**Lambda Expressions:**

If you remember the days before Java SE 8 was released, then you probably remember the anonymous classes concept. And maybe you have heard that lambda expressions are another, simpler way of writing instances of anonymous classes, in some precise cases.

Now you do not need to go through anonymous classes to understand how to write a lambda expression. you do not need anonymous classes anymore.

Writing a lambda expression breaks down to understanding three steps:

* identifying the type of the lambda expression you want to write
* finding the right method to implement
* implementing this method.

Step1: Identifying the Type of a Lambda Expression

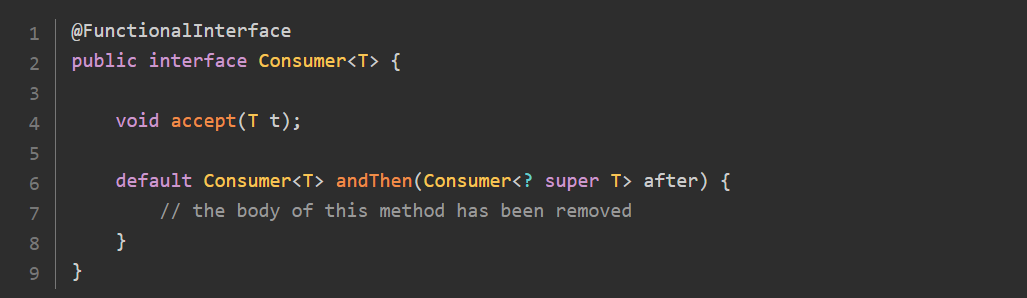
* Everything has a type in the Java language, and this type is known at compile time. So it is always possible to find the type of a lambda expression. It may be the type of a variable, of a field, of a method parameter, or the returned type of a method.
* There is a restriction on the type of a lambda expression: it has to be a functional interface. So an anonymous class that does not implement a **functional interface** cannot be written as a lambda expression.
* **Functional Interface**: functional interface is an interface that has only one abstract method.

Examples of Functional Interfaces

Let us see some examples taken from the JDK API. We have just removed the comments from the source code.



The **Runnable** interface is indeed functional, because it has only one abstract method. The **@FunctionalInterface** annotation has been added as a helper, but it is not needed.



The **Consumer<T>** interface is also functional: it has one abstract method and one default, concrete method that does not count.



The [Predicate<T>](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html)interface is a functional interface:

* it has one abstract method
* it has three default methods that do not count
* and it has two static methods that do not count neither.

**Finding the Right Method to Implement**

A lambda expression is an implementation of the only abstract method in this functional interface. So finding the right method to implement is just a matter of finding this method.



**Implementing the Right Method with a Lambda Expression**

writing the lambda itself :

This syntax is made of three elements:

* a block of parameters;
* a little piece of ASCII art depicting an arrow: ->. Note that Java uses meager arrows (->) and not fat arrows (=>);
* a block of code which is the body of the method.

Example: Suppose you need an instance of a Predicate that returns true for strings of characters that have exactly 3 characters.

Predicate<String> : It is used to give the result Boolean value



Implementing a Consumer<String> : It is used to consume the Input

Let us write a lambda that consumes a String and prints on System.out.



Implementing a Runnable

Implementing a Runnable turns out to write an implementation of void run(). This block of arguments is empty, so it should be written with parentheses. Remember: you can omit the parentheses only if you have one argument, here we have zero.

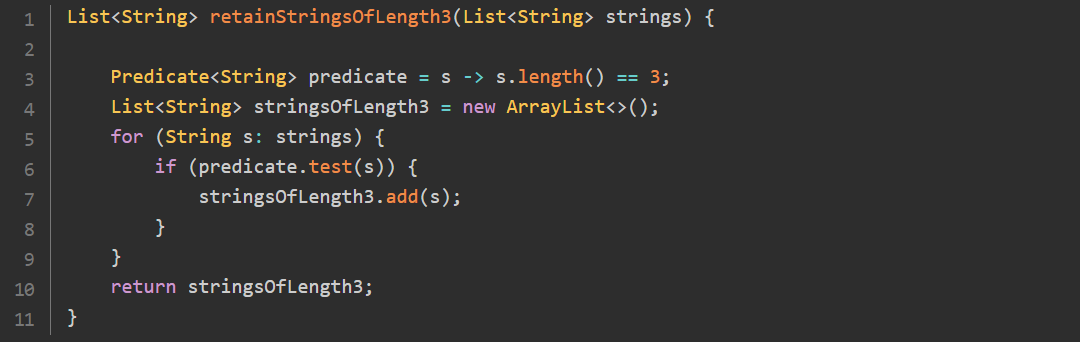
So let us write a runnable that tells us that it is running:



**Calling a Lambda Expression**

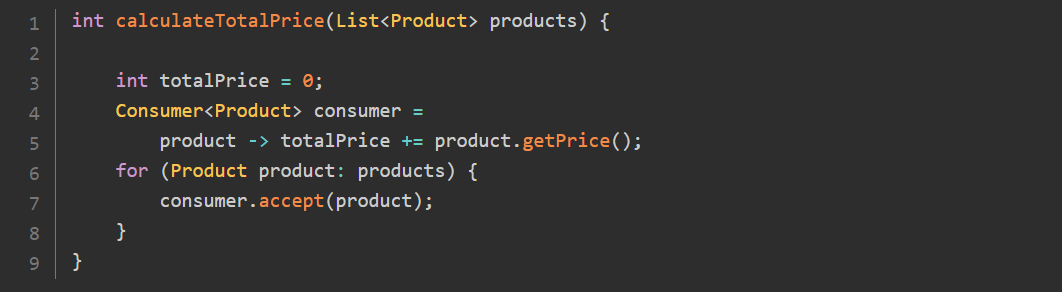
suppose that this predicate has been defined in a method. How can you use it to test if a given string of characters is indeed of length 3?

write a lambda, you need to keep in mind that this lambda is an instance of the interface Predicate. This interface defines a method called test() that takes a String and returns a boolean.



Since the [Predicate](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html) interface defines this method [boolean test(String)](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html" \l "test(T)" \t "_blank), it is perfectly legal to call the methods defined in [Predicate](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html) through a variable of type [Predicate](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html). This may look confusing at first, since this predicate variable does not look like it defines methods.

So everytime you write a lambda, you can call any method defined on the interface this lamdba is implementing. Calling the abstract method will call the code of your lambda itself, since this lambda is an implementation of that method. Calling a default method will call the code written in the interface. There is no way a lambda can override a default method.



Even if this code may look nice, trying to compile it will give you the following error on the use of totalPrice in this [Consumer](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Consumer.html) implementation:

*Variable used in lambda expression should be final or effectively final*

The reason is the following: lambdas cannot modify variables defined outside their body. They can read them, as long as they are final, that is, immutable. This process of accessing variable is called capturing: lambdas cannot capture variables, they can only capture values. A final variable is in fact a value.

You have noted that the error message tells us that the variable can be final, which is a classical concept in the Java language. It also tells us that the variable can be effectively final. This notion was introduced in Java SE 8: even if you do not explicitly declare a variable final, the compiler may do it for you. If it sees that this variable is read from a lambda, and that you do not modify it, then it will nicely add the final declaration for you.

**Using Lambdas Expressions in Your Application:**

Discovering the **java.util.function** package:

The JDK 8 also introduces a new package: **java.util.function** with functional interfaces for you to use in your application. These functional interfaces are also heavily used in the JDK API, especially in the Collections Frameworks and the Stream API. This package is in the **java.base** module.

With a little more than 40 interfaces, this package may look a little scary at first. It turns out that it is organized around four main interfaces. Understanding them gives you the key to understand all the others.

Creating or Providing Objects with Supplier<T>:

Implementing the **Supplier<T>** Interface

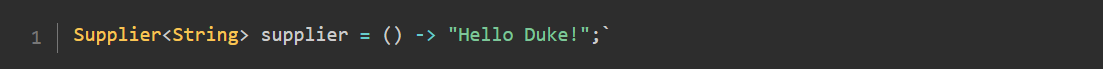
The first interface is the **Supplier<T>** interface. In a nutshell, a supplier does not take any arguments and returns an object.

We should really say: a lambda that implements the supplier interface does not take any argument and returns an object. Making shortcuts makes things easier to remember, as long as they are not confusing.

This interface is really simple: it has no default or static method, just a plain **get()** method. Here is this interface:



The following lambda is an implementation of this interface:



This lambda expression simply returns the Hello Duke! string of characters. You can also write a supplier that returns a new object every time it is invoked:



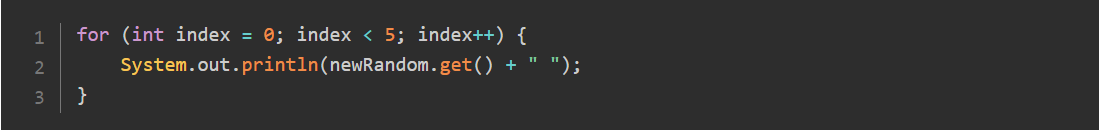
Calling the **get()** method of this supplier will invoke **random.nextInt(),** and will produce random integers. Since the seed of this random generator is fixed to the value 314L, you should see the following random integers generated:



Note that this lambda is capturing a variable from the enclosing scope: random, making this variable effectively final.

Using a **Supplier<T>:**

Note how you generated random numbers using the **newRandom** supplier in the previous example**:**



Calling the **get()** method of the Supplier interface invokes your lambda.

Using Specialized Suppliers:

Lambda expressions are used to process data in applications. How fast a lambda expression can be executed is thus critical in the JDK. Any CPU cycle that can be saved has to be saved, since it may represent a significant optimization in a real application.

Following this principle, the JDK API also offers specialized, optimized versions of the **Supplier<T>** interface.

You may have noticed that our second example supplies the **Integer** type, where the **Random.nextInt()** method returns an int. So in the code you wrote, there are two things that are happening under the hood:

* the **int** returned by the **Random.nextInt()** is first boxed into an Integer, by the auto-boxing mechanism;
* this Integer is then unboxed when assigned to the **nextRandom** variable, by the auto-unboxing mechanism.

The auto-boxing is the mechanism by which an **int** value can be directly assigned to an **Integer** object:



Under the hood, an object is created for you, wrapping that value.

The auto-unboxing does the opposite. You may assign an **Integer** to an **int** value, by unwrapping the value within the **Integer**:



This boxing / unboxing does not come for free. Most of the time, this cost will be small compared to other things your application is doing, like getting data from a database or from a remote service. But in some cases, this cost may be not acceptable, and you need to avoid paying it.

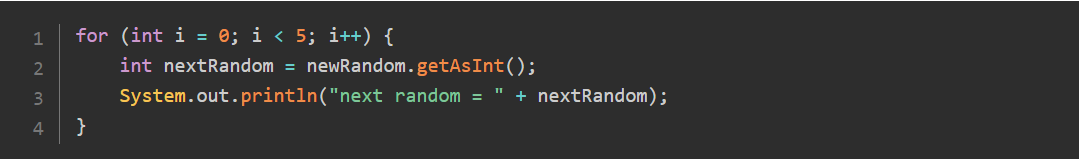
The good news is: the JDK gives you a solution with the **IntSupplier** interface. Here is this interface:



Notice that you can use the exact same code to implement this interface:



The only modification to your application code is that you need to call **getAsInt()** instead of **get():**



The result of running this code is the same, but this time no boxing / unboxing occurred: this code is more performant than the previous one.

The JDK gives you four of these specialized suppliers, to avoid unnecessary boxing / unboxing in your application: **IntSupplier, BooleanSupplier, LongSupplier and DoubleSupplier.**

You will see more of these specialized version of functional interfaces to handle primitive types. There is a simple naming convention for their abstract method: take the name of the main abstract method (**get()** in the case of the supplier), and add the returned type to it. So for the supplier interfaces we have: **getAsBoolean(), getAsInt(), getAsLong(), and getAsDouble().**

**Consuming Objects with Consumer<T>:**

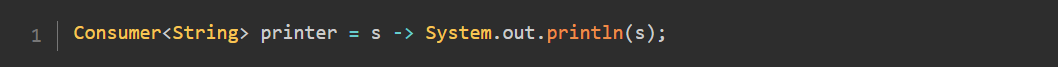
Implementing and Using Consumers:

The second interface is the **Consumer<T>** interface. A consumer does the opposite of the supplier: it takes an argument and does not return anything.

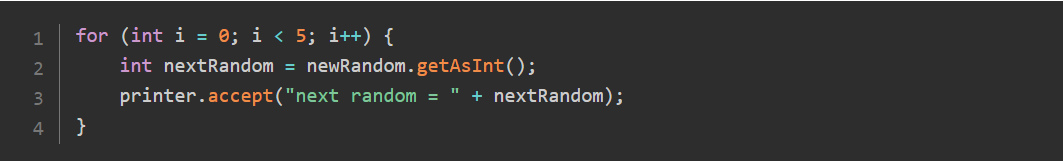
The interface is a little more complex: there are default methods in it, which will be covered later in this tutorial. Let us concentrate on its abstract method:



You already implemented consumers:



You can update the previous example with this consumer:



Using Specialized Consumers:

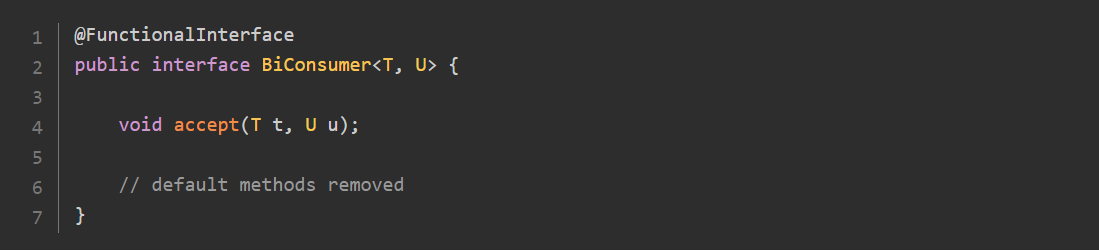
Suppose you need to print integers. Then you can write the following consumer:

Then you may face the same auto-boxing issue as with the supplier example. Is this boxing / unboxing acceptable in your application, performance-wise?

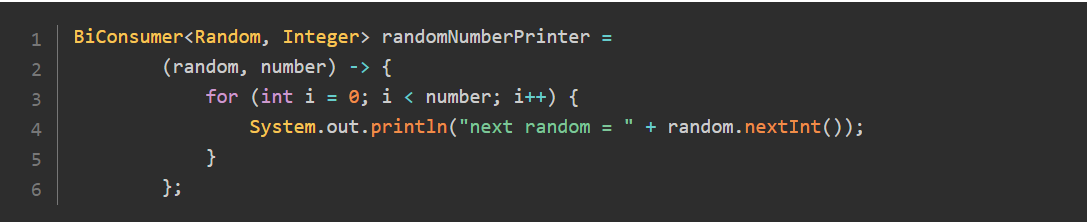
Do not worry if this is not the case, the JDK has you covered with the three specialized consumers available: **IntConsumer, LongConsumer, and DoubleConsumer**. The abstract methods of these three consumers follow the same convention as for the supplier, since the returned type is always **void**, they are all named **accept**.

**Consuming Two Elements with a BiConsumer:**

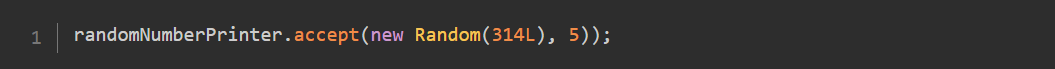
Then the JDK adds another variant of the Consumer<T> interface, which takes two arguments instead of one, called quite naturally the BiConsumer<T, U> interface. Here is this interface:



Here is an example of a biconsumer:



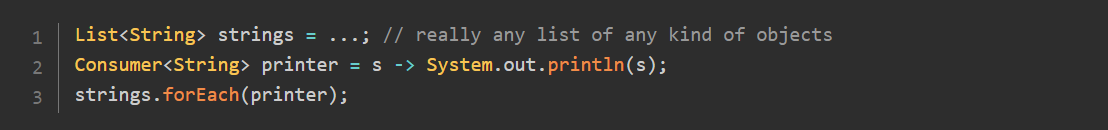
You can use this biconsumer to write the previous example differently:



There are three specialized versions of the **BiConsumer<T, U>** interface to handle primitive types: **ObjIntConsumer<T>**, **ObjLongConsumer<T>** and **ObjDoubleConsumer<T>.**

Passing a Consumer to an Iterable:

Several important methods have been added to the interfaces of the Collections Framework, that are covered in another part of this tutorial. One of them takes a **Consumer<T>** as an argument and is extremely useful: the **Iterable.forEach()** method. Here is a simple example, that you will see everywhere:



This last line of code will just apply the consumer to all the objects of the list. Here it will simply print them one by one on the console. You will see another way to write this consumer in a later part.

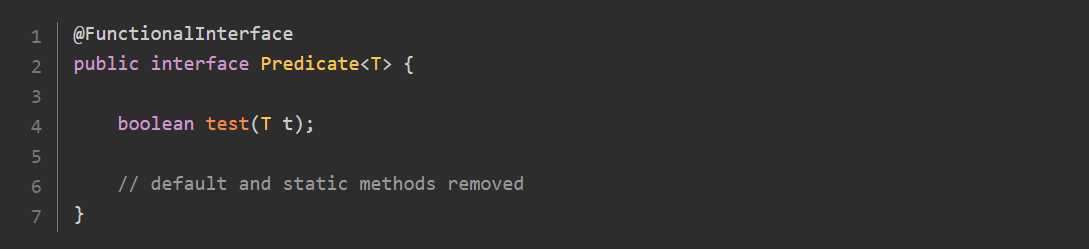
This **forEach()** method exposes a way to access an internal iteration over all the elements of any **Iterable**, passing the action you need to take on each of these elements. It is a very powerful way of doing so, and it also makes your code more readable.

Testing Objects with Predicate<T>:

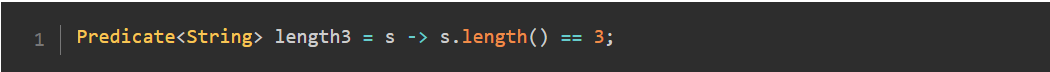
Implementing and Using Predicates:

The third interface is the **Predicate<T>** interface. A predicate is used to test an object. It is used for filtering streams in the Stream API, a topic that you will see later on.

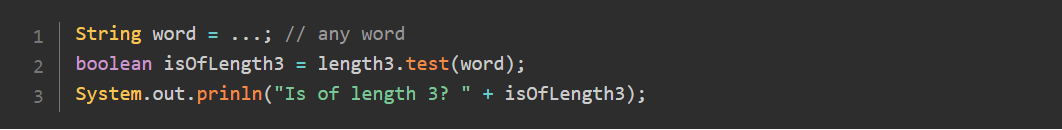
Its abstract method takes an object and returns a boolean value. This interface is again a little more complex than **Consumer<T>**: there are default methods and static methods defined on it, which you will see later on. Let us concentrate on its abstract method:



You already saw an example of a **Predicate<String>** in a previous part:

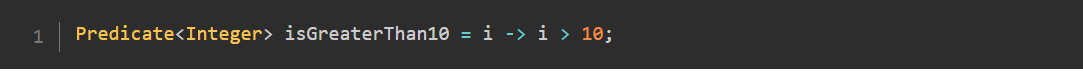


To test a given string, all you need to do is call the **test()** method of the **Predicate** interface:



Using Specialized Predicates:

Suppose you need to test integer values. You can write the following predicate:



The same goes for the consumers, the supplier, and this predicate. What this predicate takes as an argument is a reference to an instance of the **Integer** class, so before comparing this value to 10, this object is auto-unboxed. It is very convenient but comes with an overhead.

The solution provided by the JDK is the same as for suppliers and consumers: specialized predicates. Along with Predicate<String> are three specialized interfaces: **IntPredicate**, **LongPredicate**, and **DoublePredicate**. Their abstract methods all follow the naming convention. Since they all return a **boolean**, they are just named **test()** and take an argument corresponding to the interface.

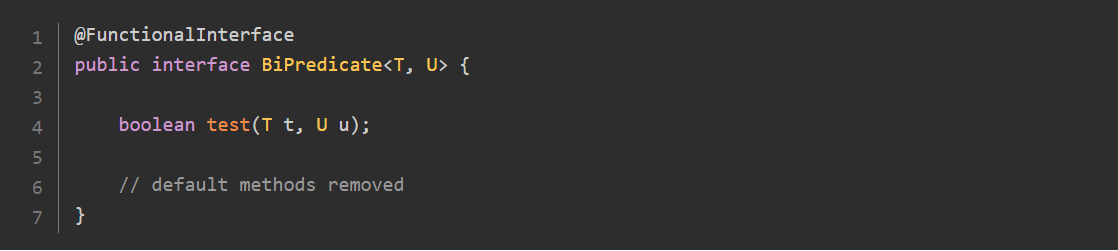
So you can write the previous example as follow:



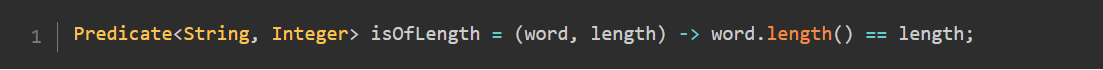
You can see that the syntax of the lambda itself is the same, the only difference is that i is now an int type instead of **Integer**.

Testing Two Elements with a BiPredicate:

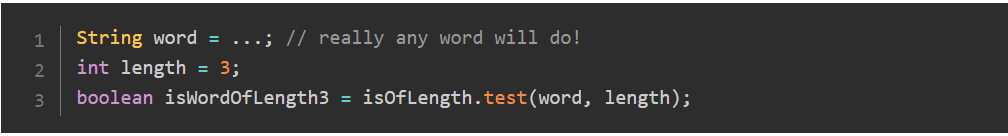
Following the convention you saw with the **Consumer<T>,** the JDK also adds a **BiPredicate<T, U>** interface, which tests two elements instead of one. Here is the interface:



Here is an example of such a **bipredicate**:



You can use this **bipredicate** with the following pattern:

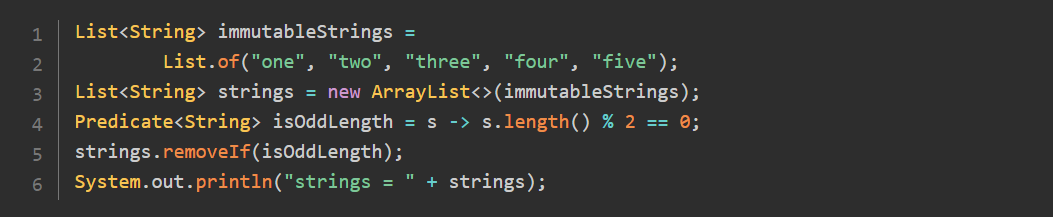


There is no specialized version of **BiPredicate<T, U>** to handle primitive types.

Passing a Predicate to a Collection:

One of the methods added to the Collections Framework takes a predicate: the **removeIf()** method. This method uses this predicate to test each element of the collection. If the result of the test is true, then this element is removed from the collection.

You can see this pattern in action in the following example:



Running this code will produce the following result:



There are several things worth pointing out on this example:

As you can see, calling **removeIf()** mutates this collection.

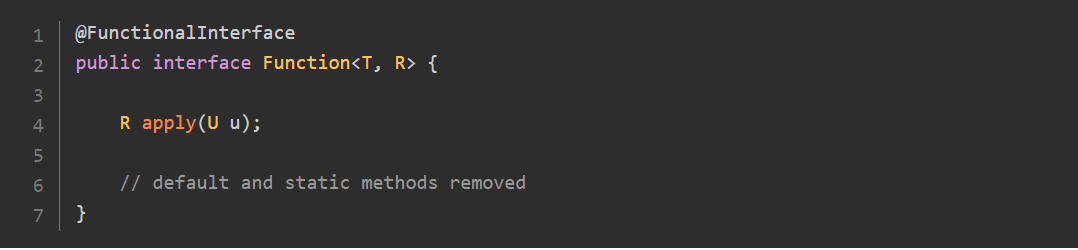
So you should not call **removeIf()** on an immutable collection, like the ones produced by the **List.of()** factory methods. You will get an exception if you do that because you cannot remove elements from an immutable collection.

**Arrays.asList()** produces a collection that behaves like an array. You can mutate its existing elements, but you are not allowed to add or remove elements from the list returned by this factory method. So calling **removeIf()** on this list will not work either.

Mapping Objects to Other Objects with Function<T, R>:

Implementing and Using Functions

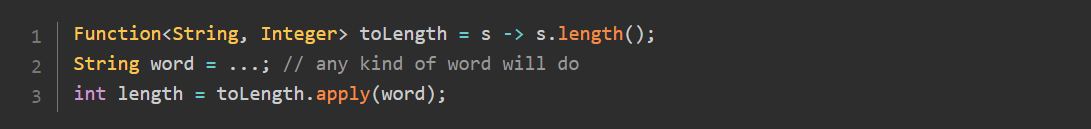
The fourth interface is the **Function<T, R>** interface. The abstract method of a function takes an object of type T and returns a transformation of that object to any other type U. This interface also has default and static methods.



Functions are used in the Stream API to map objects to other objects, A predicate can be seen as a specialized type of function, that returns a **boolean**.

Using Specialized Functions:

This is an example of a function that takes a string and returns the length of that string.



Here again, you can spot the boxing and unboxing operations in action. First, the **length()** method returns an **int**. Since the function returns an **Integer**, this int is boxed. But then the result is assigned to a variable **length** of type **int**, so the **Integer** is then unboxed to be stored in this variable.

If performance is not an issue in your application, then this boxing and unboxing is really not a big deal. If it is, you will probably want to avoid it.

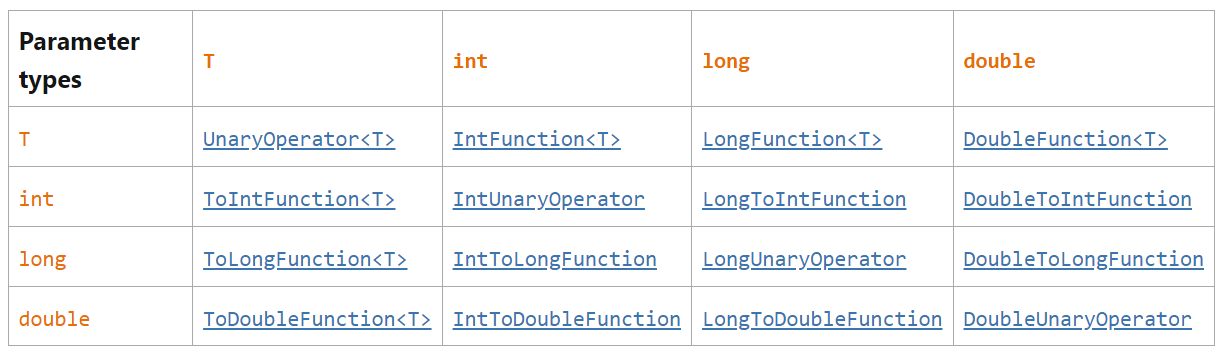
The JDK has solutions for you, with specialized versions of the **Function<T, R>** interface. This set of interfaces is more complex than the one we saw for the **Supplier**, the **Consumer<T>,** or the **Predicate** categories because specialized functions are defined both for the type of the input argument, and the returned type.

Both the input argument and the output can have four different types:

* a parameterized type **T**;
* an **int**;
* a **long**;
* a **double**.

Things do not stop here, because there is a subtlety in the design of the API. There is a special interface: **UnaryOperator<T>** which extends **Function<T, T>.** This unary operator concept is used to name the functions that take an argument of a given type, and return a result of the same type. A unary operator is just what you would expect. All the classical math operators can be **modeled** by a **UnaryOperator<T>:** the square root, all the trigonometric operators, the logarithm, and the exponential.

Here are the 16 specialized types of functions you can find in the **java.util.function** package.



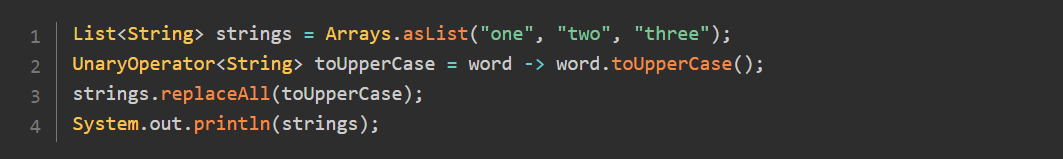
All the abstract methods of these interfaces follow the same convention: they are named after the returned type of that function. Here are their names:

* **apply()** for the functions that return a generic type T
* **applyAsInt()** if it returns the primitive type int
* **applyAsLong()** for long
* **applyAsDouble()** for double

Passing a Unary Operator to a List:

You can transform the elements of a list with a **UnaryOperator<T>.** One could wonder why a **UnaryOperator<T>** and not a basic **Function**. The answer is in fact quite simple: once declared, you cannot change the type of a list. So the function you apply can change the elements of the list, but not their type.

The method that takes this unary operator passes it to the **replaceAll()** method. Here is an example:



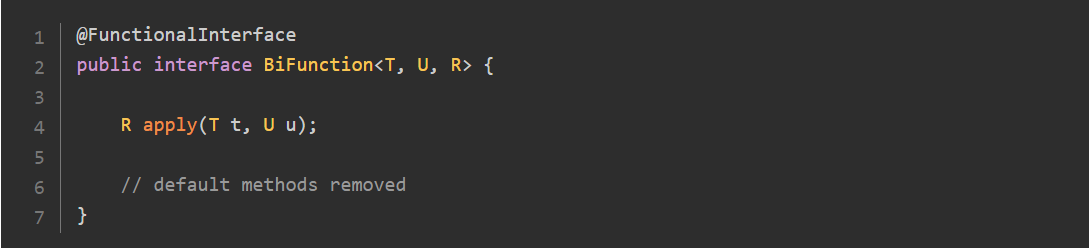
Running this code displays the following:



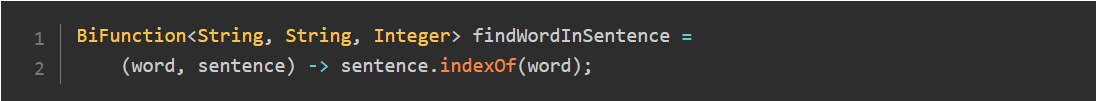
Note that this time we used a list created with the **Arrays.asList()** pattern. Indeed you do not need to add or remove any element to that list: this code just modifies each element one by one, which is possible with this particular list.

Mapping Two Elements with a BiFunction:

As for the consumer and predicate, functions have also a version that takes two arguments: the bifunction. The interface is **BiFunction<T, U, R>**, where T and U are the arguments and R the returned type. Here is the interface:



You can create a bifunction with a lambda expression:



The **UnaryOperator<T>** interface has also a sibling interface with two arguments: the **BinaryOperator<T>,** that extends **BiFunction<T, U, R>.** As you would expect, the four basic arithmetic operations can be **modeled** with a **BinaryOperator.**

A subset of all the possible specialized versions of bifunction has been added to the JDK:

* **IntBinaryOperator, LongBinaryOperator and DoubleBinaryOperator;**
* **ToIntBiFunction<T>, ToLongBiFunction<T> and ToDoubleBiFunction<T>.**

Wrapping up the Four Categories of Functional Interfaces:

The **java.util.function** package is now central in Java, because all the lambda expressions you are going to use in the Collections Framework or the Stream API implement one of the interfaces from that package.

As you saw, this package contains many interfaces and finding your way there may be tricky.

Firstly, what you need to remember is that there are 4 categories of interfaces:

* the suppliers: do not take any argument, return something
* the consumers: take an argument, do not return anything
* the predicates: take an argument, return a Boolean
* the functions: take an argument, return something

Secondly: some interfaces have versions that take two arguments instead of one:

* the biconsumers
* the bipredicates
* the bifunctions

Thirdly: some interfaces have specialized versions, added to avoid boxing and unboxing. There are too many to list them all. They are named after the type they take. For example: **IntPredicate,** or the type they return, as in **ToLongFunction<T>.** They may be named after both: **IntToDoubleFunction.**

Lastly: there are extensions of **Function<T, R>** and **BiFunction<T, U, R>** for the case where all the types are the same: **UnaryOperator<T>** and **BinaryOperator<T>,** with specialized versions for the primitive types.

**Writing Lambda Expressions as Method References:**

Sometimes you will be writing lambda expressions that are just calls to specific methods defined somewhere else.



Sometimes a lambda expression is just a reference to an existing method. In that case you can write it is as a method reference. The previous code then becomes the following:

Written like that, this lambda expression is just a reference to the **println()** method defined on the **System.out** object.

This is where the method reference syntax comes in.

Sometimes a lambda expression is just a reference to an existing method. In that case you can write it is as a method reference. The previous code then becomes the following:

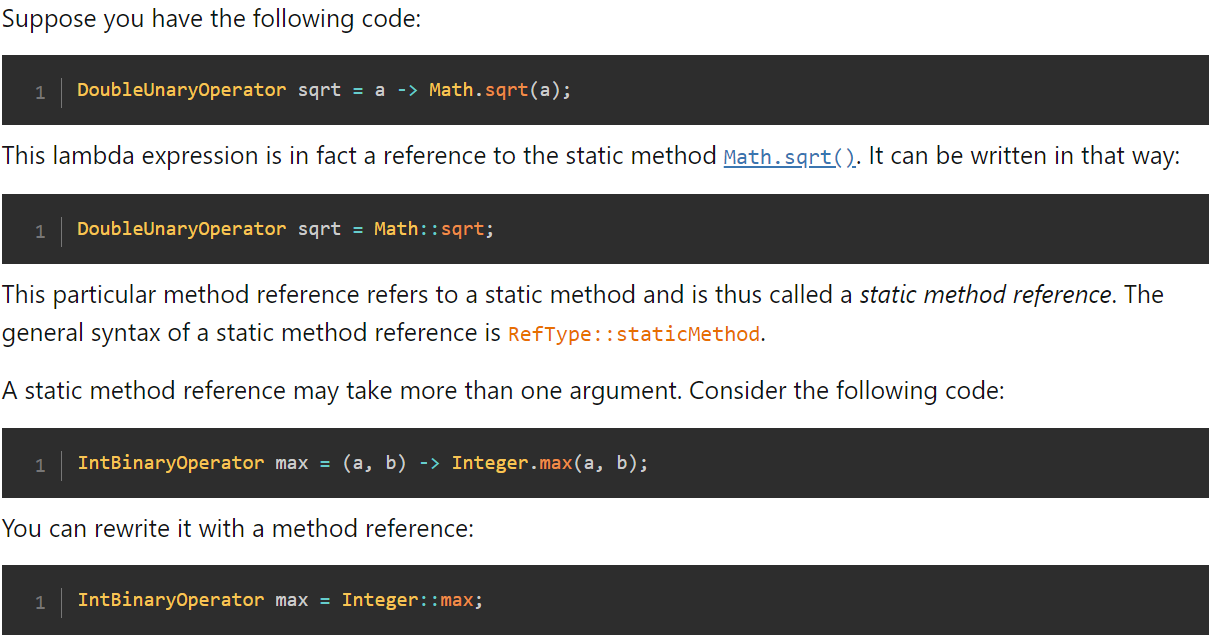


There are four categories of method references:

* Static method references
* Bound method references
* Unbound method references
* Constructor method references

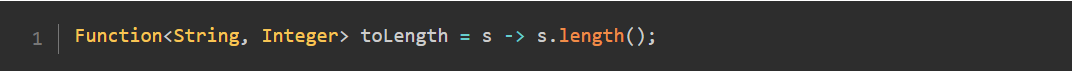
The **printer** consumer belongs to the unbound method references category.

Writing Static Method References:

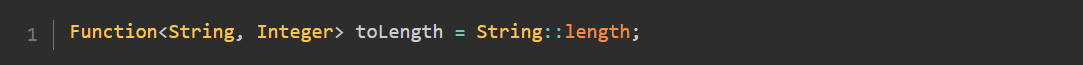


**Writing Unbound Method References:**

**Suppose you have the following code:**

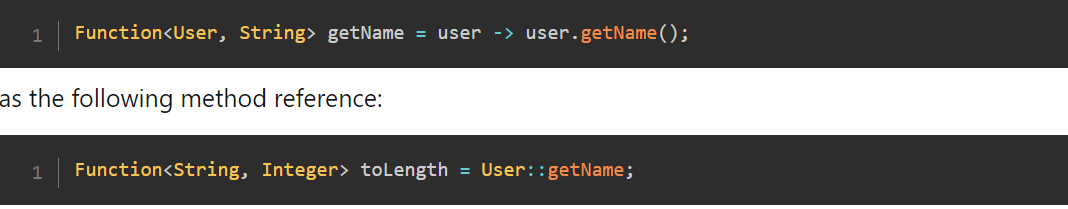


This function could be written as a **ToIntFunction<T>.** It is just a reference to the method **length()** of the class **String**. So you can write it as a method reference:



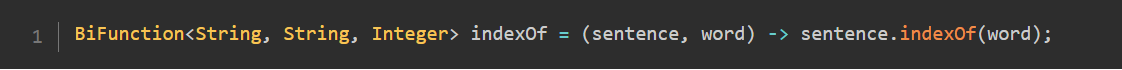
This syntax may be confusing at first since it really looks like a static call. But in fact it is not: the [**length()**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/String.html#length()) method is an instance method of the **String** class.

You can call any getter from a plain Java bean with such a method reference. Suppose you have **User** class with a **getName()** defined on it. You can then write the following function:

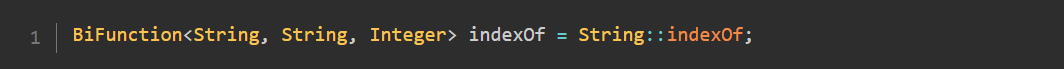


Methods That Do Not Take Any Argument:

**Example**



This lambda is in fact a reference to the **indexOf()**method of the **String**class, and can thus be written as the following method reference:

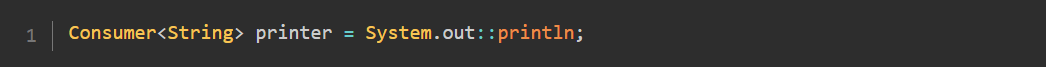


The general syntax of a unbound method reference is the following:

**RefType:instanceMethod**, where **RefType** is the name of a type, and **instanceMethod** is the name of an instance method.

Writing Bound Method References:

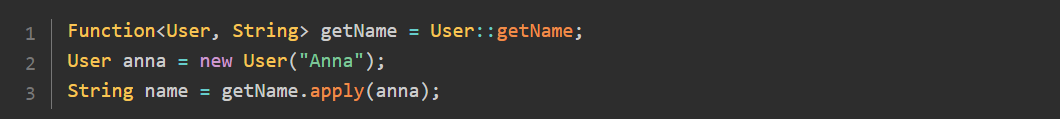
Example



This method reference is called a bound method reference. because the object on which the method is called is defined in the method reference itself.

If you consider the unbound syntax: **Person::getName**, you can see that the object on which the method is called is not part of this syntax: it is provided as an argument of the lambda expression.

Example



You can see that the function is applied to a specific instance of **User**, that is passed to the function. This function then operates on that instance.

This is not the case in the previous consumer example: the **println()** method is called on the **System.out** object, that is part of the method reference.

The general syntax of a bound method reference is the following:

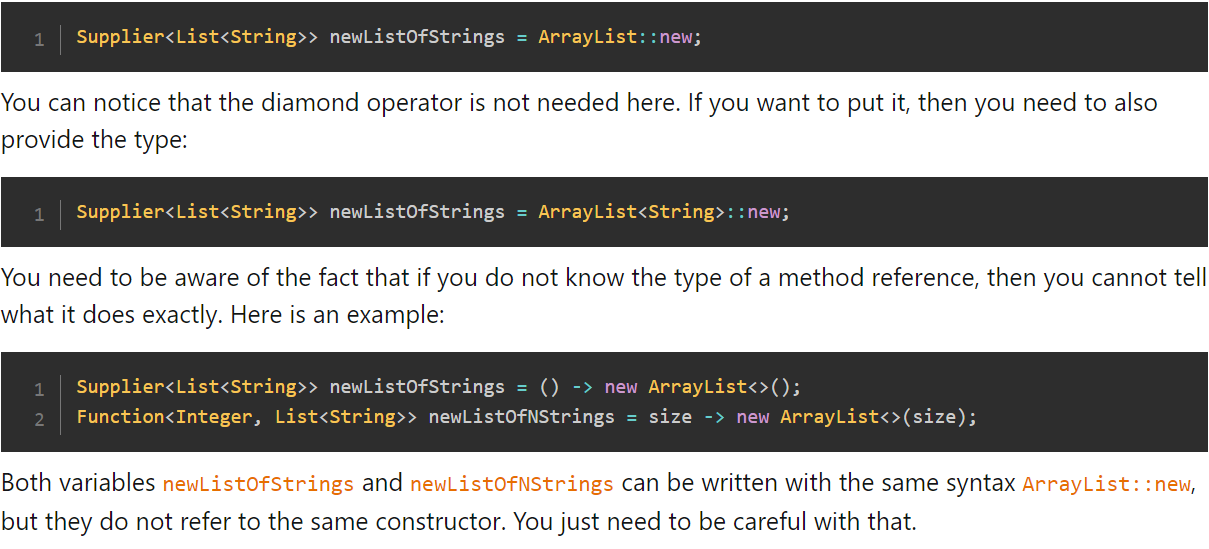
**expr:instanceMethod,** where expr is an expression that returns an object, and **instanceMethod**is the name of an instance method.

Writing Constructor Method References:

Suppose we have **Supplier<List<String>>**:

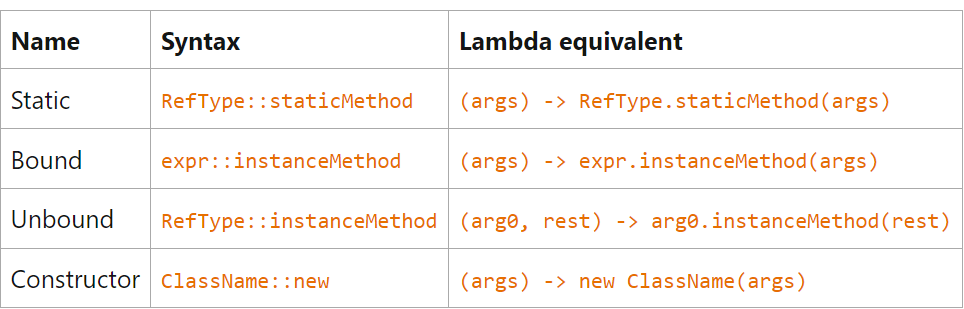


You can see that in the same way as the rest: this boils down to be a reference on the empty constructor of **ArrayList**. Well, method reference can do that. But since a constructor is not a method, this is another category of method references.



Both variables **newListOfStrings** and **newListOfNStrings** can be written with the same syntax **ArrayList::new**, but they do not refer to the same constructor. You just need to be careful with that.

Here are four types of method Reference



**Combining Lambda Expressions:**

You may have noticed the presence of default methods in the functional interfaces of the **java.util.function** package. These methods have been added to allow the combination and chaining of lambda expressions.

It help you to write simpler, more readable code.

Chaining Predicates with Default Methods:

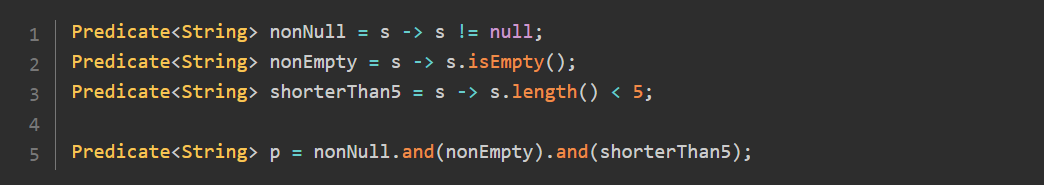
Suppose you need to process a list of strings, to keep only the strings that are non-null, non-empty, and shorter than 5 characters. The way this problem is stated is the following. You have three tests on a given string:

* non-null;
* non-empty;
* shorter than 5 characters.

Each of these tests can easily be written with a very simple, one line predicate. It is also possible to combine those three tests into one single predicate.



But the JDK allows you to write this piece of code in that way:



Combining lambda Expression is Hinding the technical complexity and exposing the intent of a code

How is this code implemented at the API level?

* and() is a method
* It is called on an instance of **Predicate<T>:** it is thus an instance method
* It takes another **Predicate<T>** as an argument
* It returns a [**Predicate<T>**](https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/function/Predicate.html)

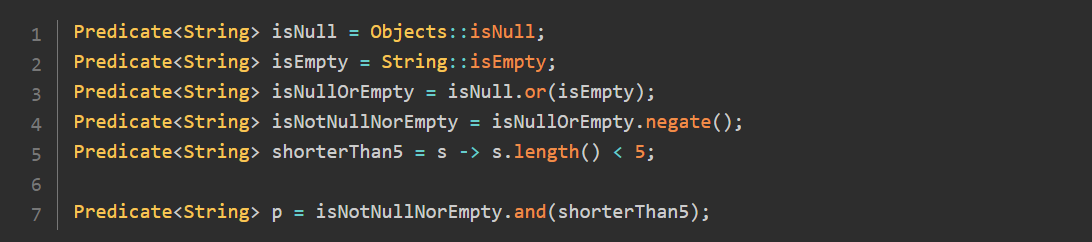
Since only one abstract method is allowed on a functional interface, this **and()** method has to be a default method.

So from the API design point of view, you have all the elements you need to create this method.

The **Predicate<T>** interface has a **and()** default method, so you do not have to do it yourself.

There is also a **or()** method that takes another predicate as an argument, and also a **negate()** method that does not take anything.

Using these, you can write the previous example in this way:

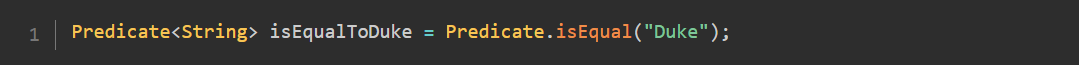


Creating Predicates with Factory Methods:

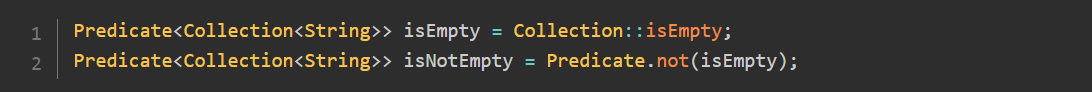
factory methods defined in functional interfaces. There are two of them on the **Predicate<T>** interface.

Example

the predicate **isEqualToDuke** tests a string of characters. The test is true when the tested string is equal to "Duke". This factory method can create predicates for any type of objects.



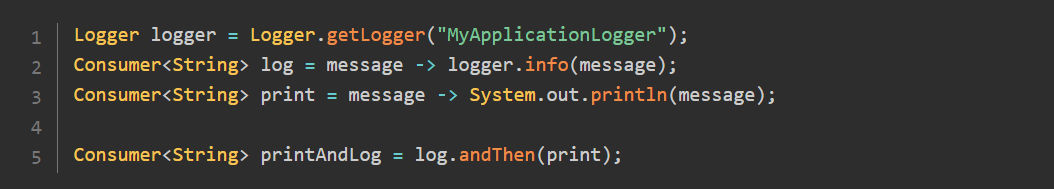
The second factory method negates the predicate given as an argument.



Chaining Consumers with Default Methods:

The **Consumer<T>** interface also has a method to chain consumers.

Example:



In this **printAndLog** is a consumer that will first pass the message to the log consumer and then pass it to the print consumer.

Chaining and Composing Functions with Default Methods:

The difference between chaining and composing is a little subtle. The result of both operations is in fact the same. What is different is the way you write it.

Suppose you have two functions **f1** and **f2**. You can chain them by calling **[f1.andThen(f2)**. Applying the resulting function to an object will first pass this object to **f1** and the result to **f2**.

The Function interface has a second default method: **f2.compose(f1)**. Written in this way, the resulting function will first process an object by passing it to the **f1** function and then the result is passed to **f2**.

What you need to realize is that to get the same resulting function, you need to call **andThen()** on **f1** or **compose()** on **f2**.

You can chain or compose functions of different types. There are obvious restrictions though: the type of the result produced by **f1** should be compatible with the type consumed by **f2**.

Creating an Identity Function**:**

The **Function<T, R>** interface also has a factory method to create the identity function, called **identity().**

Example



**Writing and Combining Comparators:**

Implementing a Comparator with a Lambda Expression:

the good old **Comparator<T>** interface introduced in JDK 2 became functional. So, implementing a comparator can be done using a lambda expression.

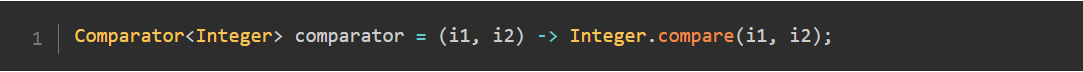
Here is the only abstract method of the **Comparator<T>** interface:



The contract of a comparator is the following:

* If **o1 < o2** then **compare(o1, o2)** should return a negative number
* If **o1 > o2** then **compare(o1, o2)** should return a positive number
* In all cases, **compare(o1, o2)** and compare(o2, o1) should have opposite signs.
* It is not strictly required that in case **o1.equals(o2)** is true, the comparison of **o1** and **o2** returns 0.

How can you create a comparator of integers, that would implement the natural order? Well, you can just use the method you saw at the beginning of this tutorial:



You may have noticed that this lambda expression can also be written with a very nice bound method reference in that way:



Refrain from implementing this comparator with **(i1 - i2)**. Even if this pattern seems to be working, there are corner cases where it will not produce a correct result.

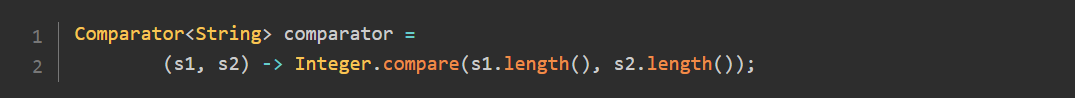
This pattern can be extended to anything you need to compare, as long as you follow the contract of the comparator.

The **Comparator** API went one step further, by providing a very useful API to create comparators in a much more readable way.

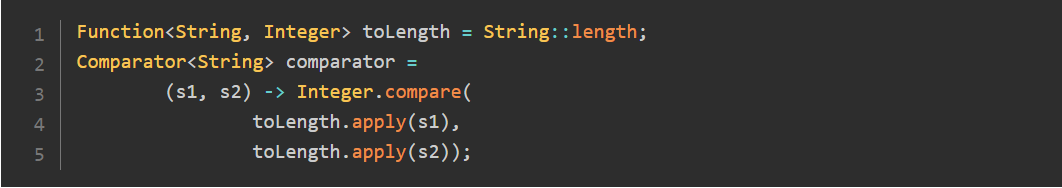
Using a Factory Method to Create a Comparator**:**

Suppose you need to create a comparator to compare strings of characters in a non-natural way: the shortest strings are lesser than the longest strings.

Such a comparator can be written in that way:



You learned in the previous part that it is possible to chain and compose lambda expressions. This code is another example of such a composition. Indeed, you can rewrite it in that way:



Now you can see that the code of this Comparator only depends on the Function called **toLength**. So it becomes possible to create a factory method that takes this function as an argument and returns the corresponding **Comparator<String>.**

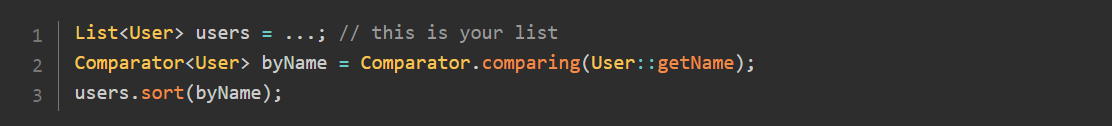
There is still a constraint of the returned type of the **toLength** function: it has to be comparable. Here it works well because you can always compare integers with their natural order, but you need to keep that in mind.

Such a factory method does exist in the JDK: it has been added to the Comparator interface directly. So you can write the previous code in that way:



This comparing() method is a static method of the Comparator interface. It takes a Function as an argument, that should return a type that is an extension of Comparable.

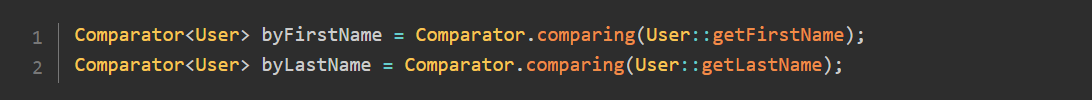
Suppose you have a User class with a **getName()** getter, and you need to sort a list of users according to their name. The code you need to write is the following:



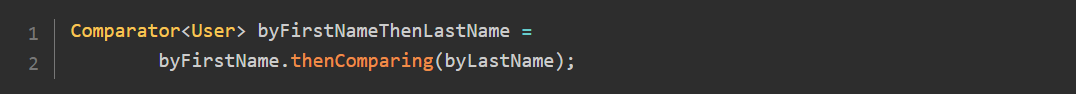
Chaining Comparators:

The company you are working for is currently very happy with the **Comparable<User>** you have delivered. But there is a new requirement in version 2: the User class now has a **firstName** and a **lastName**, and you need to produce a new Comparator to handle this change.

Writing each comparator follows the same pattern as the previous one:

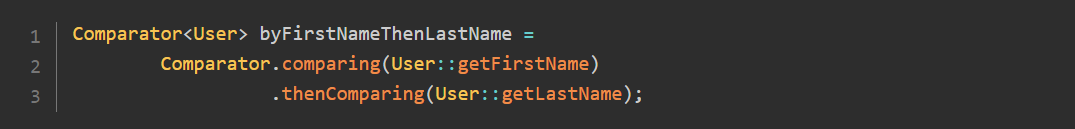


Now what you need is a way to chain them, just as you chained instances of Predicate or Consumer. The Comparator API gives you a solution to do that:



The **thenComparing()** method is a default method of the Comparator interface, that takes another comparator as an argument and returns a new comparator. When applied to two users, the comparator first **compare** these users using the **byFirstName** comparator. If the result is 0, then it will compare them using the **byLastName** comparator.

The Comparator API went one step further: since **byLastName** only depends on the **User::getLastName** function, an overload of the **thenComparing()** method has been added to the API which takes this function as an argument. So the pattern becomes the following:



With lambda expressions, method references, chaining, and composition, creating comparators has

never been so easy!

Specialized Comparators:

Boxing and unboxing or primitive types can also occur with comparators, leading to the same performance hits as in the case of the functional interfaces of the **java.util.function** package. To deal with this problem, specialized versions of the **comparing()** factory method and the **thenComparing()** default method have been added.

You can also create an instance of **Comparator<T>** with:

* comparingInt(ToIntFunction<T> keyExtractor);
* comparingLong(ToLongFunction<T> keyExtractor);
* comparingDouble(ToDoubleFunction<T> keyExtractor);.

You use these methods if you need to compare objects using a property that is a primitive type and need to avoid the boxing / unboxing of this primitive type.

There are also corresponding methods to chain **Comparator<T>:**

* thenComparingInt(ToIntFunction<T> keyExtractor);
* thenComparingLong(ToLongFunction<T> keyExtractor);
* thenComparingDouble(ToDoubleFunction<T> keyExtractor).

The idea is the same: using these methods, you can chain the comparison with a comparator built on a specialized function that returns a primitive type, without having any performance hit due to boxing / unboxing.

**Comparing Comparable Objects Using Their Natural Order:**

There are several factory methods worth mentioning in this tutorial, that will help you create simple comparators.

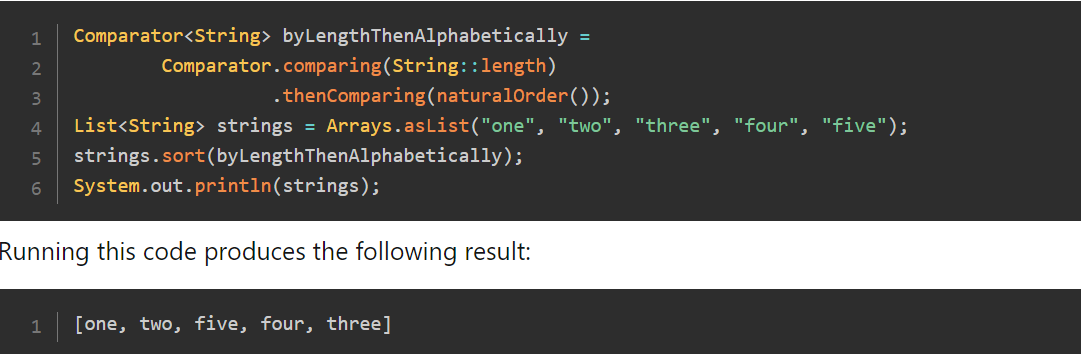
Many classes in the JDK and probably in your application too are implementing a special interface of the JDK: the **Comparable<T>** interface. This interface has one method: **compareTo(T other)** that returns an **int**. This method is used to compare this instance of **T with other**, following the contract of the **Comparator<T>** interface.

Many classes of the JDK are already implementing this interface. This is the case for all the wrapper classes of primitive types (**Integer, Long** and the like), for the **String** class, and for date and time

classes from the Date and Time API.

You can compare the instances of these classes using their natural order, that is, by using this **compareTo()** method. The Comparator API gives you a **Comparator.naturalOrder()** factory class. The comparator it builds does exactly that: it compares any Comparable object using its **compareTo()** method.

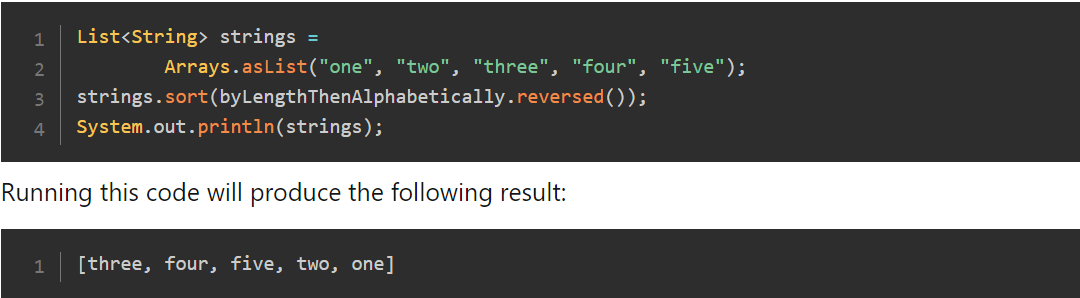
Having such a factory method can be very useful when you need to chain comparators. Here is an example, where you want to compare string of characters with their length, and then with their natural order (this example uses a static import for the **naturalOrder()** method to further improve readability):



**Reversing a Comparator:**

One major use of comparators is of course to sort lists of objects. The JDK 8 saw the addition of a method on the **List** interface especially for that: **List.sort().** This method takes a comparator as an argument.

If you need to sort the previous list in reverse order, you can use the nice **reversed()** method from the **Comparator** interface.

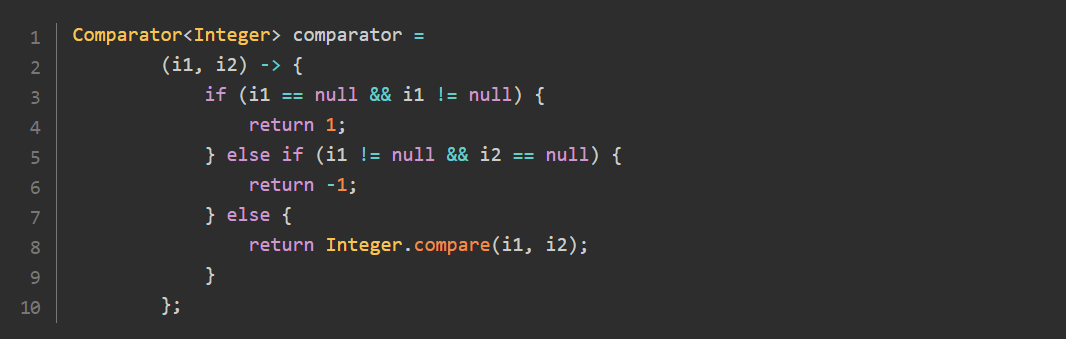


**Dealing with Null Values:**

Comparing null objects may lead to nasty **NullPointerException** while running your code, something you want to avoid.

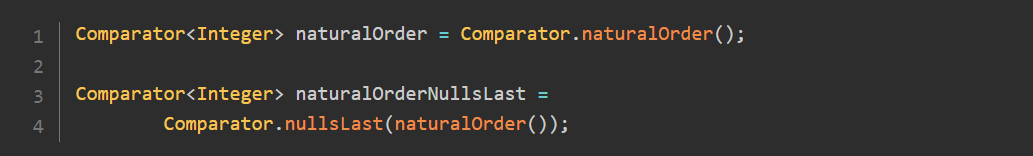
Suppose you need to write a null-safe comparator of integers to sort a list of integers. The convention you decide to follow is to push all the null values at the end of your list, meaning that a null value is greater than any other non-null value. And then you want to sort non-null values in their natural order.

Here is the kind of code you may write to implement this behavior:



You can compare this code to the first comparator you wrote at the beginning of this part, and see that readability took a big hit.

Fortunately, there is a much easier way to write this comparator, using another factory method of the **Comparator** interface.



The **nullsLast()** and its sibling method **nullsFirst()** are factory methods of the Comparator interface. Both take a **comparator** as an argument and do just that: handle the null values for you, pushing them to the end, or putting them first in your sorted list.

Example:

