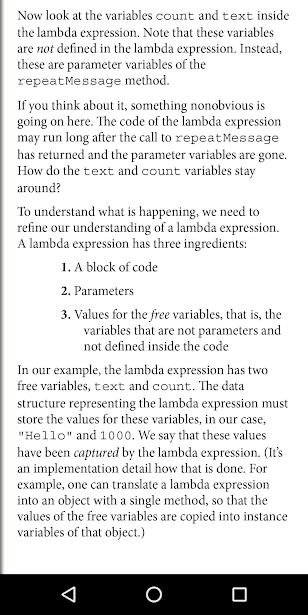
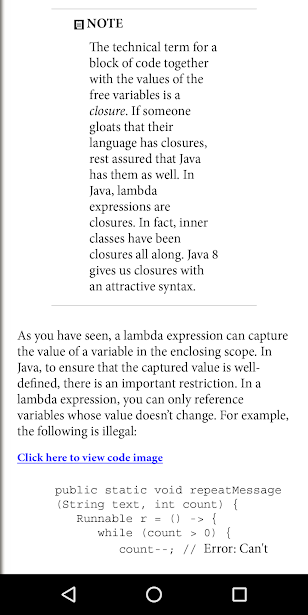
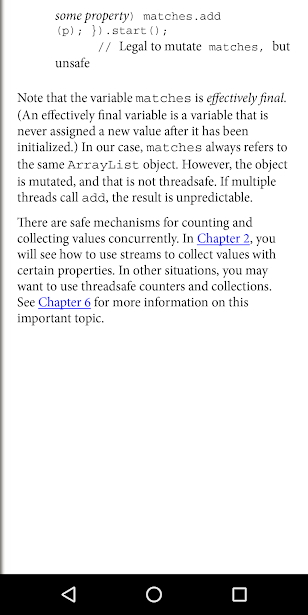
Java8

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.

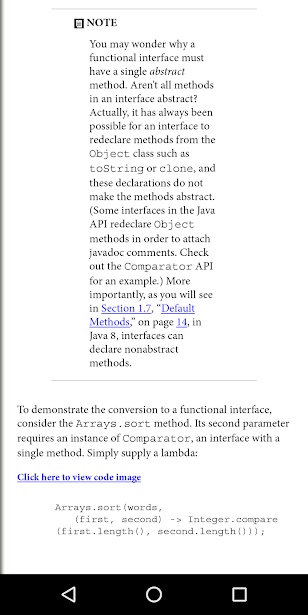


Lambdas can’t throw an Exception. Try/Catch it. Or use them for a functional interface that throws an Exception, callable instead of runnable, for e.g.





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 );

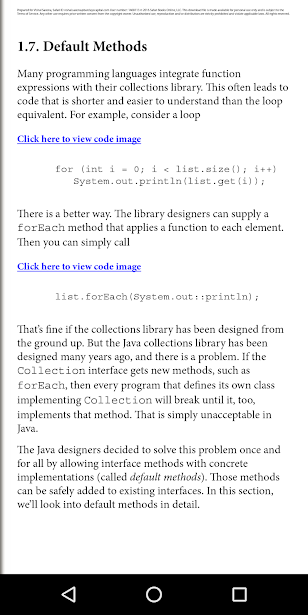
Integer[] ever = IntStream.of( data ).boxed().toArray( Integer[]::new );

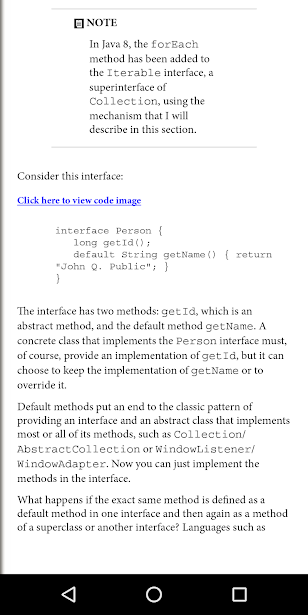
List<Integer> you = Arrays.stream( data ).boxed().collect( Collectors.toList() );

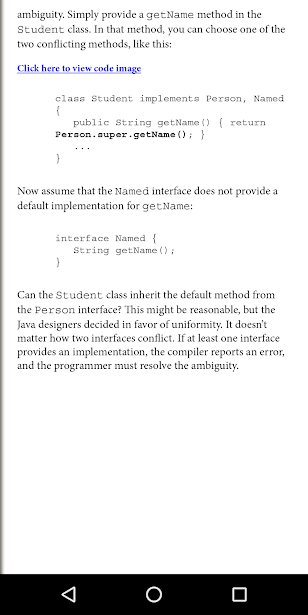
List<Integer> like = IntStream.of( data ).boxed().collect( Collectors.toList() );

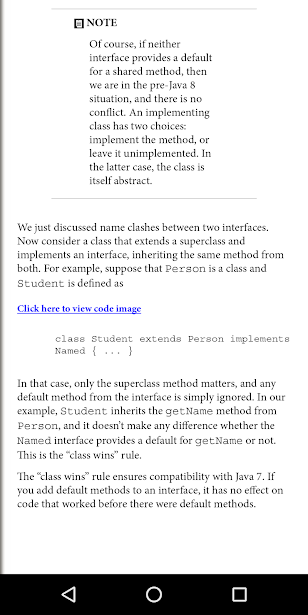
Double[] boxed = new Double[] { 1.0, 2.0, 3.0 };

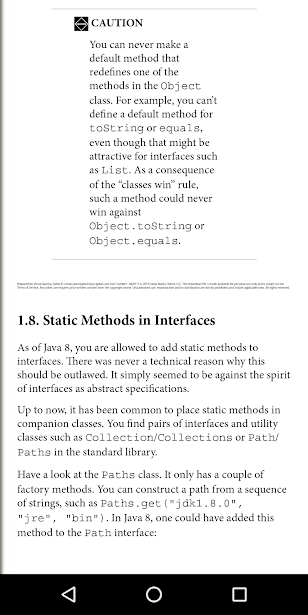
double[] unboxed = Stream.of(boxed).mapToDouble(Double::doubleValue).toArray();

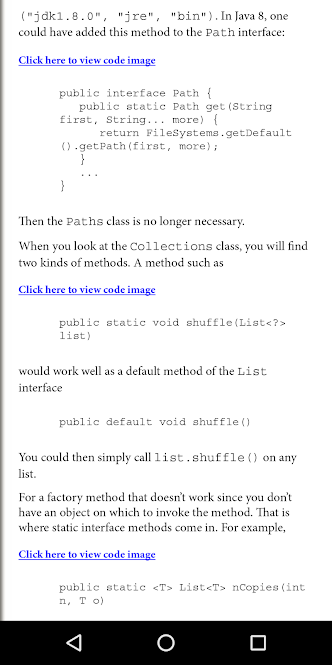


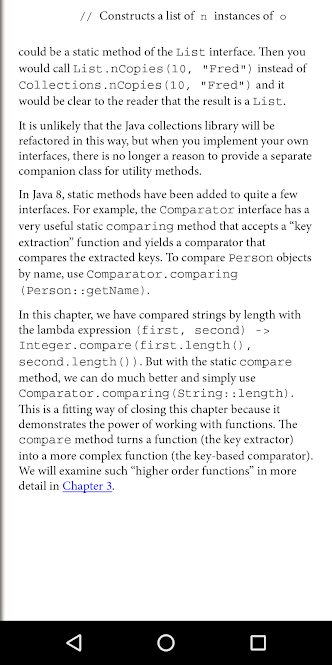


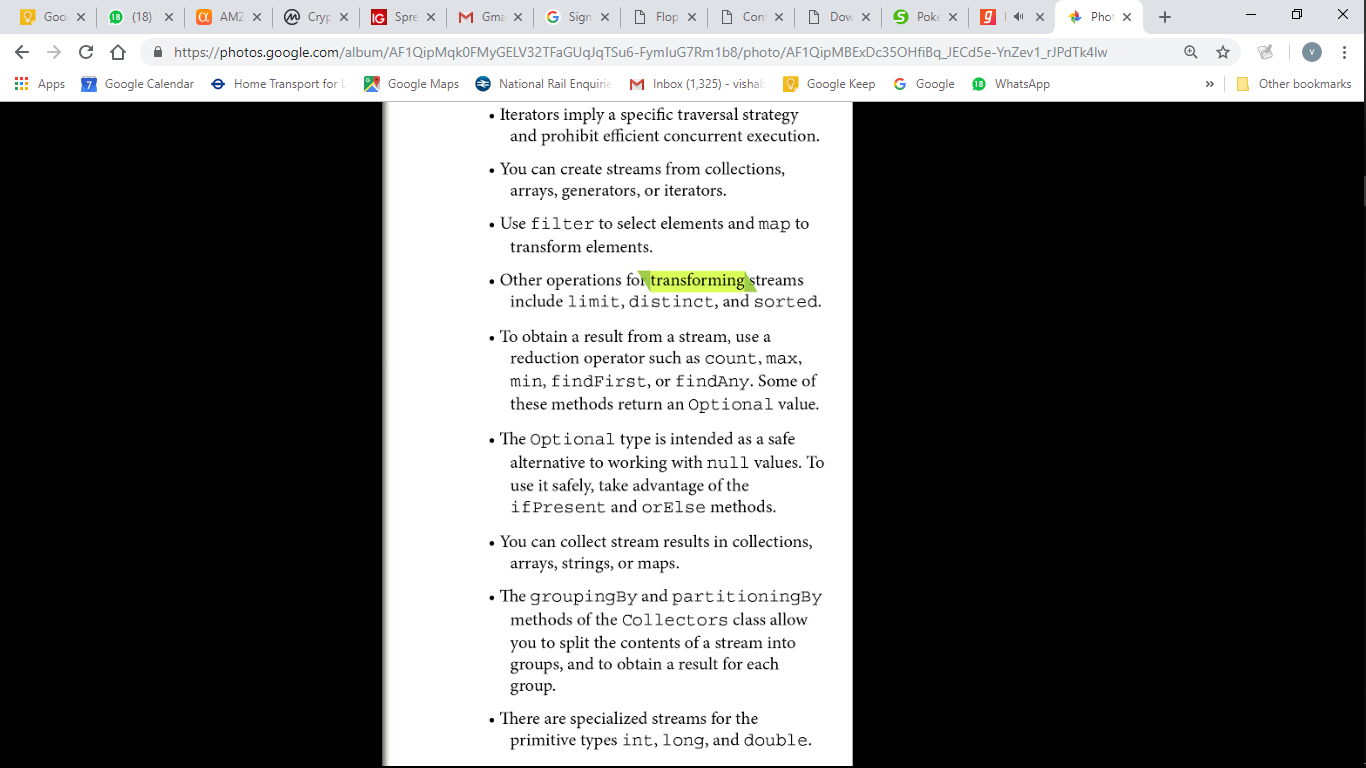


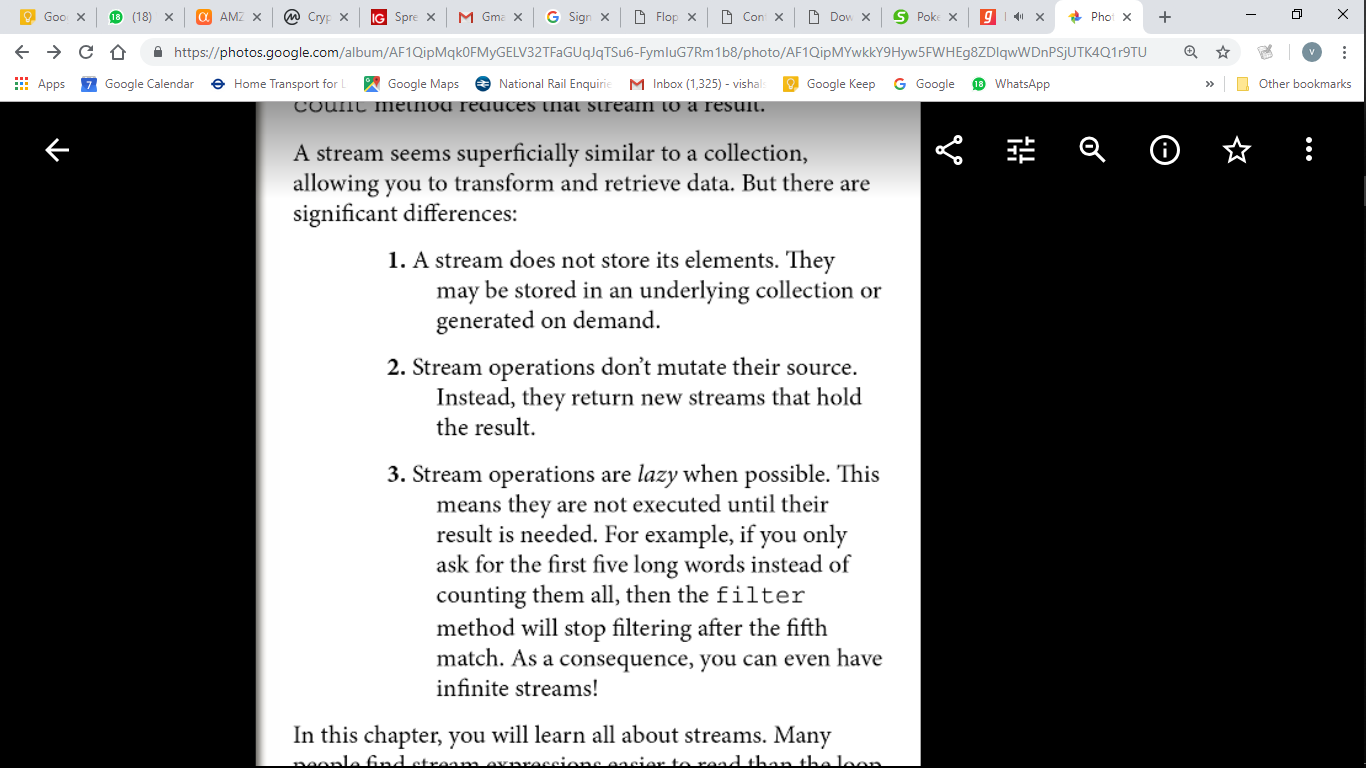


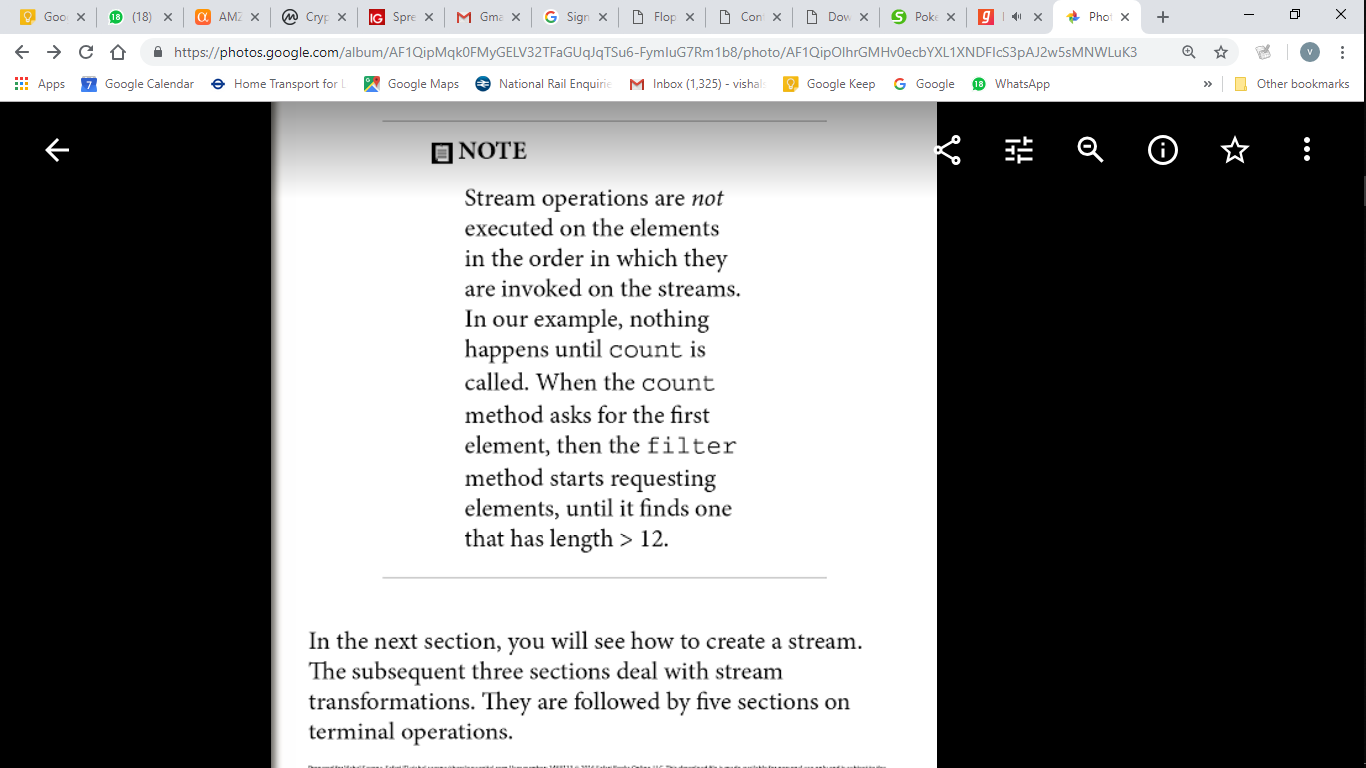


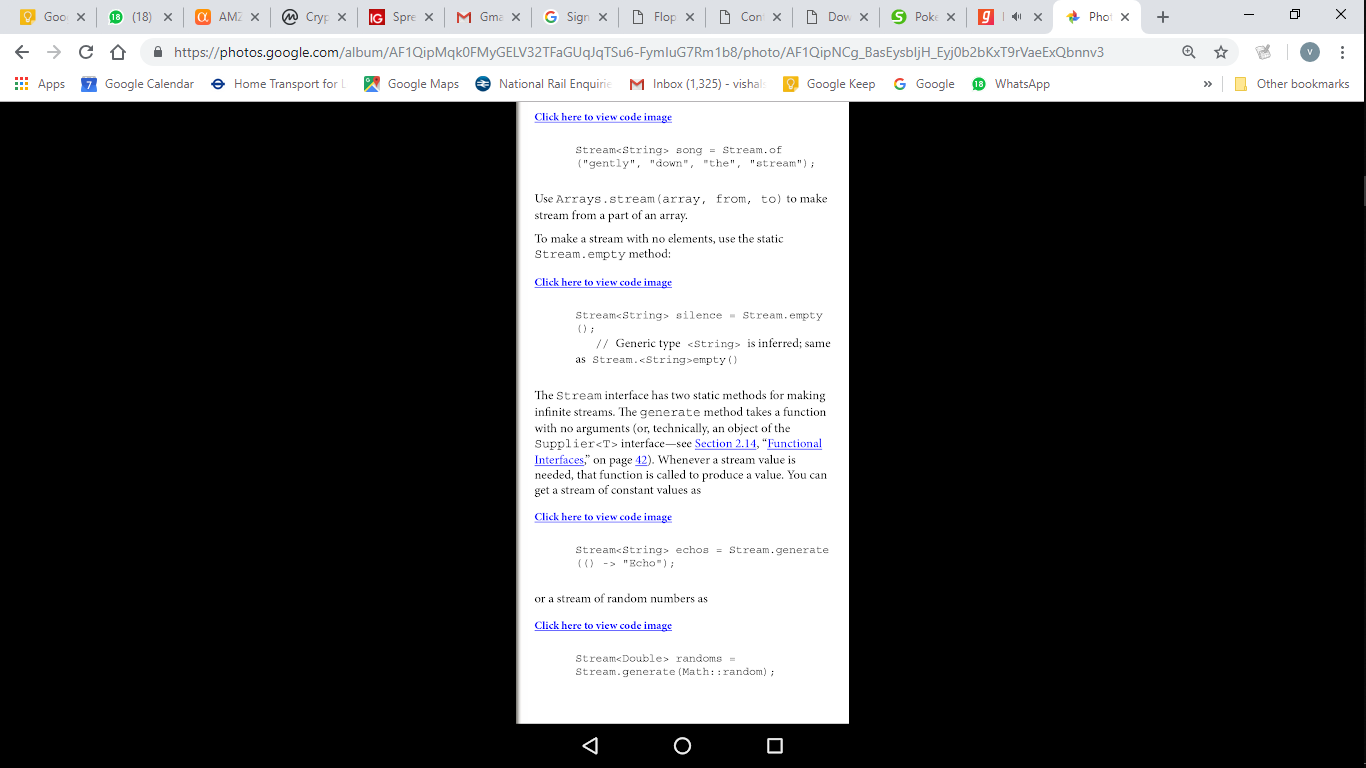


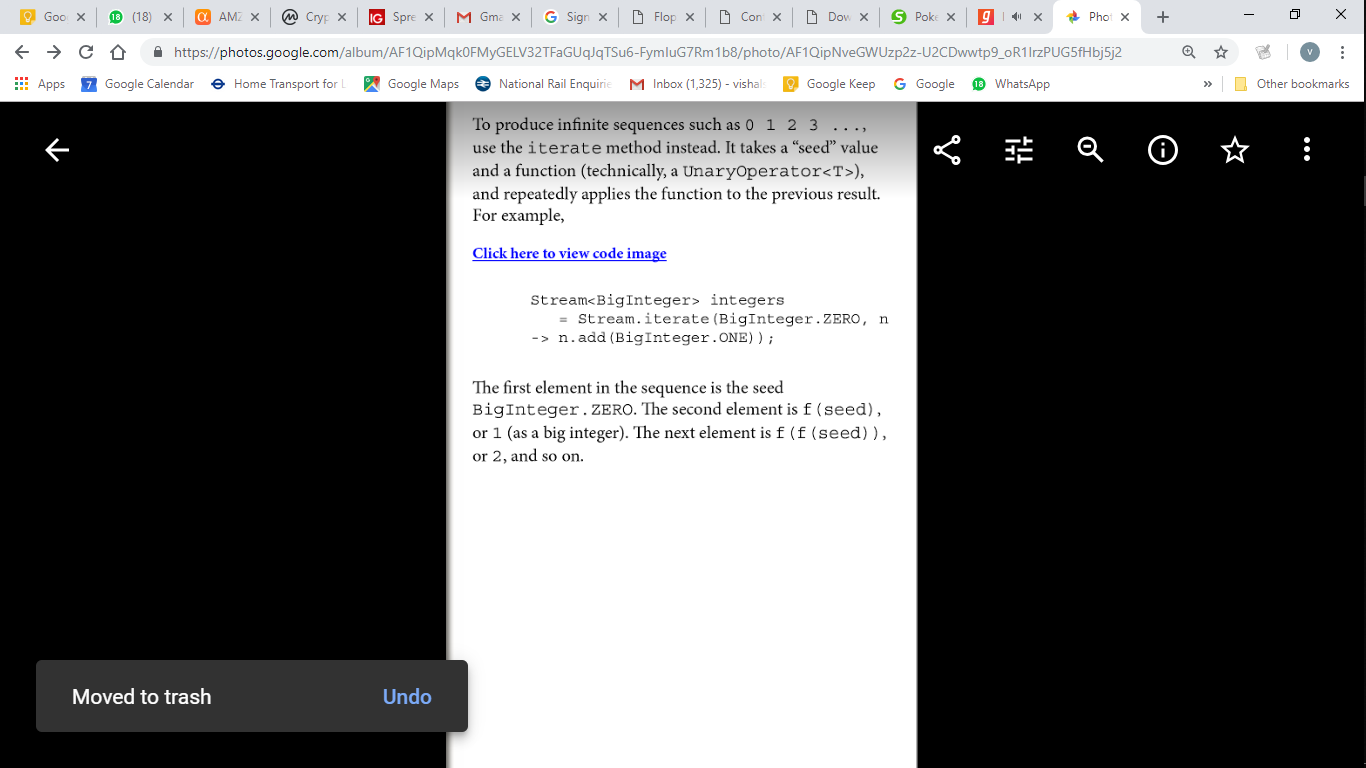












A number of methods that yield streams have been added to the API with the Java 8 release. For example, the Pattern class now has a method splitAsStream that splits a CharSequence by a regular expression. You can use the following statement to split a string into words:

[**Click here to view code image**](ms-local-stream://EpubReader_BFEA214A6AD24A7A8C9F7474E6B1BD32/Content/OEBPS/html/ch02_images.html#p024pro07a)

Stream<String> words  
   = Pattern.compile("[\\P{L}]+").splitAsStream(contents);

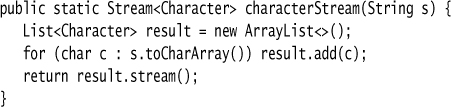
The static Files.lines method returns a Stream of all lines in a file. The Stream interface has AutoCloseable as a superinterface. When the close method is called on the stream, the underlying file is also closed. To make sure that this happens, it is best to use the Java 7 try-with-resources statement:

[**Click here to view code image**](ms-local-stream://EpubReader_BFEA214A6AD24A7A8C9F7474E6B1BD32/Content/OEBPS/html/ch02_images.html#p025pro01a)

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.

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.

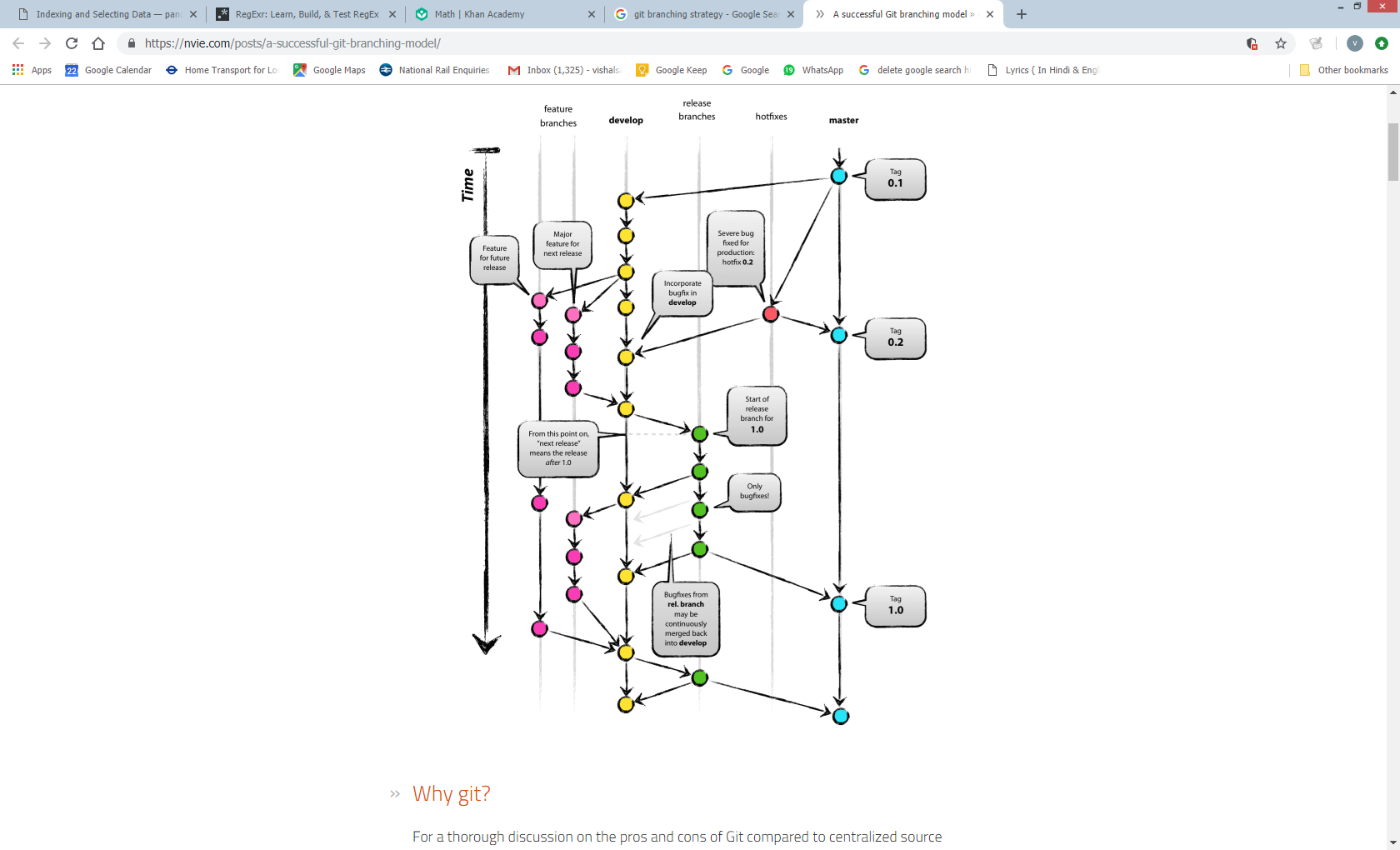
Non-reifiable types are types where information has been removed at compile-time by type erasure — invocations of generic types that are not defined as unbounded wildcards. A non-reifiable type does not have all of its information available at runtime. Examples of non-reifiable types are List<String>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