**Java Lambda Expressions & Stream API**

Table of Contents

[1 Java Lambda Expressions 2](#_Toc104033677)

[1.1 Functional vs OO 2](#_Toc104033678)

[1.2 Functions as Values 3](#_Toc104033679)

[1.3 What’s the type of these variables? 4](#_Toc104033680)

[1.4 What is the difference between these two Greeting’s? 5](#_Toc104033681)

[1.5 Type Inference 6](#_Toc104033682)

[1.6 Runnable Using Lambdas 6](#_Toc104033683)

[2 Using Functional Interfaces 9](#_Toc104033684)

[3 Exception Handling for Lambdas 11](#_Toc104033685)

[4 Handling Checked Exceptions 12](#_Toc104033686)

[5 Closures in Lambda Expressions 13](#_Toc104033687)

[5.1 this reference in Lambdas 14](#_Toc104033688)

[6 Method References 14](#_Toc104033689)

[6.1 Handling Exceptions 16](#_Toc104033690)

[6.2 Parallel Streams 16](#_Toc104033691)

[6.2.1 Performance Implications for Parallel Streams 17](#_Toc104033692)

[6.2.2 Summarized, when to use Parallel Streams? 18](#_Toc104033693)

[6.3 Stream.reduce() 19](#_Toc104033694)

[6.3.1 Identity, Accumulator and Combiner 19](#_Toc104033695)

[6.3.2 Reducing in Parallel 21](#_Toc104033696)

[6.3.3 Complex Custom Objects 21](#_Toc104033697)

1. Java Lambda Expressions

A 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.
  1. 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 must 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.

Text

Description automatically generated with low confidence

Let’s 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.

Text

Description automatically generated with medium confidence

In **Java 7 (without streams)** we can do it like this:

Text

Description automatically generated with low confidenceThis 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.

* 1. Functions as Values

Chart

Description automatically generatedInline 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.

* Data (string literals or numbers) are treated as values
* Objects are treated as values
* In Java 8, functions now can also be 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.

Text

Description automatically generated with medium confidence

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

Text

Description automatically generated with low confidenceThis 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.

Text

Description automatically generated with low confidenceWhen 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**.

A picture containing logo

Description automatically generated

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.

A picture containing text

Description automatically generated

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.



* 1. 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**.

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.

Text

Description automatically generated with medium confidence

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.

Text

Description automatically generatedText

Description automatically generated with medium confidence

* 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**.

Text

Description automatically generated with medium confidence





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 expression has 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.
* Text

  Description automatically generatedCompiler 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.
  1. 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

Text

Description automatically generated with medium confidenceAnother 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)**

A picture containing text, indoor, screenshot

Description automatically generatedA picture containing shape

Description automatically generatedA picture containing text, orange, dark

Description automatically generated



* 1. Text

     Description automatically generated with medium confidenceType Inference

The compiler does something called type inference.

If you only have one line 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.

Text

Description automatically generated with low confidence

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.

Graphical user interface, text

Description automatically generated (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.



* 1. 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.

Text

Description automatically generated



Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

Text

Description automatically generated



Graphical user interface

Description automatically generated with low confidence

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

A screenshot of a social media post

Description automatically generated

A screenshot of a cell phone

Description automatically generated

1. 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.

Table

Description automatically generatedJava 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.

A picture containing text

Description automatically generatedTable

Description automatically generatedA picture containing text

Description automatically generatedThe package name is **java.util.function**.

Table

Description automatically generated

Text

Description automatically generatedText

Description automatically generated

-> 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.

Graphical user interface, text

Description automatically generatedPredicate: Yüklem, Doğrulamak, beyan etmek, dayandırmak

Text, letter

Description automatically generated

Text

Description automatically generated

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.

A screenshot of a computer

Description automatically generated with medium confidence

A screenshot of a computer

Description automatically generated with medium confidence

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

1. Exception Handling for Lambdas

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generatedThe 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.

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.

* 1. 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: 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 generatedThe 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.

A screenshot of a cell phone

Description automatically generatedLet'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 picture containing knife

Description automatically generatedFinally, 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/>

* See also (from the sneaky-throw blog post): <https://github.com/pivovarit/throwing-function>

1. Closures in Lambda Expressions

A screenshot of a cell phone

Description automatically generatedWe 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 the statement   
i + b.

The variable 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**.

A screenshot of a cell phone

Description automatically generatedSince 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.



* 1. 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 cell phone

Description automatically generatedA screenshot of a social media post

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

1. 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.

**TODO**: this is interesting to read: <https://stackoverflow.com/questions/50748534/are-java-method-references-stable>

A screenshot of a cell phone

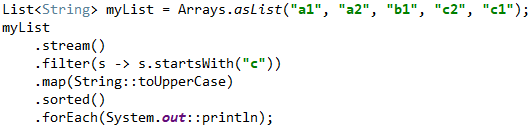
Description automatically generated

Think of method references as syntactic sugar for lambda expressions. If the lambda expression parameters and return value matches with a method, the method can be written directly, instead of the lambda expression.

Normally with LE, you would write the left one. The left and right are the same, just different ways to write it:



1. Streams

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 modifies 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.

Text

Description automatically generated

Text

Description automatically generated

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 first, then forEach, only then the second string "a2" is processed**.

**TODO** Learn More about

* anyMatch
* collectors.joining(delimeter)

**TODO**

learn more about how the results are wrapped and passed to the following stream functions.

It is similar to a promise chain in JS. Does the response of .map wrapped into an Optional?

What if the response is null?

**TODO**

* Best practices when using lambdas/streams? Try to keep it a one liner??
* I’ve already wrote about exception handling but read more about the best practices

**TODO** using Function<T, R>- <https://stackoverflow.com/questions/29945627/java-8-lambda-void-argument/29946155>

Table

Description automatically generated with medium confidenceLearn these:

Asd

* 1. Stream Examples

Text

Description automatically generated with medium confidence

asd

* 1. IntStream, DoubleStream, LongStream

**Why just** IntStream / DoubleStream / LongStream**?**

Because they are the most commonly used types.

**What is the difference between both?**

* IntStream is a stream of *primitive* int values.
* Stream<Integer> is a stream of Integer *objects*.

The list of operations available are different. E.g. IntStream has built-in concepts of **range(...)** and **sum()**, not that **sum()** is difficult to implement with Stream<Integer> using **reduce()**, but it isn't built in.

**How does the difference relate to performance?**

Boxing and unboxing does take some time, but it's not a lot. A lot of temporary boxed objects also triggers Garbage Collection a lot more often, and that's a performance drain too. It all adds up, so if the stream processes a lot of integer values in a tight "loop", the difference can be relevant.

The bigger problem is space, since the overhead of Integer is quite large. An int is **4 bytes** for the value, while an Integer is **4 bytes** for the reference plus **16 bytes** for the object, so Integer uses **20 bytes** per value, i.e. 5 times the memory.

This is especially relevant if you call toArray(), since there's a big difference between an int[] and an Integer[], space-wise.

**Some Common Ways to Use Them**

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

* 1. Terminate Operations
     1. Collect

Doc: <https://www.baeldung.com/java-8-collectors>

Graphical user interface, text, application

Description automatically generated

**TODO**: guarantees didn’t understand?

Collectors.toList/toSet

Returns a Collector that accumulates the input elements into a new List. **There are no guarantees** on the **type**, **mutability**, **serializability**, or **thread-safety** of the List returned; if more control over the returned List is required, use toCollection(Supplier).

* .collect(Collectors.*toCollection*(ArrayList::new));

Collectors.toUnmodifiableList/Set()

* Returns a Collector that accumulates the input elements into an unmodifiable List in encounter order. The returned Collector **disallows null values** and will **throw** NullPointerException if it is presented with a null value.

Collectors.toMap()

* **There are no guarantees** on the **type**, **mutability**, **serializability**, or **thread-safety** of the List returned.

The *toMap* collector can be used to collect *Stream* elements into a *Map* instance. To do this, we need to provide two functions:

* Graphical user interface, text, application

  Description automatically generatedkeyMapper (java.util.function.Function)
* valueMapper (java.util.function.Function)

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Collectors.joining()

*Joining* collector can be used for joining Stream<String> elements.   
We can also specify custom:

* separators
* prefixes
* postfixes.

Text

Description automatically generated

Collectors.summarizingDouble/Long/Int()

SummarizingDouble/Long/Int is a collector that returns a special class containing statistical information about numerical data in a Stream of extracted elements.

Collectors.groupingBy()

GroupingBy collector is used for grouping objects by some property, and then storing the results in a Map instance.

**TODO**

**Custom Collectors**

**TODO**: https://www.baeldung.com/java-8-collectors#Custom

* 1. Parallel Streams

Java 8 introduced the Stream API that makes it easy to iterate over collections as streams of data. It's also very easy to create streams that execute in parallel and make use of multiple processor cores. We might think that it's always faster to divide the work on more cores. But that is often not the case.

* **By default**, any stream operation in Java is processed **sequentially**, unless explicitly specified as parallel.
* The output of a sequential stream is predictable. For example printing: the list elements will always be printed in an ordered sequence

To make a stream parallel, add theparallel() orparallelStream() method.

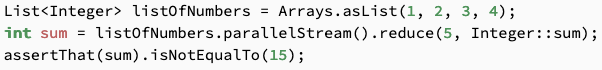
* the order of execution is out of our control. It may change every time we run the program:
* Parallel streams make use of the **fork-join** framework and its common pool of worker threads.

When we use parallelized streams, keep in mind that the operations should be:

* **associative**: the result is not affected by the order of the operands
* **non-interfering**: the operation doesn't affect the data source
* **stateless and deterministic**: the operation doesn't have state and produces the same output for a given input

Be careful about which operations can be run in parallel.   
For example, reduce() with identity **should not be used** in parallel streams.

A picture containing text, clock

Description automatically generated

In a sequential stream, the result of this operation would be 15. But since the reduce operation is handled in parallel, the number five actually gets added up in every worker thread:

* The actual result might differ depending on the number of threads used in the common fork-join pool.
* In order to fix this issue, the number five should be added outside of the parallel stream
  + 1. Performance Implications for Parallel Streams

Parallel processing may be beneficial to fully utilize multiple cores. But we also need to consider:

1. the overhead of managing multiple threads
2. memory locality
3. splitting the source and merging the results.

**Splitting the source**

Splitting the data source evenly is a necessary cost to enable parallel execution, but some data sources split better than others.

From “*Effective Java, Joshua Bloch*”: As a rule, performance gains from parallelism are best on streams over ArrayList , HashMap , HashSet , and ConcurrentHashMap instances; **arrays**; **int ranges**; and **long ranges**. What these data structures have in common is that they can all be accurately and cheaply split into subranges of any desired sizes, which makes it easy to divide work among parallel threads. (For example, the splitting characteristics of a LinkedList is awful)

Another important factor that all of these data structures have in common is that they provide good-to-excellent locality of reference when processed sequentially: sequential element references are stored together in memory. The objects referred to by those references may not be close to one another in memory, which reduces locality-of-reference. Locality-of-reference turns out to be critically important for parallelizing bulk operations: without it, threads spend much of their time idle, waiting for data to be transferred from memory into the processor’s cache. The data structures with the best locality of reference are primitive arrays because the data itself is stored contiguously in memory.

**Cache Miss**

Modern computer systems employ sophisticated multilevel caches to keep frequently used data as close (literally — speed of light is a limiting factor!) to the CPU as possible. Fetching data from L1 cache can easily be 100 times faster than fetching data from main memory. The more efficiently the CPU can predict which data will be needed next, the more cycles the CPU spends doing computation, and the fewer it spends waiting for data.

When a linear memory access pattern is detected, the hardware prefetches the next line of data from the main memory into the much faster cache, under the assumption that it will probably be needed soon. But if that data in the cache does not contain the required data, it is considered a cache miss. Now the processor has to fetch the correct data from the memory, which takes time, and the processor, or at least the thread of execution, has to wait (stall) until the instruction is fetched from main memory

**The NQ Model**

Oracle presented a simple model that can help us determine whether parallelism can offer us a performance boost. In the NQ model, **N** stands for **the number of source data elements**, while **Q** represents **the amount of computation performed per data element**.

The larger the product of **N\*Q**, the more likely we are to get a performance boost from parallelization. For problems with a trivially small Q, such as summing up numbers, the rule of thumb is that N should be greater than **10,000**. As the number of computations increases, the data size required to get a performance boost from parallelism decreases.

**How many threads are spawned in parallelStream in Java 8?**

This is probably dependent on the JVM implementation.

The Oracle's implementation of parallel stream uses the current thread and in addition to that, if needed, also the threads that compose the default fork join pool ForkJoinPool.commonPool(), which has a default size equal to one less than the number of cores of your CPU.

* + 1. Summarized, when to use Parallel Streams?

In reality, sometimes parallelism will speed up your computation, sometimes it will not, and sometimes it will even slow it down. It is best to develop first using sequential execution and then apply parallelism where:

* **(A)** you know that there's actually benefit to increased performance: This is a business problem, not a technical one. First make sure that you are absolutely sure that this will help you for sure and then look at **(B)**
* **(B)** that it will actually deliver increased performance: It is best to not guess but measure it if the technical changes really make things faster.

Use sequential streams by default and only consider parallel ones if

* You have a massive amount of items to process (or the processing of each item takes time and is parallelizable)
* You have a performance problem in the first place
* You don't already run the process in a multi-thread environment (for example: in a web container, if I already have many requests to process in parallel, adding an additional layer of parallelism inside each request could have more negative than positive effects)

Moreover, remember that parallel streams don't magically solve all the synchronization problems. If a shared resource is used by the predicates and functions used in the process, you'll have to make sure that everything is thread-safe. In particular, **side effects are things you really have to worry about if you go parallel**.

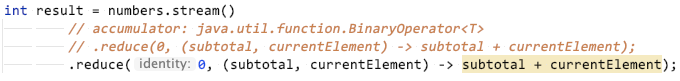
In any case, measure, don't guess! Only a measurement will tell you if the parallelism is worth it or not.

* 1. Stream.reduce()
     1. Identity, Accumulator and Combiner

The **Stream API** provides a rich repertoire of intermediate, reduction and terminal functions, which also support parallelization.

More specifically, reduction stream operations **allow us to produce one single result from a sequence of elements**, by repeatedly applying a combining operation to the elements in the sequence.

Before we look deeper into using the *Stream.reduce()* operation, let's break down the operation's participant elements into separate blocks. That way, we'll understand more easily the role that each one plays.

* ***Identity*** – an element that is the initial value of the reduction operation and the default result if the stream is empty
* ***Accumulator*** – a function that takes two parameters: a partial result of the reduction operation and the next element of the stream
* Text

  Description automatically generated with low confidence***Combiner*** – a function used to combine the partial result of the reduction operation when the reduction is parallelized or when there's a mismatch between the types of the accumulator arguments and the types of the accumulator implementation

The return value of the reduce will be passed as the subtotal for the next reduce operation.



Can be replaced with a method reference:

Chart

Description automatically generated

Of course, reduce can also be used with other types:

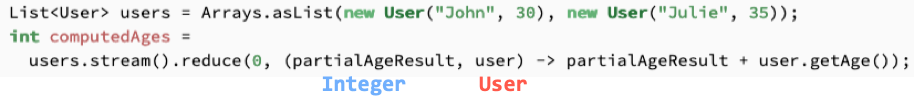


When a **stream executes in parallel**, the Java runtime splits the stream into multiple substreams. In such cases, we need to use a function to combine the results of the substreams into a single one. This is the role of the combiner — in the above snippet, it's the Integer::sum method reference.





Funnily enough, this code won't compile:



In this case, we have a stream of User objects, and the types of the accumulator arguments are Integer and User. However, the accumulator implementation is a sum of Integers, so the compiler just can't infer the type of the user parameter.

We can **fix this** issue by using a **combiner**:







**TODO me no understand – how does this work?**

To put it simply, if we use sequential streams and the types of the accumulator arguments and the types of its implementation match, we don't need to use a combiner.

* + 1. Reducing in Parallel

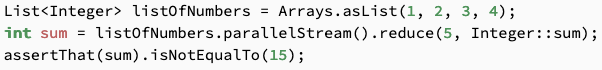
We can use reduce() on parallelized streams.

When we use parallelized streams, we should make sure that reduce() or any other aggregate operations executed on the streams are:

* **associative**: the result is not affected by the order of the operands
* **non-interfering**: the operation doesn't affect the data source
* **stateless and deterministic**: the operation doesn't have state and produces the same output for a given input

We should fulfill all these conditions to prevent unpredictable results. But even if we make sure these conditions apply, some operations might still give us wrong results with parallel streams. For example, reduce with identity **should not be used** in parallel streams:

A picture containing text, clock

Description automatically generated

In a sequential stream, the result of this operation would be 15. But since the reduce operation is handled in parallel, the number five actually gets added up in every worker thread:

* The actual result might differ depending on the number of threads used in the common fork-join pool.
* In order to fix this issue, the number five should be added outside of the parallel stream
  + 1. Complex Custom Objects

**TODO**

https://www.baeldung.com/java-stream-reduce#complex-custom-objects

Asd

**TODO**

* LongStream, IntegerStream etc.
* **.sum(), .count(), .average(), anyMatch()** and their reduce equivalents
* What else can be done with LongStream?
* Also cool: <https://www.deadcoderising.com/2015-05-19-java-8-replace-traditional-for-loops-with-intstreams/>

**TODO**: Sorting streams

#10: Sorting a stream would work just on Comparable objects - If you are trying to sort a stream of objects, a RuntimeException will be thrown if the targeted objects are not Comparable objects, i.e. implementing the compareTo(...) method.

**TODO**: learn more about parallel streams

**Continue**: 4. Reducing in Parallel

<https://www.baeldung.com/java-stream-reduce#Reducing-in-parallel>