In Java 8, functional interfaces, lambdas, and method references were added to make it easier to create function objects. The streams API was added in tandem with these language changes to provide library support for processing sequences of data elements.

**Prefer lambdas to anonymous classes**

A black text on a white background

Description automatically generated with low confidence

Anonymous classes were adequate for the classic objected-oriented design patterns requiring function objects, notably the Strategy pattern [Gamma95]. The Comparator interface represents an abstract strategy for sorting; the anonymous class above is a concrete strategy for sorting strings. The verbosity of anonymous classes, however, made functional programming in Java an unappealing prospect.

In Java 8, the language formalized the notion that interfaces with a single abstract method are special and deserve special treatment. These interfaces are now known as functional interfaces, and the language allows you to create instances of these interfaces using lambda expressions, or lambdas for short.

Lambdas are similar in function to anonymous classes, but far more concise. Here’s how the code snippet above looks with the anonymous class replaced by a lambda. The boilerplate is gone, and the behavior is clearly evident:

A picture containing text, font, screenshot, line

Description automatically generated

Note that the types of the lambda (Comparator<String>), of its parameters (s1 and s2, both String), and of its return value (int) are not present in the code. The compiler deduces these types from context, using a process known as type inference. In some cases, the compiler won’t be able to determine the types, and you’ll have to specify them. The rules for type inference are complex: they take up an entire chapter in the JLS [JLS, 18].

A picture containing text, screenshot, font, information

Description automatically generated

You might think that anonymous classes are obsolete in the era of lambdas. This is closer to the truth, but there are a few things you can do with anonymous classes that you can’t do with lambdas. Lambdas are limited to functional interfaces. If you want to create an instance of an abstract class, you can do it with an anonymous class, but not a lambda. Similarly, you can use anonymous classes to create instances of interfaces with multiple abstract methods. Finally, a lambda cannot obtain a reference to itself. In a lambda, the this keyword refers to the enclosing instance, which is typically what you want. In an anonymous class, the this keyword refers to the anonymous class instance. If you need access to the function object from within its body, then you must use an anonymous class.Lambdas share with anonymous classes the property that you can’t reliably serialize and deserialize them across implementations. Therefore, you should rarely, if ever, serialize a lambda (or an anonymous class instance).

In summary, as of Java 8, lambdas are by far the best way to represent small function objects. Don’t use anonymous classes for function objects unless you have to create instances of types that aren’t functional interfaces. Also, remember that lambdas make it so easy to represent small function objects that it opens the door to functional programming techniques that were not previously practical in Java.

**Prefer method references to lambdas**

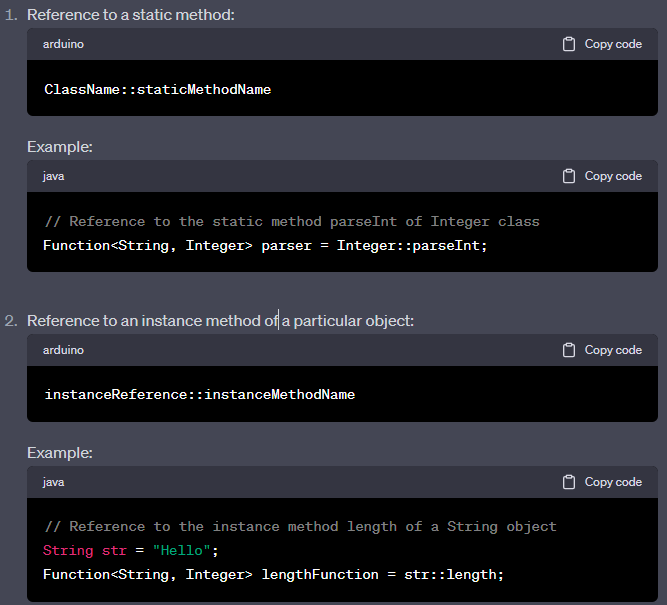
In Java, a method reference is a concise way to refer to an existing method or a constructor using the :: operator. It allows you to treat a method as a first-class object, similar to lambda expressions. There are 4 types of method references in Java, code snippets can be found below.(Next page)

Many method references refer to static methods, but there are four kinds that do not. Two of them are bound and unbound instance method references. In bound references, the receiving object is specified in the method reference. Bound references are similar in nature to static references: the function object takes the same arguments as the referenced method. In unbound references, the receiving object is specified when the function object is applied, via an additional parameter before the method’s declared parameters. Unbound references are often used as mapping and filter functions in stream pipelines (Item 45). Finally, there are two kinds of constructor references, for classes and arrays. Constructor references

serve as factory objects. All five kinds of method references are summarized in the table below:

A screenshot of a computer program

Description automatically generated with low confidence



A screenshot of a computer program

Description automatically generated with medium confidence

A screenshot of a computer program

Description automatically generated with medium confidence

Method references often provide a more succinct alternative to lambdas. Where method references are shorter and clearer, use them; where they aren’t, stick with lambdas.

**Favor the use of standard functional interfaces**

If one of the standard functional interfaces does the job, you should generally use it in preference to a purpose-built functional interface. This will make your API easier to learn, by reducing its conceptual surface area, and will provide significant interoperability benefits, as many of the standard functional interfaces provide useful default methods

There are forty-three interfaces in java.util.Function. You can’t be expected to remember them all, but if you remember six basic interfaces, you can derive the rest when you need them. The basic interfaces operate on object reference types. The Operator interfaces represent functions whose result and argument types are the same. The Predicate interface represents a function that takes an argument and returns a boolean. The Function interface represents a function whose argument and return types differ. The Supplier interface represents a function that takes no arguments and returns (or “supplies”) a value. Finally, Consumer represents a function that takes an argument and returns nothing, essentially consuming its argument. The six basic functional interfaces are summarized below:

A picture containing text, screenshot, font, number

Description automatically generated

In summary, now that Java has lambdas, it is imperative that you design your APIs with lambdas in mind. Accept functional interface types on input and return them on output. It is generally best to use the standard interfaces provided in java.util.function.Function, but keep your eyes open for the relatively rare cases where you would be better off writing your own functional interface.

**Use streams judiciously**

The streams API was added in Java 8 to ease the task of performing bulk operations, sequentially or in parallel. This API provides two key abstractions: the stream, which represents a finite or infinite sequence of data elements, and the stream pipeline, which represents a multistage computation on these elements. The elements in a stream can come from anywhere. Common sources include collections, arrays, files, regular expression pattern matchers, pseudorandom number generators, and other streams. The data elements in a stream can be object references or primitive values. Three primitive types are supported: int, long, and double.

A stream pipeline consists of a source stream followed by zero or more intermediate operations and one terminal operation. Each intermediate operation transforms the stream in some way, such as mapping each element to a function of that element or filtering out all elements that do not satisfy some condition. Intermediate operations all transform one stream into another, whose element type may be the same as the input stream or different from it. The terminal operation performs a final computation on the stream resulting from the last intermediate operation, such as storing its elements into a collection, returning a certain element, or printing all of its elements.

Stream pipelines are evaluated lazily: evaluation doesn’t start until the terminal operation is invoked, and data elements that aren’t required in order to complete the terminal operation are never computed. This lazy evaluation is what makes it possible to work with infinite streams. Note that a stream pipeline without a terminal operation is a silent no-op, so don’t forget to include one. By default, stream pipelines run sequentially. Making a pipeline execute in parallel is as simple as invoking the parallel method on any stream in the pipeline,

but it is seldom appropriate to do so (Item 48).

The streams API is sufficiently versatile that practically any computation can be performed using streams, but just because you can doesn’t mean you should. When used appropriately, streams can make programs shorter and clearer; when used inappropriately, they can make programs difficult to read and maintain. There are no hard and fast rules for when to use streams, but there are heuristics.

Using helper methods is even more important for readability in stream pipelines than in iterative code because pipelines lack explicit type information and named temporary variables. When you start using streams, you may feel the urge to convert all your loops into streams, but resist the urge. While it may be possible, it will likely harm the readability and maintainability of your code base. As a rule, even moderately complex tasks are best accomplished using some combination of streams and iteration, as illustrated by the Anagrams programs above. So refactor existing code to use streams and use them in new code only where it makes sense to do so.

If a computation is best expressed using these techniques, then it’s probably not a

good match for streams.

From a code block, you can read or modify any local variable in scope; from a lambda, you can only read final or effectively final variables [JLS 4.12.4], and you can’t modify any local variables.

From a code block, you can return from the enclosing method, break or

continue an enclosing loop, or throw any checked exception that this method is declared to throw; from a lambda you can do none of these things.

If a computation is best expressed using these techniques, then it is a good candidate

for streams.

Uniformly transform sequences of elements

Filter sequences of elements

Combine sequences of elements using a single operation (for example to add them, concatenate them, or compute their minimum)

Accumulate sequences of elements into a collection, perhaps grouping them by some common attribute

Search a sequence of elements for an element satisfying some criterion

In summary, some tasks are best accomplished with streams, and others with iteration. Many tasks are best accomplished by combining the two approaches. There are no hard and fast rules for choosing which approach to use for a task, but there are some useful heuristics. In many cases, it will be clear which approach to use; in some cases, it won’t. If you’re not sure whether a task is better served by streams or iteration, try both and see which works better.

**Prefer side-effect-free functions in streams**

In order to obtain the expressiveness, speed, and in some cases parallelizability that streams have to offer, you have to adopt the paradigm as well as the API. The most important part of the streams paradigm is to structure your computation as a sequence of transformations where the result of each stage is as close as possible to a pure function of the result of the previous stage. A pure function is one whose result depends only on its input: it does not depend on any mutable state, nor does it update any state. In order to achieve this, any function objects that you pass into stream operations, both intermediate and terminal, should be free of side-effects.

A picture containing text, font, screenshot, line

Description automatically generated

A forEach operation that does anything more than present the result of the computation performed by a stream is a “bad smell in code,” as is a lambda that mutates state.The forEach operation should be used only to report the result of a stream computation, not to perform the computation. Occasionally, it makes sense to use forEach for some other purpose, such as adding the results of a stream computation to a preexisting collection.

The improved code uses a collector, which is a new concept that you have to learn in order to use streams. The Collectors API is intimidating: it has thirtynine methods, some of which have as many as five type parameters. The good news is that you can derive most of the benefit from this API without delving into its full complexity. For starters, you can ignore the Collector interface and think of a collector as an opaque object that encapsulates a reduction strategy. In this context, reduction means combining the elements of a stream into a single object. The object produced by a collector is typically a collection (which accounts for the name collector).

The collectors for gathering the elements of a stream into a true Collection are straightforward. There are three such collectors: toList(), toSet(), and toCollection(collectionFactory). They return, respectively, a set, a list, and a programmer-specified collection type. Armed with this knowledge, we can write a stream pipeline to extract a top-ten list from our frequency table. Note that we haven’t qualified the toList method with its class, Collectors. It is customary and wise to statically import all members of Collectors because it makes stream pipelines more readable.

A picture containing text, font, screenshot, line

Description automatically generated

So what about the other thirty-six methods in Collectors? Most of them exist to let you collect streams into maps, which is far more complicated than collecting them into true collections. Each stream element is associated with a key and a value, and multiple stream elements can be associated with the same key.

The simplest map collector is toMap(keyMapper, valueMapper), which takes two functions, one of which maps a stream element to a key, the other, to a value. We used this collector in our fromString implementation in Item 34 to make a map from the string form of an enum to the enum itself:

A picture containing text, font, screenshot, white

Description automatically generated

This simple form of toMap is perfect if each element in the stream maps to a unique key. If multiple stream elements map to the same key, the pipeline will terminate with an IllegalStateException.

The three-argument form of toMap is also useful to make a map from a key to a chosen element associated with that key. For example, suppose we have a stream of record albums by various artists, and we want a map from recording artist to best-selling album. This collector will do the job.

A black text on a white background

Description automatically generated with low confidence

Note that the comparator uses the static factory method maxBy, which is statically imported from BinaryOperator. This method converts a Comparator<T> into a BinaryOperator<T> that computes the maximum implied by the specified comparator. In this case, the comparator is returned by the comparator construction method comparing, which takes the key extractor function Album::sales. This may seem a bit convoluted, but the code reads nicely. Loosely speaking, it says, “convert the stream of albums to a map, mapping each artist to the album that has the best album by sales.” This is surprisingly close to the problem statement.

Another use of the three-argument form of toMap is to produce a collector that imposes a last-write-wins policy when there are collisions. For many streams, the results will be nondeterministic, but if all the values that may be associated with a key by the mapping functions are identical, or if they are all acceptable, this collector’s s behavior may be just what you want:

A picture containing text, font, white, line

Description automatically generated

The third and final version of toMap takes a fourth argument, which is a map factory, for use when you want to specify a particular map implementation such as an EnumMap or a TreeMap. There are also variant forms of the first three versions of toMap, named toConcurrentMap, that run efficiently in parallel and produce ConcurrentHash- Map instances.

In addition to the toMap method, the Collectors API provides the groupingBy method, which returns collectors to produce maps that group elements into categories based on a classifier function. The classifier function takes an element and returns the category into which it falls. This category serves as the element’s map key. The simplest version of the groupingBy method takes only a classifier and returns a map whose values are lists of all the elements in each category. This is the collector that we used in the Anagram program in Item 45 to generate a map from alphabetized word to a list of the words sharing the alphabetization:

words.collect(groupingBy(word -> alphabetize(word)))

The final Collectors method is joining, which operates only on streams of CharSequence instances such as strings. In its parameterless form, it returns a collector that simply concatenates the elements. Its one argument form takes a single CharSequence parameter named delimiter and returns a collector that joins the stream elements, inserting the delimiter between adjacent elements. If you pass in a comma as the delimiter, the collector returns a comma-separated values string (but beware that the string will be ambiguous if any of the elements in the stream contain commas). The three argument form takes a prefix and suffix in addition to the delimiter. The resulting collector generates strings like the ones that you get when you print a collection, for example [came, saw, conquered].

In summary, the essence of programming stream pipelines is side-effect-free function objects. This applies to all of the many function objects passed to streams and related objects. The terminal operation forEach should only be used to report the result of a computation performed by a stream, not to perform the computation. In order to use streams properly, you have to know about collectors. The most important collector factories are toList, toSet, toMap, groupingBy, and joining.

**Prefer Collection over Stream as return type.**

Many methods return sequences of elements. Prior to Java 8, the obvious return types for such methods were the collection interfaces Collection, Set, and List; Iterable; and the array types. Usually, it was easy to decide which of these types to return. The norm was a collection interface. If the method existed solely to enable for-each loops or the returned sequence couldn’t be made to implement some Collection method (typically, contains(Object)), the Iterable interface was used. If the returned elements were primitive values or there were stringent performance requirements, arrays were used. In Java 8, streams were added to the platform, substantially complicating the task of choosing the appropriate return type for a sequence-returning method.

when writing a method that returns a sequence of elements, remember that some of your users may want to process them as a stream while others may want to iterate over them. Try to accommodate both groups. If it’s feasible to return a collection, do so. If you already have the elements in a collection or the number of elements in the sequence is small enough to justify creating a new one, return a standard collection such as ArrayList. Otherwise, consider implementing a custom collection as we did for the power set. If it isn’t feasible to return a collection, return a stream or iterable, whichever seems more natural. If, in a future Java release, the Stream interface declaration is modified to extend Iterable, then you should feel free to return streams because they will allow for both stream processing and iteration.

**Use caution when making streams parallel**

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. The abstraction used by the streams library to perform this task is the spliterator, which is returned by the spliterator method on Stream and Iterable

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.

The nature of a stream pipeline’s terminal operation also affects the effectiveness of parallel execution. If a significant amount of work is done in the terminal operation compared to the overall work of the pipeline and that operation is inherently sequential, then parallelizing the pipeline will have limited effectiveness. The best terminal operations for parallelism are reductions, where all of the elements emerging from the pipeline are combined using one of Stream’s reduce methods, or prepackaged reductions such as min, max, count, and sum. The shortcircuiting operations anyMatch, allMatch, and noneMatch are also amenable to parallelism. The operations performed by Stream’s collect method, which are known as mutable reductions, are not good candidates for parallelism because the overhead of combining collections is costly. If you write your own Stream, Iterable, or Collection implementation and you want decent parallel performance, you must override the spliterator method and test the parallel performance of the resulting streams extensively.

A screenshot of a computer program

Description automatically generated with medium confidence

In summary, do not even attempt to parallelize a stream pipeline unless you have good reason to believe that it will preserve the correctness of the computation and increase its speed. The cost of inappropriately parallelizing a stream can be a program failure or performance disaster. If you believe that parallelism may be justified, ensure that your code remains correct when run in parallel, and do careful

performance measurements under realistic conditions. If your code remains correct and these experiments bear out your suspicion of increased performance, then and only then parallelize the stream in production code.