**Java Lambda Expressions & Stream API**

Lambda expression in computer programming, also called **anonymous function**, is a function defined, and possibly called, without being bound to an identifier or class.

**Why Lambdas**?

* **Enables functional programming**: It’s a paradigm shift. We are used to writing OO programs using java, and lambdas enables us to use functional programming with java. We can not think of functions as entities passed around
* **Readable and concise code**: In certain situations, lambdas eliminate the need of some boilerplate code.
* **Easier to use API’s and libraries:** for instance passing behaviour to Collection API is easier using lambdas
* **Enables support for parallel processing**: Writing code that can run on our processors in the fastest way is a big deal.

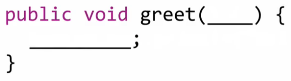
**Functional vs OO**

**Do we need functional programming?**

Functional programming allows us to write better/more readable code, this means maintainable code. There is a reason that we don’t write assembly code, it is hard to read, write and maintain. In certain situations functional programming allows us to write elegant code.

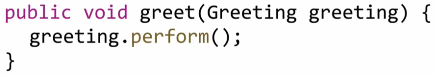
**Some problems with OO programming**: and how functional programming (new feature in java) tries to solve these.

* **Everything is an object**. You cannot have a piece of logic that exist in isolation. It has to be a part of a component(class/object). Most of the time this is not a problem, but sometimes it can be. As a java developer when solving a problem you tend to think in nouns, things, objects rather than actions/verbs. i.e. you need to write a method that greets someone. You cannot just put a function in isolation. You must create a Greeter class.



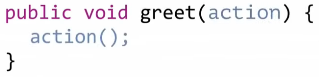
Lets say we want our method to take input parameters and those params will tell the greet method what to do.

1. One way is to have the greet method contain all possible combinations of all it can do and let the input argument be a switch. But this is not elegant design. (case(‘type’))
2. Better way: have the behaviour itself passed as an argument. And the greet method will not contain anything. It will just take the greet behaviour and executes it.



In **Java 7** we can do it like this:

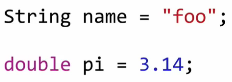
This kind of solves our problem. But there is some extra work that we are doing. We are not just passing a behaviour, we are passing a thing(object) that has a behaviour. We are not passing the perform method directly, it is a part of the object that we pass.



Lambdas take the direct way. It enables us to just pass the action and it directly executes the action.

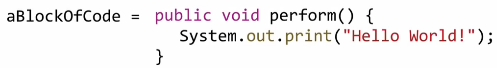
With lambdas we can achieve exactly this. Lambdas lets you create these entities which are just functions, they are called **lambda expressions**. A function which does not belong to a class, functions which exist in isolation. The best part is, those functions can be treated as values, they can be passed around like variables. This can be confusing at first.

**Functions as Values**

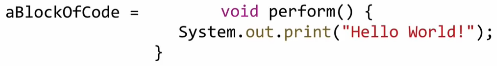
Inline values. Foo is a string which we wrote inline and will be assigned to the variable. Name contains a value which is the string “foo”. So data acts like values in Java. You can assign it to variables and do different operations, similarly objects are also treated as values.

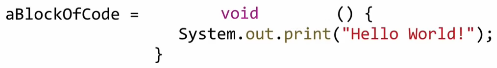
So can we assign a piece of code to a variable? Meaning the variable won’t be the execution return of the block of code, it will be the block itself. A piece of code becomes a value that gets assigned to a variable, wherever the variable goes, the block goes with it.

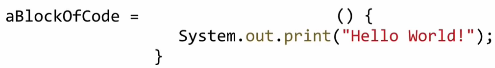
This is possible in **Java 8** using lambdas. You can write a lambda expression which just does this. Once you do this, you can take that variable and pass it around and have a different code execute it.

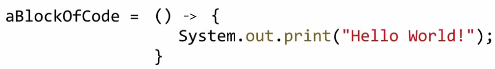


Our goal is to assign a function, a block of code to a variable like so

This code above has a lot of extra things we might not need. For example **public**. It makes sense when a function is part of the class. But if a function exists in isolation it does not make sense to use **public**.

When we assign something to a variable, we can access it using the variable name. This means it doesn’t need this other method name **perform**.

The java compiler is smart enough to look a piece of code and tell what value it is returning (if any). So with lambdas we don’t need to explicitly write it, the compiler will figure it out. In our case, the block doesn’t return any value, so the compiler can tell it has a **void** return type.



So now we have the required elements for a lambda expression.



If the body of your lambda expression is **just one line**, you don’t need the curly braces.

Just like you can pass an inline String value to a method, you can pass in an “**inline**” lambda expression too.



Passing parameters:

If your lambda expression is just one line you **can’t** write return.

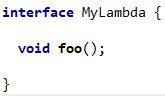


**What’s the type of these variables?**

The creators didn’t create a new type. Instead they decided to use an already existing type, the **interface**.

In order to use lambda expressions:

1. Create an interface, the name doesn’t matter (use an out of the box interface from **java.lang.function**)
2. Create exactly one abstract method in this interface which has the exact same signature as the lambda we will declare.

****

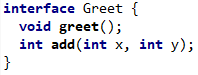
So in our example above, no input params, return type void. The code below is valid now, the compiler/IDE does not complain.



If we try to change arguments we will get an error since in our interface we don’t have a method with arguments.



* If the signature of the lambda we want to use matches the signature of abstract method of an interface then we can use that interface. Meaning that many interfaces could be used / swapped for one expression. Meaning the **Greet** interface can also be used as a type for **myVar** (first example above).
* **Important!** We **CANNOT** use interfaces with **multiple abstract methods** for our lambda expressions (default or static methods inside an interface are fine)! The compiler will be confused. Can’t know which method we meant to use. Our interface **must have only ONE abstract method**.





**Functional Interface**: An interface which has **exactly one abstract method**.

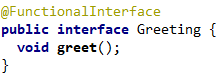
Before **Java 8** you always had abstract methods inside interfaces but now we have **default methods + static methods** (methods with implementation) as well.

* In order to use an interface for a lambda, the interface must be a functional interface. Doesn’t matter how many default or static method it has. When it has exactly one abstract method, then it can be used as a type for a lambda expression.

But normal interfaces can change over time, maybe someone wants to add another abstract method. If we will have lambda expressions depending on that interface then that will be bad for us.

If the consumer of an interface is a class, the new methods can be implemented in that class. But if it’s a lambda expression, then that lambda expressionhas no choice, it won’t work. They are counting on that the interface will have always just one method.

You have a way to declare an interface as a **functional interaface** with the help of an annotation. **RECOMMENDED**

* **@FunctionalInterface** from the **Java.lang** package **->** no need for an import.
* Compiler doesn’t need this. This is an indicator for other developers to keep this interface as a functional interface and don’t try to add new abstract methods to it. Or for devs who will consume this interface.

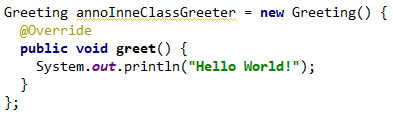
**What is the difference between these two Greeting’s?**



* This is how you execute lambda expressions. By calling the interface method on it, just as if it were an instance of a class.
* Also, how you can pass around lambda expressions is the same as any other object reference.
* So, they are almost the same. In the object, we provided the behaviour in the class and for our lambda expression we just wrote the behaviour (created the function) inline on its own (no class) and gave it to a variable.
* In a way, we are just implementing a function for a particular interface and not implementing a class.

**How do we know it is not an actual class implementation under the hood?**

The difference is hard to find but they are still different. One is an instance and one is a lambda expression

**Another way**: Anonymous inner class. What this does is it creates an inline instance of this class that we provided and will assign it to the variable.

This is **not** a **HelloWorldGreeter** class. It is just an anonymous class which provides an inline implementation for **Greeting** so we can create and return an instance back. So, an instance which implements the Greeting interface, but we have no exact class for it, hence anonymous.

A screenshot of a cell phone

Description automatically generated

For most purposes you can think that lambda expressions are just a shortcut for creating anonymous inner classes (instances). But it’s not exactly true. There are things that the inner class does which are different from lambda expressions. **So -> they are different things**. Will look at those differences later.

* When you compile a program with anonymous inner classes, you will see different .class files in the target directory. For instance, you have a class named **LambdaExample**, that will produce **LambdaExample.class**, and for each inner class defined in there, the compiler will create a new .class file with names like **LambdaExample$1.class**, **LambdaExample$1.class**, …
* It is important to note that **lambdas are NOT syntactic sugar for anonymous inner classes**. Because the purpose was not to have the overhead of creating all those classes, which also means you wont have the overhead of deleting/GC all those classes

**Pass lambda expressions around (SAME as any object reference)**





**Type Inference**

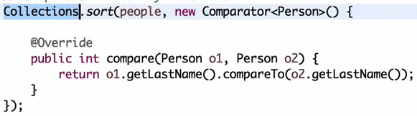
The compiler does something called type inference.

If you only have one parameter in a method, when writing the lambda expression you don’t have to tell the return type. Because the compiler can figure the rest out by itself.



A picture containing knife, table

Description automatically generatedThe compiler also can figure out the parameter types from the interface, so no need to write them as well. Collections.sort() can take 2 Arguments, one of them is a **Comparator** which has a method that takes 2 Arguments of the same type and returns an integer. That’s why the compiler can figure out the types, without us needing to write the types explicitly.

 (Person p1, Person p2) -> ... would also work





* Comparator seems to have 2 abstract methods. But since equals is an inherited method from Object, it doesn’t count as abstract.



**Runnable Using Lambdas**

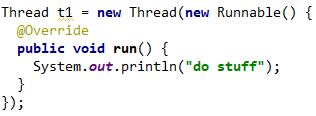
We know that we are reusing interfaces for lambdas and there is no new type for lambda expressions. There are several reasons for that.

* **Backward compatibility**: If they would have created a new type, you could only be able to use those types in code which is already using them. For example, you want to use a library which uses LE. You want to pass a LE to that library. If you had a new function type, you would have to re-write this library so it would expect that new type.

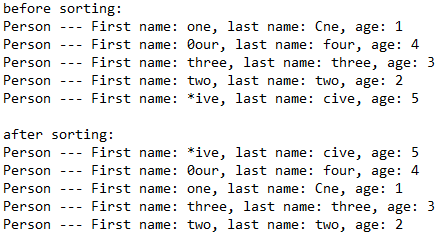
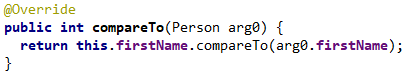
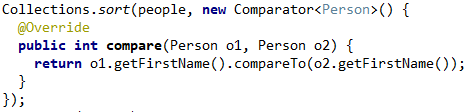
With interfaces we can use lambdas in place of the anonymous inner classes and compatible method signatures which accepts the interface. No need to re-write anything.

**Thread-Runnable**

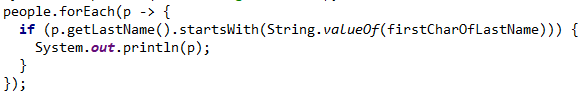
You need to create an instance of **Runnable** and pass it to the Thread object. Normally I always used an anonymous inner class for that but now lambdas can be used as well because **Runnable** is an interface with ONE abstract method.



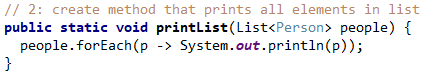
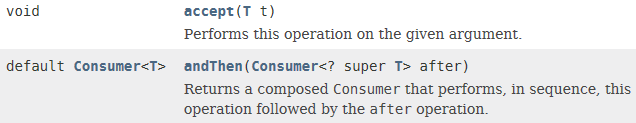


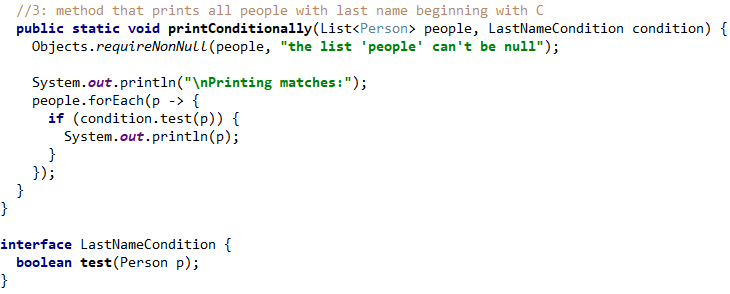












A screenshot of a social media post

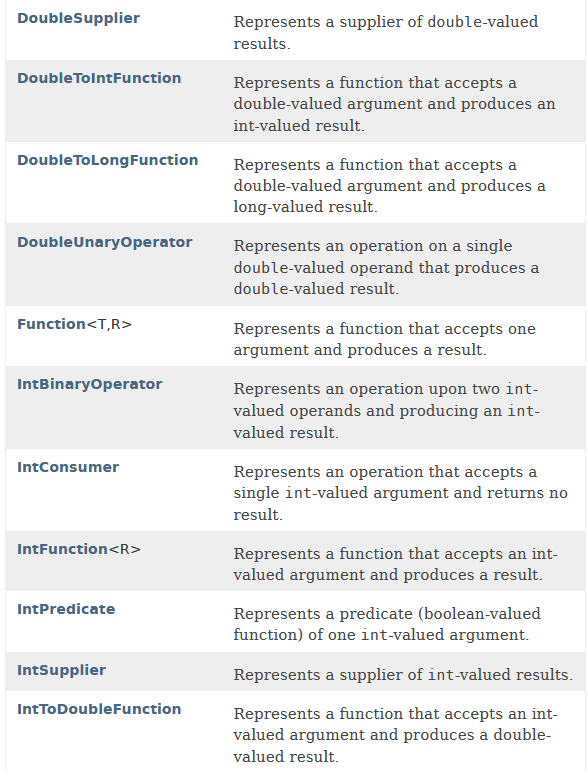
Description automatically generated

A screenshot of a cell phone

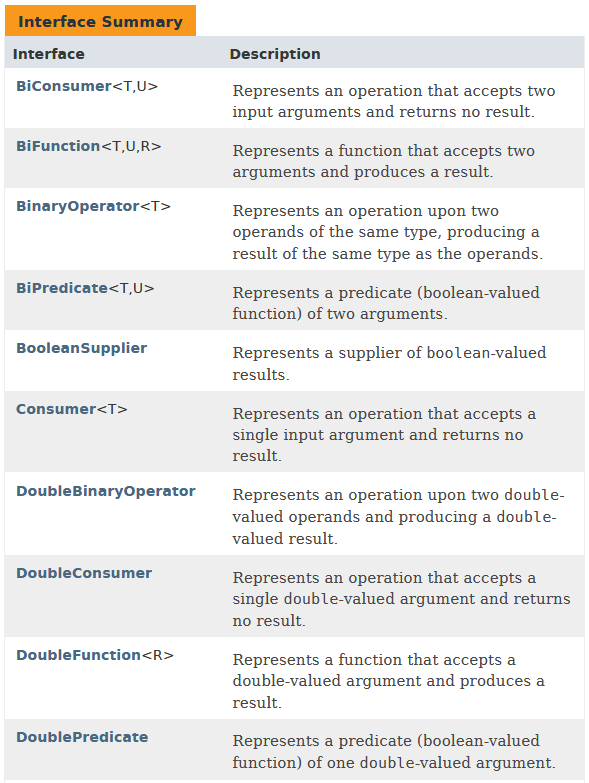
Description automatically generated

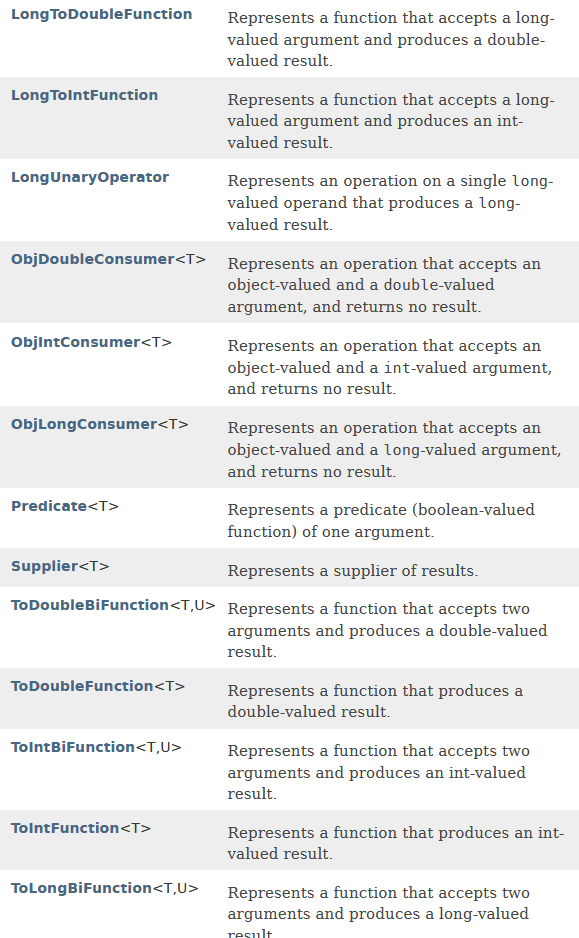
**Using Functional Interfaces**

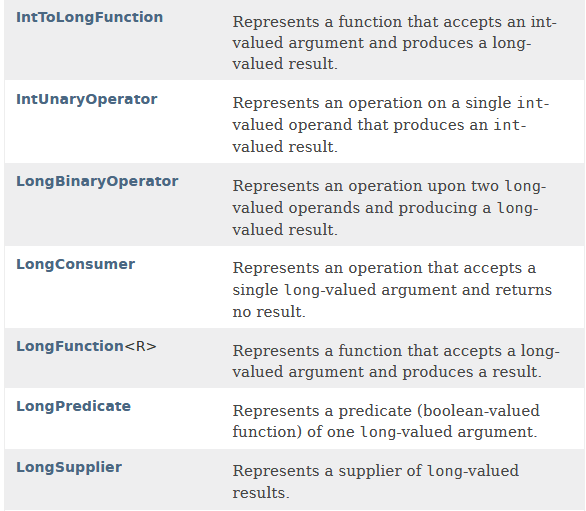
In our last example we had a ‘LastNameCondition’ interface with a test method. This seems like extra work, creating an extra interface. We just created an interface for the sole purpose of using a lambda expression.

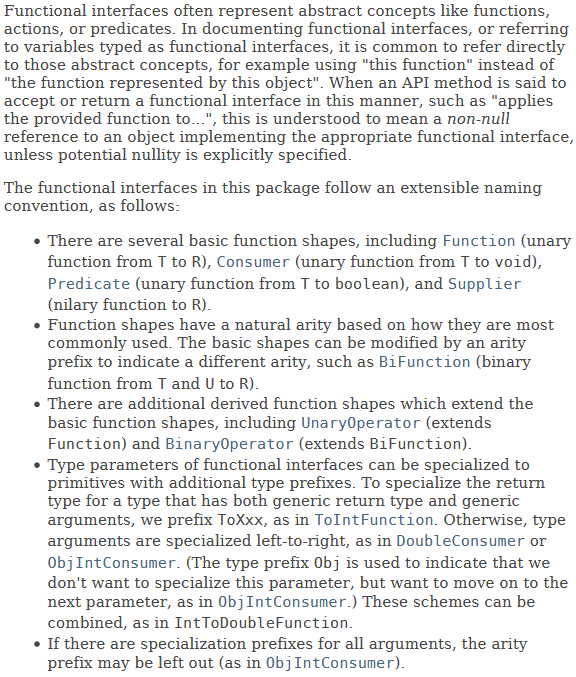
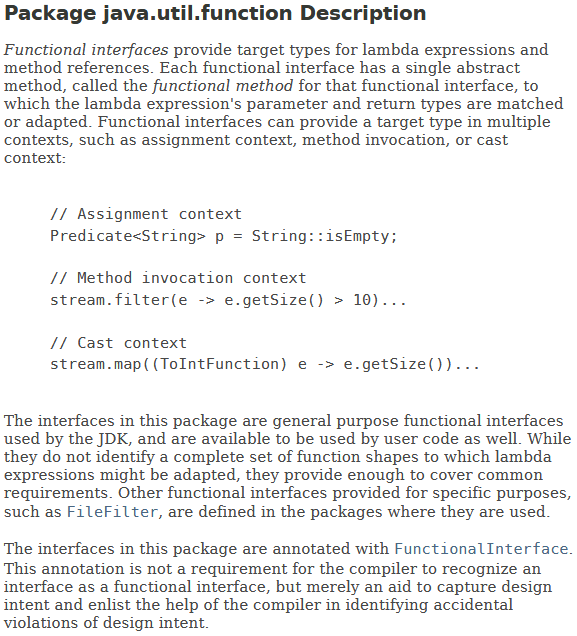
Java 8 provides some out-of-the-box interfaces for different purposes, to address some of these common scenarios. The idea is to use one of these if you have a similar scenario.

The package name is **java.util.function**.



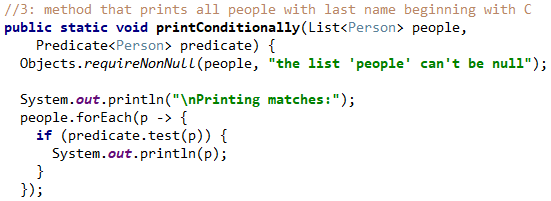
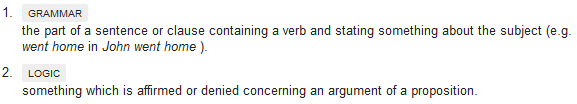




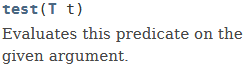


-> So for our example we can use a Predicate interface defined here. It takes an object and creates a boolean. So we don’t have to create a new interface for that.

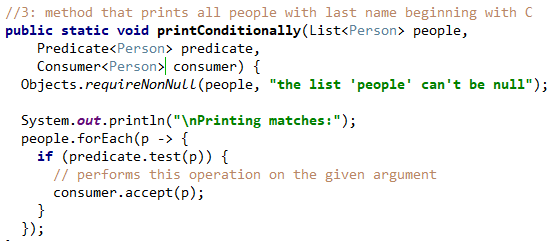
Predicate: Yüklem, Doğrulamak, beyan etmek, dayandırmak







Lets say in this example we also want to pass the behaviour to our ‘printConditionally’ method, instead of printing the person. And since we have the behaviour passed in, we could also pass different behaviours.





**Important Rules**:

* **Keep your lambda expressions one line!** If you need to open curly braces that means you also have to write a return statement and the block gets too large instantly. Keep in mind that if you use a lot of lambda expressions, you are effectively duplicating code, repeating yourself, even if it’s one line. It is also really hard to test. So a good way is to extract a method for common lambda logic and call that method with a method reference. That way you can keep your expression nice and tidy, avoid duplicate logic and you can test your methods.
* Use method references wherever you can

**Exception Handling for Lambdas**

The Functional Interfaces provided by the JDK don't deal with exceptions very well. One reason is because the default functional interfaces don’t really declare a throw in the method signature, which means the lambda can’t really throw a checked exception– and the code becomes verbose and cumbersome when it comes to handling them.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A close up of a logo

Description automatically generatedA screenshot of a cell phone

Description automatically generated

The wrapper method works as expected but, you may argue that it's basically removing the try-catch block from lambda expression and moving it to another method and it doesn't reduce the actual number of lines of code being written.

This is true in this case where the wrapper is specific to a particular use case (just for one Type: Integer and one Exception: ArithmeticException) but we can make use of generics to improve this method and use it for a variety of other scenarios:

A screenshot of a social media post

Description automatically generated

A picture containing knife

Description automatically generated

As we can see, this iteration of our wrapper method takes two arguments, the lambda expression and the type of Exception to be caught. This lambda wrapper is capable of handling all data types, not just Integers, and catch any specific type of exception and not the superclass Exception.

Also, notice that we have changed the name of the method from lambdaWrapper to consumerWrapper. It's because this method only handles lambda expressions for Functional Interface of type Consumer. We can write similar wrapper methods for other Functional Interfaces like Function, BiFunction, BiConsumer and so on.

**Handling Checked Exceptions**

Let's modify the example from the previous section and instead of printing to the console, let's write to a file. Note that the method may throw the IOException.

A screenshot of a cell phone

Description automatically generated

* Think about it like this: the everything to the right of the lambda arrow **i -> {… body …}** is the **body of the abstract method**, defined in the functional interface. In this example, forEach takes a **Consumer<T>** interface and the abstract method is **void accept(T t)**. Since this body might throw an IOException, it needs to be handled immediately or the containing method (in this case **void accept(T t)**) have to declare a throw. But it does not have a throw in its declaration.

A picture containing knife

Description automatically generated

The most straightforward way would be to use a try-catch block, wrap the checked exception into an unchecked exception and rethrow it:

But we are again at the same place we started at, like the last examples.

Let's create a custom functional interface with a single accept method that throws an exception. And now, let's implement a wrapper method that's able to rethrow the exception. This method takes a **ThrowingConsumer** and returns a **Consumer**, so that we can use the standard methods which accepts a **Consumer** as a parameter.

A screenshot of a cell phone

Description automatically generatedA picture containing knife

Description automatically generated

Finally, we're able to simplify the way we use the writeToFile method

This is still a kind of a workaround, but the end result looks pretty clean and is definitely easier to maintain. Both, the ThrowingConsumer and the throwingConsumerWrapper are generic and can be easily reused in different places of our application.

But we can modify this a bit to easily handle any specific exception we want.

A screenshot of a cell phone

Description automatically generated



* Note, that the above code **handles only IOException, whereas any other kind of exception is rethrown as a RuntimeException**.
* **The previous example did not handle anything**. It just rethrew all exceptions as a runtime exception!
* Another way would be to explore the **sneaky-throws hack**. <https://4comprehension.com/sneakily-throwing-exceptions-in-lambda-expressions-in-java/>

**Closures in Lambda Expressions**

A screenshot of a cell phone

Description automatically generated

We can do something like this, and the output will be 30

The important point here is, the anonymous class Process does not have a field called b in its scope. Where the method is getting executed, the **doProcess** method body also does not have that field inside its block.

So the compiler goes one level above, where the method is called to get the local variable b from there and inject that value into i + b.

The varialb b needs to be final or effectively final, which means that you have to guarantee that the variable won’t change until it is used inside inner class method.

In Java 7 or before, you **were forced** to declare variable **b** as **final**.

Since Java 8, you it doesn’t enforce you. If a variable is effectively final, like the example to the right, the compiler can tell it is not changing.

A screenshot of a cell phone

Description automatically generated



**this reference in Lambdas**

There is a difference how the **this** keyword is interpreted between lambdas and anonymous inner classes. This is also a true indication that a lambda is not just a syntactical sugar for anonymous inner classes.

A screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**The value of the this reference inside a lambda expression is the same as if you would use it outside the lambda expression**.

Since in this example we are in a static method, the compiler complains, but if we were in an instance method, the this reference would refer to the instance object. And **this.toString()** would give “ThisReferenceExample instance”

A picture containing knife, table

Description automatically generated

**Method References**

Sometimes, a lambda expression does nothing but call an existing method. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this; they are compact, easy-to-read lambda expressions for methods that already have a name.

A screenshot of a cell phone

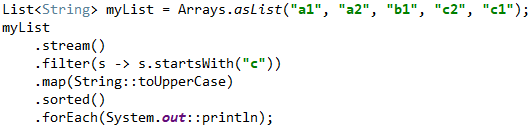
Description automatically generated

name.

asd

**Streams**

TODO

Stream operations are either **intermediate** or **terminal**.

* **Intermediate operations:** return a stream so we can chain multiple intermediate operations without using semicolons.
* **Terminal operations** are either void or return a non-stream result.

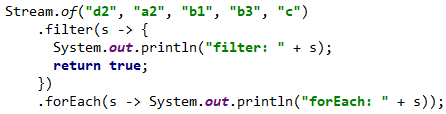
In the above example filter, map and sorted are **intermediate** **operations** whereas forEach is a **terminal** **operation**. For a full list of all available stream operations see the Stream Javadoc. Such a chain of stream operations as seen in the example above is also known as **operation pipeline**.

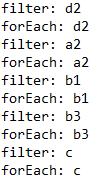
Most stream operations accept some kind of lambda expression parameter, a functional interface specifying the exact behavior of the operation. Most of those operations must be both **non-interfering** and **stateless**. What does that mean?

A function is **non-interfering** when it does not modify the underlying data source of the stream, e.g. in the above example no lambda expression does modify myList by adding or removing elements from the collection.

A function is **stateless** when the execution of the operation is deterministic, e.g. in the above example no lambda expression depends on any mutable variables or states from the outer scope which might change during execution.

…. TODO





* The order of the result might be surprising. A naive approach would be to execute the operations horizontally one after another on all elements of the stream. **But instead each element moves along the chain vertically. The first string "d2" passes filter then forEach, only then the second string "a2" is processed**. Asd

**To Learn More about**

* anyMatch
* collectors.joining(delimeter)