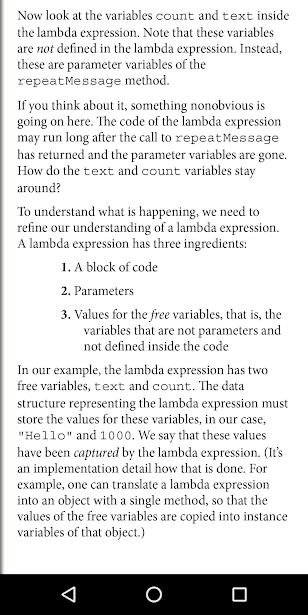
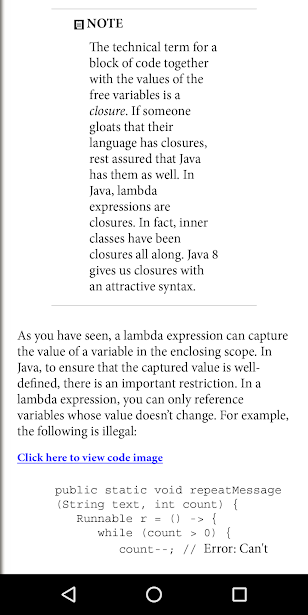
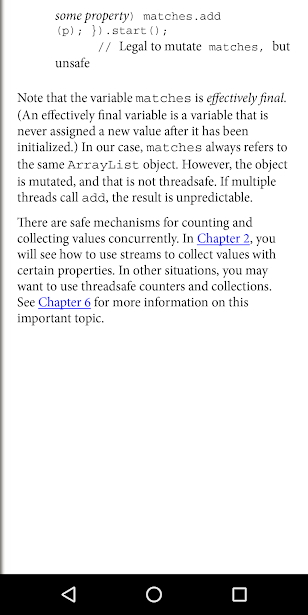
Java8

Lambda expressions are just block of code to be executed later. These are computable functions and the name lambda comes from the Greek letter ^ as used in Principa Mathematica to denote free variables.

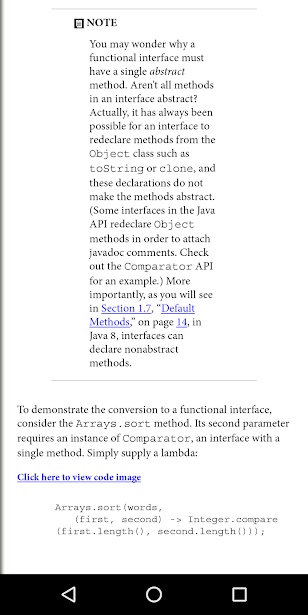


Lambdas can’t throw an Exception. Try/Catch it. Or use them for a functional interface that throws an Exception, callable instead of runnable, for e.g.





Functional interfaces are interfaces with a single abstract method. All lambdas are converted to Functional Interfaces. Not all methods on an interface need to be abstract. Methods from Object like equals and toString can be redeclared in an interface as non abstract.



MethodReferences are lambdas but already existing functions. System.out:println where one parameter is passed; Math:pow where two parameters are passed; String::compareToIgnoreCase where two parameters are passed and it’s called on the instance of first parameter and the second parameter is passed as an argument to the method call; this::equals, super::methodName, EnclosingClassInstance::this, EnclosingClassInstance::super when in inner class.

Constructor references – lines.stream(Contribution::new), compiler matches String expiry, String maturity, String strike. Int[]::new where the single argument to the lambda is the length of the array.

Array constructor references are useful to overcome a limitation of Java. It is not possible to construct an array of a generic type T. The expression new T[n] is an error since it would be erased to new Object[n].

double] dobs = new double[] {1.43,2.343,3,3.3432,3.342,4342.34,43.4324}

Double[] Dobs = Arrays.stream( dobs ).boxed().toArray( Double[]::new );

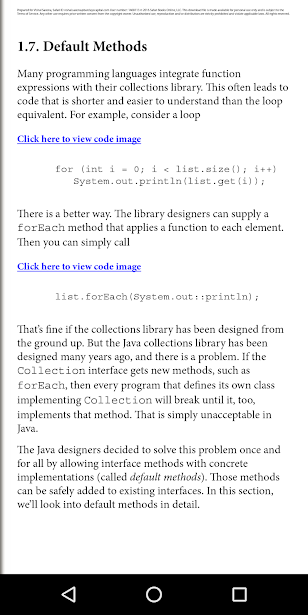
Integer[] ever = IntStream.of( data ).boxed().toArray( Integer[]::new );

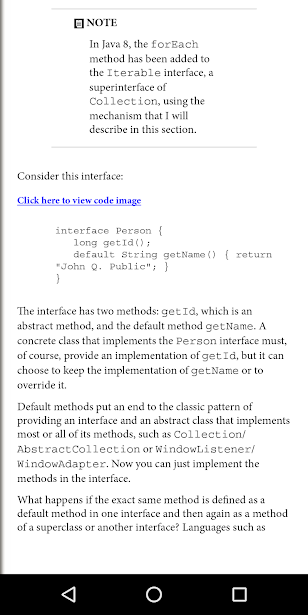
List<Integer> you = Arrays.stream( data ).boxed().collect( Collectors.toList() );

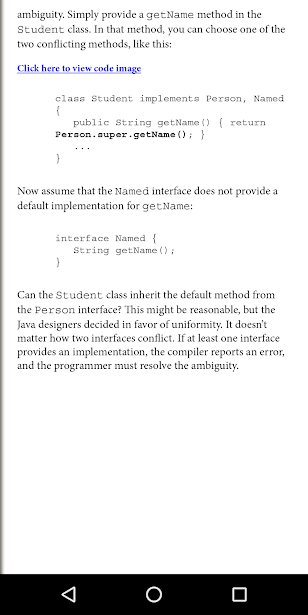
List<Integer> like = IntStream.of( data ).boxed().collect( Collectors.toList() );

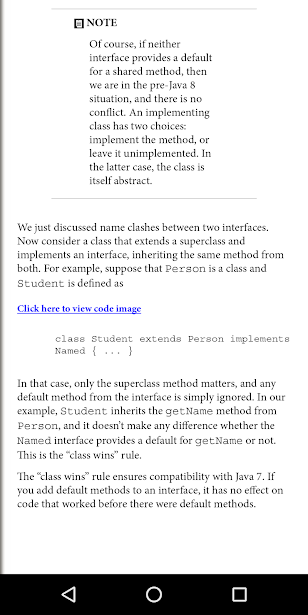
Double[] boxed = new Double[] { 1.0, 2.0, 3.0 };

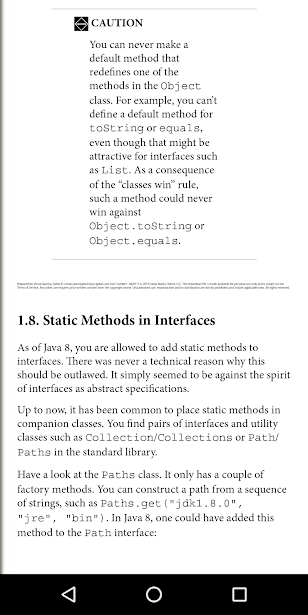
double[] unboxed = Stream.of(boxed).mapToDouble(Double::doubleValue).toArray();

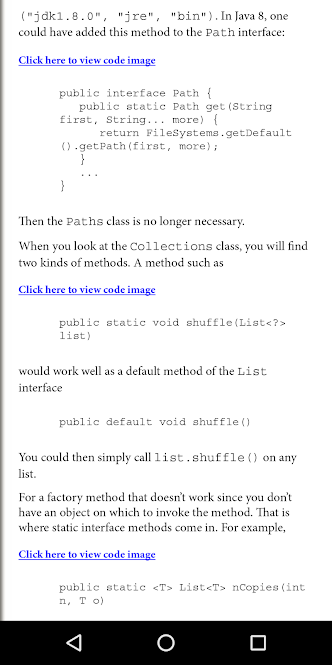


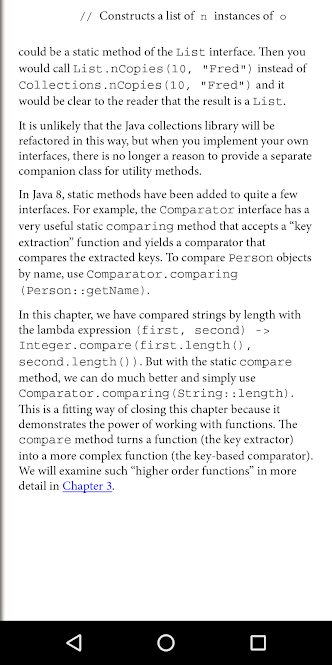


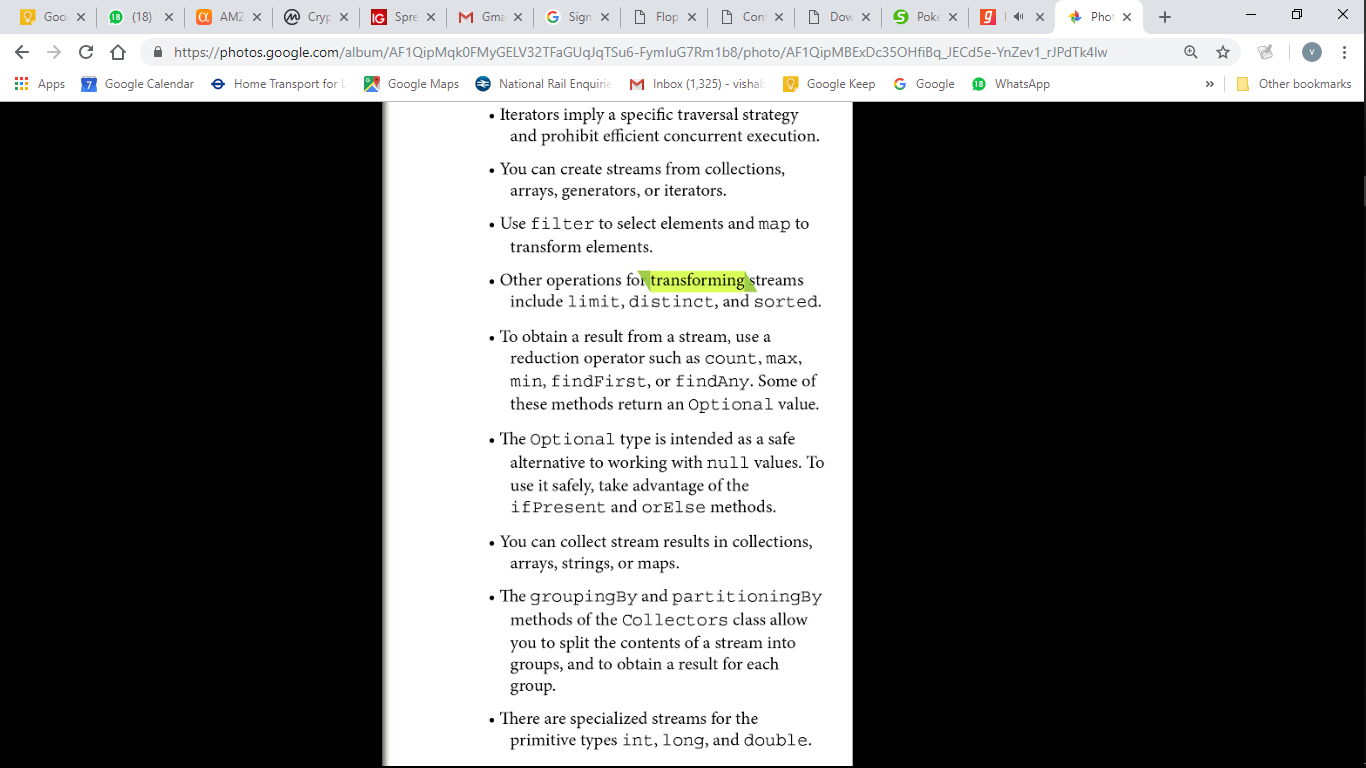


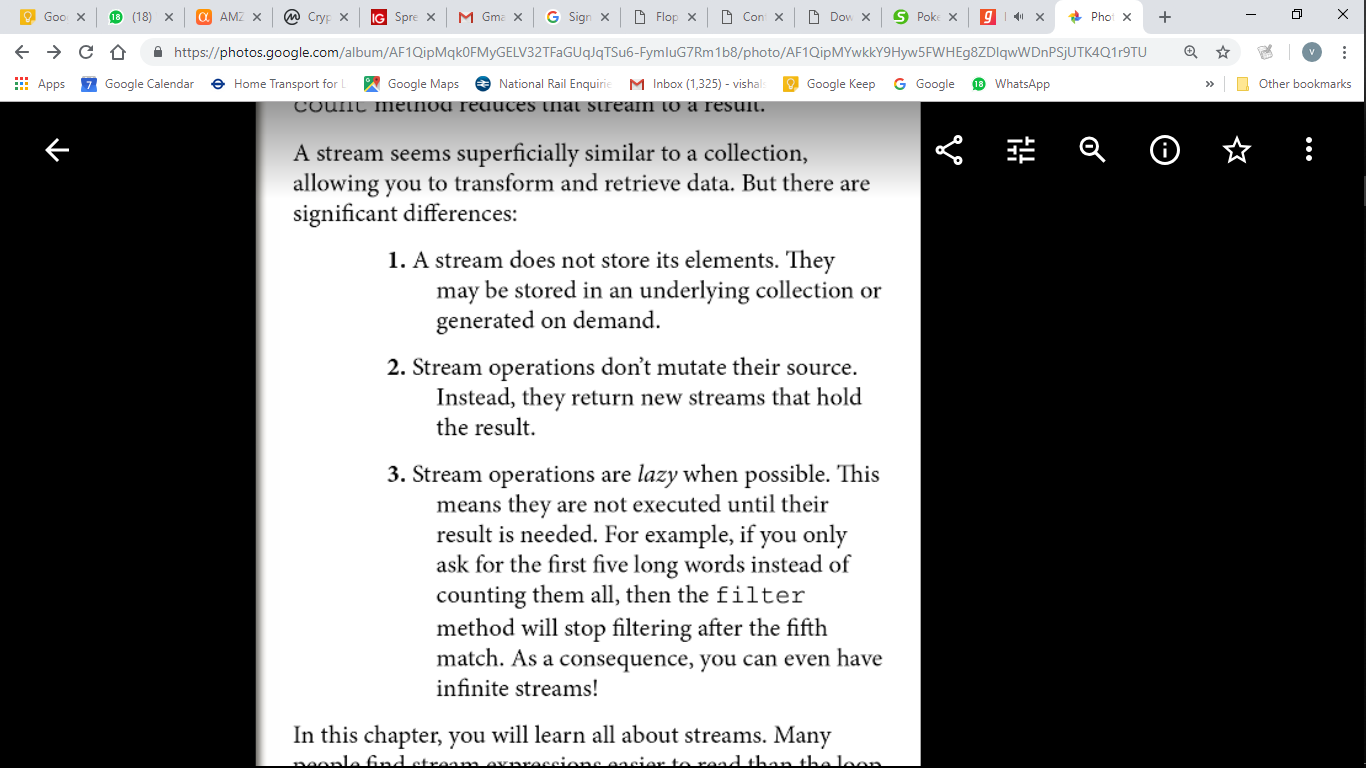


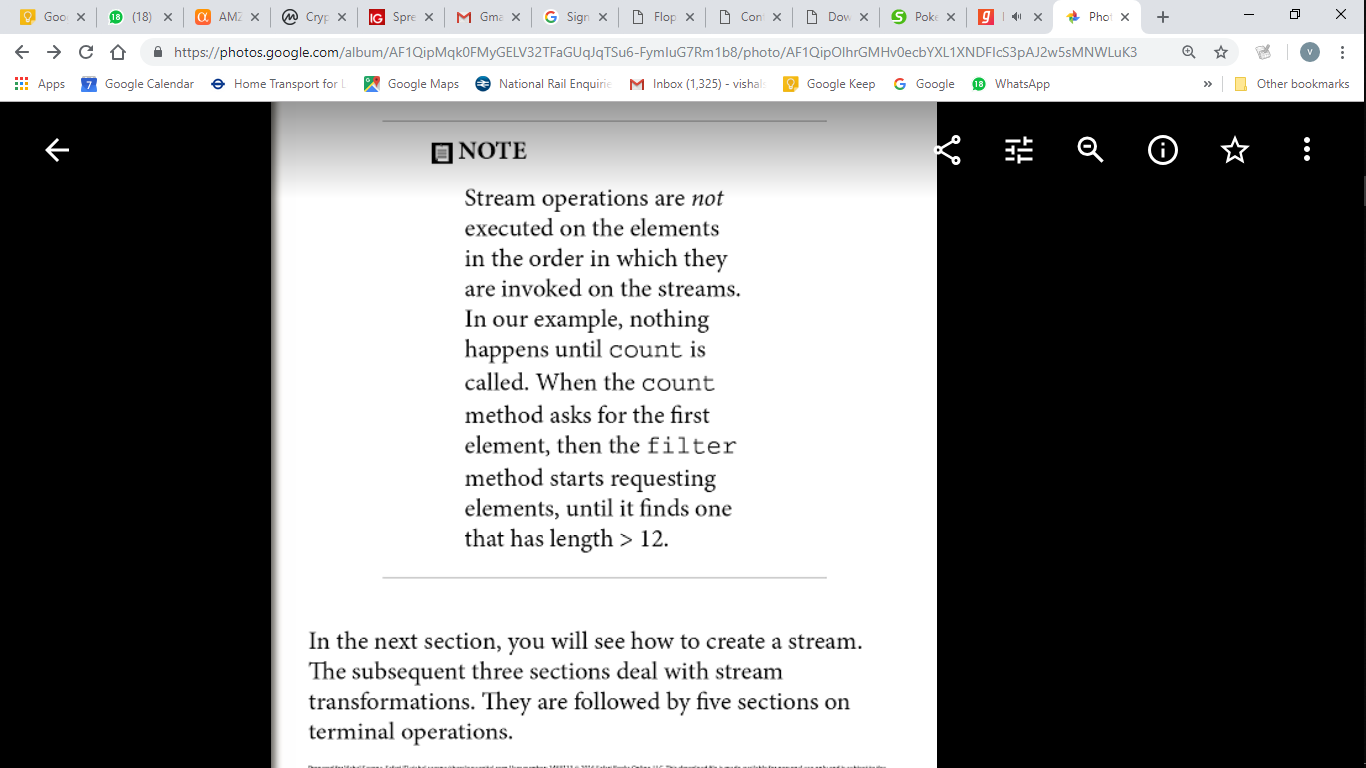


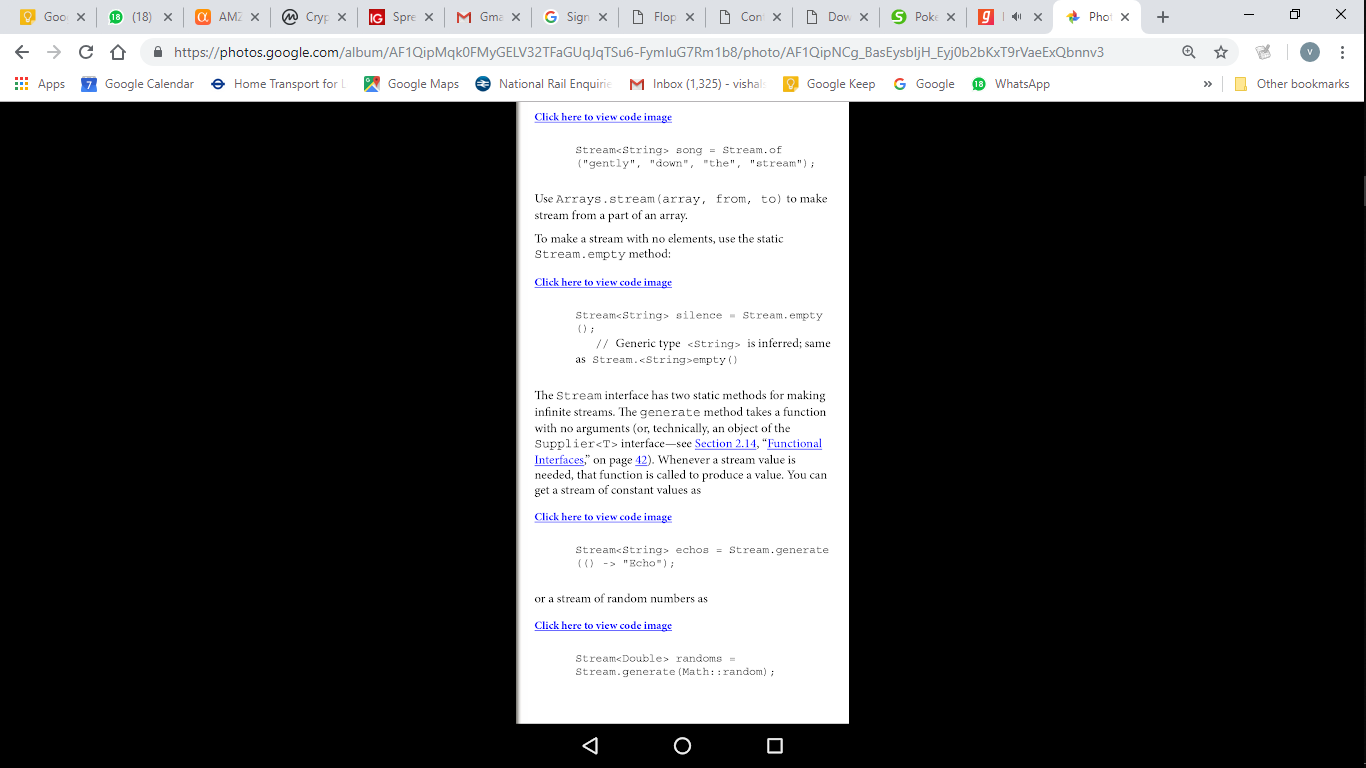


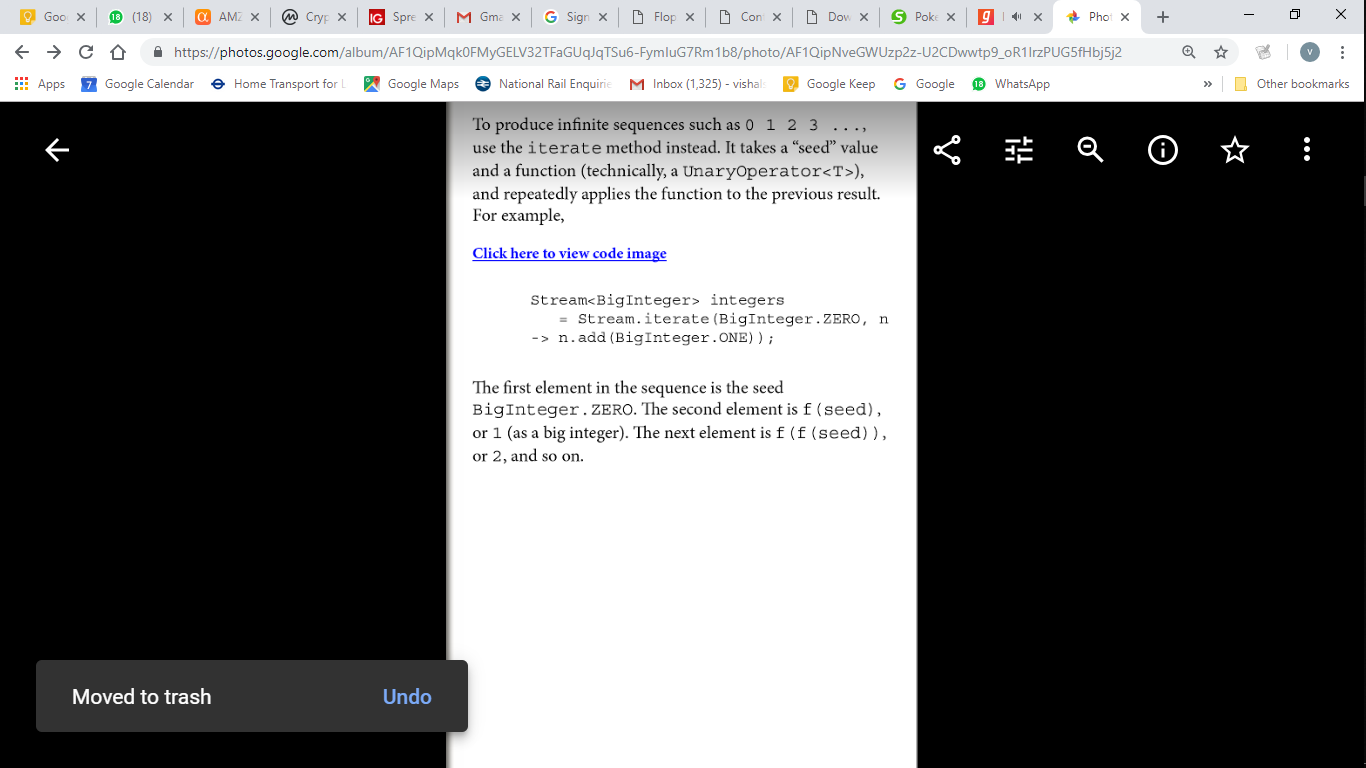




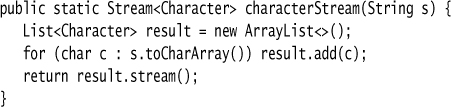








A number of methods that yield streams have been added to the API with the Java 8 release. For example, the Pattern class now has a method splitAsStream that splits a CharSequence by a regular expression. You can use the following statement to split a string into words:Stream<String> words  
   = Pattern.compile("[\\P{L}]+").splitAsStream(contents);The static Files.lines method returns a Stream of all lines in a file. The Stream interface has AutoCloseable as a superinterface. When the close method is called on the stream, the underlying file is also closed. To make sure that this happens, it is best to use the Java 7 try-with-resources statement:try (Stream<String> lines = Files.lines(path)) {  
   *Do something with* lines  
}The stream, and the underlying file with it, will be closed when the try block exits normally or through an exception.However I mostly use File.readAllLines() that doesn’t return a Stream but a list and that method takes care of closing the IO resource.When you use map, a function is applied to each element, and the return values are collected in a new stream. Now suppose that you have a function that returns a Stream







You may find a flatMap method in classes other than streams. It is a general concept in computer science. Suppose you have a generic type G (such as Stream) and functions f from some type T to G<U> and g from U to G<V>. Then you can compose them, that is, first apply f and then g, by using flatMap. This is a key idea in the theory of monads. But don’t worry—you can use flatMap without knowing anything about monads.

##### 2.4. Extracting Substreams and Combining Streams

The call stream.limit(n) returns a new stream that ends after n elements (or when the original stream ends if it is shorter). This method is particularly useful for cutting infinite streams down to size. For example,

Stream<Double> randoms = Stream.generate(Math::random).limit(100); yields a stream with 100 random numbers.The call stream.skip(n) does the exact opposite. It discards the first n elements. Stream<String> words = Stream.of(contents.split("[\\P{L}]+")).skip(1);

You can concatenate two streams with the static concat method of the Stream class:

Stream<Character> combined = Stream.concat(  
   characterStream("Hello"), characterStream("World"));  
   // Yields the stream ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']

Of course, the first stream should not be infinite—otherwise the second wouldn’t ever get a chance. The peek method yields another stream with the same elements as the original, but a function is invoked every time an element is retrievedObject[] powers = Stream.iterate(1.0, p -> p \* 2)  
   .peek(e -> System.out.println("Fetching " + e))  
   .limit(20).toArray();When an element is actually accessed, a message is printed. This way you can verify that an infinite stream is processed lazily.

##### 2.5. Stateful Transformations

The stream transformations of the preceding sections were stateless. When an element is retrieved from a filtered or mapped stream, the answer does not depend on the previous elements. There are also a few stateful transformations. For example, the distinct method returns a stream that yields elements from the original stream, in the same order, except that duplicates are suppressed. The stream must obviously remember the elements that it has already seen. Stream<String> uniqueWords  
   = Stream.of("merrily", "merrily", "merrily", "gently").distinct();  
   // Only one "merrily" is retained The sorted method must see the entire stream and sort it before it can give out any elements—after all, the smallest one might be the last one. Clearly, you can’t sort an infinite stream. Stream<String> longestFirst =  
   words.sorted(Comparator.comparing(String::length).reversed());The Collections.sort method sorts a collection in place, whereas Stream.sorted returns a new sorted stream.

##### Reductions

…are terminal operations. After a terminal operation has been applied, the stream ceases to be usable. Unlike count which always has a value, max/min/findFirst/findAny/anyMatch may not have a value if the stream is empty/filtered (count will be 0), hence to avoid NPE, Optional is returned.

Optional<String> largest = words.max(String::compareToIgnoreCase);  
if (largest.isPresent())  
   System.out.println("largest: " + largest.get());

Optional<String> startsWithQ  
   = words.filter(s -> s.startsWith("Q")).findFirst();

Optional<String> startsWithQ  
   = words.**parallel**().filter(s -> s.startsWith("Q")).**findAny**();

boolean aWordStartsWithQ  
   = words.**parallel**().**anyMatch**(s -> s.startsWith("Q"));

There are also methods allMatch and noneMatch that return true if all or no elements match a predicate. These methods always examine the entire stream, but they still benefit from being run in parallel.

##### 2.7. The Optional Type

An Optional<T> object is either a wrapper for an object of type T or for no object. It is intended as a safer allternative than a reference of type T that refers to an object or null. But it is only safer if you use it right.The get method gets the wrapped element if it exists, or throws a NoSuchElementException if it doesn’t. Therefore, if (optionalValue.isPresent()) optionalValue.get().someMethod();

is no easier than if (value != null) value.someMethod(); The key to using Optional effectively is to use a method that either consumes the correct value or produces an alternative.

Besides the isPresent method, there is an ifPresent method that accepts a function. If the optional value exists, it is passed to that function. Otherwise, nothing happens. Instead of using an if statement, you call

optionalValue.ifPresent(v -> *Process* v);

optionalValue.ifPresent(v -> results.add(v));

optionalValue.ifPresent(results::add);

When calling this version of ifPresent, no value is returned. If you want to process the result, use map instead:

Optional<Boolean> added = optionalValue.map(results::add);

Now added has one of three values: true or false wrapped into an Optional, if optionalValue was present, or an empty optional otherwise.

String result = optionalString.orElse("");

// The wrapped string, or "" if none

You can also invoke code to compute the default,

String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));

// The function is only called when needed

Or, if you want to throw another exception if there is no value,

String result = optionalString.orElseThrow(NoSuchElementException::new);

// Supply a method that yields an exception object

To create Optional

public static Optional<Double> inverse(Double x) {

return x == 0 ? Optional.empty() : Optional.of(1 / x);

}

The ofNullable method is intended as a bridge from the use of null values to optional values.Optional.ofNullable(obj) returns Optional.of(obj) if obj is not null, and Optional.empty() otherwise.

Optional<U> result = s.f().flatMap(T::g);

If s.f() is present, then g is applied to it. Otherwise, an empty Optional<U> is returned.

Clearly, you can repeat that process if you have more methods or lambdas that yield Optional values.You can then build a pipeline of steps that succeeds only when all parts do, simply by chaining calls to flatMap. For example, consider the safe inverse method of the preceding section. Suppose we also have a safe square root:

public static Optional<Double> squareRoot(Double x) {

return x < 0 ? Optional.empty() : Optional.of(Math.sqrt(x));

}

Then you can compute the square root of the inverse as

Optional<Double> result = inverse(x).flatMap(MyMath::squareRoot);

or, if you prefer,

Optional<Double> result =

Optional.of(-4.0).flatMap(Test::inverse).flatMap(Test::squareRoot);

If either the inverse method or the squareRoot returns Optional.empty(), the result is empty. flatMap method in the Stream interface was used to compose two methods that yield streams, by flattening out the resulting stream of streams. The Optional.flatMap method works in the same way if you consider an optional value to be a stream of size zero or one.

2.8. Reduction Operations

The simplest form takes a binary function and keeps applying it, starting with the first two elements.

Optional<Integer> sum = values.reduce((x, y) -> x + y)

In this case, the reduce method computes v0 + v1 + v2 + ..., where the vi are the stream elements. The method returns an Optional because there is no valid result if the stream is empty.In this case, you can write values.reduce(Integer::sum) instead of values.reduce((x, y) -> x + y). In general, if the reduce method has a reduction operation op, the reduction yields v0 op v1 op v2 op ...,

where we write vi op vi + 1 for the function call op(vi, vi + 1). The operation should be associative: It

shouldn’t matter in which order you combine the elements. This allows efficient reduction with parallel streams. There are many associative operations that might be useful in practice, such as sum and product, string concatenation, maximum and minimum, set union and intersection. An example of an operation that is not associative is subtraction. For example, (6 – 3) – 2 ≠ 6 – (3 – 2).

Often, there is an identity e such that e op x = x, and you can use that element as the start of the computation. For example, 0 is the identity for addition. Then call the second form of reduce:

Stream<Integer> values = ...;

Integer sum = values.reduce(0, (x, y) -> x + y);

// Computes 0 + v0 + v1 + v2 + ...

The identity value is returned if the stream is empty, and you no longer need to deal with the Optional class. Now suppose you have a stream of objects and want to form the sum of some property, such as all lengths in a stream of strings. You can’t use the simple form of reduce. It requires a function (T, T) -> T, with the same types for the arguments and the result. But in this situation, you have two types. The stream elements have type String, and the accumulated result is an integer. There is a form of reduce that can deal with this situation. First, you supply an “accumulator” function (total, word) -> total + word.length(). That function is called repeatedly, forming the cumulative total. But when the computation is parallelized, there will be multiple computations of this kind, and you need to combine their results. You supply a second function for that purpose. The complete call is:

int result = words.reduce(0,

(total, word) -> total + word.length(),

(total1, total2) -> total1 + total2);

In practice, you probably won’t use the reduce method a lot. It is usually easier to map to a stream of numbers and use one of its methods to compute sum, max, or min. In this particular example, you could have called words.mapToInt(String::length).sum(), which is both simpler and more efficient, since it doesn’t involve boxing.

2.9. Collecting Results

When you are done with a stream, you often just want to look at the results instead of reducing them to a value. You can call the iterator method, which yields an old-fashioned iterator that you can use to visit the elements. Or you can call toArray and get an array of the stream elements. Since it is not possible to create a generic array at runtime, the expression stream.toArray()

returns an Object[] array. If you want an array of the correct type, pass in the array constructor: String[] result = words.toArray(String[]::new);

// words.toArray() has type Object[]

Now suppose you want to collect the results in a HashSet. If the collection is parallelized, you can’t put the elements directly into a single HashSet because a HashSet object is not threadsafe. For that reason, you can’t use reduce. Each segment needs to start out with its own empty hash set, and reduce only lets you supply one identity value. Instead, use collect. It takes three arguments:

1. A supplier to make new instances of the target object, for example, a constructor for a hash set

2. An accumulator that adds an element to the target, for example, an add method

3. A combiner that merges two objects into one, such as addAll

The target object need not be a collection. It could be a StringBuilder or an object that tracks a count and a sum. HashSet<String> result = stream.collect(HashSet::new, HashSet::add, HashSet::addAll);

In practice, you don’t have to do that because there is a convenient Collector interface for these three functions, and a Collectors class with factory methods for common collectors. To collect a stream

into a list or set, you can simply call List<String> result = stream.collect(Collectors.toList()); or Set<String> result = stream.collect(Collectors.toSet()); If you want to control which kind of set you get, use the following call instead: TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new)); Suppose you want to collect all strings in a stream by concatenating them. You can call String result = stream.collect(Collectors.joining()); If you want a delimiter between elements, pass it to the joining method: String result = stream.collect(Collectors.joining(", "));

If your stream contains objects other than strings, you need to first convert them to strings, like this: String result = stream.map(Object::toString).collect(Collectors.joining(", "));

If you want to reduce the stream results to a sum, average, maximum, or minimum, then use one of the methods summarizing(Int|Long|Double). These methods take a function that maps the stream

objects to a number and yield a result of type (Int|Long|Double)SummaryStatistics, with methods for obtaining the sum, average, maximum, and minumum.

IntSummaryStatistics summary = words.collect(

Collectors.summarizingInt(String::length));

double averageWordLength = summary.getAverage();

double maxWordLength = summary.getMax();

So far, you have seen how to reduce or collect stream values. But perhaps you just want to print them or put them in a database. Then you can use the forEach method:

stream.forEach(System.out::println); The function that you pass is applied to each element. On a parallel stream, it’s your responsibility to ensure that the function can be executed concurrently. We discuss this in On a parallel stream, the elements can be traversed in arbitrary order. If you want to execute them in stream order, call forEachOrdered instead. Of course, you might then give up

most or all of the benefits of parallelism. The forEach and forEachOrdered methods are terminal operations. You cannot use the stream again after calling them. If you want to continue using the stream, use peek

2.10. Collecting into Maps

Map<Integer, String> idToName = people.collect(Collectors.toMap(Person::getId, Person::getName));

In the common case that the values should be the actual elements, use Function.identity() for the second function.

Map<Integer, Person> idToPerson = people.collect(Collectors.toMap(Person::getId, Function.identity()));

If there is more than one element with the same key, the collector will throw an IllegalStateException. You can override that behavior by supplying a third function argument that determines the value for the key, given the existing and the new value. Your function could return the existing value, the new value, or a combination of them.

Here, we construct a map that contains, for each language in the available locales, as key its name in your default locale (such as "German"), and as value its localized name (such as "Deutsch").

Stream<Locale> locales = Stream.of(Locale.getAvailableLocales());

Map<String, String> languageNames = locales.collect(

Collectors.toMap(

l -> l.getDisplayLanguage(),

l -> l.getDisplayLanguage(l),

(existingValue, newValue) -> existingValue));

We don’t care that the same language might occur twice—for example, German in Germany and in Switzerland, and we just keep the first entry. However, suppose we want to know all languages in a given country. Then we need a Map<String,Set<String>>. For example, the value for "Switzerland" is the set [French, German,Italian]. At first, we store a singleton set for each language. Whenever a new language is found for a given country, we form the union of the existing and the new set.

Map<String, Set<String>> countryLanguageSets = locales.collect(Collectors.toMap(

l -> l.getDisplayCountry(),

l -> Collections.singleton(l.getDisplayLanguage()),

(a, b) -> { // Union of a and b

Set<String> r = new HashSet<>(a);

r.addAll(b);

return r; }));

If you want a TreeMap, then you supply the constructor as the fourth argument. You must provide a merge function. Here is one of the examples from the beginning of the section, now yielding a TreeMap:

Map<Integer, Person> idToPerson = people.collect(Collectors.toMap(Person::getId,Function.identity(),(existingValue, newValue) -> { throw new IllegalStateException(); },TreeMap::new));

For each of the toMap methods, there is an equivalent toConcurrentMap method that yields a concurrent map. A single concurrent map is used in the parallel collection process. When used with a parallel stream, a shared map is more efficient than merging maps, but of course, you give up ordering.

2.11. Grouping and Partitioning

In the preceding section, you saw how to collect all languages in a given country. But the process was a bit tedious. You had to generate a singleton set for each map value, and then specify how to merge the existing and new values. Forming groups of values with the same characteristic is very common, and the groupingBy method supports it directly.

Map<String, List<Locale>> countryToLocales = locales.collect(Collectors.groupingBy(Locale::getCountry));

List<Locale> swissLocales = countryToLocales.get("CH");

// Yields locales [it\_CH, de\_CH, fr\_CH]

The function Locale::getCountry is the classifier function of the grouping.

When the classifier function is a predicate function (that is, a function returning a boolean value), the stream elements are partitioned into two lists: those where the function returns true and the complement. In this case, it is more efficient to use partitioningBy instead of groupingBy. For example, here we split all locales into those that use English, and all others:

Map<Boolean, List<Locale>> englishAndOtherLocales = locales.collect(Collectors.partitioningBy(l -> l.getLanguage().equals("en")));

List<Locale>> englishLocales = englishAndOtherLocales.get(true);

If you call the groupingByConcurrent method, you get a concurrent map that, when used with a parallel stream, is concurrently populated. This is entirely analogous to the toConcurrentMap method.

The groupingBy method yields a map whose values are lists. If you want to process those lists in some way, you supply a “downstream collector.” For example, if you want sets instead of lists, you can use the Collectors.toSet collector:

Map<String, Set<Locale>> countryToLocaleSet = locales.collect(groupingBy(Locale::getCountry, toSet()));

Several other collectors are provided for downstream processing of grouped elements:

Map<String, Long> countryToLocaleCounts = locales.collect(groupingBy(Locale::getCountry, counting()));

• summing(Int|Long|Double) takes a function argument, applies the function to the downstream elements, and produces their sum. For example,

Map<String, Integer> stateToCityPopulation = cities.collect(groupingBy(City::getState, summingInt(City::getPopulation)));

• maxBy and minBy take a comparator and produce maximum and minimum of the downstream elements.

Map<String, City> stateToLargestCity = cities.collect(groupingBy(City::getState,maxBy(Comparator.comparing(City::getPopulation))));

• mapping applies a function to downstream results, and it requires yet another collector for processing its results.

Map<String, Optional<String>> stateToLongestCityName = cities.collect(groupingBy(City::getState,mapping(City::getName,maxBy(Comparator.comparing(String::length)))));

The mapping method also yields a nicer solution to a problem from the preceding section, to gather a set of all languages in a country.

Map<String, Set<String>> countryToLanguages = locales.collect(groupingBy(l -> l.getDisplayCountry(),mapping(l -> l.getDisplayLanguage(),toSet())));

In the preceding section, I used toMap instead of groupingBy. In this form, you don’t need to worry about combining the individual sets. If the grouping or mapping function has return type int, long, or double, you can collect elements into a summary statistics object

Map<String, IntSummaryStatistics> stateToCityPopulationSummary = cities.collect(groupingBy(City::getState,summarizingInt(City::getPopulation)));

Finally, the reducing methods apply a general reduction to downstream elements. There are three forms: reducing(binaryOperator), reducing(identity,binaryOperator), and reducing(identity, mapper, binaryOperator). In the first form, the identity is null. (Note that this is different from the forms of Stream::reduce, where the method without an identity parameter yields an Optional result.) In the third form, the mapper function is applied and its values are reduced.

Map<String, String> stateToCityNames = cities.collect(groupingBy(City::getState,reducing("", City::getName,(s, t) -> s.length() == 0 ? t : s + ", " + t)));

As with Stream.reduce, Collectors.reducing is rarely necessary. In this case, you canachieve the same result more naturally as Click here to view code image

Map<String, String> stateToCityNames = cities.collect(groupingBy(City::getState,mapping(City::getName,joining(", "))));

Frankly, the downstream collectors can yield very convoluted expressions. You should only use them inconnection with groupingBy or partitioningBy to process the “downstream” map values. Otherwise, simply apply methods such as map, reduce, count, max, or min directly on streams.

2.12. Primitive Type Streams

So far, we have collected integers in a Stream<Integer>, even though it is clearly inefficient to wrap each integer into a wrapper object. The same is true for the other primitive types double, float,

long, short, char, byte, and boolean. The stream library has specialized types IntStream, LongStream, and DoubleStream that store primitive values directly, without using wrappers. If you want to store short, char, byte, and boolean, use an IntStream, and for float, use a DoubleStream. The library designers didn’t think it was worth adding another five stream types.

To create an IntStream, you can call the IntStream.of and Arrays.stream methods:

IntStream stream = IntStream.of(1, 1, 2, 3, 5);

stream = Arrays.stream(values, from, to); // values is an int[] array

As with object streams, you can also use the static generate(Math::random) and iterate(0, s -> s++) methods. In addition, IntStream and LongStream have static methods range and rangeClosed that generate integer ranges with step size one:

IntStream zeroToNinetyNine = IntStream.range(0, 100); // Upper bound is excluded

IntStream zeroToHundred = IntStream.rangeClosed(0, 100); // Upper bound is included

The CharSequence interface has methods codePoints and chars that yield an IntStream of the Unicode codes of the characters or of the code units in the UTF-16 encoding.

String sentence = "\uD835\uDD46 is the set of octonions.";

// \uD835\uDD46 is the UTF-16 encoding of the letter , unicode U+1D546

IntStream codes = sentence.codePoints();

// The stream with hex values 1D546 20 69 73 20 ...

When you have a stream of objects, you can transform it to a primitive type stream with the mapToInt, mapToLong, or mapToDouble methods. For example, if you have a stream of strings and want to process their lengths as integers, you might as well do it in an IntStream:

Stream<String> words = ...;

IntStream lengths = words.mapToInt(String::length);

To convert a primitive type stream to an object stream, use the boxed method:

Stream<Integer> integers = IntStream.range(0, 100).boxed();

Generally, the methods on primitive type streams are analogous to those on object streams. Here are the most notable differences:

• The toArray methods return primitive type arrays.

• Methods that yield an optional result return an OptionalInt, OptionalLong, or OptionalDouble. These classes are analogous to the Optional class, but they have methods getAsInt, getAsLong, and getAsDouble instead of the get method.

• There are methods sum, average, max, and min that return the sum, average, maximum, and minimum. These methods are not defined for object streams. The summaryStatistics method yields an object of type IntSummaryStatistics, LongSummaryStatistics, or DoubleSummaryStatistics that can simultaneously report the sum, average, maximum, and minimum of the stream.

The Random class has methods ints, longs, and doubles that return primitive type streams of random numbers.

Parallel Streams

Streams make it easy to parallelize bulk operations. The process is mostly automatic, but you need to follow a few rules. First of all, you must have a parallel stream. By default, stream operations create sequential streams, except for Collection.parallelStream(). The parallel method converts any sequential stream into a parallel one. For example:

Stream<String> parallelWords = Stream.of(wordArray).parallel();

As long as the stream is in parallel mode when the terminal method executes, all lazy intermediate stream operations will be parallelized. When stream operations run in parallel, the intent is that the same result is returned as if they had run serially. It is important that the operations are stateless and can be executed in an arbitrary order. Here is an example of something you cannot do.

int[] shortWords = new int[12];

words.parallel().forEach(s -> { if (s.length() < 12) shortWords[s.length()]++; });

// Error—race condition!

This is very, very bad code. The function passed to forEach runs concurrently in multiple threads, updating a shared array. That’s a classic race condition. If you run this program multiple times, you are quite likely to get a different sequence of counts in each run, each of them wrong. It is your responsibility to ensure that any functions that you pass to parallel stream operations are threadsafe. In our example, you could use an array of AtomicInteger objects for the counters. Or you could simply use the facilities of the streams library and group strings by length. By default, streams that arise from ordered collections (arrays and lists), from ranges, generators, and iterators, or from calling Stream.sorted, are ordered. Results are accumulated in the order of the original elements, and are entirely predictable. If you run the same operations twice, you will get exactly the same results. Ordering does not preclude parallelization. For example, when computing stream.map(fun), the stream can be partitioned into n segments, each of which is concurrently processed. Then the results are reassembled in order.

Some operations can be more effectively parallelized when the ordering requirement is dropped. By calling the Stream.unordered method, you indicate that you are not interested in ordering. One operation that can benefit from this is Stream.distinct. On an ordered stream, distinct retains the first of all equal elements. That impedes parallelization—the thread processing a segment can’t know which elements to discard until the preceding segment has been processed. If it is acceptable to retain any of the unique elements, all segments can be processed concurrently (using a shared set to track duplicates). You can also speed up the limit method by dropping ordering. If you just want any n elements from a stream and you don’t care which ones you get, call

Stream<T> sample = stream.parallel().unordered().limit(n);

As discussed “Collecting into Maps,” on page 34, merging maps is expensive. For that reason, the Collectors.groupingByConcurrent method uses a shared concurrent map. Clearly, to benefit from parallelism, the order of the map values will not be the same as the stream order. Even on an ordered stream, that collector has a “characteristic” of being unordered, so that it can be used efficiently without having to make the stream unordered. You still need to make the stream parallel, though:

Map<String, List<String>> result = cities.parallel().collect(Collectors.groupingByConcurrent(City::getState));

// Values aren’t collected in stream order

It is very important that you don’t modify the collection that is backing a stream while carrying out a stream operation (even if the modification is threadsafe). Remember that streams don’t collect their own data—the data is always in a separate collection. If you were to modify that collection, the outcome of the stream operations would be undefined. The JDK documentation refers to this requirement as noninterference. It applies both to sequential and parallel streams. To be exact, since intermediate stream operations are lazy, it is possible to mutate the collection up to the point when the terminal operation executes. For example, the following is correct:

List<String> wordList = ...;

Stream<String> words = wordList.stream();

wordList.add("END"); // Ok

long n = words.distinct().count();

But this code is not:

Stream<String> words = wordList.stream();

words.forEach(s -> if (s.length() < 12) wordList.remove(s));

// Error—interference

"PECS"

is from the collection's point of view. If you are *only* pulling items from a generic collection, it is a producer and you should use extends; if you are *only* stuffing items in, it is a consumer and you should use super. If you do both with the same collection, you shouldn't use either extends or super.

Suppose you have a method that takes as its parameter a collection of things, but you want it to be more flexible than just accepting a Collection<Thing>.

**Case 1: You want to go through the collection and do things with each item.**  
Then the list is a **producer**, so you should use a Collection<? extends Thing>.

The reasoning is that a Collection<? extends Thing> could hold any subtype of Thing, and thus each element will behave as a Thing when you perform your operation. (You actually cannot add anything to a Collection<? extends Thing>, because you cannot know at runtime which *specific*subtype of Thing the collection holds.)

**Case 2: You want to add things to the collection.**  
Then the list is a **consumer**, so you should use a Collection<? super Thing>.

The reasoning here is that unlike Collection<? extends Thing>, Collection<? super Thing> can always hold a Thing no matter what the actual parameterized type is. Here you don't care what is already in the list as long as it will allow a Thing to be added; this is what ? super Thingguarantees.

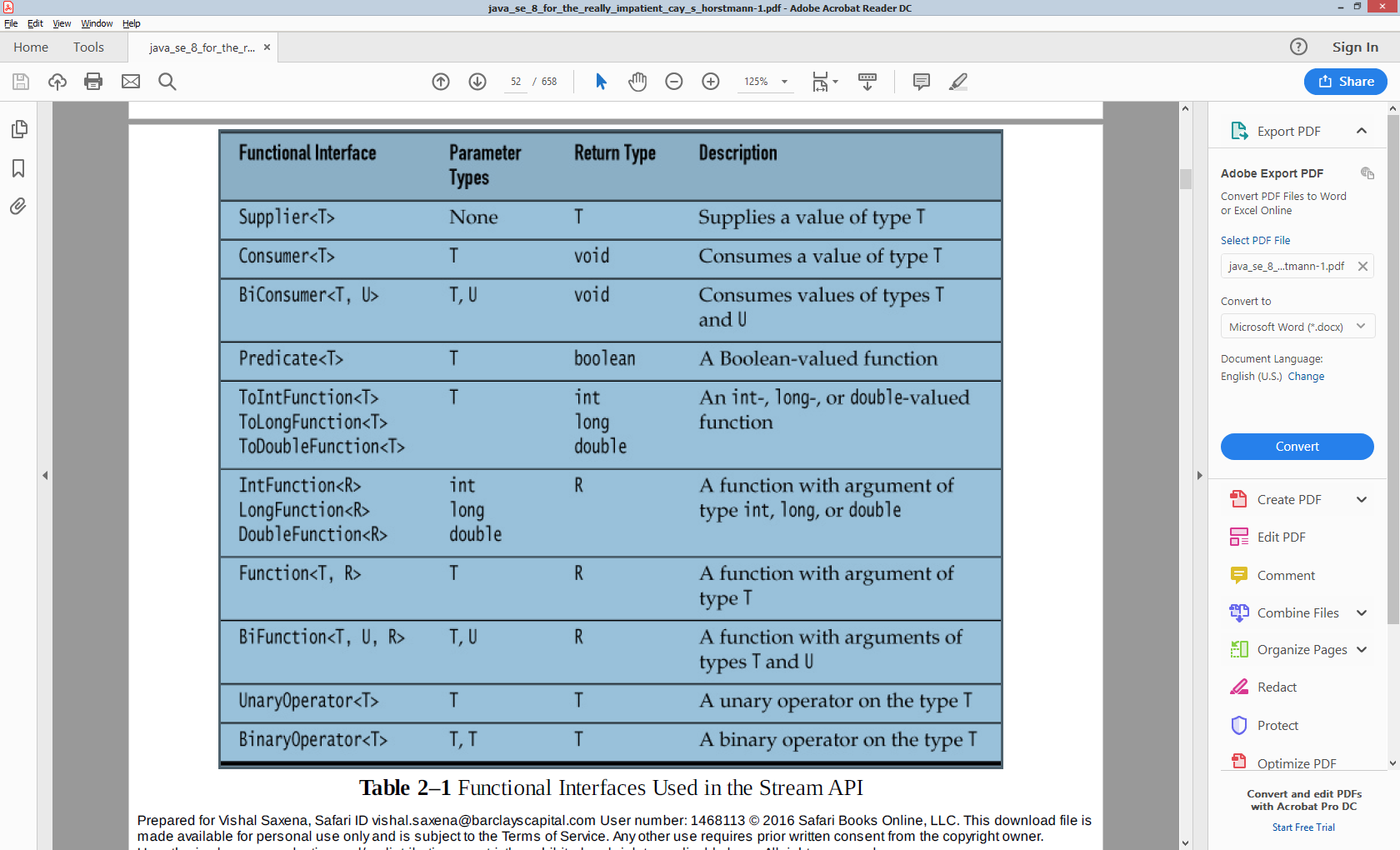
When you look closely at the declaration of Stream.filter, you will note the wildcard type Predicate<? super T>. This is common for function parameters. For example, suppose Employee is a subclass of Person, and you have a Stream<Employee>. You can filter the stream (where T is Employee) with a Predicate<Employee>, a Predicate<Person>, or a Predicate<Object>. This flexibility is particularly

important for supplying method references. For example, you may want to use Person::isAlive to filter a Stream<Employee>. That only works because of wild card in the parameter of the filter method.

When working with generic functional interfaces, use ? super wildcards for argument types, ? extends wildcards for return types.

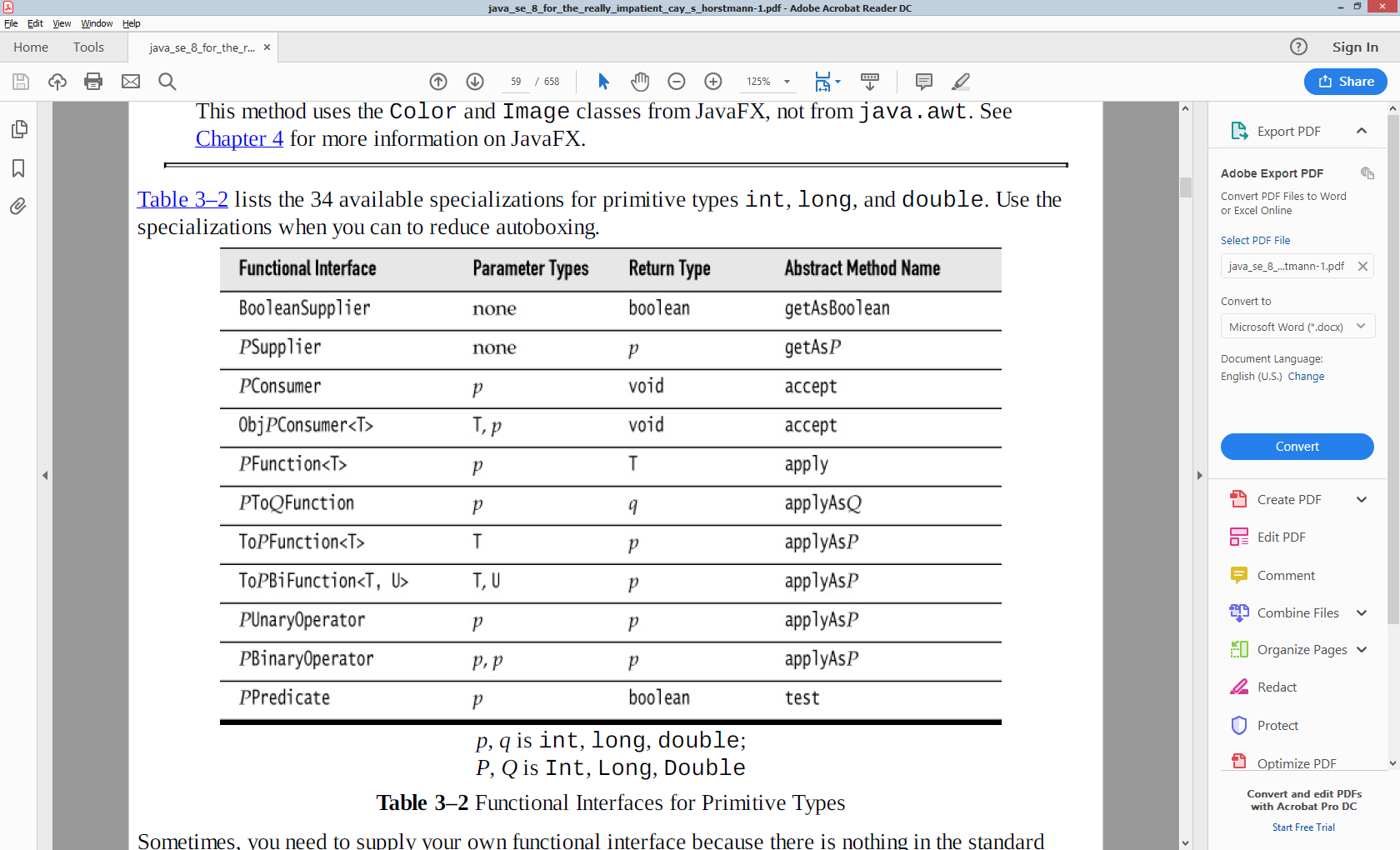
Functional interfaces table

that occur as parameters of the Stream and Collectors methods:



Runnable void void

Callable void T throws Exception



Deferred Execution

The point of all lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you’d do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as

• Running the code in a separate thread • Running the code multiple times • Running the code at the right point in an algorithm (for example, the comparison operation in sorting) • Running the code when something happens (a button was clicked, data has arrived, and so on) • Running the code only when necessary

Deferring logging messages is such a good idea that the Java 8 library designers beat me to it. The info method, as well as the other logging methods, now have variants that accept a Supplier<String>. You can directly call logger.info(() -> "x: " + x + ", y:" + y).

public static void repeat(int n, IntConsumer action) {for (int i = 0; i < n; i++) action.accept(i);} vs public static void repeat(int n, Runnable action) {for (int i = 0; i < n; i++) action.run();}

We tell the action in which iteration it occurs, which might be useful information. The action needs to capture that input in a parameter.

repeat(10, i -> System.out.println("Countdown: " + (9 - i)));

Another example : button.setOnAction(event -> action);

Exception Handling in Lambdas

When an exception is thrown in a lambda expression, it is propagated to the caller. There is nothing special about executing lambda expressions, of course. They are simply method calls on some object that implements a functional interface. Often it is appropriate to let the exception bubble up to the caller.

public static void doInOrder(Runnable first, Runnable second) { first.run(); second.run(); }

If first.run() throws an exception, then the doInOrder method is terminated, second is never run, and the caller gets to deal with the exception. But now suppose we execute the tasks asynchronously.

public static void doInOrderAsync(Runnable first, Runnable second) {

Thread t = new Thread() {

public void run() {

first.run();

second.run();

}

};

t.start();

}

If first.run() throws an exception, the thread is terminated, and second is never run. However, the doInOrderAsync returns right away and does the work in a separate thread, so it is not possible to

have the method rethrow the exception. In this situation, it is a good idea to supply a handler:

public static void doInOrderAsync(Runnable first, Runnable second, Consumer<Throwable> handler) {

Thread t = new Thread() {

public void run() {

try {

first.run();

second.run();

} catch (Throwable t) {

handler.accept(t);

}

}

};

t.start();

}

Now suppose that first produces a result that is consumed by second. We can still use the handler.

public static <T> void doInOrderAsync(Supplier<T> first, Consumer<T> second, Consumer<Throwable> handler) {

Thread t = new Thread() {

public void run() {

try {

T result = first.get();

second.accept(result);

}

catch (Throwable t) {

handler.accept(t);

}

}

};

t.start();

}

Alternatively, we could make second a BiConsumer<T, Throwable> and have it deal with the exception from first. It is often inconvenient that methods in functional interfaces don’t allow checked exceptions. Of course, your methods can accept functional interfaces whose methods allow checked exceptions, such as Callable<T> instead of Supplier<T>. A Callable<T> has a method that is declared as T call() throws Exception. If you want an equivalent for a Consumer or a Function, you have to create it yourself.

public static <T> Supplier<T> unchecked(Callable<T> f) {

return () -> {

try {

return f.call();

}

catch (Exception e) {

throw new RuntimeException(e);

}

catch (Throwable t) {

throw t;

}

};

}

Then you can pass a unchecked(() -> new String(Files.readAllBytes(Paths.get("/etc/passwd")), StandardCharsets.UTF\_8)) to a Supplier<String>, even though the readAllBytes method throws an IOException.

That is a solution, but not a complete fix. For example, this method cannot generate a Consumer<T> or a Function<T, U>. You would need to implement a variation of unchecked for each functional

interface.

##### Comparable,Comparator,Comparators

Comparing is the natural ordering. Collections.sort(ArrayList<Structure>) can only in place sort if it Structure implements Comparable.compareTo(T) returning -ve|0|+ve. If however that natural ordering is to be overwritten(or multiple sorting ways required) or doesn’t exist in library code, pass a Comparator.comparet(T,T) to the Collections.sort. Comparators.comparing(keyExtractorFunction) in Java8 will do the comparison provided the keyExtractor function.

static <T,U extends Comparable<? super U>> Comparator<T> comparing(

   Function<? super T,? extends U> keyExtractor)

There is another option that facilitates overriding the natural ordering of the sort key by providing the *Comparator*that creates a custom ordering for the sort key:

|  |  |
| --- | --- |
| 1  2  3 | static <T,U> Comparator<T> comparing(    Function<? super T,? extends U> keyExtractor,      Comparator<? super U> keyComparator) |

Co – contra variance is to do with how the reference type varies with the instance type. A reference to parent type holding a child instance type is covariance in type conversion.

**Arrays**

Arrays in Java are **covariant** in the type of the objects they hold. In other words, Clazz[] can hold SubClazzobjects.

Clazz**[]** array **=** **new** Clazz**[**10**];**

array**[**0**]** **=** **new** SubClazz**();**

They are also **covariant** in the type of the array itself. You can directly assign a SubClazz[] type to a Clazz[].

Clazz**[]** array **=** **new** SubClazz**[**10**];**

Be careful though; the above line is dangerous. Although the type of the array variable is Clazz[], the actual array object on the heap is a SubClazz[]. For that reason, the following code compiles fine but throws a java.lang.ArrayStoreException at runtime:

Clazz**[]** array **=** **new** SubClazz**[**10**];**

array**[**0**]** **=** **new** Clazz**();**

## Overriding methods

The overriding method is **covariant** in the return type and **invariant** in the argument types. That means that the return type of the overriding method can be a subclass of the return type of the overridden method, but the argument types must match exactly.

**public** **interface** **Parent** **{**

**public** Clazz **act(**Clazz argument**);**

**}**

**public** **interface** **Child** **extends** Parent **{**

**@Override**

**public** SubClazz **act(**Clazz argument**);**

**}**

If the argument types aren’t identical in the subclass then the method will be overloaded instead of overridden. You should always use the @Override annotation to ensure that this doesn’t happen accidentally.

## Generics

Unless bounds are involved, generic types are **invariant** with respect to the parameterized type. So you can’t do covariant ArrayLists like this:

ArrayList**<**Clazz**>** ary **=** **new** ArrayList**<**SubClazz**>();** *// Error!*

The normal rules apply to the type being parameterized:

List**<**Clazz**>** list **=** **new** ArrayList**<**Clazz**>();**

Unbounded wildcards allow assignment with any type parameter:

List**<?>** list **=** **new** ArrayList**<**Clazz**>();**

Bounded wildcards affect assignment like you might expect:

List**<?** **extends** Clazz**>** list **=** **new** ArrayList**<**SubClazz**>();**

List**<?** **super** Clazz**>** list2 **=** **new** ArrayList**<**Object**>();**

Java is smart enough that more restrictive type bounds are commensurable with less restrictive type bounds when appropriate:

List**<?** **super** Clazz**>** clazzList**;**

List**<?** **super** SubClazz**>** subClazzList**;**

subClazzList **=** clazzList**;**

Type parameter bounds work the same way, [although they cannot be lower-bounded](http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeParameters.html#FAQ107). If you have multiple upper bounds on a type parameter, you can upcast to any of them, as expected:

**interface** **A** **{}**

**interface** **B** **{}**

**interface** **C** **extends** A**,** B **{}**

**public** **class** **Holder<**T **extends** A **&** B**>** **{**

T member**;**

**}**

A member1 **=** **new** Holder**<**C**>().**member**;**

B member2 **=** **new** Holder**<**C**>().**member**;**

C member3 **=** **new** Holder**<**C**>().**member**;**

You can add or remove the type parameters from the return type of an overriding method and it will still compile:

**public** **interface** **Parent** **{**

**public** List **echo();**

**}**

**public** **interface** **Child** **extends** Parent **{**

**@Override**

**public** List**<**String**>** **echo();**

**}**

**public** **interface** **Parent** **{**

**public** List**<**String**>** **echo();**

**}**

**public** **interface** **Child** **extends** Parent **{**

**@Override**

**public** List **echo();**

**}**

Wildcards can be present in the types of method arguments. If you want to override a method with a wildcard-typed argument, the overriding method must have an identical type parameter. You cannot be “more specific” with the overriding method:

**public** **interface** **Parent** **{**

**public** **void** **act(**List**<?** **extends** List**>** a**);**

**}**

**public** **interface** **Child** **extends** Parent **{**

**@Override**

**public** **void** **act(**List**<?** **extends** ArrayList**>** a**);** *// Error!*

**}**

Also, you can replace any type-parameterized method argument with a non-type-parameterized method argument in the subclass and it will still be considered an override:

**public** **interface** **Parent** **{**

**public** **void** **act(**List**<?** **extends** Number**>** a**);**

**}**

**public** **interface** **Child** **extends** Parent **{**

**@Override**

**public** **void** **act(**List a**);**

**}**

<https://briangordon.github.io/2014/09/covariance-and-contravariance.html>

# Generics

## Non-Reifiable Types

A reifiable type is a type whose type information is fully available at runtime. This includes primitives, non-generic types, raw types, and invocations of unbound wildcards.

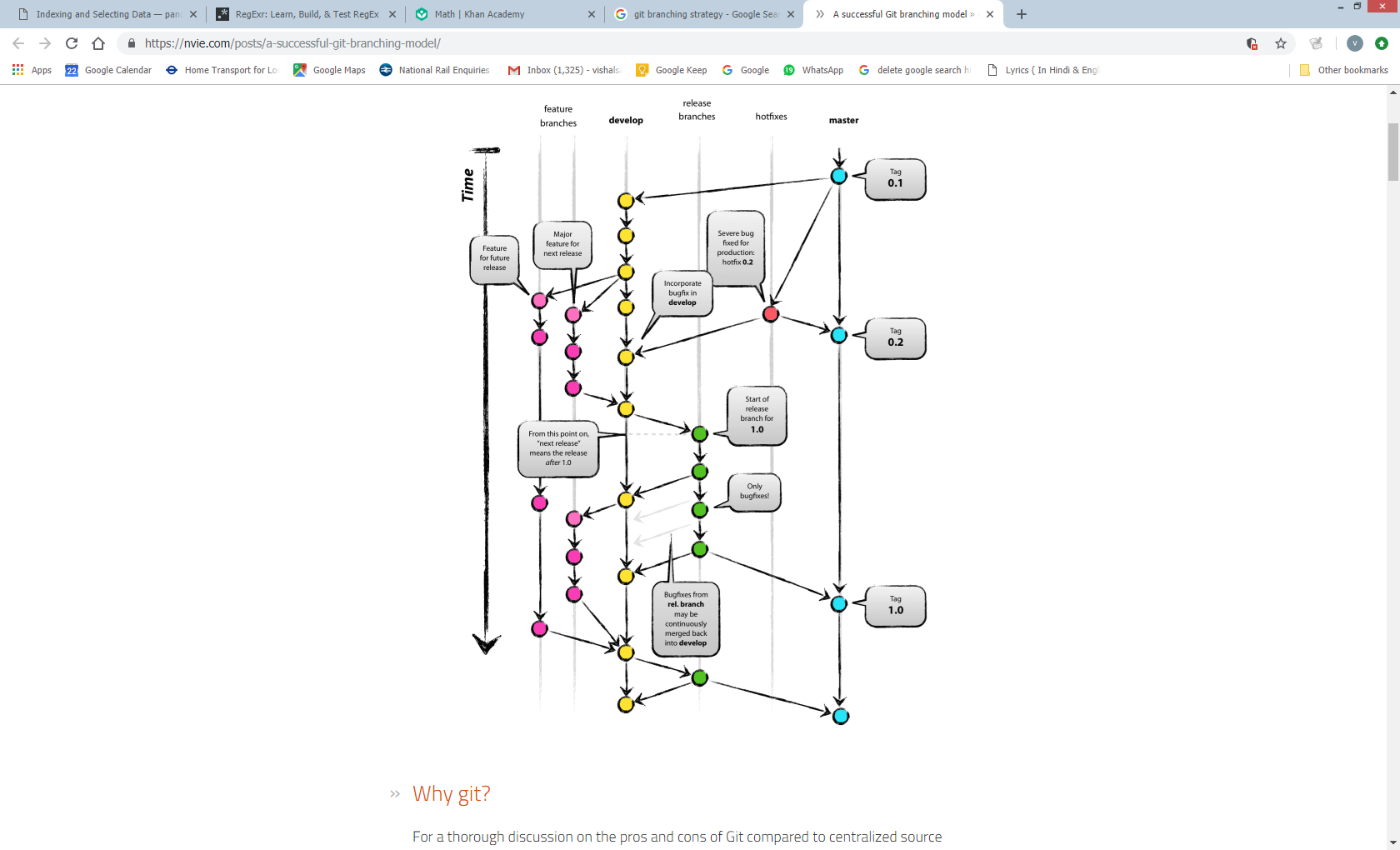
Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String> and List<Number>; the JVM cannot tell the difference between these types at runtime. As shown in [Restrictions on Generics](https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html), there are certain situations where non-reifiable types cannot be used: in an instanceof expression, for example, or as an element in an array.

**public class** GenericsArray<T> {  
  
 T[] **array** = (T[]) **new** Object[4];  
  
 **public static void** main(String[] args) {  
  
 GenericsArray<Integer> ga = **new** GenericsArray<>();  
 ga.**array**[0]= 1;  
 ga.**array**[1] = **""**; //compile error  
  
 }  
}

https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html#cannotCast

<http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeParameters.html#FAQ107>

# Git Branching Strategy



# Java APIs

* In currently latest JDK6 release/build (b27), the [Scanner](http://docs.oracle.com/javase/6/docs/api/java/util/Scanner.html) has a smaller buffer ([1024 chars](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b27/java/util/Scanner.java#350)) as opposed to the [BufferedReader](http://docs.oracle.com/javase/6/docs/api/java/io/BufferedReader.html) ([8192 chars](http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b27/java/io/BufferedReader.java#80)), but it's more than sufficient.

As to the choice, use the Scanner if you want to **parse** the file, use the BufferedReader if you want to **read** the file line by line. Also see the introductory text of their aforelinked API documentations.

* **Parsing** = interpreting the given input as tokens (parts). It's able to give back you specific parts directly as int, string, decimal, etc. See also all those nextXxx() methods in Scanner class.
* **Reading** = dumb streaming. It keeps giving back you all characters, which you in turn have to manually inspect if you'd like to match or compose something useful. But if you don't need to do that anyway, then reading is sufficient.