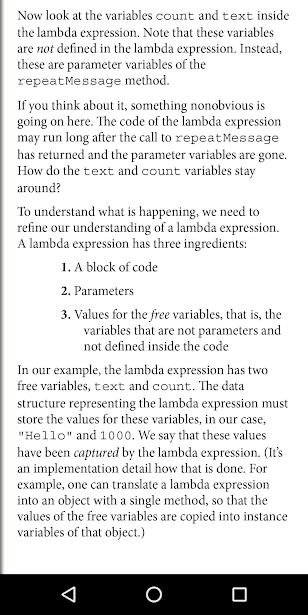
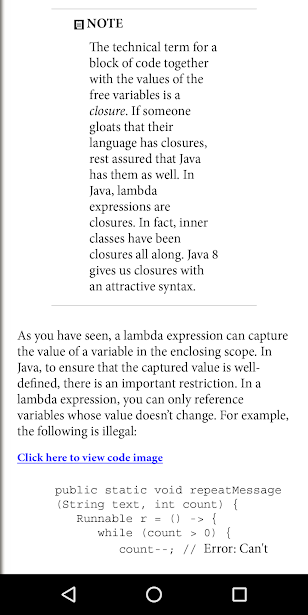
### Java8

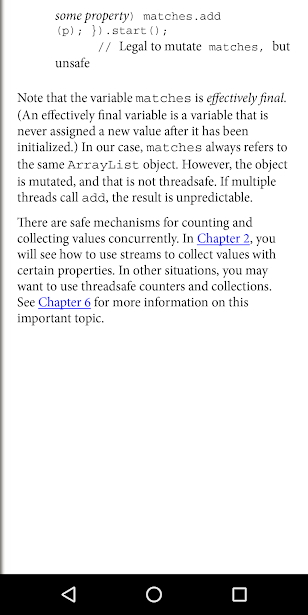
Lambda expressions

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.



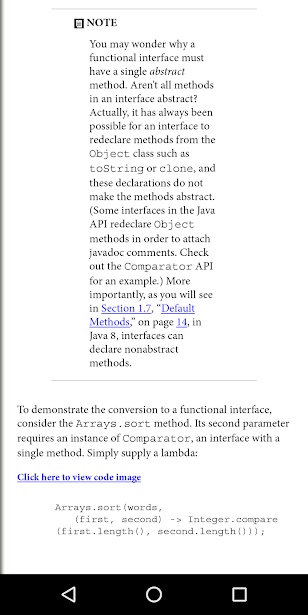
Lambdas can’t throw an Exception. Try/Catch it. Or use them for a functional interface that throws an Exception, callable instead of runnable, for e.g.





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 );

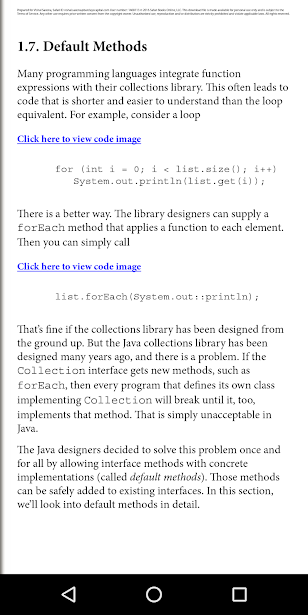
List<Integer> you = Arrays.stream( data ).boxed().collect( Collectors.toList() );

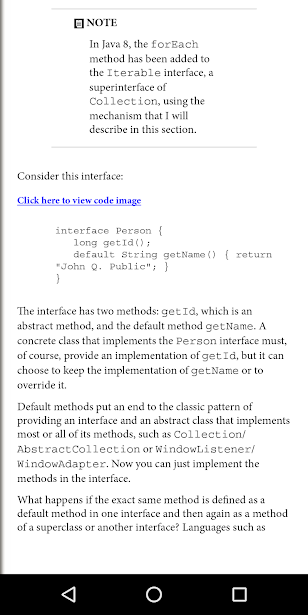
List<Integer> like = IntStream.of( data ).boxed().collect( Collectors.toList() );

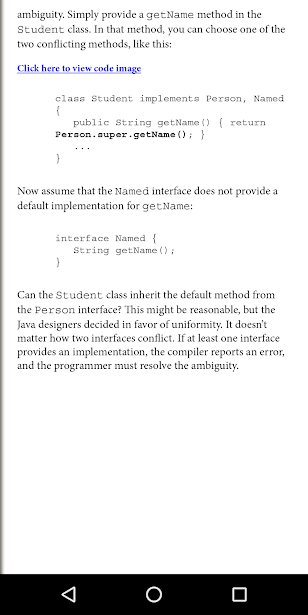
Double[] boxed = new Double[] { 1.0, 2.0, 3.0 };

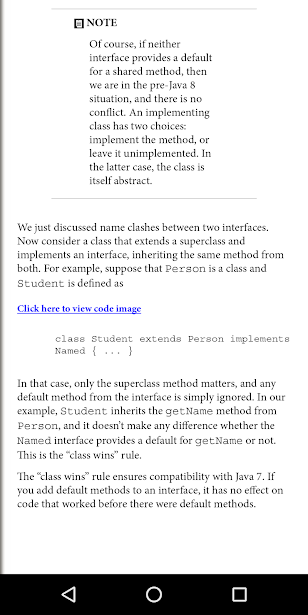
double[] unboxed = Stream.of(boxed).mapToDouble(Double::doubleValue).toArray();

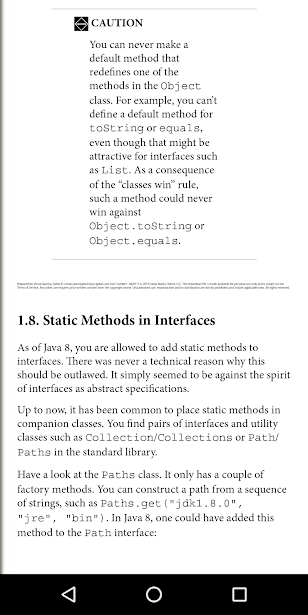
Default Method/Static Methods in Interfaces

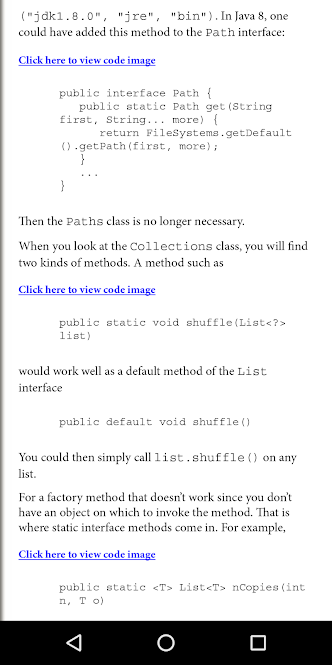


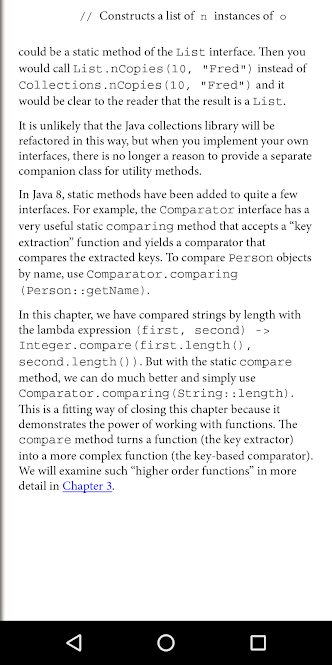


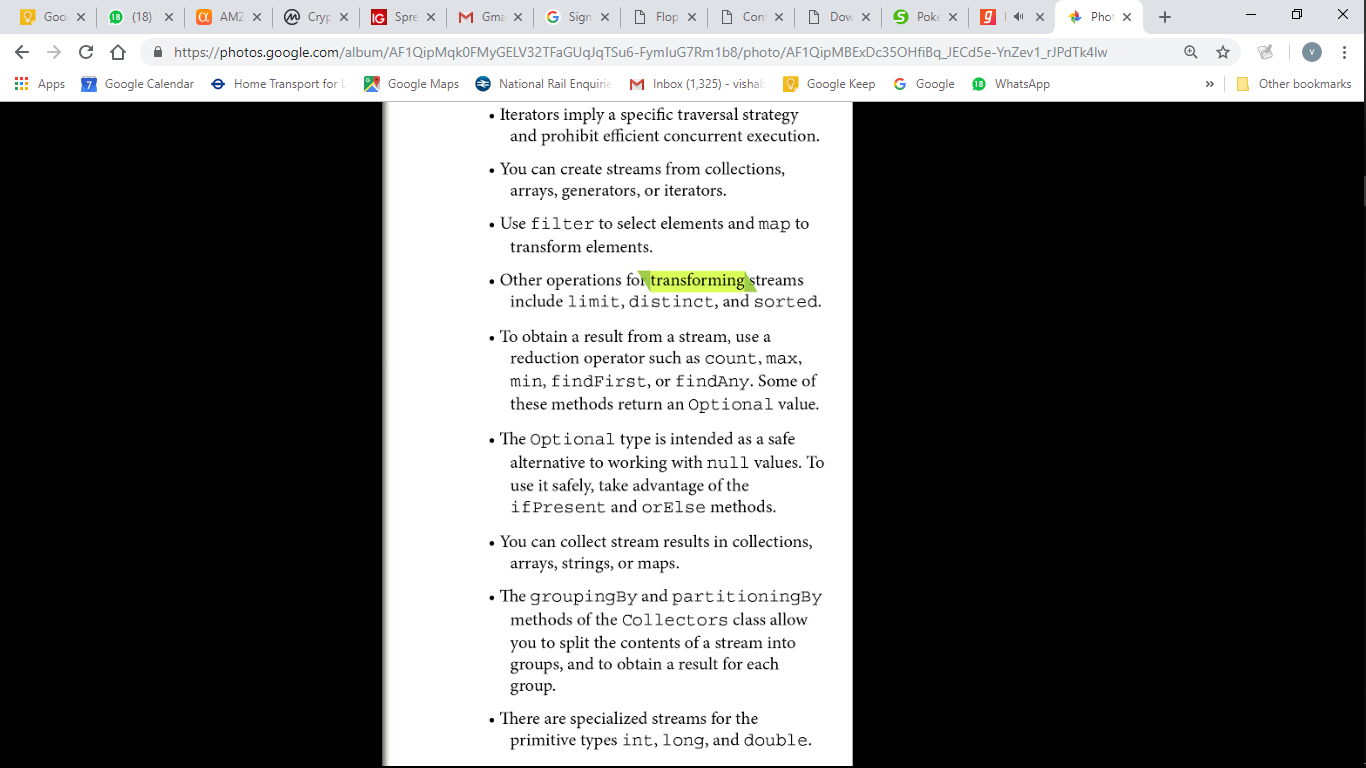




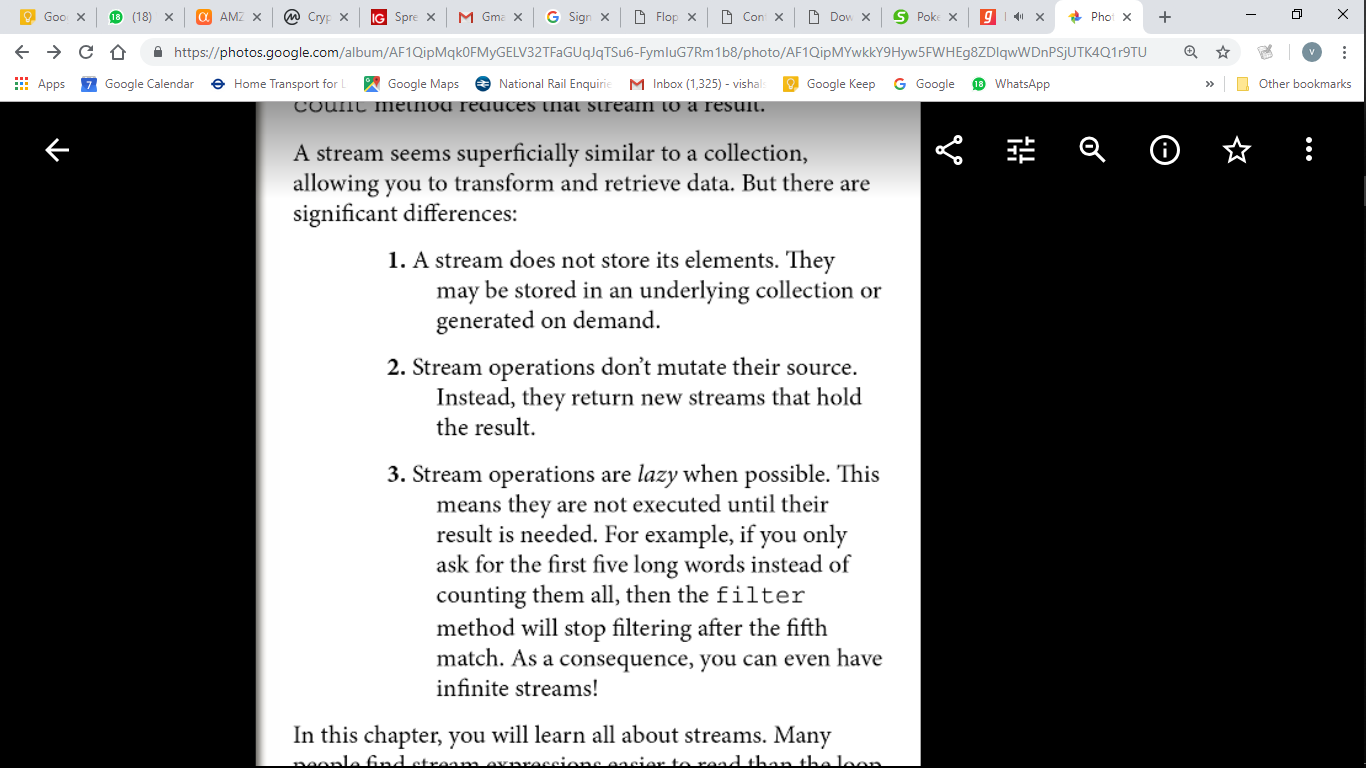


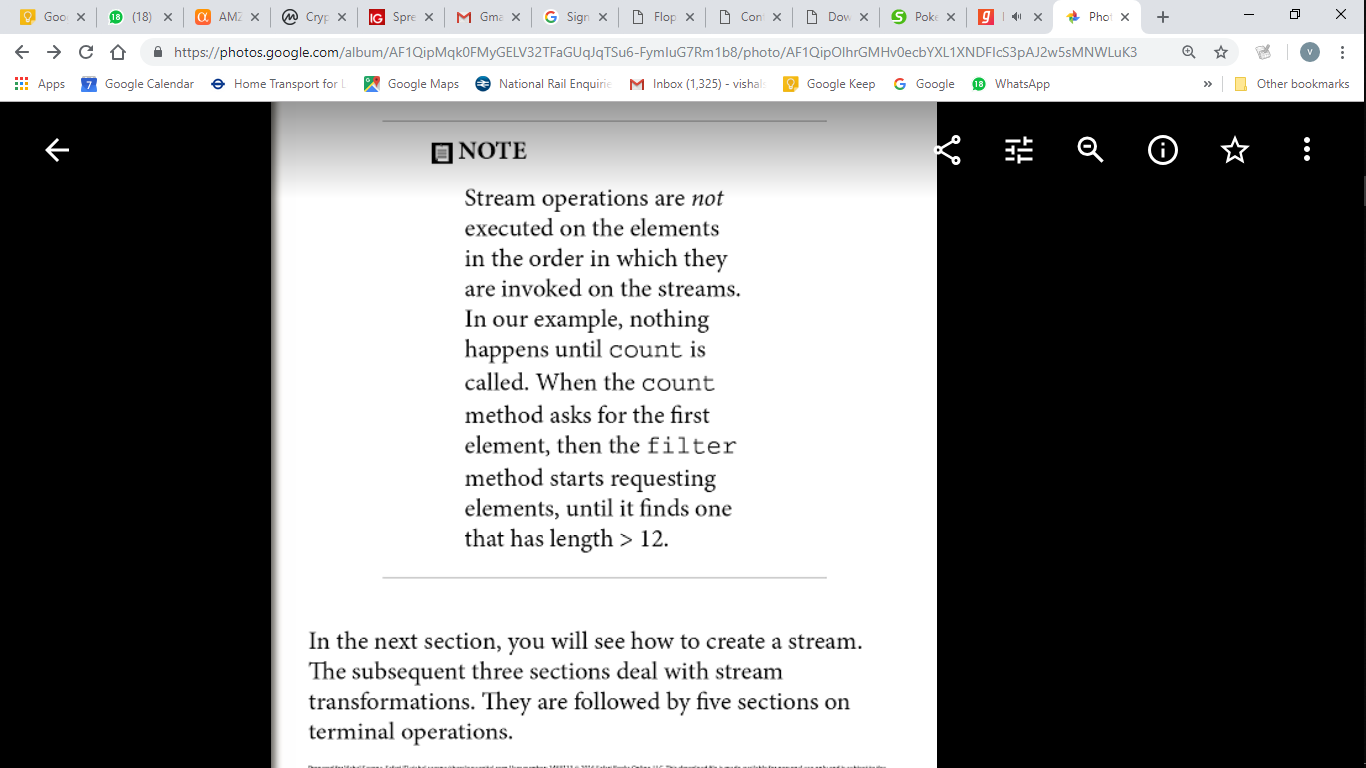


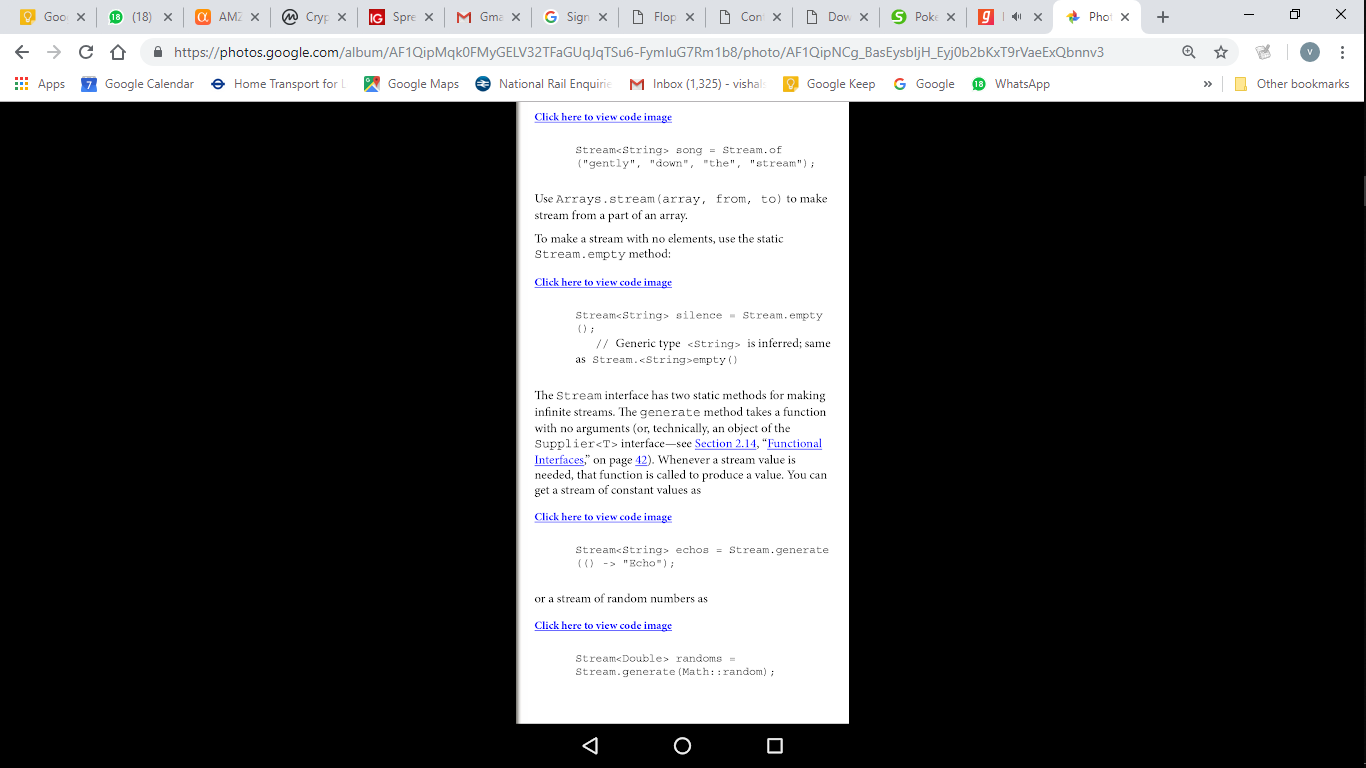


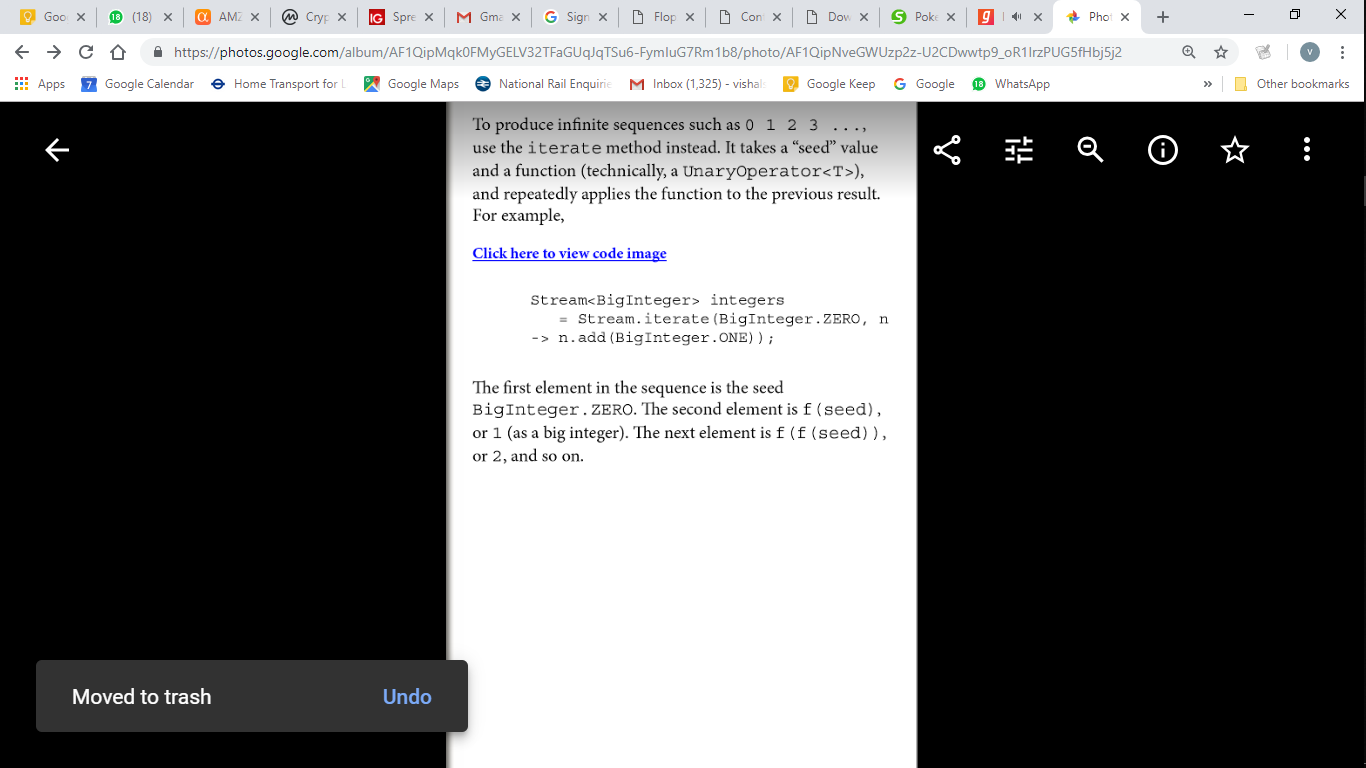


Streams

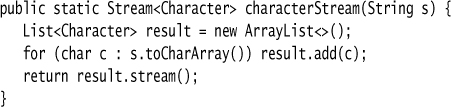








A number of methods that yield streams have been added to the API with the Java 8 release. For example, the Pattern class now has a method splitAsStream that splits a CharSequence by a regular expression. You can use the following statement to split a string into words:Stream<String> words  
   = Pattern.compile("[\\P{L}]+").splitAsStream(contents);The 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"

Producer extends Consumer Super. [See Co-Contra Variance](#_Use-site_variance)

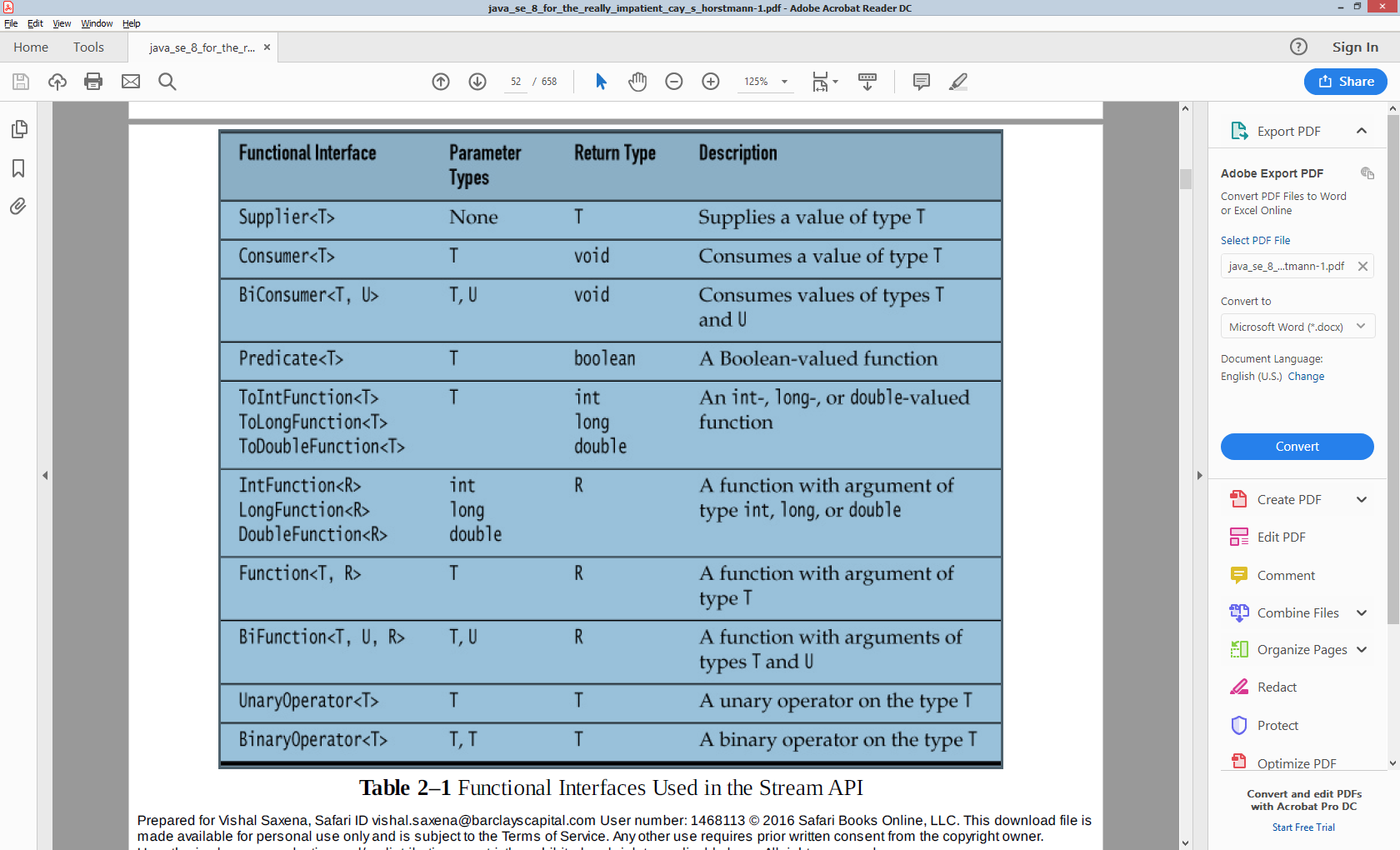
We say that the type parameter T of List<T> is invariant.

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

default <V> BiFunction<T,U,V> andThen(Function<? super R,? extends V> after)

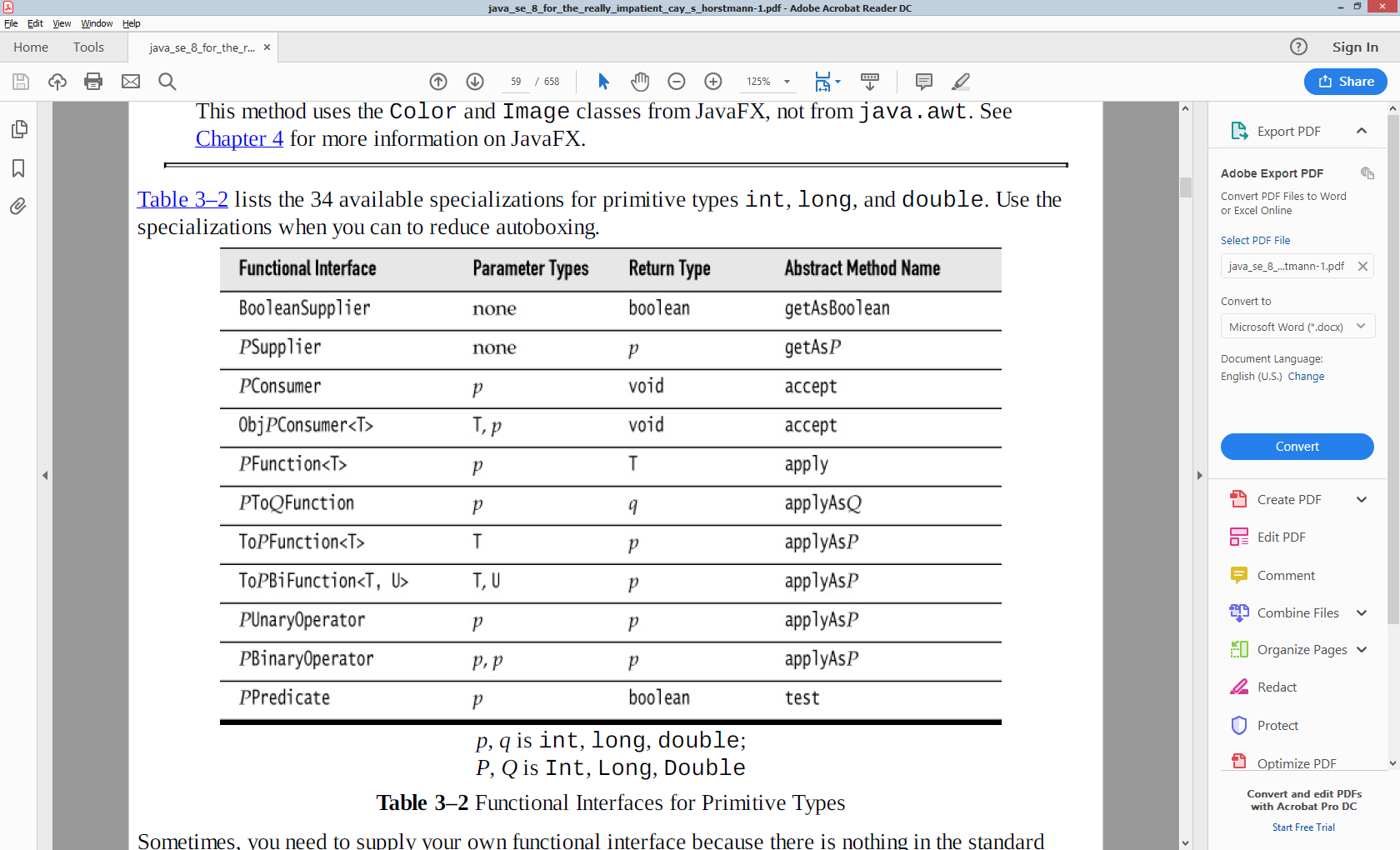
Functional interfaces table

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.

Stream.toArray

One of the unhappy consequences of type erasure is that you cannot construct a generic array at runtime. For example, the toArray() method of Collection<T> and Stream<T> cannot call T[] result = new T[n]. Therefore, these methods return Object[] arrays. In the past, the solution was to provide a second method that accepts an array. That array was either filled or used to create a new one via reflection. For example, Collection<T> has a method toArray(T[] a). With lambdas, you have a new option, namely to pass the constructor. That is what you do with streams:

String[] result = words.toArray(String[]::new);

Monadic Operations

Instead of the map that takes as argument a Function (T,R) that converts T->R and returns a Stream<R> a flatMap is a monadic operation that takes a Function (T, Stream<R>):

<R> Stream<R> flatMap(Function<? super T,? extends Stream<? extends R>> mapper)

Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. Each mapped stream is closed after its contents have been placed into this stream. (If a mapped stream is null an empty stream is used, instead.) The flatMap() operation has the effect of applying a one-to-many transformation to the elements of the stream, and then flattening the resulting elements into a new stream.

Stream<String> lines = Files.lines(path, StandardCharsets.UTF\_8);

Stream<String> words = lines.flatMap(line -> Stream.of(line.split(" +")));

Same thing for:

<U> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U> flatMap([Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html),[Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U>> mapper)

If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. This method is similar to [map(Function)](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#map-java.util.function.Function-), but the provided mapper is one whose result is already an Optional, and if invoked, flatMap does not wrap it with an additional Optional

public <U> [Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U> flatMap([Function](https://docs.oracle.com/javase/8/docs/api/java/util/function/Function.html)<? super [T](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html),[Optional](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html)<U>> mapper)

If a value is present, apply the provided Optional-bearing mapping function to it, return that result, otherwise return an empty Optional. This method is similar to [map(Function)](https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html#map-java.util.function.Function-), but the provided mapper is one whose result is already an Optional, and if invoked, flatMap does not wrap it with an additional Optional.

##### 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.

“Why can’t I use a List<Apple> as a List<Fruit>?” (given that Apple is a subclass of Fruit) Their reasoning usually goes like this: “An apple is a fruit, so a basket of apples is a fruit basket, right?”

List<Apple> apples = new ArrayList<>();

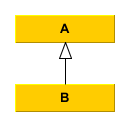
List<Fruit> fruits = apples;

fruits.add(new Orange());

// what's an orange doing here?!

Apple apple = apples.get(0);

The difference is the “flow” direction of objects of type *T* in their interface:

1. If a generic interface has only methods that return objects of type *T*, but don’t consume objects of type *T*, then assignment from a variable of *Type<B>* to a variable of *Type<A>* can make sense. This is called **covariance**. Examples are: I*terable<T>, Iterator<T>, Supplier<T>*
2. If a generic interface has only methods that consume objects of type *T*, but don’t return objects of type *T*, then assignment from a variable of *Type<A>* to a variable of *Type<B>* can make sense. This is called **contravariance**. Examples are: *Comparable<T>, Consumer<T>*
3. If a generic interface has both methods that return and methods that consume objects of type *T* then it should be **invariant**. Examples are: *List<T>, Set<T>*

Java doesn’t allow covariance or contravariance for generic types by default. They’re invariant by default. But there are ways to achieve co- and contravariance.

Use-site variance

In Java you can express co- and contravariance with wildcards like <? extends A> and <? super B>.

Producer<B> producerOfB = /\*...\*/;

Producer<? extends A> producerOfA = producerOfB; // legal

A a = producerOfA.produce();

// producerOfB = producerOfA; // still illegal

Consumer<A> consumerOfA = /\*...\*/;

Consumer<? super B> consumerOfB = consumerOfA; // legal

consumerOfB.consume(new B());

// consumerOfA = consumerOfB; // still illegal

This is called **use-site variance**, because the annotation is not placed where the type is declared, but where the type is used.

Declaration-site variance

In C# you can use the in and out keywords on a type parameter to indicate variance:

interface IProducer<out T> // Covariant

{

    T produce();

}

 interface IConsumer<in T> // Contravariant

{

    void consume(T t);

}

IProducer<B> producerOfB = /\*...\*/;

IProducer<A> producerOfA = producerOfB;  // now legal

// producerOfB = producerOfA;  // still illegal

IConsumer<A> consumerOfA = /\*...\*/;

IConsumer<B> consumerOfB = consumerOfA;  // now legal

// consumerOfA = consumerOfB;  // still illegal

Arrays are covariant, not invariant, even though a T[] has the same problem as a List<T>would have if it was covariant: Unfortunately, this code compiles but throws an exception at runtime

Apple[] apples = new Apple[10];

Fruit[] fruits = apples;

fruits[0] = new Orange();

Apple apple = apples[0];

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>

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.

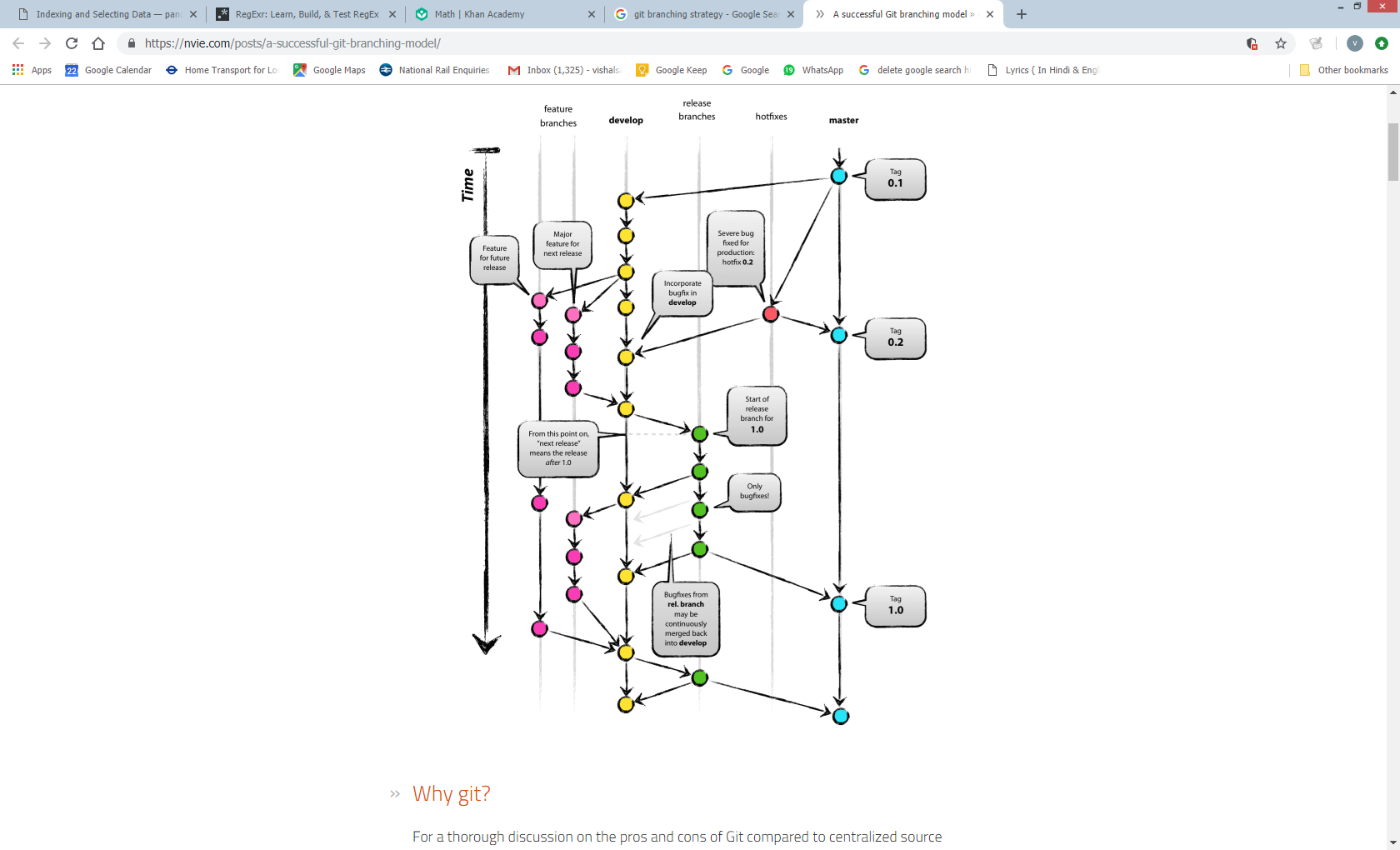
Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String> 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.

java8 new Date and Time Api

That’s where things get complicated. Java 1.0 had a Date class that was, in hindsight,unbelievably naïve, and had most of its methods deprecated in Java 1.1 when a Calendar class was introduced. Its API wasn’t stellar, its instances were mutable, and it didn’t deal with issues such as leap seconds. The third time is a charm, and the java.time API that is introduced in Java 8 has remedied the flaws of the past and should serve us for quite some time.

The key points of this chapter are:

• All java.time objects are immutable.

• An Instant is a point on the time line (similar to a Date).

• In Java time, each day has exactly 86,400 seconds (i.e., no leap seconds).

• A Duration is the difference between two instants.

• LocalDateTime has no time zone information.

• TemporalAdjuster methods handle common calendar computations, such as finding the first

Tuesday of a month.

• ZonedDateTime is a point in time in a given time zone (similar to GregorianCalendar).

• Use a Period, not a Duration, when advancing zoned time, in order to account for daylight

savings time changes.

• Use DateTimeFormatter to format and parse dates and times.

Instant, Duration

In Java, an Instant represents a point on the time line. An origin, called the epoch, is arbitrarily set at midnight of January 1, 1970 at the prime meridian This is the same convention used in the Unix/POSIX time. Starting from that origin, time is measured in 86,400 seconds per day, forwards and backwards, in nanosecond precision. The Instant values go back as far as a billion years (Instant.MIN). The largest value, Instant.MAX, is December 31 of the year 1,000,000,000. The static method call Instant.now() gives the current instant. You can compare two instants with the equals and compareTo methods in the usual way, so you can use instants as timestamps.

Instant start = Instant.now();

runAlgorithm();

Instant end = Instant.now();

Duration timeElapsed = Duration.between(start, end);

long millis = timeElapsed.toMillis();

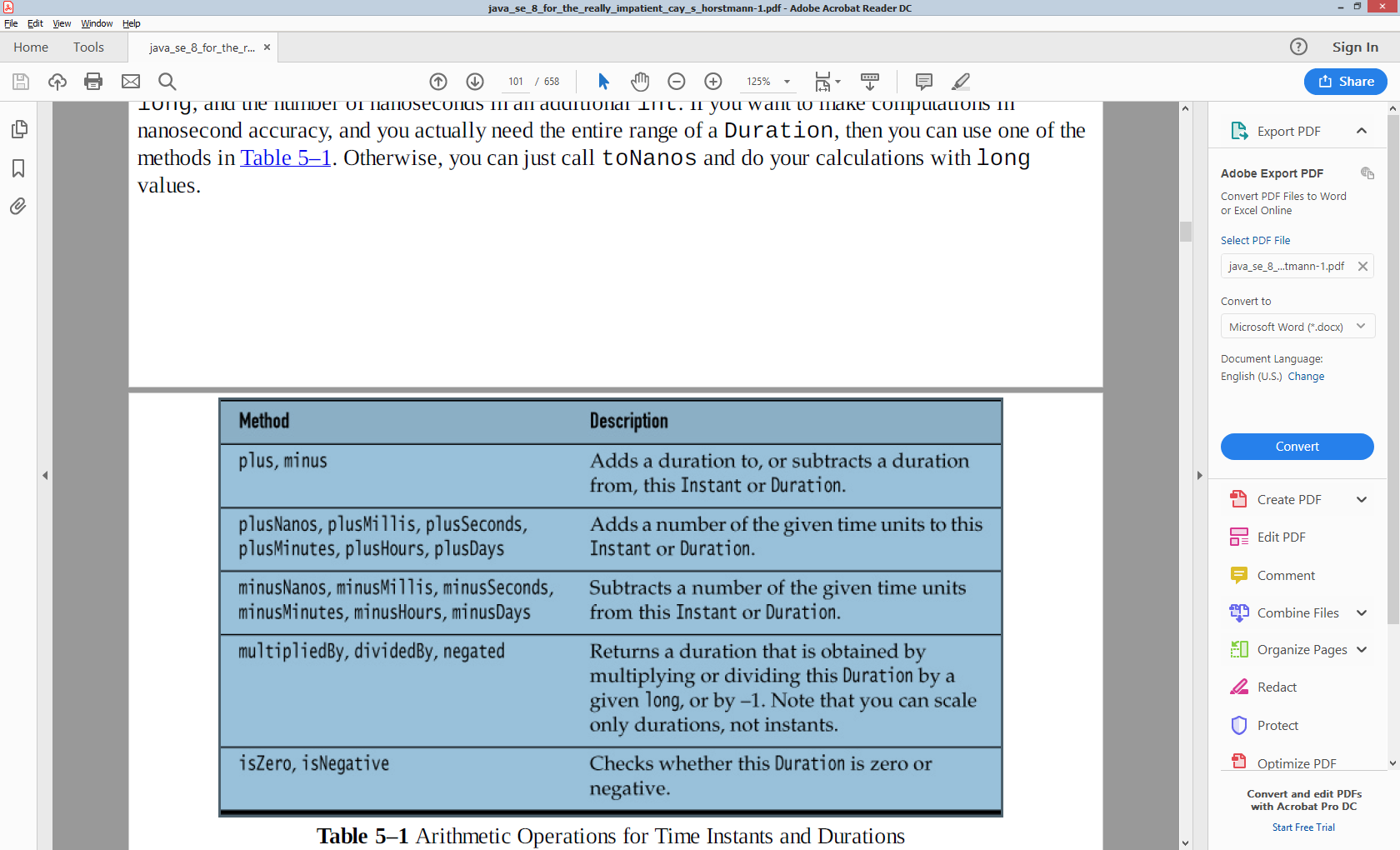
Duration timeElapsed2 = Duration.between(start2, end2);

boolean overTenTimesFaster =

timeElapsed.multipliedBy(10).minus(timeElapsed2).isNegative();

// Or timeElapsed.toNanos() \* 10 < timeElapsed2.toNanos()

The Instant and Duration classes are immutable, and all methods, such as multipliedBy or minus, return a new instance.



LocalDate / ZonedDateTime

There are two kinds of human time in the new Java API, local date/time and zoned time. Local date/time has a date and/or time of day, but no associated time zone information. Local date/time has a date and/or time of day, but no associated time zone information. A local date is, for example, June 14, 1903 (the day on which Alonzo Church,inventor of the lambda calculus, was born). Since that date has neither a time of day nor time zone information, it does not correspond to a precise instant of time. In contrast, July 16, 1969, 09:32:00 EDT (the launch of Apollo 11) is a zoned date/time, representing a precise instant on the time line. There are many calculations where time zones are not required, and in some cases they can even be a hindrance. Suppose you schedule a meeting every week at 10:00. If you add 7 days (that is, 7 × 24 × 60 ×60 seconds) to the last zoned time, and you happen to cross the daylight savings time boundary, the meeting will be an hour too early or too late!

For that reason, the API designers recommend that you do not use zoned time unless you really want to represent absolute time instances. Birthdays, holidays, schedule times, and so on are usually best represented as local dates or times.

A LocalDate is a date, with a year, month, and day of the month. To construct one, you can use the now or of static methods:

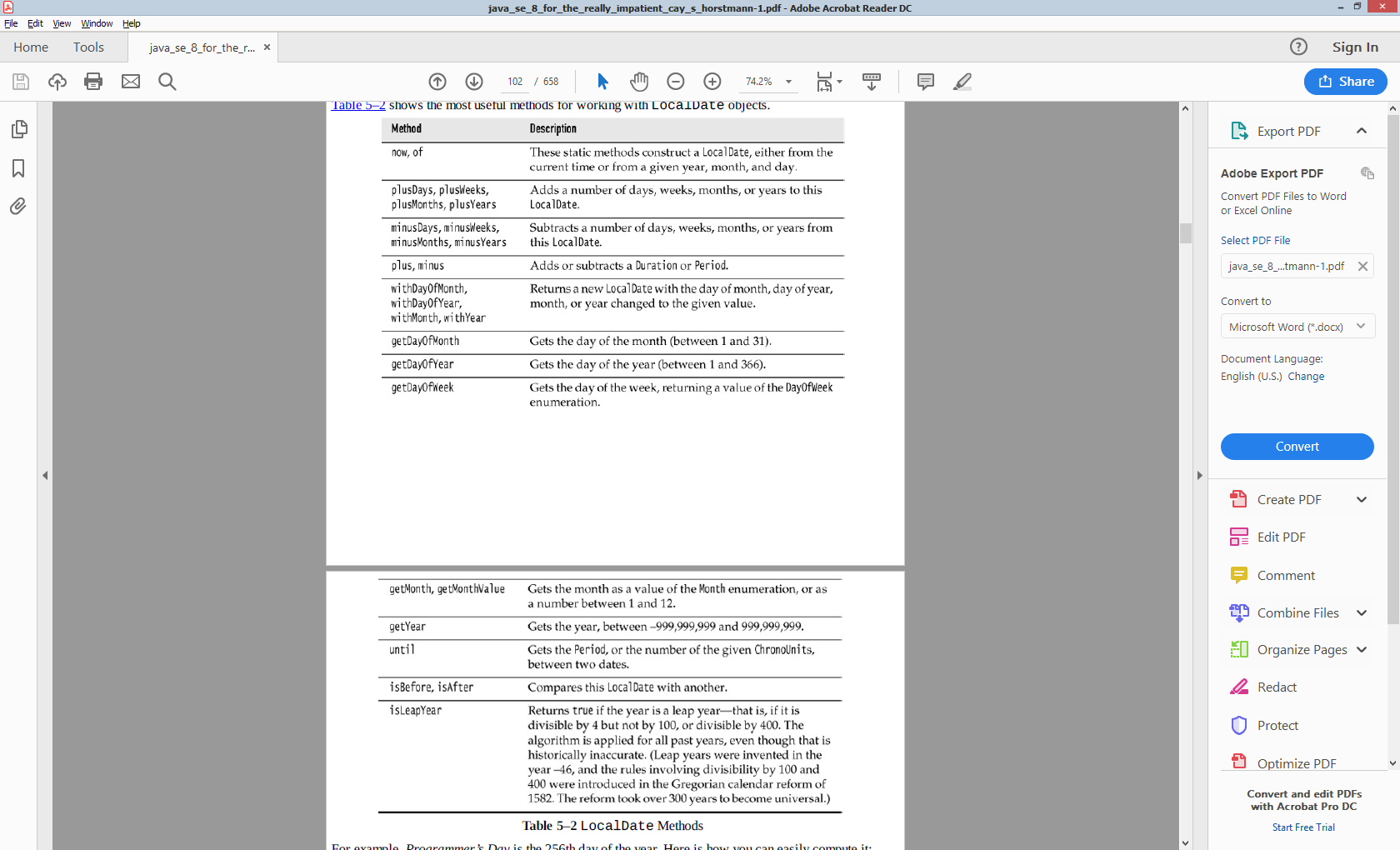
LocalDate today = LocalDate.now(); // Today's date

LocalDate alonzosBirthday = LocalDate.of(1903, 6, 14);

alonzosBirthday = LocalDate.of(1903, Month.JUNE, 14);

// Uses the Month enumeration

Unlike the irregular conventions in Unix and java.util.Date, where months are zero-based and years are counted from 1900, you supply the usual numbers for the month of year. Alternatively, you can use the Month enumeration.



For example, Programmer’s Day is the 256th day of the year. Here is how you can easily compute it:

LocalDate programmersDay = LocalDate.of(2014, 1, 1).plusDays(255);

// September 13, but in a leap year it would be September 12

Recall that the difference between two time instants is a Duration. The equivalent for local dates is a Period, which expresses a number of elapsed years, months, or days. You can call birthday.plus(Period.ofYears(1)), to get the birthday next year. Of course, you can also just call birthday.plusYears(1). But birthday.plus(Duration.ofDays(365)) won’t produce the correct result in a leap year. The until method yields the difference between two local dates.

independenceDay.until(christmas) yields a period of 5 months and 21 days. That is actually not terribly useful because the number of days per month varies. To find the number of days, use independenceDay.until(christmas, ChronoUnit.DAYS) // 174 days

Some methods in Table 5–2 could potentially create nonexistent dates. For example, adding one month to January 31 should not yield February 31. Instead of throwing an exception, these methods return the last valid day of the month. For example, LocalDate.of(2016, 1, 31).plusMonths(1) and LocalDate.of(2016, 3, 31).minusMonths(1) yield February 29, 2016.

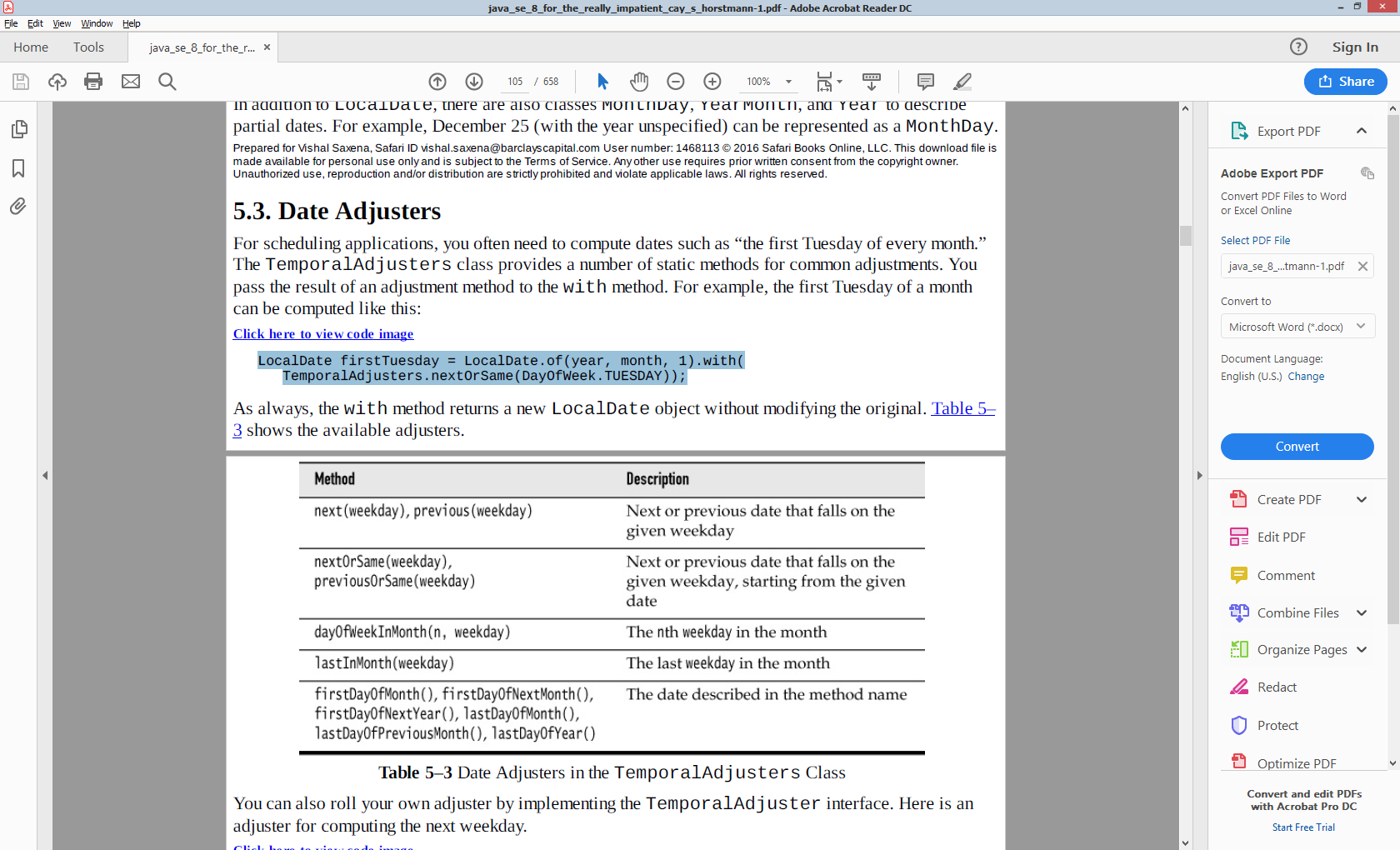
The getDayOfWeek yields the weekday, as a value of the DayOfWeek enumeration. DayOfWeek.MONDAY has the numerical value 1, and DayOfWeek.SUNDAY has the value 7.

LocalDate.of(1900, 1, 1).getDayOfWeek().getValue() yields 1. The DayOfWeek enumeration has convenience methods plus and minus to compute weekdays modulo 7. For example, DayOfWeek.SATURDAY.plus(3) yields

DayOfWeek.TUESDAY. The weekend days actually come at the end of the week. This is different from java.util.Calendar, where Sunday has value 1 and Saturday value 7. In addition to LocalDate, there are also classes MonthDay, YearMonth, and Year to describe partial dates. For example, December 25 (with the year unspecified) can be represented as a MonthDay.

For scheduling applications, you often need to compute dates such as “the first Tuesday of every month.” The TemporalAdjusters class provides a number of static methods for common adjustments. You pass the result of an adjustment method to the with method.

LocalDate firstTuesday = LocalDate.of(year, month, 1).with(TemporalAdjusters.nextOrSame(DayOfWeek.TUESDAY));



You can also roll your own adjuster by implementing the TemporalAdjuster interface. Here is an adjuster for computing the next weekday.

TemporalAdjuster NEXT\_WORKDAY = w -> {

LocalDate result = (LocalDate) w;

do {

result = result.plusDays(1);

} while (result.getDayOfWeek().getValue() >= 6);

return result;

};

LocalDate backToWork = today.with(NEXT\_WORKDAY);

Note that the parameter of the lambda expression has type Temporal, and it must be cast to LocalDate. You can avoid this cast with the ofDateAdjuster method that expects a lambda of type UnaryOperator<LocalDate>.

TemporalAdjuster NEXT\_WORKDAY = TemporalAdjusters.ofDateAdjuster(w -> {

LocalDate result = w; // No cast

do {

result = result.plusDays(1);

} while (result.getDayOfWeek().getValue() >= 6);

return result;

});

A LocalTime represents a time of day, such as 15:30:00. You can create an instance with the now or of methods:

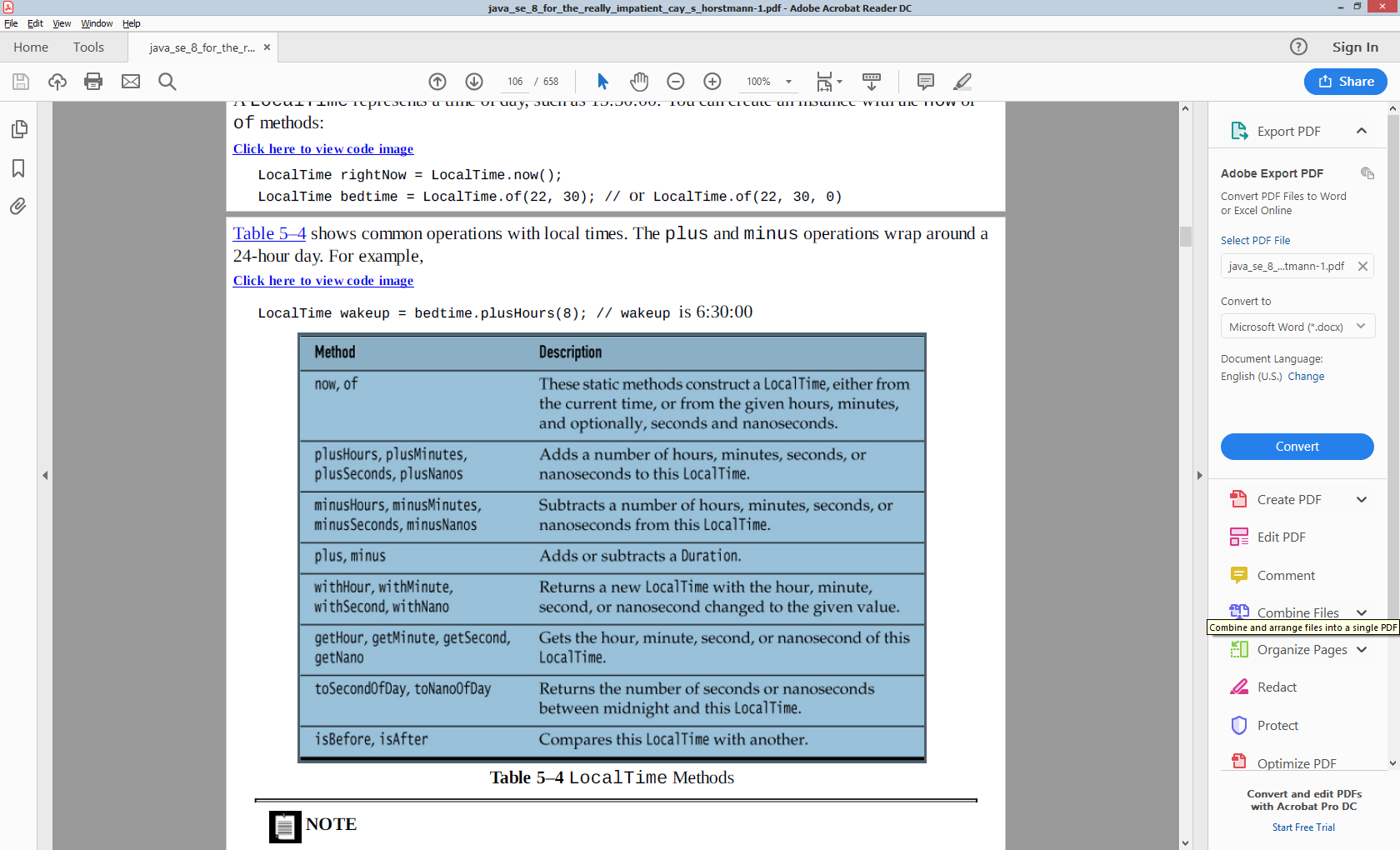
LocalTime rightNow = LocalTime.now();

LocalTime bedtime = LocalTime.of(22, 30); // or LocalTime.of(22, 30, 0)

LocalTime wakeup = bedtime.plusHours(8); // wakeup is 6:30:00

Table 5–4 LocalTime Methods

LocalTime doesn’t concern itself with AM/PM. That silliness is left to a formatter There is a LocalDateTime class, representing a date and time. That class is suitable for storing points in time in a fixed time zone, for example, for a schedule of classes or events. However, if you need to make calculations that span the daylight savings time, or if you need to deal with users in different time zones, you should use the ZonedDateTime class that we discuss next.



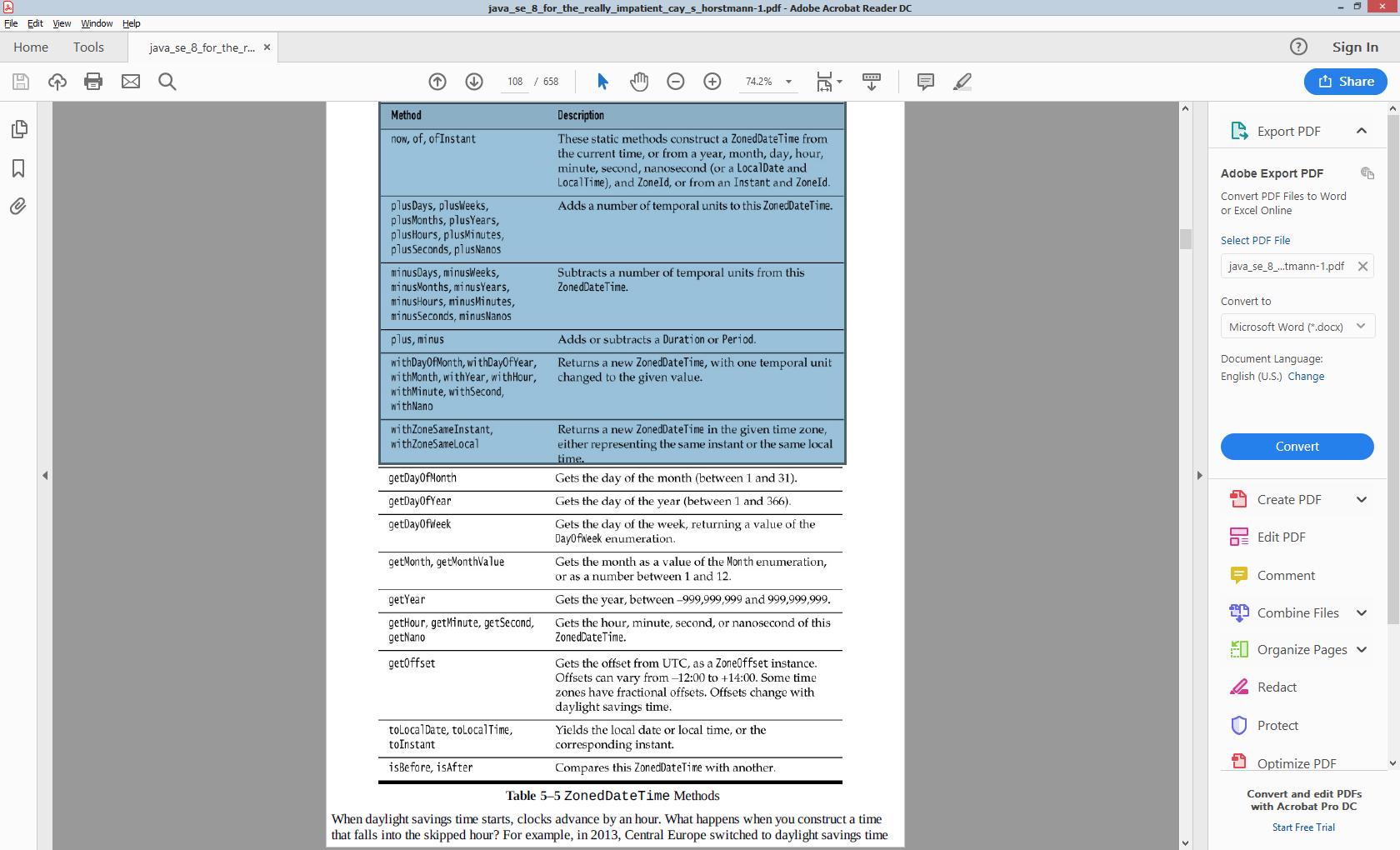
Given a time zone ID, the static method ZoneId.of(id) yields a ZoneId object. You can use that object to turn a LocalDateTime object into a ZonedDateTime object by calling local.atZone(zoneId), or you can construct a ZonedDateTime by calling the static method ZonedDateTime.of(year, month, day, hour, minute, second, nano, zoneId). For example,

ZonedDateTime apollo11launch = ZonedDateTime.of(1969, 7, 16, 9, 32, 0, 0,ZoneId.of("America/New\_York"));

// 1969-07-16T09:32-04:00[America/New\_York]

This is a specific instant in time. Call apollo11launch.toInstant to get the Instant. Conversely, if you have an instant in time, call instant.atZone(ZoneId.of("UTC")) to get the ZonedDateTime at the Greenwich Royal Observatory, or use another ZoneId to get it elsewhere on the planet.

Many of the methods of ZonedDateTime are the same as those of LocalDateTime. Most are straightforward, but daylight savings time introduces some complications. See table below:



When daylight savings time starts, clocks advance by an hour. What happens when you construct a time that falls into the skipped hour? For example, in 2013, Central Europe switched to daylight savings time on March 31 at 2:00. If you try to construct nonexistent time March 31 2:30, you actually get 3:30.

ZonedDateTime skipped = ZonedDateTime.of(LocalDate.of(2013, 3, 31),LocalTime.of(2, 30),ZoneId.of("Europe/Berlin"));

Conversely, when daylight time ends, clocks are set back by an hour, and there are two instants with the same local time! When you construct a time within that span, you get the earlier of the two.

ZonedDateTime ambiguous = ZonedDateTime.of(LocalDate.of(2013, 10, 27), // End of daylight savings time

LocalTime.of(2, 30),ZoneId.of("Europe/Berlin"));

// 2013-10-27T02:30+02:00[Europe/Berlin]

ZonedDateTime anHourLater = ambiguous.plusHours(1);

// 2013-10-27T02:30+01:00[Europe/Berlin]

An hour later, the time has the same hours and minutes, but the zone offset has changed. You also need to pay attention when adjusting a date across daylight savings time boundaries. For example, if you set a meeting for next week, don’t add a duration of seven days:

ZonedDateTime nextMeeting = meeting.plus(Duration.ofDays(7));

// Caution! Won't work with daylight savings time

Instead, use the Period class.

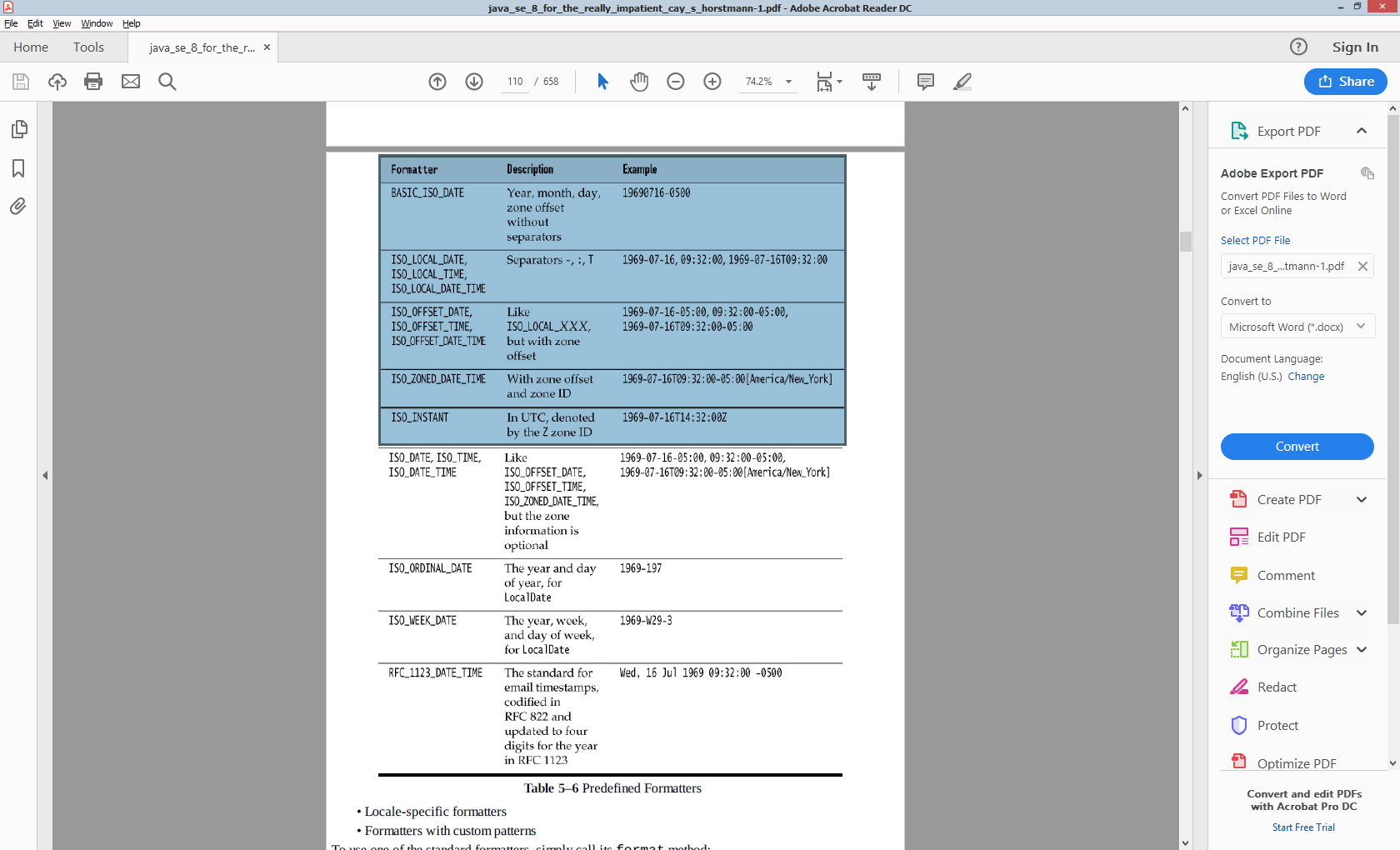
ZonedDateTime nextMeeting = meeting.plus(Period.ofDays(7)); // OK

There is also an OffsetDateTime class that represents times with an offset from UTC,but without time zone rules. That class is intended for specialized applications that specifically require the absence of those rules, such as certain network protocols. For human time, use ZonedDateTime.

*DateTimeFormatter*

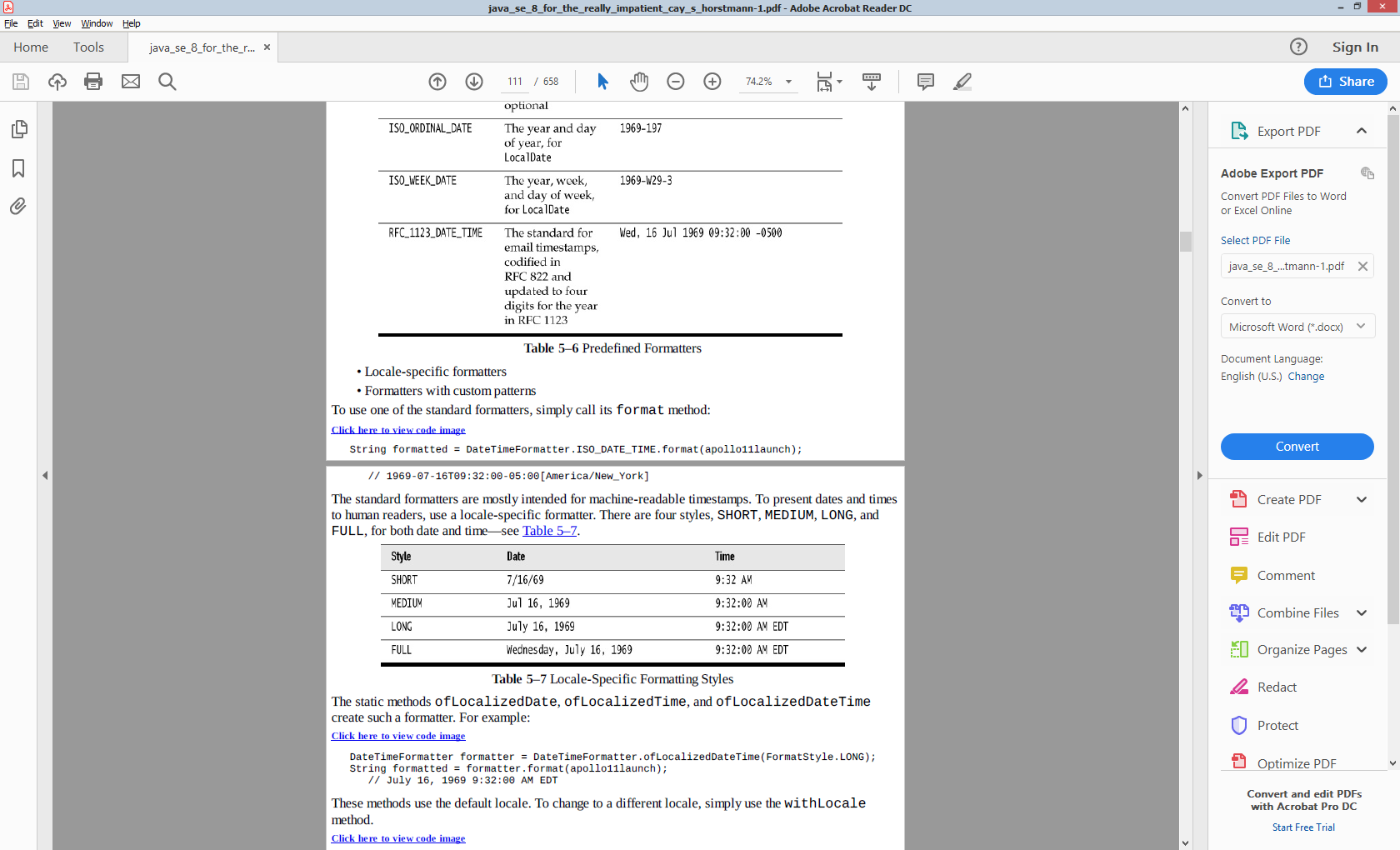
three kinds of formatters to print a date/time value:

* Predefined standard formatters The standard formatters are mostly intended for machine-readable timestamps.



String formatted = DateTimeFormatter.ISO\_DATE\_TIME.format(apollo11launch);

* locale-specific formatters, to present to humans



The static methods ofLocalizedDate, ofLocalizedTime, and ofLocalizedDateTime

create such a formatter.

DateTimeFormatter formatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG);

String formatted = formatter.format(apollo11launch);

// July 16, 1969 9:32:00 AM EDT

These methods use the default locale. To change to a different locale, simply use the withLocale method.

formatted = formatter.withLocale(Locale.FRENCH).format(apollo11launch);

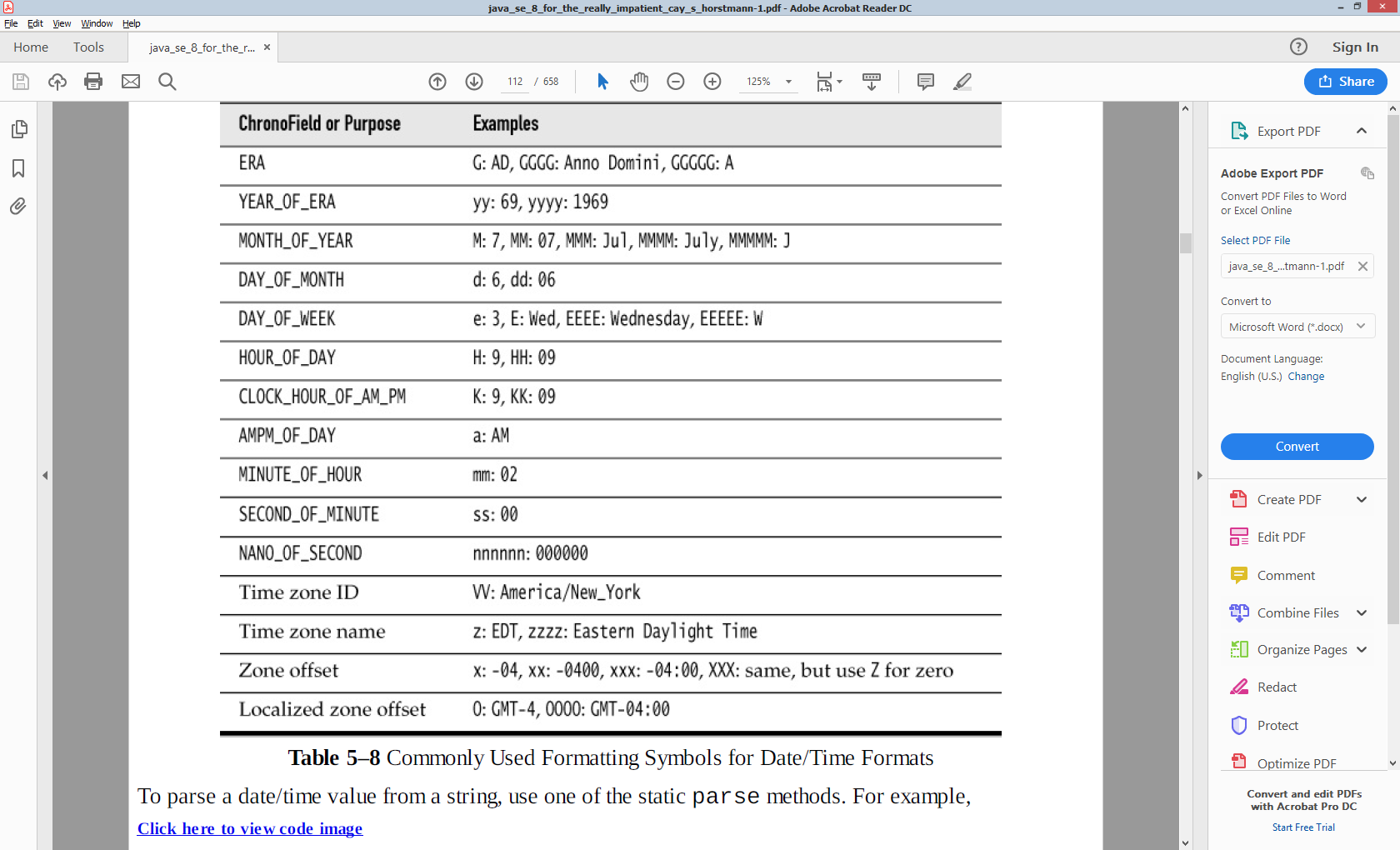
// 16 juillet 1969 09:32:00 EDT

The java.time.format.DateTimeFormatter class is intended as a replacement for java.util.DateFormat. If you need an instance of the latter for backwards compatibility, call formatter.toFormat().

* roll your own date format by specifying a pattern.

formatter = DateTimeFormatter.ofPattern("E yyyy-MM-dd HH:mm");

formats a date in the form Wed 1969-07-16 09:32. Each letter denotes a different time field, and the number of times the letter is repeated selects a particular format, according to rules that are arcane and seem to have organically grown over time.



To parse a date/time value from a string, use one of the static parse methods.

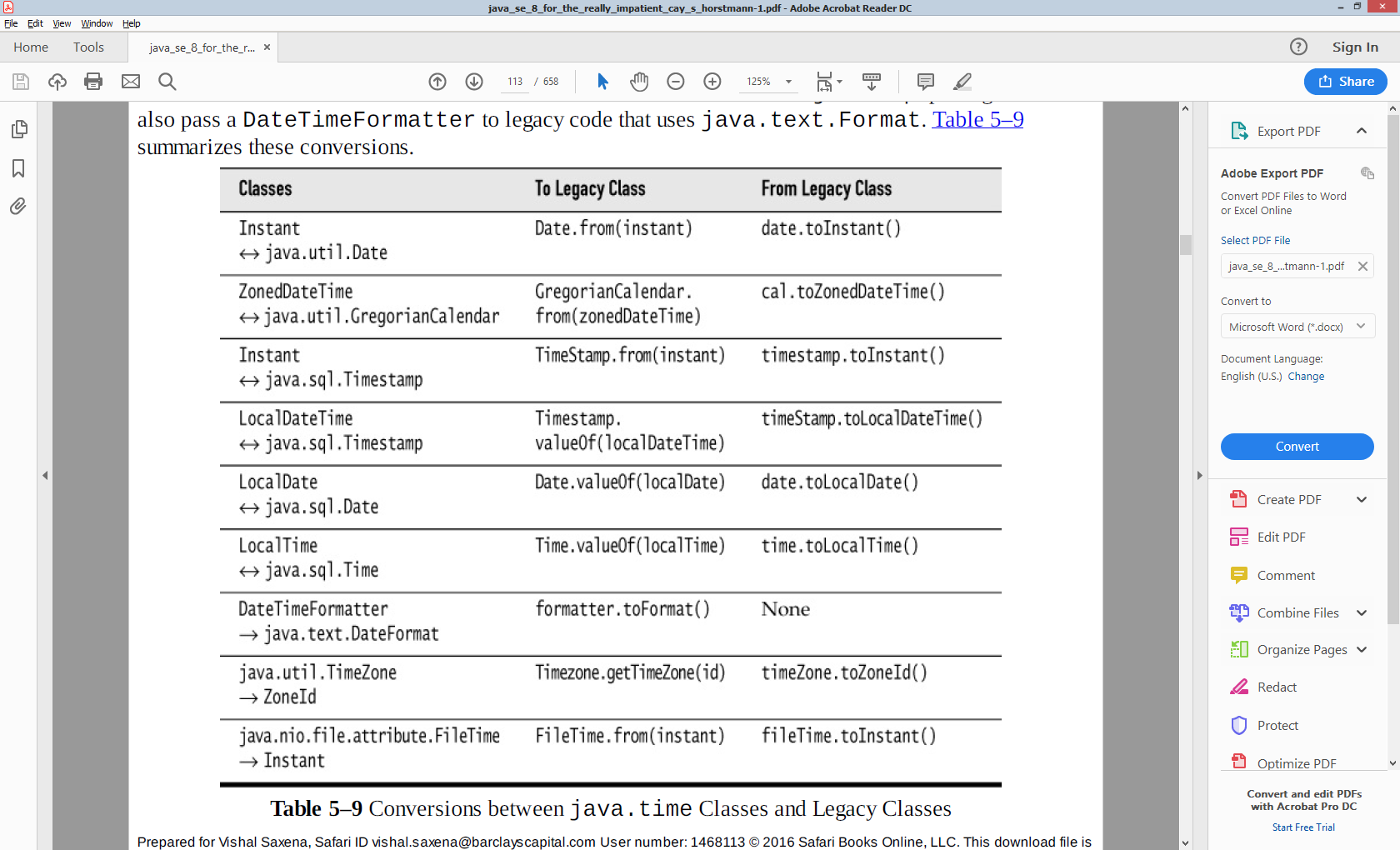
LocalDate churchsBirthday = LocalDate.parse("1903-06-14");

ZonedDateTime apollo11launch =

ZonedDateTime.parse("1969-07-16 03:32:00-0400",

DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ssxx"));

The first call uses the standard ISO\_LOCAL\_DATE formatter, the second one a custom formatter.



Concurrency

Early Java releases had minimal support for concurrency, and programmers busily created code with deadlocks and race conditions. The

robust java.util.concurrent package didn’t appear until Java 5. That package gives us threadsafe collections and thread pools, allowing many application programmers to write concurrent programs without using locks or starting threads. Unfortunately, java.util.concurrent is a mix of useful utilities for the application programmer and power tools for library authors, without much effort to separate the two.

Runnable, Callable, Future Executors & ExecutorService

public interface Runnable {

void run();

}

public interface Callable<V> {

V call() throws Exception;

}

public interface Supplier<V>{

V get();

}// used in CompleteableFuture factory methods.

Runnable task = () -> { ... }; ExecutorService executor = ...; executor.execute(task);

public interface Callable<V> {

V call() throws Exception;

}

ExecutorService executor = Executors.newFixedThreadPool(); Callable<V> task = ...; Future<V> result = executor.submit(task);

Executor has a single method: void execute(Runnable command)

ExecutorService extends Executor

However submit does awkwardly takes a Runnable too, unlike an execute which only takes Runnable.

<T> Future<T> submit(Callable<T> task)

Future<?> submit(Runnable task)

Submits a Runnable task for execution and returns a Future representing that task. Future.get returns a null

<T> Future<T> submit(Runnable task, T result)

Submits a Runnable task for execution and returns a Future representing that task.

Future<V>

V get() throws InterruptedException, ExecutionException

V get(long timeout, TimeUnit unit)throws InterruptedException, ExecutionException, TimeoutException;

boolean cancel(boolean mayInterruptIfRunning)

boolean isCancelled() //false if call throws an Exception

boolean isDone() //false if call throws an Exception

The get method blocks until it returns, or throws an exception, or a timeout happens. If the call returns a value get returns, if call throws the checked exception, get wraps it into an ExecutionException, if time out happens it throws a TimeOutException.

The cancel method attempts to cancel the task – if not scheduled won’t schedule it, if mayInterruptIfRunning is true the thread running the task is interrupted (i.e. co-operative interrupt status is set)

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

Executes the given tasks, returning a list of Futures holding their status and results when all complete or the timeout expires, whichever happens first. Future.isDone() is true for each element of the returned list. Upon return, tasks that have not completed are cancelled. Note that a completed task could have terminated either normally or by throwing an exception. The results of this method are undefined if the given collection is modified while this operation is in progress.

<T> T invokeAny(Collection<? extends Callable<T>> tasks)

<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)

Executes the given tasks, returning the result of one that has completed successfully (i.e., without throwing an exception), if any do before the given timeout elapses. Upon normal or exceptional return, tasks that have not completed are cancelled. The results of this method are undefined if the given collection is modified while this operation is in progress.

Here’s a short summary of how ExecutorServices that are created by the factory methods in the Executors class differ from each other. I hope this will shed some light on how do they resolve the important questions above. **newFixedThreadPool(int nThreads)** – n threads will process tasks at the time, when the pool is saturated, new tasks will get added to a queue without a limit on size. Good for CPU intensive tasks. *The ExecutorService though has to be expilcitly shutdown*. **newWorkStealingPool(int parallelism)** – will create and shut down threads dynamically to accommodate the required parallelism level. It also tries to reduce the contention on the task queue, so can be really good in heavily loaded environments. Also good when your tasks create more tasks for the executor, like recursive tasks. **newSingleThreadExecutor()** – creates an unconfigurable `newFixedThreadPool(1)`, so you know that only one thread will process everything. Good when you really need predictability and sequential tasks completion. **newCachedThreadPool()** – doesn’t put tasks into a queue. Consider this as the same as using a queue with the maximum size of 0. When all current threads are busy, it creates another thread to run the task. Sometimes it can reuse threads. Good for denial of service attacks on your own servers. The problem with a cached thread pool is that it doesn’t know when to stop spawning more and more threads. Imagine the situation where you have computationally intensive tasks that you submit into this executor. The more threads that consuming the CPU, the slower every individual task takes to process. This has a domino effect in that it means more work gets backlogged. As the result you’ll end up with more and more threads spawned making task processing even slower. This negative feedback loop is a hard problem to solve. *ExcecutorService though after a little while shutsdown on it’s own.* So for almost all intents and purposes, **Executors::newFixedThreadPool(int nThreads)** should be your goto choice for when you need a thread pool. For computationally intensive tasks it will probably give you close to optimal throughput and for IO heavy tasks you won’t be that much worse than anything else. At least until you profile that you got a problem with this kind of executor, you shouldn’t mindlessly try to optimize it.

CompletebleFutures

Extract/modify wrapped value

Typically futures represent piece of code running by other thread. But that's not always the case. Sometimes you want to create a Future representing some event that you know will occur, e.g. JMS message arrival. So you have Future<Message> but there is no asynchronous job underlying this future. You simply want to complete (resolve) that future when JMS message arrives, and this is driven by an event. In this case you can simply create CompletableFuture, return it to your client and whenever you think your results are available, simply complete() the future and unlock all clients waiting on that future.

public CompletableFuture<String> ask() {

final CompletableFuture<String> future = new CompletableFuture<>();

//...return future;

}

Notice that this future is not associated wtih any Callable<String>, no thread pool, no asynchronous job. If now the client code calls ask().get() it will block forever. If it registers some completion callbacks, they will never fire. So what's the point? Now you can say: future.complete("42") ...and at this very moment all clients blocked on Future.get() will get the result string. Also completion callbacks will fire immediately. This comes quite handy when you want to represent a task in the future, but not necessarily computational task running on some thread of execution. CompletableFuture.complete() can only be called once, subsequent invocations are ignored. But there is a back-door called CompletableFuture.obtrudeValue(...) which overrides previous value of the Future with new one. Use with caution. Sometimes you want to signal failure. As you know Future objects can handle either wrapped result or exception. If you want to pass some exception further, there is CompletableFuture.completeExceptionally(ex) (and obtrudeException(ex) evil brother that overrides the previous exception). completeExceptionally() also unlock all waiting clients, but this time throwing an exception from get(). And finally there is also CompletableFuture.getNow(valueIfAbsent) method that doesn't block but if the Future is not completed yet, returns default value.

Creating and obtaining CompletableFuture

OK, so is creating CompletableFuture manually our only option? Not quite. Just as with normal Futures we can wrap existing task with CompletableFuture using the following family of factory methods:

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier);

static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor);

static CompletableFuture<Void> runAsync(Runnable runnable);

static CompletableFuture<Void> runAsync(Runnable runnable, Executor executor);

Methods that do not take an Executor as an argument but end with ...Async will use ForkJoinPool.commonPool() (global, general purpose pool introduces in JDK 8).

final CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> { //...long running... return "42";}, executor);

Transforming and acting on one CompletableFuture (thenApply)

CompletableFuture is superior to Future but you haven't yet seen why? It allows just like as in Scala and JavaScript registering asynchronous callbacks when future is completed. We don't have to wait and block until it's ready. We can simply say: run this function on a result, when it arrives. Moreover, we can stack such functions, combine multiple futures together, etc. For example if we have a function from String to Integer we can turn CompletableFuture<String> to CompletableFuture<Integer without unwrapping it. This is achieved with thenApply() family of methods:

<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn);

<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn);

<U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor);

As stated before ...Async versions are provided for most operations on CompletableFuture - that first method will apply function within the same thread in which the future completed while the remaining two will apply it asynchronously in different thread pool.

CompletableFuture<String> f1 = //...

CompletableFuture<Integer> f2 = f1.thenApply(Integer::parseInt);

CompletableFuture<Double> f3 = f2.thenApply(r -> r \* r \* Math.PI);

or in one statement CompletableFuture<Double> f3 = f1.thenApply(Integer::parseInt).thenApply(r -> r \* r \* Math.PI);

You see a sequence of transformations here. From String to Integer and then to Double. But what's most important, these transformations are neither executed immediately nor blocking. They are simply remembered and when original f1 completes they are executed for you. If some of the transformations are time-consuming, you can supply your own Executor to run them asynchronously. Notice that this operation is equivalent to monadic map in Scala. (>>=) :: (Monad m) => m a -> (a -> m b) -> m b Running code on completion (thenAccept/thenRun)

CompletableFuture<Void> thenAccept(Consumer<? super T> block); CompletableFuture<Void> thenRun(Runnable action);

These two methods are typical "final" stages in future pipeline. They allow you to consume future value when it's ready. While thenAccept() provides the final value, thenRun executes Runnable which doesn't even have access to computed value. Example: future.thenAcceptAsync(dbl -> log.debug("Result: {}", dbl), executor); log.debug("Continuing");

...Async variants are available as well for both methods, with implicit and explicit executor. I can't emphasize this enough: thenAccept()/thenRun() methods do not block (even without explicit executor). Treat them like an event listener/handler that you attach to a future and that will execute some time in the future. "Continuing" message will appear immediately, even if future is not even close to completion.

Error handling of single CompletableFuture:

So far we only talked about result of computation. But what about exceptions? Can we handle them asynchronously as well? Sure!

CompletableFuture<String> safe = future.exceptionally(ex -> "We have a problem: " + ex.getMessage());

exceptionally() takes a function that will be invoked when original future throws an exception. We then have an opportunity to recover by transforming this exception into some value compatible with Future's type. Further transformations of safe will no longer yield an exception but instead a String returned from supplied function. A more flexible approach is handle() that takes a function receiving either correct result or exception: handle() is called always, with either result or exception argument being not-null. This is a one-stop catch-all strategy.

CompletableFuture<Integer> safe = future.handle((ok, ex) -> { if (ok != null) {return Integer.parseInt(ok); } else {

log.warn("Problem", ex); return -1; }});

Combining two CompletableFuture together

Asynchronous processing of one CompletableFuture is nice but it really shows its power when multiple such futures are combined together in various ways.

Combining (chaining) two futures(thenCompose())

Sometimes you want to run some function on future's value (when it's ready). But this function returns future as well. CompletableFuture should be smart enough to understand that the result of our function should now be used as top-level future, as opposed to CompletableFuture<CompletableFuture<T>>. Method thenCompose() is thus equivalent to flatMap in Scala:

<U> CompletableFuture<U> thenCompose(Function<? super T,CompletableFuture<U>> fn);

...Async variations are available as well. Example below, look carefully at the types and the difference between thenApply() (map) and thenCompose() (flatMap) when applying a calculateRelevance() function returning CompletableFuture<Double>:

CompletableFuture<Document> docFuture = //...

CompletableFuture<CompletableFuture<Double>> f =

docFuture.thenApply(this::calculateRelevance);

CompletableFuture<Double> relevanceFuture =

docFuture.thenCompose(this::calculateRelevance);

private CompletableFuture<Double> calculateRelevance(Document doc) //...

thenCompose() is an essential method that allows building robust, asynchronous pipelines, without blocking or waiting for intermediate steps.

Transforming values of two futures (thenCombine())

While thenCompose() is used to chain one future dependent on the other, thenCombine combines two independent futures when they are both done:

<U,V> CompletableFuture<V> thenCombine(CompletableFuture<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)

...Async variations are available as well. Imagine you have two CompletableFutures, one that loads Customer and other that loads nearest Shop. They are completely independent from each other,but when both of them are completed, you want to use their values to calculate Route. Here is a stripped example:

CompletableFuture<Customer> customerFuture = loadCustomerDetails(123);

CompletableFuture<Shop> shopFuture = closestShop();

CompletableFuture<Route> routeFuture =

customerFuture.thenCombine(shopFuture, (cust, shop) -> findRoute(cust, shop));

So you get the idea. We have customerFuture and shopFuture. Then routeFuture wraps them and "waits" for both to complete. When both of them are ready, it runs our supplied function that combines results (findRoute()). Thus routeFuture will complete when two underlying futures are resolved and findRoute() is done. //...private Route findRoute(Customer customer, Shop shop) //...

Waiting for both CompletableFutures to complete

If instead of producing new CompletableFuture combining both results we simply want to be notified when they finish, we can use thenAcceptBoth()/runAfterBoth() family of methods (...Async variations are available as well). They work similarly to thenAccept() and thenRun() but wait for two futures instead of one:

<U> CompletableFuture<Void> thenAcceptBoth(CompletableFuture<? extends U> other, BiConsumer<? super T,? super U> block)

CompletableFuture<Void> runAfterBoth(CompletableFuture<?> other, Runnable action) Imagine that in the example above, instead of producing new CompletableFuture<Route> you simply want send some event or refresh GUI immediately. This can be easily chieved with thenAcceptBoth(): customerFuture.thenAcceptBoth(shopFuture, (cust, shop) -> { final Route route = findRoute(cust, shop); //refresh GUI with route});

Why can't I simply block on these two futures? Like here:

Future<Customer> customerFuture = loadCustomerDetails(123);

Future<Shop> shopFuture = closestShop();

findRoute(customerFuture.get(), shopFuture.get());

Well, of course you can. But the whole point of CompletableFuture is to allow asynchronous, event driven programming model instead of blocking and eagerly waiting for result. So functionally two code snippets above are equivalent, but the latter unnecessarily occupies one thread of execution.

Waiting for first CompletableFuture to complete:

Another interesting part of the CompletableFuture API is the ability to wait for first (as opposed to all) completed future. This can come handy when you have two tasks yielding result of the same type and you only care about response time, not which task resulted first. API methods (...Async variations are available as well):

As an example say you have two systems you integrate with. One has smaller average response times but high standard deviation. Other one is slower in general, but more predictable. In order to take best of both worlds (performance and predictability) you call both systems at the same time and wait for the first one to complete. Normally it will be the first one, but in case it became slow, second one finishes in an acceptable time:

CompletableFuture<String> fast = fetchFast();

CompletableFuture<String> predictable = fetchPredictably(); fast.acceptEither(predictable, s -> {System.out.println("Result: " + s); });

s represents String reply either from fetchFast() or from fetchPredictably(). We neither know nor care.

Transforming first completed:

applyToEither() is an older brother of acceptEither(). While the latter simply calls some piece of code when faster of two futures complete, applyToEither() will return a new future. This future will complete when first of the two underlying futures complete. API is a bit similar (...Async variations are available as well):

<U> CompletableFuture<U> applyToEither(CompletableFuture<? extends T> other, Function<? super T,U> fn).

The extra fn function is invoked on the result of first future that completed. I am not really sure what's the purpose of such a specialized method, after all one could simply use: fast.applyToEither(predictable).thenApply(fn). Since we are stuck with this API but we don't really need extra function application, I will simply use Function.identity() placeholder:

CompletableFuture<String> fast = fetchFast();

CompletableFuture<String> predictable = fetchPredictably();

CompletableFuture<String> firstDone = fast.applyToEither(predictable, Function.<String>identity());

firstDone future can then be passed around. Notice that from the client perspective the fact that two futures are actually behind firstDone is hidden. Client simply waits for future to complete and applyToEither() takes care of notifying the client when any of the two finish first.

Combining multiple CompletableFuture together:

So we now know how to wait for two futures to complete (using thenCombine()) and for the first one to complete (applyToEither()). But can it scale to arbitrary number of futures? Sure, using static helper methods:

static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)

static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)

allOf() takes an array of futures and returns a future that completes when all of the underlying futures are completed (barrier waiting for all). anyOf() on the other hand will wait only for the fastest of the underlying futures. Please look at the generic type of returned futures. Not quite what you would expect? We will take care of this issue in the next article.

Simply put, it's because CompletableFuture is a monad and a functor. Wrapped value is a way to visualize context.

Functors apply a function to a wrapped value. fmap :: (Functor f) => (a -> b) -> f a -> f b

Applicatives apply a wrapped function to a wrapped value. (<\*>) :: (Applicative f) => f (a -> b) -> f a -> f b

Monads apply a function that returns a wrapped value to a wrapped value. (>>=) :: (Monad m) => m a -> (a -> m b) -> m b

ThreadInterruption

Task cancellation is cooperative. Each thread has an interrupted status that indicates that someone would like to “interrupt” the thread. There is no precise definition of what interruption means, but most programmers use it to indicate a cancellation request. A Runnable can check for this status, which is typically done in a loop. When the thread is interrupted, the run method simply ends. Sometimes, a thread becomes temporarily inactive. That can happen if a thread waits for a value to be computed by another thread or for input/output, or if it goes to sleep to give other threads a chance. If the thread is interrupted while it waits or sleeps, it is immediately reactivated—but in this case, the interrupted status is not set. Instead, an InterruptedException is thrown. This is a checked exception, and you must catch it inside the run method of a Runnable. The usual reaction to the exception is to end the run method:

Runnable task = () -> {

try {

while (more work todo) {

if (Thread.currentThread().isInterrupted()) return;

Thread.sleep(millis);

}

}

catch (InterruptedException ex) {

Thread.currentThread().interrupt();

}

};

When you catch the InterruptedException in this way, there is no need to check for the interrupted status. If the thread was interrupted outside the call to Thread.sleep, the status is set and the

Thread.sleep method throws an InterruptedException as soon as it is called.

public void interrupt()

Interrupts this thread. Unless the current thread is interrupting itself, which is always permitted, the checkAccess method of this thread is invoked, which may cause a SecurityException to be thrown. If this thread is blocked in an invocation of the wait(), wait(long), or wait(long, int) methods of the Object class, or of the join(), join(long), join(long, int), sleep(long), or sleep(long, int), methods of this class, then its interrupt status will be cleared and it will receive an InterruptedException. If this thread is blocked in an I/O operation upon an InterruptibleChannel then the channel will be closed, the thread's interrupt status will be set, and the thread will receive a ClosedByInterruptException. If this thread is blocked in a Selector then the thread's interrupt status will be set and it will return immediately from the selection operation, possibly with a non-zero value, just as if the selector's wakeup method were invoked. If none of the previous conditions hold then this thread's interrupt status will be set. Interrupting a thread that is not alive need not have any effect.

public static boolean interrupted()

Tests whether the current thread has been interrupted. The interrupted status of the thread is cleared by this method. In other words, if this method were to be called twice in succession, the second call would return false (unless the current thread were interrupted again, after the first call had cleared its interrupted status and before the second call had examined it). A thread interruption ignored because a thread was not alive(there has not been a t.start() yet or the t.run method has ended) at the time of the interrupt will be reflected by this method returning false.

public boolean isInterrupted()

Tests whether this thread has been interrupted. The interrupted status of the thread is unaffected by this method. A thread interruption ignored because a thread was not alive at the time of the interrupt will be reflected by this method returning false.

ThreadLocal

For example, the NumberFormat class is not threadsafe. Suppose we have a static variable public static final NumberFormat currencyFormat = NumberFormat.getCurrencyInstance();

If two threads execute an operation such as String amountDue = currencyFormat.format(total); then the result can be garbage since the internal data structures used by the NumberFormat instance can be corrupted by concurrent access. You could use a lock or provide a synchronized method to ensure atomic access to the shared NumberFormat variable. Alternatively, you could construct a local NumberFormat object whenever you need it, but that is also wasteful.

public static final ThreadLocal<NumberFormat> currencyFormat = ThreadLocal.withInitial(() -> NumberFormat.getCurrencyInstance());

String amountDue = currencyFormat.get().format(total);

Thread Properties

When a thread terminates due to an uncaught exception, the exception is passed to the thread's uncaught exception handler. By default, its stack trace is dumped to System.err, but you can install your own handler. A daemon is a thread that has no other role in life than to serve others. This is useful for threads that send timer ticks or clean up stale cache entries. When only daemon threads remain, the virtual machine exits.

Process

Up to now, you have seen how to execute Java code in separate threads within the same program. Sometimes, you need to execute another program. For this, use the ProcessBuilder and Process classes. The Process class executes a command in a separate operating system process and lets you interact with its standard input, output, and error streams. The ProcessBuilder class lets you configure a Process object. The ProcessBuilder class is a more flexible replacement for the Runtime.exec calls. Start the building process by specifying the command that you want to execute. You can supply a List<String> or simply the strings that make up the command. ProcessBuilder builder = new ProcessBuilder("gcc", "myapp.c"); The first string must be an executable command, not a shell builtin. For example, to run the dir command in Windows, you need to build a process with strings "cmd.exe", "/C", and "dir". Each process has a working directory, which is used to resolve relative directory names. By default, a process has the same working directory as the virtual machine, which is typically the directory from which you launched the java program. You can change it with the directory method: builder = builder.directory(path.toFile()); Each of the methods for configuring a ProcessBuilder returns itself, so that you can chain commands. Ultimately, you will call Process p = new ProcessBuilder(command).directory(file).start(); Next, you will want to specify what should happen to the standard input, output, and error streams of the process. By default, each of them is a pipe that you can access with OutputStream processIn = p.getOutputStream(); InputStream processOut = p.getInputStream(); InputStream processErr = p.getErrorStream(); The input stream of the process is an output stream in the JVM! You write to that stream, and whatever you write becomes the input of the process. Conversely, you read what the process writes to the output and error streams. For you, they are input streams. You can specify that the input, output, and error streams of the new process should be the same as the JVM. If the user runs the JVM in a console, any user input is forwarded to the process, and the process output shows up in the console. Call builder.inheritIO() to make this setting for all three streams. If you only want to inherit some of the streams, pass the value ProcessBuilder.Redirect.INHERIT to the redirectInput, redirectOutput, or redirectError methods. For example, builder.redirectOutput(ProcessBuilder.Redirect.INHERIT); You can redirect the process streams to files by supplying File objects: builder.redirectInput(inputFile) .redirectOutput(outputFile) .redirectError(errorFile) The files for output and error are created or truncated when the process starts. To append to existing files, use builder.redirectOutput(ProcessBuilder.Redirect.appendTo(outputFile)); It is often useful to merge the output and error streams, so you see the outputs and error messages in the sequence in which the process generates them. Call builder.redirectErrorStream(true) to activate the merging. If you do that, you can no longer call redirectError on the ProcessBuilder or getErrorStream on the Process. Finally, you may want to modify the environment variables of the process. Here, the builder chain syntax breaks down. You need to get the builder's environment (which is initialized by the environment variables of the process running the JVM), then put or remove entries. Map<String, String> env = builder.environment(); env.put("LANG", "fr\_FR"); env.remove("JAVA\_HOME"); Process p = builder.start(); After you have configured the builder, invoke its start method to start the process. If you configured the input, output, and error streams as pipes, you can now write to the input stream and read the output and error streams. For example, Process process = new ProcessBuilder("/bin/ls", "-l").directory(Paths.get("/tmp").toFile()).start(); try (Scanner in = new Scanner(process.getInputStream())) { while (in.hasNextLine()) System.out.println(in.nextLine()); }There is limited buffer space for the process streams. You should not flood the input, and you should read the output promptly. If there is a lot of input and output, you may need to produce and consume it in separate threads. To wait for the process to finish, call int result = process.waitFor(); or, if you don't want to wait indefinitely, long delay = ...; if (process.waitfor(delay, TimeUnit.SECONDS)) {int result = process.exitValue();} else {process.destroyForcibly();} The first call to waitFor returns the exit value of the process (by convention,0 for success or a nonzero error code). The second call returns true if the process didn't time out. Then you need to retrieve the exit value by calling the exitValue method. Instead of waiting for the process to finish, you can just leave it running and occasionally call isAlive to see whether it is still alive. To kill the process, call destroy or destroyForcibly. The difference between these calls is platform dependent. On Unix, the former terminates the process with SIGTERM, the latter with SIGKILL. (The supportsNormalTermination method returns true if the destroy method can terminate the process normally.) Finally, you can receive an asynchronous notification when the process has completed. The call process.onExit() yields a CompletableFuture<Process> that you can use to schedule any action. process.onExit().thenAccept( p -> System.out.println("Exit value: " + p.exitValue())); To get more information about a process that your program started, or any other process that is currently running on your machine, use the ProcessHandle interface. You can obtain a ProcessHandle in four ways: 1. Given a Process object p, p.toHandle() yields its ProcessHandle. 2. Given a long operating system process ID, ProcessHandle.of(id) yields the handle of that process. 3. ProcessHandle.current() is the handle of the process that runs this Java virtual machine. 4. ProcessHandle.allProcesses() yields a Stream<ProcessHandle> of all operating system processes that are visible to the current process. Given a process handle, you can get its process ID, its parent process, its children, and its descendants: long pid = handle.pid(); Optional<ProcessHandle> parent = handle.parent(); Stream<ProcessHandle> children = handle.children(); Stream<ProcessHandle> descendants = handle.descendants(); The Stream<ProcessHandle> instances that are returned by the allProcesses, children, and descendants methods are just snapshots in time. Any of the processes in the stream may be terminated by the time you get around to seeing them, and other processes may have started that are not in the stream. The info method yields a ProcessHandle.Info object with methods for obtaining information about the process. Optional<String[]> arguments() Optional<String> command() Optional<String> commandLine() Optional<String> startInstant() Optional<String> totalCpuDuration() Optional<String> user() All of these methods return Optional values since it is possible that a particular operating system may not be able to report the information. For monitoring or forcing process termination, the ProcessHandle interface has the same isAlive, supportsNormalTermination, destroy, destroyForcibly, and onExit methods as the Process class. However, there is no equivalent to the waitFor method.