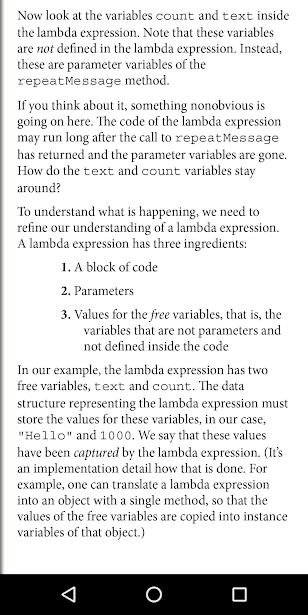
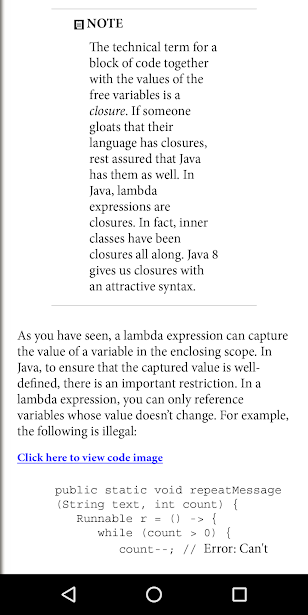
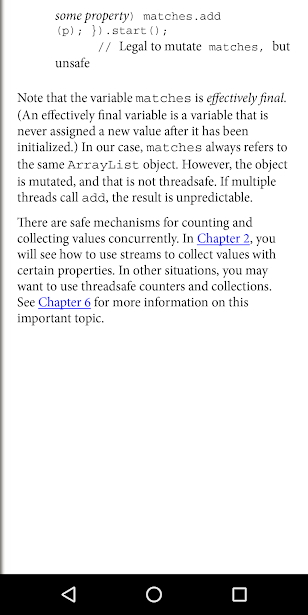
Java8

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

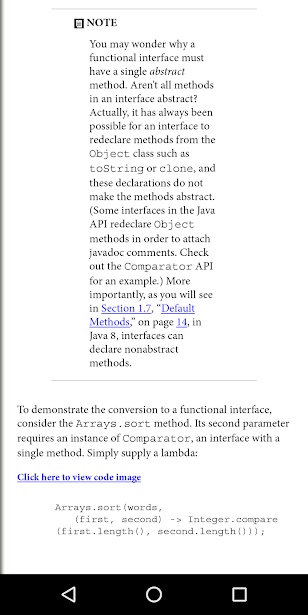


Lambdas can’t throw an Exception. Try/Catch it. Or use them for a functional interface that throws an Exception, callable instead of runnable, for e.g.





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



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

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

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

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

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

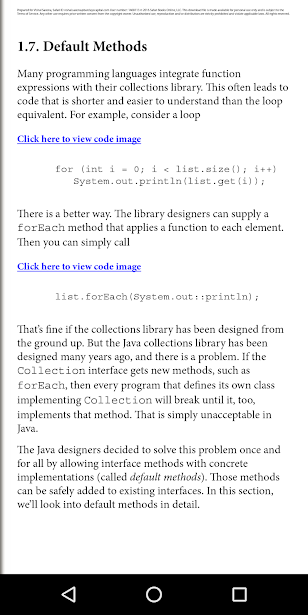
Integer[] ever = IntStream.of( data ).boxed().toArray( Integer[]::new );

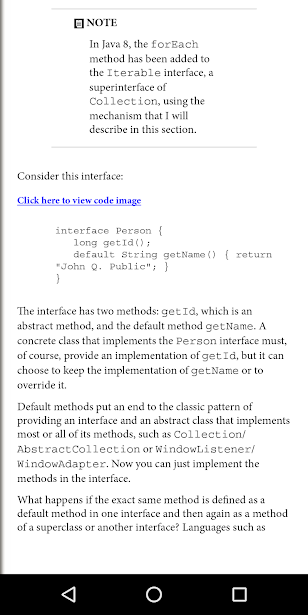
List<Integer> you = Arrays.stream( data ).boxed().collect( Collectors.toList() );

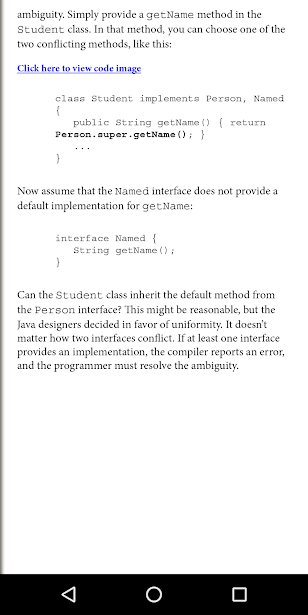
List<Integer> like = IntStream.of( data ).boxed().collect( Collectors.toList() );

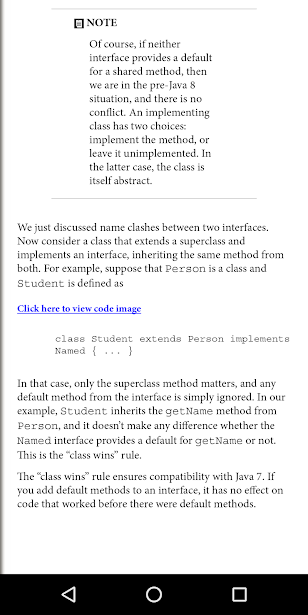
Double[] boxed = new Double[] { 1.0, 2.0, 3.0 };

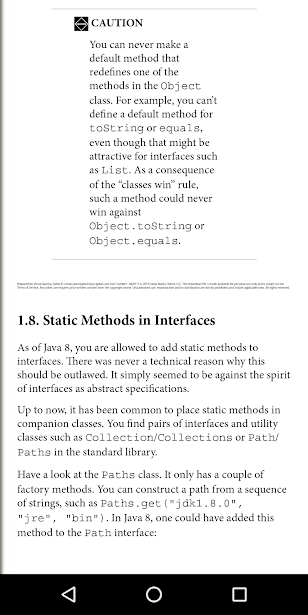
double[] unboxed = Stream.of(boxed).mapToDouble(Double::doubleValue).toArray();

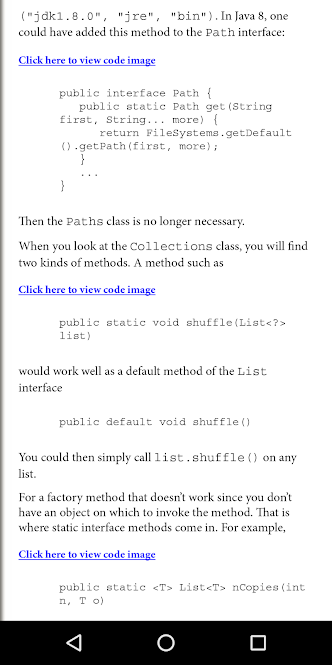


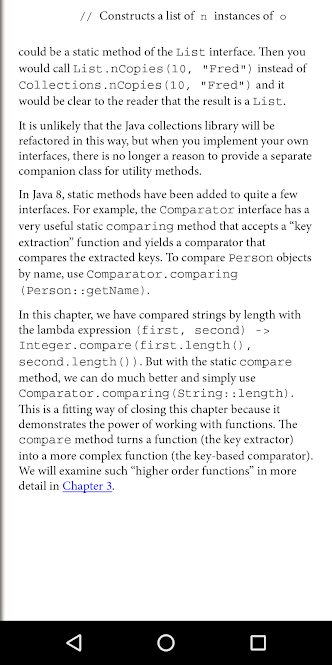


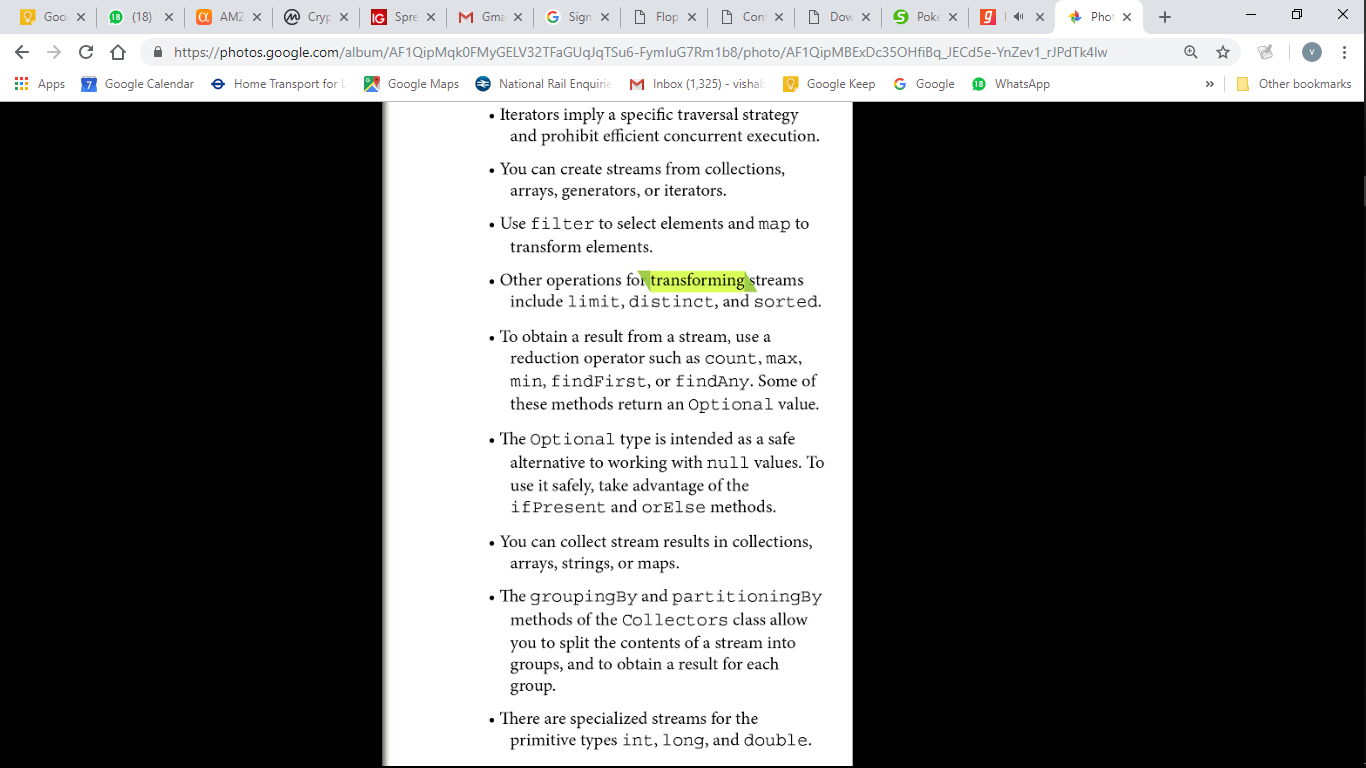


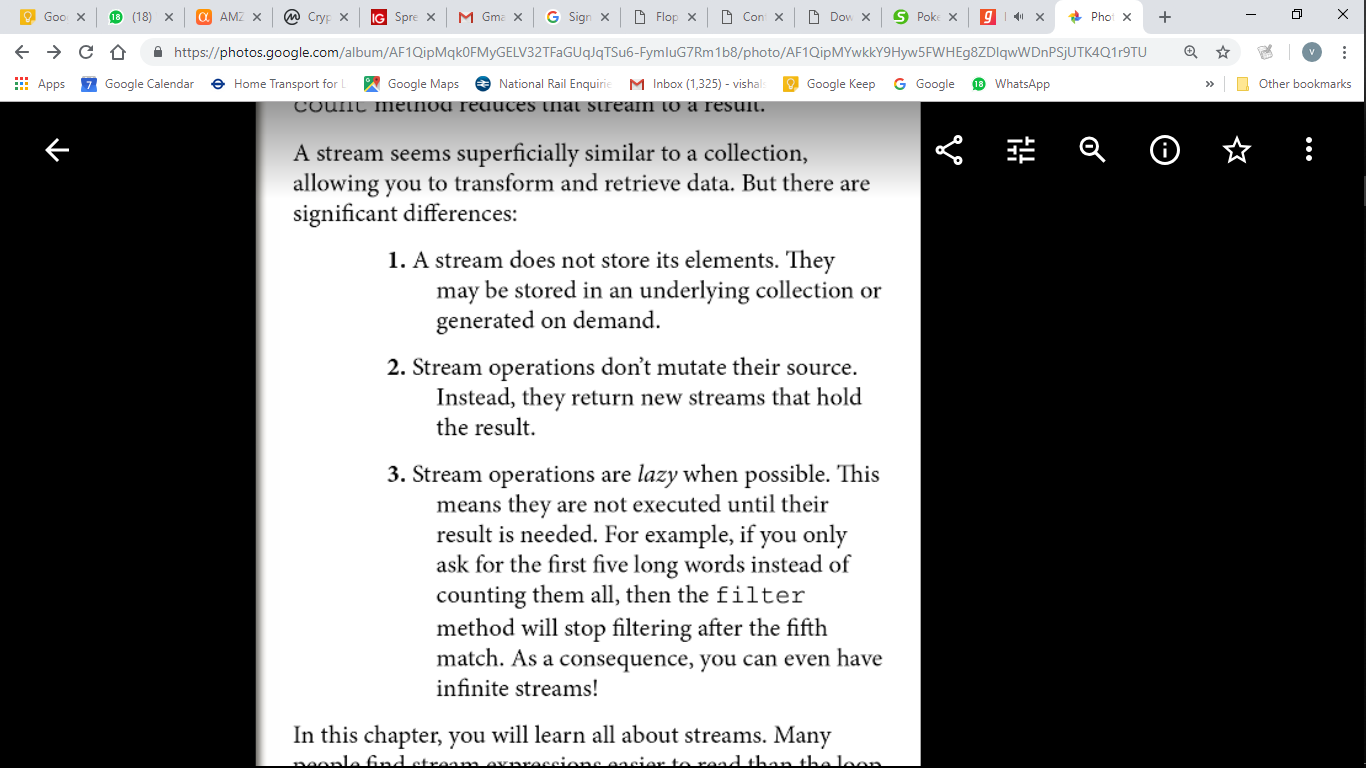


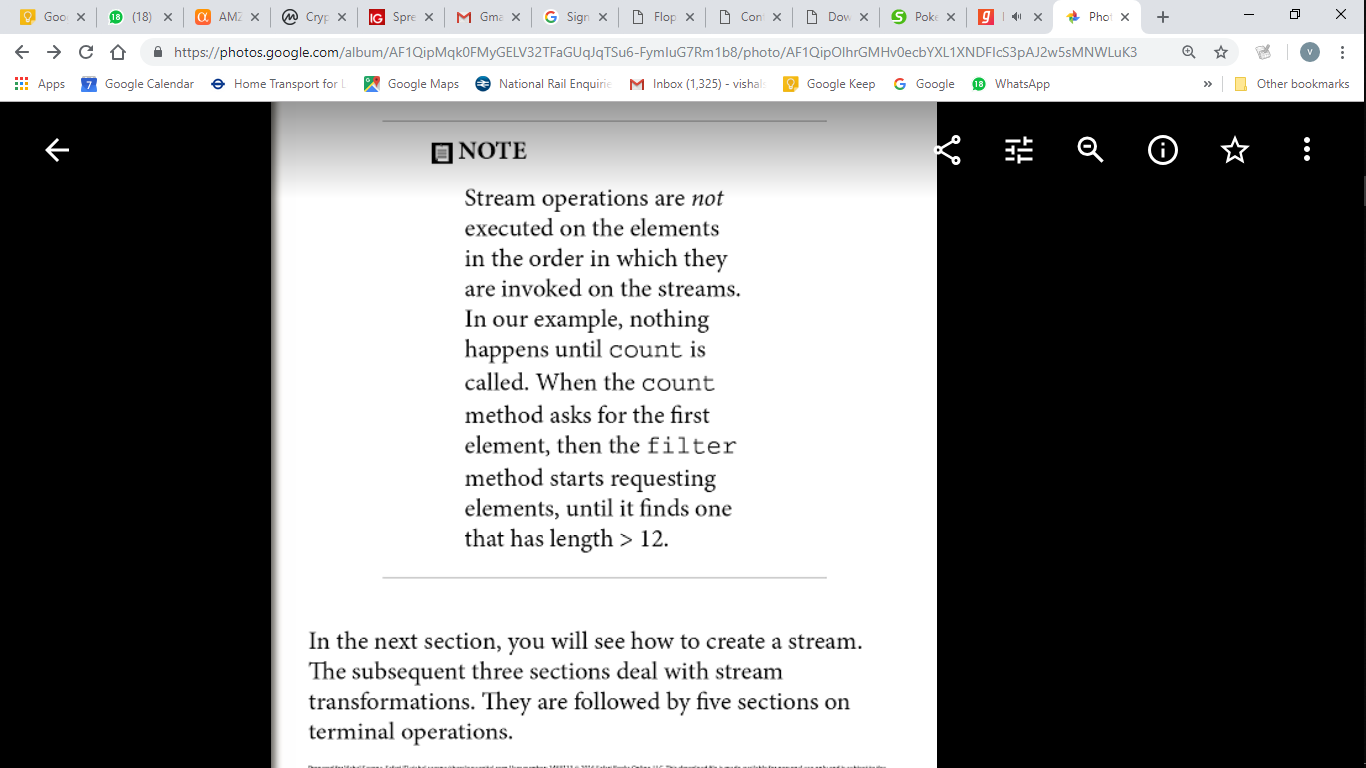


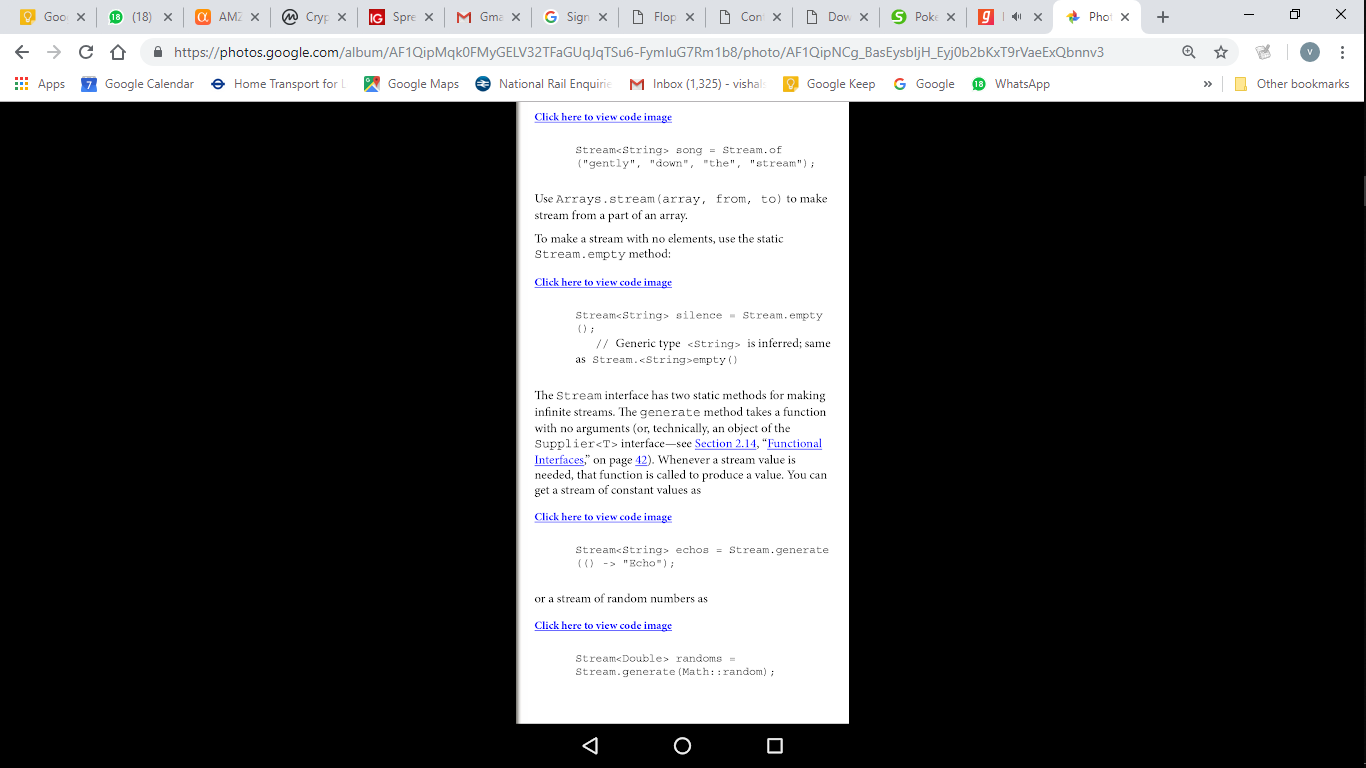


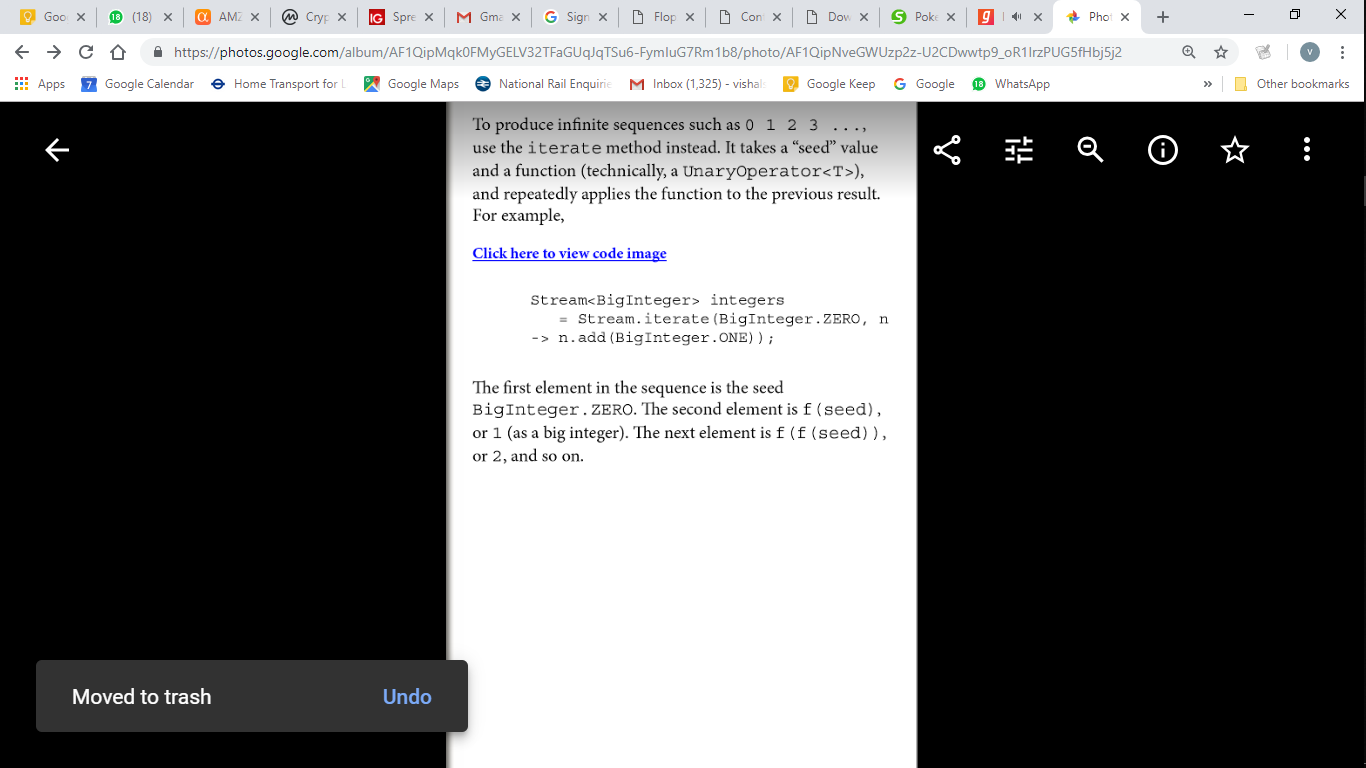




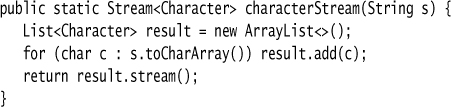








A number of methods that yield streams have been added to the API with the Java 8 release. For example, the Pattern class now has a method splitAsStream that splits a CharSequence by a regular expression. You can use the following statement to split a string into words:Stream<String> words  
   = Pattern.compile("[\\P{L}]+").splitAsStream(contents);The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







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

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

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

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

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

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

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

##### 2.5. Stateful Transformations

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

##### Reductions

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

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

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

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

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

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

##### 2.7. The Optional Type

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

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

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

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

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

optionalValue.ifPresent(results::add);

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

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

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

String result = optionalString.orElse("");

// The wrapped string, or "" if none

You can also invoke code to compute the default,

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

// The function is only called when needed

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

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

// Supply a method that yields an exception object

To create Optional

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

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

}

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

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

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

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

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

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

}

Then you can compute the square root of the inverse as

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

or, if you prefer,

Optional<Double> result =

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

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

##### 2.8. Reduction Operations

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

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

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

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

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

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

Stream<Integer> values = ...;

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

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

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

int result = words.reduce(0,

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

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

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

##### Comparable,Comparator,Comparators

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

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

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

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

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

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

**Arrays**

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

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

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

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

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

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

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

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

## Overriding methods

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

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

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

**}**

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

**@Override**

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

**}**

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

## Generics

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

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

The normal rules apply to the type being parameterized:

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

Unbounded wildcards allow assignment with any type parameter:

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

Bounded wildcards affect assignment like you might expect:

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

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

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

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

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

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

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

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

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

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

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

T member**;**

**}**

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

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

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

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

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

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

**}**

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

**@Override**

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

**}**

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

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

**}**

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

**@Override**

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

**}**

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

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

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

**}**

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

**@Override**

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

**}**

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

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

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

**}**

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

**@Override**

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

**}**

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

# Generics

## Non-Reifiable Types

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

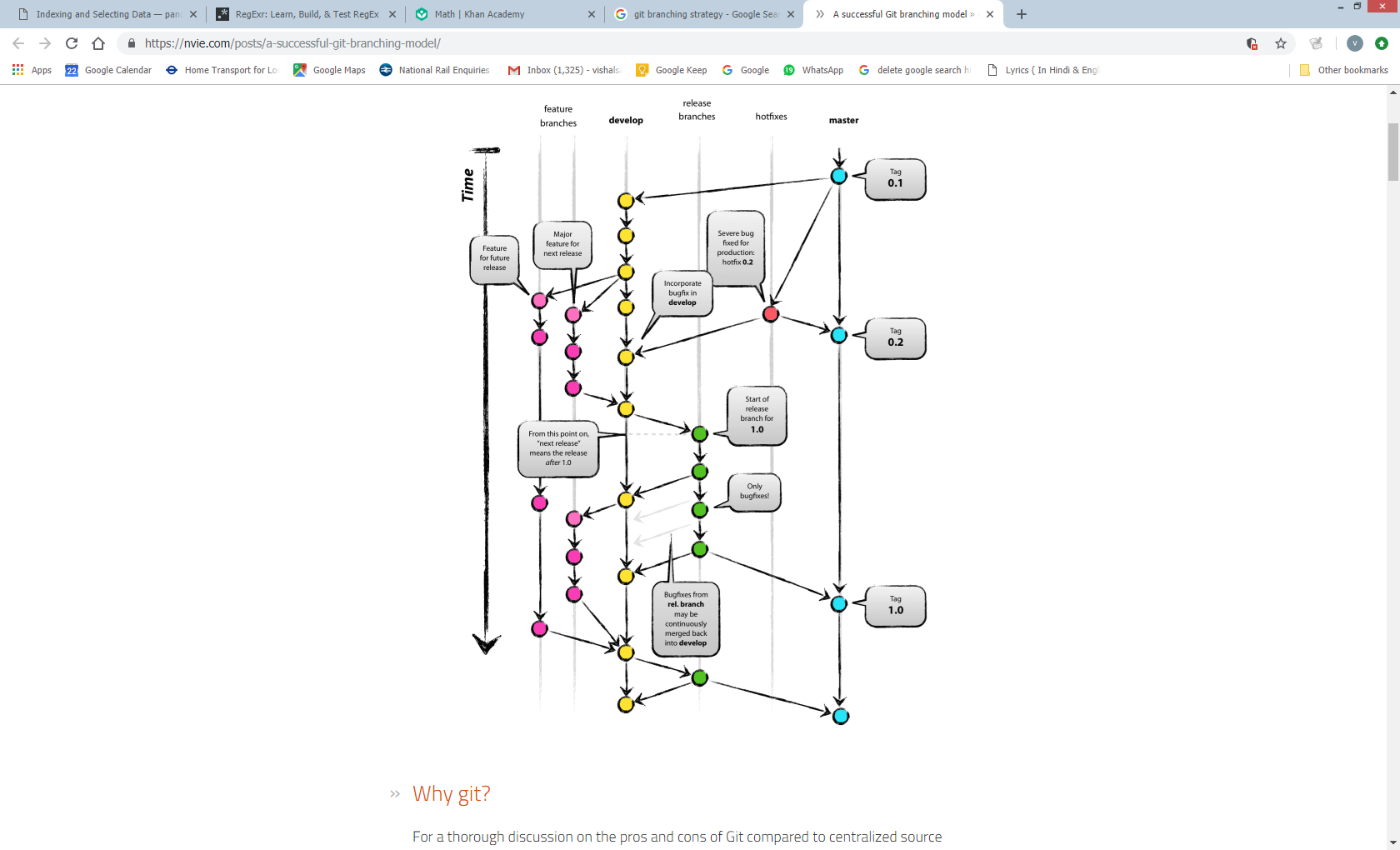
Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String>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

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

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

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

# Git Branching Strategy



# Java APIs

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

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

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