`	If you use var for one parameter, you must use var for all parameters.

Lambda expressions that return values

This slide shows the two rules for returning values from a lambda expression.

Lambda Expression	Description
`(a, b) -> a + b;`	when not using curly braces, the return keyword is unnecessary, and will throw a compiler error.

`(a, b) -> {`
`  var c = a + b;`
`  return c;`
`  }`
	If you use a statement block, meaning you use the curly braces, a return is required.

java.util.function

Java provides a library of functional interfaces in the java.util.function package.

We've looked at one already, the Consumer interface.

I'll look at another of these interfaces now, the BinaryOperator, in code.

The Four categories of Functional Interfaces

It's a good idea to know the four basic types of functional interfaces in the java.util.function package.

There are over forty interfaces in this package.

This slide shows the four categories, with the simplest method shown.

These can all be categorized as one of the following types.

Interface Category	Basic Method Signature	Purpose
Consumer	`void accept(T t)`	execute code without returning data
Function	`R apply(T t)`	return a result of an operation or function
Predicate	`boolean test(T t)`	test if a condition is true or false
Supplier	`T get()`	return an instance of something

The Consumer interface

On this slide, I'm showing the two most common Consumer interfaces, and the functional method on each.

The Consumer interface takes one argument of any type.

The BiConsumer interface takes two arguments, of two different types.

Interface Name	Method Signature
Consumer	`void accept(T t)`
BiConsumer	`void accept(T t, U u)`

A Consumer Lambda Expression Example

This slide shows an example consumer lambda expression. It takes one argument and executes a single statement.

No result is returned.

Example Lambda Expression for Consumer	Consumer Method
`s -> System.out.println(s)`	`void accept(T t)`

The Predicate Interface

The predicate interfaces take one or two arguments, and always returns a boolean value.

They are used to test a condition, and if the condition is true, to perform an

Interface Name	Method Signature
Predicate	`boolean test(T t)`
BiPredicate	`boolean test(T t, U u)`

A Predicate Lambda Expression Example

In this example, the expression takes a String, and tests if it's equal to the literal text "Hello", ignoring case. It returns either true or false.

Example Lambda Expression for Consumer

```
s -> s.equalsIgnoreCase("Hello")
```

The Function interface

On this slide, I'm showing four of the most common interfaces in this category.

Each has a return type, shown here as either T, or R, which stands for result, meaning a result is expected for any of these.

In addition to Function and BiFunction, there is also UnaryOperator and BinaryOperator.

You can think of the UnaryOperator as a Function Interface, but where the argument type is the same as the result type.

Interface Name	Method Signature	Interface Name	Method Signature
`Function<T,R>`	`R apply(T t)`	`UnaryOperator<T>`	`T apply(T t)`
`BiFunction<T,U,R>`	`R apply(T t, U u)`	`BinaryOperator<T>`	`T apply(T t1, T t2)`

The Function interface

The BinaryOperator is a BiFunction interface, where both arguments have the same type, as does the result, which is why the result is shown as T, and not R.

This reminds us that the result will be the same type as the arguments to the methods.

I've also included the type parameters with each interface name on this slide, because I wanted you to see that the result for a Function or BiFunction, is declared as the last type argument.

For UnaryOperator and BinaryOperator, there is only one type argument declared, because the types of the arguments and results will be the same

Interface Name	Method Signature	Interface Name	Method Signature
Function<T,R>	R apply(T t)	UnaryOperator<T>	T apply(T t)
BiFunction<T,U,R>	R apply(T t, U u)	BinaryOperator<T>	T apply(T t1, T t2)

A Function Interface Lambda Expression Example

On this slide, I'm showing an example of a lambda expression which targets a Function interface.

This lambda expression takes a String, s, and splits that String on commas, returning an array of String.

In this case, the argument type, T, is a String, and the result, R, is an array of String.

To demonstrate how to declare a variable of this type, I'm showing a variable declaration as well.

Example Lambda Expression for Function	Function Method	Variable Declaration for this example
`s -> s.split(",");`	`R apply(T t)`	`Function<String, String[]> f1;`

The Supplier Interface

The supplier interface takes no arguments but returns an instance of some type, T.

Interface Name	Method Signature
Supplier	`T get()`

You can think of this as kind of like a factory method.

It will produce an instance of some object.

However, this doesn't have to be a new or a distinct result returned.

A Supplier Lambda Expression Example

In the example I'm showing you on this slide, I'm using the Random class to generate a random Integer.

This method takes no argument, but lambda expressions can use final or effectively final variables in their expressions, which I'm demonstrating here.

The variable random is an example of a variable from the enclosing code.

Example Lambda Expression for Consumer
`() -> random.nextInt(1, 100)`

Valid Lambda Declarations for different number of argument

This slide shows the many varieties of declaring a parameter type in a lambda expression.

Parentheses are required in all but the one case, where the functional method has a single argument, and you don't specify a type, or use var.

Arguments in Functional Method	Valid lambda syntax
None	`() -> statement`

One	`s -> statement`
	`(s) -> statement`
	`(var s) -> statement`
	`(String s) -> statement`

Two
• When using var, all arguments must use var.
• When specifying explicit types, all arguments must specify explicit types.
	`(s, t) -> statement`
	
	`(var s, var t) -> statement`
	
	`(String s, List t) -> statement`

Valid Lambda Declarations for different number of argument

When using var as the type, every argument must use var.

When specifying explicit types, every argument must include a specific type.

Arguments in Functional Method	Valid lambda syntax
None	`() -> statement`

One	`s -> statement`
	`(s) -> statement`
	`(var s) -> statement`
	`(String s) -> statement`

Two
• When using var, all arguments must use var.
• When specifying explicit types, all arguments must specify explicit types.
	`(s, t) -> statement`
	
	`(var s, var t) -> statement`
	
	`(String s, List t) -> statement`

Lambda Mini Challenges

This challenge video is going to be a little different, and consist of several small tasks, to help you really practice creating and using lambda expressions.

I'll assign a task for you to try on your own, and then we will solve it together.

Then I'll present the next one, and so on.

Mini Challenge 1

Write the following anonymous class that you can see on screen, as a lambda expression.

Try to do it manually on your own, rather than relying on IntelliJ's tools to do it for you.

```
Consumer<String> printTheParts = new Consumer<String>() {

    @Override
    public void accept(String sentence) {
        String[] parts = sentence.split(" ");
        for (String part : parts) {
            System.out.println(part);
        }
    }
};
```

Mini Challenge 2

Write the following method as a lambda expression.

```
public static String everySecondChar(String source) {

    StringBuilder returnVal = new StringBuilder();
    for (int i = 0; i < source.length(); i++) {
        if (i % 2 == 1) {
            returnVal.append(source.charAt(i));
        }
    }
    return returnVal.toString();
}
```

In other words, create a variable, using a type that makes sense for this method.

Don't worry about executing it though.

Mini Challenge 3

The lambda expression we created in Challenge 2 doesn't do anything right now.

I want you to write the code to execute this lambda expression's functional method, using 1234567890, as the argument to that method, and print the result out

```
UnaryOperator<String> everySecondChar = source -> {
    StringBuilder returnVal = new StringBuilder();
    for (int i = 0; i < source.length(); i++) {
        if (i % 2 == 1) {
            returnVal.append(source.charAt(i));
        }
    }
    return returnVal.toString();
};
```

Mini Challenge 4

Instead of executing this function directly, suppose we want to pass it to a method.

Write another method on your class, called everySecondCharacter.

This method should accept a Function, or UnaryOperator, as a parameter, as well as a second parameter that lets us pass, "1234567890".

In other words, don't hard code that String in your method code.

The method code should execute the functional method on the first argument, passing it the value of the String passed, from the enclosing method.

It should return the result of the call to the functional method.

Lambda Expressions, What you need to understand

1) Declare lambda variables, or pass lambdas directly to methods that are targets.

I show two examples here.

Local Variable Declaration	Method argument
`Function<String, String> myFunction = s -> s.concat(" " + suffix);`	`list.forEach(s -> System.out.println(s));`

2) Create methods that can be targets for lambda expressions.

Mini Challenge 5

Call the method you created from Challenge 4, passing the lambda variable we created earlier, and the string 1234567890, then print the result returned from the method.

Mini Challenge 6

Write a lambda expression that is declared with the Supplier interface.

This lambda should return the String, "I love Java", and assign it to a variable called, iLoveJava.

Mini Challenge 7

As with the Function example, the Supplier lambda won't do anything until we use it.

Remember, lambdas represent deferred execution of snippets of code.

Use this Supplier to assign a String, "I love Java", to a variable called supplierResult.

Print that variable to the console.

Lambda Expression Challenge

This challenge is designed to exercise your skills with methods on Arrays and ArrayLists, that are targets for lambda expressions.

First, I want you to create an array of String, which is populated with first names, in mixed case.

Include at least one name, which is spelled the same backwards, and forwards, like Bob, or Anna.

Use **Arrays.setAll**, or **List.replaceAll**, to change the values in this array.

If you use List methods, you'll need **a list backed by the array**, so that changes get made to the initial array.

Lambda Expression Challenge

Using one of those two methods, perform the following functions on the elements in the array, with appropriate lambda expressions.

- Transform names to all uppercase.
- Add a randomly generated middle initial and include a period.
- Add a last name that is the reverse of the first name.

Print your array or the array elements, after each change, **using the forEach method, at least once**.

Finally, create a new modifiable ArrayList from your names array, removing any names where the last name equals the first name.

Use removeIf with a lambda expression to perform this last operation.

What's a Method Reference?

Java gives us an alternative syntax to use for this second kind of lambda that uses named methods.

These are called method references.

These provide a more compact, easier-to-read lambda expression, for methods that are already defined on a class.

For the last couple of videos, I've been ignoring IntelliJ's warnings and hints, whenever I've used System.out.println in a lambda expression.

IntelliJ is issuing the warnings because they can be replaced with a method reference.

Why are these statements equal?

At first glance, it's not really obvious why a method reference has this syntax.

Lambda Expression	Method Reference
`s -> System.out.println(s)`	`System.out::println`

A method reference abstracts the lambda expression even further, eliminating the need to declare formal parameters.

We also don't have to pass arguments to the method in question, in this case println.

A method reference has double colons, between the qualifying type, and the method name.

In this example of a Consumer interface, not only is the method inferred, but the parameters are as well.

What methods can be used in method references?

Methods which can be used as method references are based on the context of the lambda expression.

This means the method reference, is again dependent on the targeted interface's method.

You can reference a static method on a class.

You can reference an instance method from either an instance external to the expression, or an instance passed as one of the arguments.

Or you can reference a constructor, by using new as the method.

Method references can be used to increase the readability of your code.

Deferred Method Invocation

When you create variables that are lambda expressions or method references, it's important to remember that the code isn't invoked at that point.

The statement or code block gets invoked at the point in the code that the targeted functional method is called.

Some Terminology for the next couple of Slides

A Type Reference refers to a class name, an interface name, an enum name, or a record name.

Static methods are usually called using Type References but can also be called by instances in our code.

This is NOT true for method references.

Static methods, in method references and lambda expressions, must be invoked using a reference type only.

Some Terminology for the next couple of Slides

There are two ways to call an instance method.

The first is when you refer to the method with an instance derived from the enclosing code.

This instance is declared outside of the method reference.

The System.out::println method reference is an example of this.

You'll find that some web sites call this instance a Bounded Receiver, and I actually like that terminology as a differentiator.

A Bounded Receiver is an instance derived from the enclosing code, used in the lambda expression on which the method will be invoked.

Some Terminology for the next couple of Slides

The second way to call an instance method is where the confusion starts.

The instance used to invoke the method will be the first argument passed to the lambda expression or method reference when executed.

This is known in some places as the Unbounded Receiver.

It gets dynamically bound to the first argument, which is passed to the lambda expression when the method is executed.

Unfortunately, this looks an awful lot like a static method reference using a reference type.

Some Terminology for the next couple of Slides

This means there are two method references that resemble each other but have **two very different meanings**.

The first actually does call a static method and uses a reference type to do it.

You saw this earlier, when I used the sum method on the Integer wrapper class.

```
Integer::sum
```

This is a Type Reference (Integer is the type), which will invoke a static method.

This is easy to understand.

Some Terminology for the next couple of Slides

But there is another, which you'll see when I start working with String method references in particular.

Here, I show a method reference for the concat method on String.

```
String::concat
```

Now, you know by now I hope that the concat method, isn't a static method on String. It's an instance method that is called on a specific String object.

So why is this method reference even valid?

You could never call concat from the String class directly, because it needs to be called on a specific instance.

Some Terminology for the next couple of Slides

I just said, not two slides ago, that instance methods can't be called using Reference Types.

But the example shown here, is a special syntax, when it's declared in the right context, meaning when it's associated to the right type of interface.

```
String::concat
```

This is valid when you use a method reference in the context of an **unbounded receiver**.

Some Terminology for the next couple of Slides

Remember, the unbounded receiver means that the first argument becomes the instance used on which the method gets invoked.

```
String::concat
```

Any method reference that uses String::concat, must be in the context of a two-parameter functional method.

The first parameter is the String instance on which the concat method gets invoked, and the second argument is the String argument passed to the concat method.

Four Types of Method References

This chart shows the four different types of method references, with method reference examples, and a corresponding lambda expression.

Type	Syntax	Method Reference Example	Corresponding Lambda Expression
static method	`ClassName::staticMethodName(p1, p2, ... pn)`	`Integer::sum`	`(p1, p2) ->  p1 + p2`
instance method of a particular (Bounded) object	`ContainingObject::instanceMethodName(p1, p2,...pn)`	`System.out::println`	`p1 -> System.out.println(p1)`
instance method of an arbitrary (Unbounded) object (as determined by p1)	`ContainingType[=p1] ::instanceMethodName(p2, ... pn)`	`String::concat`	`(p1, p2) -> p1.concat(p2).`
constructor	`ClassName::new`	`LPAStudent::new`	`() -> new LPAStudent()`

Method Reference Examples (No arguments and one argumen

This chart shows some of the valid ways to use method references when assigned to different interface types.

These interface types have no arguments in the case of a Supplier, and one argument for the other interfaces.

	No Args	One Argument		
Types of Method References	**Supplier**	**Predicate**	**Consumer**	**Function** **UnaryOperator** **et. al.**
Reference Type (Static)				
Reference Type (Constructor)	`Employee::new`		**n/a**	`Employee::new`
Bounded Receiver (Instance)			`System.out::println`	
Unbounded Receiver (Instance)	**n/a**	`String::isEmpty`	`List::clear`	`String::length`

Method Reference Examples (No arguments and one argumen

If a cell is empty, it's not because it's not valid, but because are many possibilities.

n/a means not applicable, so a Supplier or an interface method that has no arguments, can never be a target for the unbounded receiver type of method reference.

	No Args	One Argument		
Types of Method References	**Supplier**	**Predicate**	**Consumer**	**Function** **UnaryOperator** **et. al.**
Reference Type (Static)				
Reference Type (Constructor)	`Employee::new`		**n/a**	`Employee::new`
Bounded Receiver (Instance)			`System.out::println`	
Unbounded Receiver (Instance)	**n/a**	`String::isEmpty`	`List::clear`	`String::length`

Method Reference Examples (Two Arguments)

This chart shows some of the valid ways to use method references when assigned to different interface types.

These interface types have two arguments, and therefore it's more common to see the unbounded receiver method references used for these.

	Two Arguments		
Types of Method References	**BiPredicate**	**BiConsumer**	**BiFunction** **BinaryOperator** **et. al.**
Reference Type (Static)			`Integer::sum`
Reference Type (Constructor)			`Employee::new`
Bounded Receiver (Instance)		`System.out::printf`	`new Random()::nextInt`

Unbounded Receiver (Instance)	`String::equals`	`List::add`	`String::concat`
			
			`String::split`

Method and Lambda Challenge

First, create an array of names, in mixed case, as you did in the Lambda Expression Challenge.

Create a list of Function interfaces, or alternately UnaryOperator, which will contain all the operations you want executed on each name in your array.

Do something similar to what we did in the Lambda Expression challenge:

- Make each name upper case,
- Add a random middle initial,
- Add a last name, which should be the reverse of the first.

In addition to this, add some custom transformations of your own.

Method and Lambda Challenge

Use a mix of lambda expressions, and method references.

Create a method that takes the name array, and the function list, and applies each function to each name, using the transform method on String, to do this.

All changes should be applied to the original array.

Make sure you explore as many transformations as you can, trying as many different types of method references as you can think of.

Convenience Methods

In this video, you'll learn how to do something similar, using what are called convenience methods.

These are default methods on some of the functional interfaces I've been covering in the last few videos.

The Consumer, Predicate, and Function interfaces all come with these methods, as does the Comparator, which I'll also include here.

Convenience Methods on functional interfaces in java.util.function package

Category of Interface	Convenience method example	Notes
Function	function1.**andThen**(function2)	Not implemented on IntFunction, DoubleFunction, LongFunction
Function	function2.**compose**(function1)	Only implemented on Function & UnaryOperator
Consumer	consumer1.**andThen**(consumer2)	
Predicate	predicate1.**and**(predicate2)	
Predicate	predicate1.**or**(predicate2)	
Predicate	predicate1.**negate**()	

For andThen, and compose, on the Function category of interfaces, any Interim functions are not required to have the same type arguments.

Instead, one function's output becomes the next function's input, and the next function's output is not constrained to any specific type, except the last function executed in the chain.

Convenience Methods on functional interfaces in java.util.function package

Category of Interface	Convenience method example	Notes
Function	function1.**andThen**(function2)	Not implemented on IntFunction, DoubleFunction, LongFunction
Function	function2.**compose**(function1)	Only implemented on Function & UnaryOperator
Consumer	consumer1.**andThen**(consumer2)	
Predicate	predicate1.**and**(predicate2)	
Predicate	predicate1.**or**(predicate2)	
Predicate	predicate1.**negate**()	

The Consumer's andThen method is different, because it never returns a result, so you use this when you're chaining methods independent of one another.

The Predicate methods always return a boolean, which will combine the output of the two expressions, to obtain a final boolean result.

Comparator's additional helper methods

I want to cover the additional convenience methods now, since, as you can see from this table, many take a functional interface instance, as an argument.

Type of Method	Method Signature
static	Comparator **comparing**(Function keyExtractor)
static	Comparator naturalOrder()
static	Comparator reverseOrder()
default	Comparator **thenComparing**(Comparator other)
default	Comparator **thenComparing**(Function keyExtractor)
default	Comparator **reversed**()

There is a comparing static method, and an overloaded default method named thenComparing, and finally a default reversed method.