Java 8 (Pluralsight)

Monday, September 3, 2018

12:14 PM

***Points Dig More Read Again***

* + ***Lamda expression simple example***

|  |  |
| --- | --- |
| *FileFilter filter = new FileFilter() {*  *@Override*  *public boolean accept(File pathname) {*  *return pathname.getName().endsWith(".java");*  *}*  *};* | *FileFilter filterLambda = (File pathname) -> pathname.getName().endsWith(".java");* |
| *Runnable runnable = new Runnable() {*  *@Override*  *public void run() {*  *for (int i = 0 ; i < 3 ; i++) {*  *System.out.println(*  *"Hello world from thread [" +*  *Thread.currentThread().getName() + "]");*  *}*  *}*  *};* | *Runnable runnableLambda = () -> {*    *for (int i = 0 ; i < 3 ; i++) {*  *System.out.println(*  *"Hello world from thread [" +*  *Thread.currentThread().getName() +*  *"]");*  *}*  *} ;* |
| *Comparator<String> comp = new Comparator<String>() {*  *@Override*  *public int compare(String s1, String s2) {*  *return Integer.compare(s1.length(), s2.length());*  *}*  *};* | *Comparator<String> compLambda = (String s1, String s2) ->*  *Integer.compare(s1.length(), s2.length());* |

***Questions about Lambdas***

* + ***What is the type of a lambda expression?***
    - *a functional interface*
  + ***What is a functional interface?***
    - *A functional interface is an interface with only one abstract method*

***Functional Interface***

* + *A functional interface can be annotated*

* + ***Can a lambda be put in a variable?***
    - *yes!*
    - *Comparator<String> c = (String s1, String s2) -> Integer.compare(s1.length(), s2.length());*
    - *Consequences: a lambda can be taken as a method parameter, and can be returned by a method*

* + ***Is a lambda expression an object?***
    - *The answer is complex, but no*
    - *Exact answer: a lambda is an object without an identity*
    - *Let’s compare the following:*
      * *Comparator<String> c = (String s1, String s2) -> Integer.compare(s1.length(), s2.length());*

* + *Comparator<String> c =* ***new*** *Comparator<String>(String s1, String s2) {*

*Public boolean compareTo(String s1, String s2) {*

*Integer.compare(s1.length(), s2.length());*

*}*

*};*

* + *A lambda expression is created without using «new»*

* + *It is just here for convenience, the compiler can tell me whether the interface is functional or not*

*Lamda Expression:*

*Ex 1:*

*Consumer<String> c = s -> System.out.println(s);*

*Or*

*Consumer<String> c = System.out::println;*

*Note: "::" operator denotes you will be invoking the println method with a parameter, which name you don't specify explicitly:*

*Ex 2:*

*Comparator<Integer> c = (i1, i2) -> Integer.compare(i1, i2);*

*OR*

*Comparator<Integer> c = Integer::compare*

*Ex3 :*

*List<Customer> list = ….;*

*List.forEach(customer -> System.out.println(customer));*

*or*

*List<Customer> list = ….;*

*List.forEach( System.out::println);*

***Functional Interfaces Toolbox***

*Note: if its functional interface you can use it as lamda expression*

* + *The JDK 8 provides us a new Toolbox of Functional Interfaces*
  + *This is a New package: java.util.function, With a rich set of functional interfaces. In fact, there are 43 of them*
  + *We are just going to divide them up into four categories.*
    - ***Supplier***
      * *The first category is the Supplier.*
        + *The supplier is just a single interface that doesn't take any object and that provides a new object.*
    - ***Consumer***
      * *The second category is the reverse of the supplier, the Consumer.* 
        + *The consumer accepts an object and doesn't return anything. Now think of System.out.println. This is the first example of a consumer.*
        + *in this category, we also have the BiConsumer. A BiConsumer is a special kind of consumer that takes two objects instead of one. Now, those two objects don't have to be of the same type. They can be different.*

* + ***Predicate***
    - *The third category is the Predicate.*
    - *A predicate takes an object as a parameter and returns a Boolean.*
    - *Same as the BiConsumer, a predicate can be a BiPredicate that takes two parameters, instead of one. Same thing, the two parameters don't have to be of the same type.*
  + ***Function*** 
    - *the last category is the Function category. A function takes an object as a parameter and returns another object.*
    - *We also have the BiFunction that takes two objects instead of one, and that still returns one object, of course. Now, you can see that the BiFunction is typed with three generate types, T, U, R.*
    - *in the function category, I have special cases of function. The first special case is called the Unary Operator. The Unary Operator is a special function that takes an object and returns another object of the same type. So, the identity method, for instance, is a special type of Unary Operator, and, if we have the Unary Operator for the function, you can guess that we also have the Binary Operator for the BiFunction. A Binary Operator takes two objects of the same kind and returns a third object of the same kind as its parameters.*

***Changing the Way Interfaces Work?***

***Can I Process This Data with Lambdas?***

*yes!*

*It has been done in Java 8, and this is what we're going to see now.*

*We can, indeed, write this code, take a list of customers, and call this new method, forEach, defined on the Iterable interface, so this method is available for all the Iterables of the collection framework, and pass a consumer to this method, like System.out::println.*

*But, then, there's another question that we need to ask. Where does this forEach method come from? Because the way interface is working in Java 7 is the following: If I add a method to a given interface, when an existing interface, then this method should be added to all the imple- mentation of this interface. So, if I've got five classes that implement that interface, those five classes need to provide an implementation for this new method. And, this is where the problem is. Adding a forEach method on the Iterable interface implies to refactor all the existing implementations of Iterable.*

*It might be possible to do that inside the JDK, because the people who had their hand on the modification of this Iterable interface, also have their hand on all the implementation inside the JDK, but all the custom implementations of Iterable in all the Java applications, would have to do that, too, and this is just not possible.*

*The Iterable interface has been around for 15 years. There is, I guess, a code absolutely everywhere that is implementing that interface and that will not be refactored, if that interface is modified.*

*Let's state the question again. How can we Add Methods to Iterable, and namely, that forEach method without breaking all the existing implementations and without modifying them? I want to be able to add that forEach method to Iterable, without modifying any other the imple- mentations of Iterable and without breaking the code.*

*The code should still compile and should still execute correctly. This is not possible in Java 7. This is not the way interfaces work in Java 7.*

*Refactoring these implementations is not an option, just because, there are too many of them. They are scattered in all the existing Java applications, so we can't ask all the applications to refactor themselves.*

***Default and Static Methods in Java 8 Interfaces***

*If I can't put the implementation of that forEach method in ArrayList, for instance, then the choice has been made to put it directly in the interface. We can see in Iterable interface that the forEach method is, in fact, a new kind of method that you can put in an interface called a default method.*

*This default method is just a regular method with the default keyword in front of the return type that is declared. Here it is, default void forEach, then the parameters, then the block of implementation of that method.*

*Putting code in an interface is a new thing we can do in Java 8, and it is a revolution in the Java language. These Default Methods is a new Java 8 concept. It allows us to change the old interfaces without breaking the existing implementations, and this is what we are going to do.*

*It also allows new patterns of code, and we are going to see examples of that right now. And, by the way, because interfaces and the way they work are being revisited, Static methods are also allowed in Java 8 interfaces.*

*This is also a new concept. So far, I could put static final fields, that is, constants, in interfaces in Java. Now I can also put static methods.*

*New Patterns: The Predicate Interface Example*

*So, let us see Examples Of New Patterns allowed with these two new concepts of default method and of static methods in interfaces. We saw the Predicate interface. It is part of the functional interfaces toolbox available in Java 8 in this Java utile function package, so let us take two of those predicates. The first one returns true. The string is of length less than 20, and the second one, greater than 10. We can change these two predicates with the boolean operation, and, and, what is this and stuff? Well, it is a method on the predicate interface, and it is a default method. When I write a predicate, when I implement a predicate, I do not need to provide an implementation of this and method, and, if I check the predicate interface, I can see the implementation of this and method. What does it take? It takes another predicate that should not be null. This is the first test inside the implementation, and, after that, what does it return? It just tests the variable with the first predicates and tests the variable with the second one, with the and boolean operation between those two tests. So, it is pretty simple and pretty straightforward to implement such methods. It has been done in the JVK. We have such default methods in all the functional interfaces of the toolbox we saw, and it is a good thing to just have a look at this code, just to take it as an example for us to get ideas on how to build new APIs with Java 8. We also have an isEqual method in predicates. What does it do? It returns a predicate that is a static method of the predicate's interface, and it returns a predicate that will just compare the given string to the target string that predicate is built on. Here is the implementation of that isEqual method. It takes the object as a target and just returns the result of target.equals the object passed as a parameter of the lambda expression.*

***Java 8 Stream API and Collectors***

*Note: Stream does not hold any data*

* + ***The<Consuming> operations: forEach() and peek()***
  + ***The<mapping> operations: map() and flatMap()***
  + ***The<filter> operation: filter()***
  + ***The<reduction> operations***
    - ***Aggregation: reduce(), max(), min()…..***
    - ***Mutable reduction: Collect, Collectors***

* + ***The Map / Filter / Reduce Algorithm***
    - *What is the Map/Filter/Reduce go to them about*
    - *It is very widely used in data processing nowadays*

*Example*

*Average Age of the people older than 20*

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

* + *list of Person instances and we want to compute the average age of the age of the people from this list older than 20.*
  + ***1st step*** *is called the* ***mapping step.***
    - *The mapping takes a List <Person> and returns a List <Integer>.*
    - *This list will contain the people we are working on, and the return list, the list of the ages of those people.*
    - *What is important to stress is that the size of both lists is the same. If we have a list of 50 people, we'll get a list of 50 ages, and, if it is a list in the Java sense, a list is different from a collection. In the list, I've got a first element, a second element. The older of the elements in the list is important, when the order of the elements in a collection is not, then the order of the ages must match the orders of the people in the input list.*

*Note: List must be in order for better results. If it's not in order you may get different result. If same number in two times not sure which number return*

* + ***2nd step*** *is called the* ***filtering step****.* 
    - *The filtering step takes our list of ages, which is a list of integers, and returns, also, a list of integers. In fact, in that list, I have all the edges that have passed the filtering step.*
    - *If my filtering is just a predicate edge greater than 20, then, in the return list, I have all the ages greater than 20.*

*Note:*

*The first step took a list of a given type<Persons> and returned the list of another type<Integer> with persons age,*

*The second step takes a list of a given type<Integer> and returns a list of the same type<Integer>.*

* + ***3rd step,*** *This is called the* ***reduction step****.* 
    - *We'll just say that it is equivalent to a* ***SQL aggregation****.*
    - *What is a SQL aggregation?*
      * *It's, the sum of the elements or the max, or the mean, or the average, something like that.*

* + ***What Is a Stream?***
    - *from the pure technical point of view. The stream is a Java-type interface. This type is T. It means that we can have streams of integer, streams of person, streams of customer, streams of strings, etc., etc.*

**public interface** Stream<T> **extends** BaseStream<T, Stream<T>>{

//////

}

* + *We could think of a stream as being a collection, but, in fact, it is completely different from a collection. A stream is not a collection, even if it might look like a collection,*
  + *What does a Stream exactly do?*
    - *It gives it a way to efficiently process data inside the Java Virtual Machine. It can efficiently process large amounts of data. For instance, it can process data in parallel, which is interesting, if I have large amounts of data, but it is also efficient to process small amounts of data. Even if I have only a few objects to process in my stream, it is still interesting and efficient to use a stream.*
  + *What does efficiently mean?*
    - *it means two things. First of all, this data can be processed in parallel, and I would say automatically processed in parallel. I don't have to write any kind of technical code to process my data in parallel. The parallelization of the algorithm doesn't have to be written by me as a developer.*

*Why do I need to process my data in parallel?*

* + *The answer is simple. It's just to leverage the computing power of multiple CPUs. Nowadays, all the CPU will use are multiple CPUs, and I want to leverage that.*
  + *The second thing, all the process is conducted in a Pipeline, and this is extremely important to understand, too, because it will avoid unnecessary intermediary computations. Even if I have several operations on a given stream of data, all these operations are conducted on the one pass over the data.*

* + ***Definition of a Stream in Java 8***
    - *What is a Stream and how can I build one?* 
      * *The stream is an object on which I can define operations, and by operations, you can think of a map, a filter, or a reduce operation.*
      * *It's also an object that does not hold any data, and this is a big difference with the collection object.*
      * *It's an object that should not change the data it processes. Why that?*
        + *Just because I want to be able to process this data in parallel, that is, to distribute this data on the several CPUs I have or, at least, on the several calls of my CPU. So, since I don't want to be bothered by any visibility issue that would be solved with atomic variables, volatile variables, or synchronization, I decide that a stream is not allowed to change the data it processes.*
      * *Now, if I build myself my own implementation of stream, and, if I decide to violate this rule and to modify the data I am processing, my code will still compile. The JVM will execute it, and I will run into big problems, exceptions, crashes, or corrupted data. So, this is just a rule. It is not enforced, not by the compiler, nor the JVM itself.*
      * *Object able to process data in one pass. What does it mean?*
        + *Suppose I have a stream defined on a collection of person, the collection of person I used in the map/filter/reduce example, for instance, then I define three operations on it--the map, the filter, and the reduce. If I want to be efficient, I don't want to create any kind of intermediary collections to process this data, so it has to be processed in one pass.*
      * *this object should be optimized from the algorithm point of view and able to process data in parallel. What does it mean to be optimized from the algorithm point of view?*
        + *It means that every time I declare an operation on it, the implementation of that operation, implementation that isn't a JDK, of course, should be optimal, as far as algorithm is concerned.*

* + ***Building and Consuming a Stream***
    - *How Can We Build a Stream?* 
      * *we have many patterns to build streams. the most useful one.*

*List<Person> persons = ....;*

*Stream<Person> stream = Person.stream();*

* + *What can we do with it?*
    - *We can call, for instance, the forEach method defined on the stream interface and pass it a consumer.*
    - *Here that consumer just prints out all the elements of the list. That forEach method takes an instance of Consumer as an argument. Consumer is a functional interface from the Java.util.function package, which is the toolbox that contains all the main functional interfaces provided in the JDK 8.*
  + *Consumer interface.*
    - *It's a functional interface, so it has only one abstract method. This is the definition of a functional interface.*

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

*Consumer<String> c1 = s -> list.add(s);*

*Consumer<String> c2 = s -> System.****out****.println(s);*

* + *It can be implemented by a lambda expression.* 
    - *Consumer<T> c = p -> System.out.println(p);*
  + *it can also be written as a method reference*
    - *Consumer<T> c = System.out::println;*

***Chain consumers.***

**public class** ChainConsumers {

**public static void** main(String... args) {

List<String> strings =

Arrays.*asList*("one", "two", "three", "four", "five");

List<String> result = **new** ArrayList<>();

Consumer<String> c1 = System.***out***::println;

Consumer<String> c2 = result::add;

strings.forEach(c1.andThen(c2));

System.***out***.println("size of result = " + result.size());

}

}

* + *Consumer<String> c3 = c1.andThen(c2);*
  + *This Consumer takes a strings and adds it to our list, and the second Consumer, c2, just prints out the strings that is passed to it as a parameter. We can also write those to Consumer with method reference.*
  + *we can just chain them, c3 = c1.andThen(c2). This will chain the Consumer c1 with the Consumer c2 by using the default method, and Then, defined on the Consumer interface.*
  + *So, if I want to declare several consumers on the single stream, because the forEach method does not return anything,*
  + *the only way I can do that is by chaining the consumers into one and passing them as a parameter to the forEach method, as it is the case on that example.*

* + ***Filtering a Stream***

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

Stream<Person> stream = list.stream();

Stream<Person> filtered = stream.filter(person -> person.getAge() >20);

* + *what does a filter do?*
    - *It takes a stream, defined on a source of data, and it filters out part of that data following a predicate.*
    - *The predicate we are using here, just checking that the age of that person is greater than 20.*
    - *we open a stream on the list of person and declare the filtered stream by calling the filter method on the stream interface.*

* + *It takes a predicate as a parameter. Here is the Predicate. It's a regular lambda expression*

Stream<Person> p = person -> person.getAge() >20

* + *Predicate interface*

@FunctionalInterface

**public interface** Predicate<T> {

**Boolean** test(T t);

// default methods

**static**<T> Predicate<T> isEqual(Object o) { ... }

}

* + *it's a functional interface, provided in the Java.util.function package,*
  + *It has a single method called test that takes an object as a parameter and that returns a boolean.*
  + *we have a bunch of default methods*

**default** Predicate<T> and(Predicate<? **super** T> other){…}

**default** Predicate<T> or(Predicate<? **super** T> other){…}

**default** Predicate<T> negate(){…}

* + *We have the and, the or, and the negate.*
  + *How does it work?* 
    - *Well, let's take three predicates, p1, p2, and p3.*
    - *I can define a new predicate p by chaining those predicates, p1.and(p2).or(p3), and it will compute p1 and p2 or p3.*

Predicate<Integer> p1 = i-> i > 20;

Predicate<Integer> p2 = i-> i < 30;

Predicate<Integer> p3 = i-> i == 0;

Predicate<Integer> p = p1.and(p2).or(p3);

Predicate<Integer> p = p3.or(p1).and(p2);

* + *We have to be a bit careful with this way of writing things, priority is like (p1 and p2) or p3 ; (p3 or p1) and p2*
  + *I also have a static method in the predicate interface called isEqual.*
  + W*hat does this isEqual method do?*

Predicate<String> p = Predicate.*isEqual* ("two");

Stream<String> stream1 = Stream.*of* ("one","two","three" );

Stream<String> stream2 = stream1.filter(p);

* + *It creates a new Predicate by comparing the objects passed as a parameter. Here, it's in the string of characters two, and it can be used to filter out streams of strings. Here, I take my predicate isEqual two.I open a stream on this var org arrays, one, two, three, then define the filtering and, hopefully, the stream2 will only hold the strings of character two. Let us notice that the off method of the stream interface is a static method, so it's a new way of declaring patterns of code on interfaces that are used here, that leverage the file in Java 8. I can write static methods in interfaces and is also another way of creating streams in Java. The stream that is written every time by the filter method is a new instance of stream, so the stream1 and the stream2 objects are different objects.*

*Ex: Predicate consumer filter*

**public class** PredicatesConsumerFilter {

**public static void** main(String[] args) {

Stream<String> stream = Stream.*of*("One", "Two", "Three", "Four", "five", "Six", "One", "Two");

Predicate<String> p1 = s -> s.length() > 3;

Predicate<String> p2 = Predicate.*isEqual*("Two");

Predicate<String> p3 = Predicate.*isEqual*("Three");

stream

.filter(p2.or(p3))

.forEach(s -> System.***out***.println(s));

}

}

* + *Intermediary and Terminal Operations*

**public class** PredicatesConsumerFilter {

**public static void** main(String[] args) {

Stream<String> stream = Stream.*of*("One", "Two", "Three", "Four", "five", "Six", "One", "Two");

Predicate<String> p1 = s -> s.length() > 3;

Predicate<String> p2 = Predicate.*isEqual*("Two");

Predicate<String> p3 = Predicate.*isEqual*("Three");

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

stream

.peek(System.***out***::println) // intermediary Operation

.filter(p2.or(p3))

.forEach(list::add); //final Operation

System.***out***.println("Done!");

System.***out***.println("Size= "+ list.size());

}

}

* + *We take stream, and we are going to peek it with the System.out::println operation.*
  + *Then, filter with p1.or(p2), same case as previously, and then peek again, but this time we will add the filter content to our list*

.peek(System.***out***::println) // intermediary Operation

.filter(p2.or(p3))

.peek(list::add); //final Operation

* + *Since the peek method returns a stream and the filter method returns also a stream, both are intermediary operations, so if I execute that code, nothing should happen,*
  + *This piece of code System.out::println hasn't been executed. The size is still zero. It means that list.add hasn't been executed either.*
  + *Now, if we replace the peek call by a forEach call,*

.peek(System.***out***::println) // intermediary Operation

.filter(p2.or(p3))

.forEach(list::add); //final Operation

* + *what is the difference between the peek and the forEach?*
    - *The forEach doesn't return anything, so it is not an intermediary operation. It is a final operation.*
    - *This time, this final operation should trigger the processing of the data that the stream is connected to in one pass over the data. So, I should see the execution of this System.out.println.*
  + *It should print Because it is executed before the filter step, then the filter step will occur. We only keep the two and the three strings from that stream forEach will add the content to that stream.*
  + *forEach is not lazy. peek and filter operations were lazy.*

***The Map Operation***

* + *It implements the first step of the map/filter/reduce algorithm*

List<Person> list = Arrays.*asList*();

Stream<String> stream = list.stream();

Stream<String> names = stream.map(person -> person.getName())

* + *The map operation returns a Stream, so we can safely assume that it is an intermediary operation.*
  + *it does not execute anything. It does not process any data. It is only a declaration*
  + *it takes a function as an argument.*
  + *Map is functional interface with method called apply. This method takes an object as a parameter, and returns another object.*
  + *In our example, the parameter would be an instance of person, and that return object would be an integer (Age).*
  + *We also have a set of default methods to chain and compose mappings.*

@FunctionalInterface

**public interface** Function<T, R> {

R apply(T t);

**default** <V> Function<V, R> compose(Function<V, T> before);

**default** <V> Function<T, V> andThen(Function<R, V> after);

}

* + *Namely, we have two default methods, compose & and Then.*
  + *Beware the generics! You have to be extra careful when you design such methods, if you want your call to be allowed by extension of the person class, for instance.*
  + *I have one static utility method called identity. What does an identity do? Well, it's quite obvious. It takes an object and returns that same object.*

* + *The Flatmap Operation*

* + *Flat Mapping operation is a bit tricky to understand. signature of this method.* 
    - *Signature:* 
      * *<R> Stream<R> flatMap(Function<T, Stream<R>> flatMapper);*
      * *<R> Stream<R> map(Function<T, R> mapper);*

* + *flat Map takes a function as an argument, the same kind of function as the map method.*
  + ***map takes an object and returns another object, whereas the flat map takes an object and returns as a return type, a stream of objects.***

**public class** MapFlatMapEx {

**public static void** main(String[] args) {

List<Integer> list1 = Arrays.*asList*(1, 2, 3, 4, 5, 6, 7);

List<Integer> list2= Arrays.*asList*(2, 4, 6);

List<Integer> list3 = Arrays.*asList*(3, 5, 7);

List<List<Integer>> list = Arrays.*asList*(list1, list2, list3);

Function<List<?>,Integer> size = List::size;

Function<List<Integer>, Stream<Integer>> flatMapper = l -> l.stream();

list.stream()

.map(l -> l.size())

.forEach(System.***out***::println); //Output: 7 3 3

list.stream()

.map(size)

.forEach(System.***out***::println); //Output: 7 3 3

list.stream()

.flatMap(flatMapper)

.forEach(System.***out***::println); //Output: 1 2 3 4 5 6 7 2 4 6 3 5 7

}

}

***Reduction, Functions, and Bifunctions***

* + *last step of our map/filter/reduce algorithm is the Reduction step.*
  + ***What about that step? How can we model it on the stream API?***
    - *There are 2 kinds of reduction included in the Stream API.*
    - *The 1st kind is the basic and classical SQL operation, like the min, the max, the sum, the average, etc…*

*List<Integer> list = ......;*

*Stream<Integer> stream = age.stream();*

*Integer sum = stream.reduce(0, (age1,age2) -> age1 + age2); // 0 here is first value as initialization if age 22, 23 result will 0+22+23*

* + *Let's take a Stream of Integer built on the ages list, that will probably hold the ages produce step, and we just want to make the sum of all the integers in that stream.*
  + *So, we call the reduce method. It takes a first parameter here in blue, 0, the second parameter, that is the reduction lambda expression. That reduction takes two Integers, age1, and age2, and returns the sum of them.*
  + *The 1st argument, this 0 should be the identity element of the reduction operation.*
  + *The 2nd argument is the reduction operation of type BinaryOperator of T. Here, T is the type integer.*

@FunctionalInterface

**public interface** BiFunction<T, U, R>{

R apply(T t, U u);

}

* + *In fact, BinaryOperator is a special case of a BiFunction. A BiFunction looks like a function. It takes two objects of type T and U here and returns an object of type R.*

@FunctionalInterface

**public interface** BinaryOperator<T> **extends** BiFunction<T,T,T> {

R apply(T t1, T t2);

}

* + *The BinaryOperator is just an extension of a BiFunction where all those three types are, in fact, the same. So, a BiFunction takes two objects as parameters of the same type and returns an object, also, of the same type.*

***Optionals***

* + *Suppose that the reduction step is the max operation.*

BinaryOperator<Integer> sum = (i1,i2) -> i1 > i2 ? i1 : i2;

* + *The problem is that the max operation doesn't have an identity element, and its mandatory to provide the identity element of the reduction operation as first argument of the reduction method. So, the max of an empty Stream cannot be defined in that way.*
  + *Then, what happens if I want to call that?*

*List<Integer> ages;*

*Stream<Integer> stream = age.stream();*

*….max = stream.max(Comparator.naturalOrder());*

* + *we call it stream.max and pass as a parameter, a Comparator. Now, the naturalOrder static method from the Comparator interface will return the Comparator that will just compare comparable objects. An integer is a comparable object, so it can be compared with that Comparator.*
  + *So, what is the return type of this max method?*
    - *If it is int, the primitive type from the Java language, then the default value is 0, and 0 is clearly not the identity element of the max method.*
    - *If the return type is Integer, then the default value is null, and I certainly do not want to return a null value, in that case, because I will have to check ever in my code, if the return value is null to avoid null pointer exceptions.*
    - *The return type is called optional, Optional of Integer.*
  + *What is Optional?*

*List<Integer> ages;*

*Stream<Integer> stream = age.stream();*

***Optional<Integer>*** *max = stream.max(Comparator.naturalOrder());*

* + *It is a new concept in Java 8. It is a class and we can see it as a wrap type. You know the wrap type from the Java technology. The int primitive type is associated with the integer class, and, so on, for the long, for the double, for the float, etc. Well, the optional of integer looks like a wrap type, the only difference being that, in a wrap type, I always have a value, whereas, in an optional, I might not have a value. Returning an optional means there might be no result, and this is exactly what I want to mean here, because if I take the max of an empty stream, I do not know what is the max. I do not know what value to return. In fact, no value is suitable, in that case.*

***Pattern for the Optionals***

* + *How can I use an Optional?* 
    - *I have several patterns.*
    - *The first one and the easiest one is to call the isPresent method.*

* + *Optional<String> opt = stream.max(Comparator.naturalOrder());*
  + ***if****(opt.isPresent()) {*
    - *String s = opt.get();*
  + *}*
  + ***else*** *{*

* + *}*
  + *opt.isPresent())*
    - *The isPresent method will return true, if I have a value. If no value returns false*
  + *opt.get()*
    - *I can get it by calling the get method*
  + *String s = opt.orElse("");*
    - *I can also call the orElse method. This orElse method encapsulates both calls. In fact, it is just to call isPresent and will call the get method for me, if there is an object.*

* + *String s = opt.orElseThrow(MyException::****new****);*
    - *I can also throw an exception if I want to. So, I also have this orElseThrow method that will build a new exception.*

***Reduction Operations***

* + *Available reductions*
    - *max()*
    - *min()*
    - *count() -- return number of elements in a stream.*
  + *Boolean reductions*
    - *allMatch()*
    - *noneMatch()*
    - *anyMatch()*

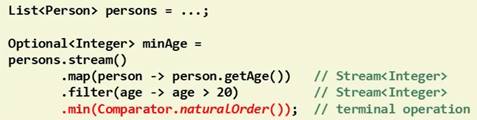
*Note: All those three methods take predicates as parameters, and the allMatch method will return true, if the predicate returns true for all the elements of that stream.*

* + *Reductions that return an optional*
    - *findFirst()*
    - *findAny()*

*Note: Reductions are intermediary operations reductions do not return a new stream. reductions are should call terminal operations.*

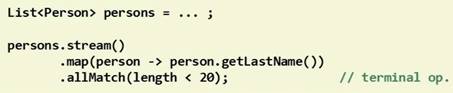
***Terminal Operations***

*Example: compute the age of the younger person older than 20*



* + *Let's take list of person*
  + ***First Step #*** *I will call the map methods and pass the person gives person.getAge lambda expression, so it will take my list of person and brings me back a list of integer, with the ages of the person in it.*
  + ***Second Step #*** *I call the filter method with the predicate age > 20, and this will filter out all the people younger than 20 from the stream. Those two methods return a stream of integers, so they are intermediary operations. They do not trigger the processing of the data,*
  + ***last step #*** *I call the Terminal Operation, a reduction step. In this case, it is the min method. That will compute the smallest integer from my stream. Since this last operation is terminal, it will trigger all the processing of the data, and the processing of this data will be conducted in one pass over the person elements of my list.*

*Example*



* + *map the list of person in the list of integer, and get the last name of each person.*
  + *check if all the elements of this list of string of characters have a length less than 20.*
  + *Now, since the allMatch step and the mapping step are conducted in one pass, it allows a nice optimization. If I see that the first person has already a last name of a length greater than 20, I do not have to compute the last name of the subsequent instances of person in my list.*
  + *It would not have been possible, if I would have conducted first the mapping step and then the allMatch step.*

***Example #***

**public class** ReductionExample {

**public static void** main(String[] args) {

List<Integer> list = Arrays.*asList*(10, 10, 10);

list.stream().reduce(0, (i1,i2) -> i1 + i2);

//Same line can be written using method reference

Integer result = list.stream().reduce(0, Integer::*sum*); // output: 30

List<Integer> list1 = Arrays.*asList*(10, 10, 10);

Integer result1 = list1.stream().reduce(100, Integer::*sum*); // output: 130

List<Integer> list2 = Arrays.*asList*(-10, -10);

Integer result2 = list2.stream().reduce(0, Integer::*max*); // output: 0

//Optional

List<Integer> list3 = Arrays.*asList*(-10, -10);

Optional<Integer> result3 = list3.stream().reduce(Integer::*max*); // output: Optional[-10]

List<Integer> list4 = Arrays.*asList*();

Optional<Integer> result4 = list4.stream().reduce(Integer::*max*); // output: Optional.empty

System.***out***.println(result4);

}

}

***Java 8 Date API and Time API***

*Java 7 : java.util.Date and java.sql.Date*

* + *Instant*
  + *Duration*
  + *LocalDate*
  + *Period*
  + *DateAdjuster*
  + *LocalTime*

*The Instant Class and the Duration Class*

*java.time. It has new key concepts. It completely replaces the java.util.Date class and the Calendar class*

*The first concept of this new API is the concept of Instant.*

*An Instant is a new class, it represents a point on the time line.*



* + *Instant 0 is the January the 1st, 1970 at midnight GMT*
  + *Instant.MIN is 1 billion years ago*
  + *Instant.MAX is Dec. 31 of the year 1,000,000,000*
  + *Instant.now() Now is a static method of the Instant class which represents the current Instant*

***Instant Object***

*Instant Object is immutable*

*Instant time = Instant.now();*

***Duration Object***

***public class*** *dummy {*

***public static void*** *main(String[] args) {*

*DateTimeFormatter formatter =*

*DateTimeFormatter.ofLocalizedDateTime( FormatStyle.****SHORT*** *)*

*.withLocale( Locale.****UK*** *)*

*.withZone( ZoneId.systemDefault() );*

*Instant Start = Instant.now();*

*Instant end = Instant.now().plus(3, ChronoUnit.****DAYS****);*

*System.****out****.println(formatter.format(end));*

*Duration elapsed = Duration.between(Start, end);*

***long*** *hours = elapsed.toHours();*

*System.****out****.println("Hours: "+ hours);*

*}*

*}*

*A Duration is the time elapsed between two different Instants.*

* + *toNanos(), toMilis(), toSeconds(), toMinutes(), toHours(), toDays()*
  + *minusNanos(), minusMilis(), minusSeconds(), minusMinutes()……..*
  + *plusNanos(), plusMilis(), plusSeconds(), plusMinutes(),……*
  + *multipliedBy(), dividedBy(),negated()*
  + *isZero(), isNegative()*

***LocalDate Class and the Period Class***

* + *There are still many cases that are not covered by the Instant concept.*
  + *Ex: if I say "Shakespeare was born Apr. 23rd, 1564", it is not covered by the Instant concept. Because this is just a date in the past, it doesn't have the nanoseconds precision,*
  + *Ex: I can say "let us meet at 1 pm and have lunch together". This is not an Instant neither, it's just an hour in a day*
  + *To cover above concepts java8 has LocalDate.*

***LocalDate #***

* + *A LocalDate is just a date with a day precision and not a nanosecond precision.*

*LocalDate now = LocalDate.now();*

*System.****out****.println("Now: "+now);*

*LocalDate dOB = LocalDate.of(1985, Month.****AUGUST****, 28);*

*System.****out****.println("DOB: "+dOB);*

***Period #***

* + *The first concept is the concept of Period.*
  + *The duration was the amount of time between two Instants*
  + *The Period is the amount of time between two LocalDates*

*LocalDate dOB = LocalDate.of(1985, Month.****AUGUST****, 28);*

*Period period = dOB.until(now);*

*System.****out****.println("# Period: "+period.getYears());*

***long*** *days = dOB.until(now, ChronoUnit.****DAYS****);*

*System.****out****.println("# days =" + days);*

***long*** *months = dOB.until(now, ChronoUnit.****MONTHS****);*

*System.****out****.println("# Months =" + months);*

***DateAdjuster/Temporal Adjusters***

*This is useful to add or subtract an amount of time to an Instant or a LocalDate.*

*LocalDate dateAdjuster = LocalDate.now();*

*LocalDate nextSunday = dateAdjuster.with(TemporalAdjusters.next(DayOfWeek.****SUNDAY****));*

*System.****out****.println("Next Sunday: "+nextSunday);*

*It have a bunch of static methods in the TemporalAdjusters class to adjust an Instant or a LocalDate.*

* + *firstDayOfMonth(), lastDayOfMonth(),*
  + *firstDayOfNextMonth(), lastDayOfNextYear().*
  + *firstDayOfNextMonth(), firstDayOfNextYear()*
  + *next(DayOfWeek.****SUNDAY****));*
  + *nextOrSame(DayOfWeek.****SUNDAY****));*
  + *previous(DayOfWeek.****SUNDAY****));*
  + *previousOrSame(DayOfWeek.****SUNDAY****));*

***LocalTime***

*The 6th concept is the concept of LocalTime. A LocalTime is just a time in a day.*

*LocalTime localTimeNow = LocalTime.now();*

*System.****out****.println("Local Time Now: " +localTimeNow);*

*LocalTime time = LocalTime.of(10, 20);*

*System.****out****.println("Time: " +time);*

*if I suppose that the time to go to bed is 23 at night, I can call that bedTime = LocalTime.of(23, 0). Then the wakeUpTime will be 8 hours later, so it's bedTime.plusHours 10 and it will represent 7:00 a.m.*

*LocalTime bedTime = LocalTime.of(23, 0);*

*LocalTime wakeupTime = bedTime.plusHours(8);*

***ZonedTime***

* + *There are many time zones, in fact, considering that we also have summer time and winter time, etc.*
  + *The zones are available from a class called ZoneId. I can get the Set of all the zones by calling the getAvailableZoneIds.*

*Set<String> allZonesIds = ZoneId.getAvailableZoneIds();*

* + *Special time zone,*

*ZoneId ukTimeZone =ZoneId.of("Europe/London");*

* + *Then, how can I link a special time to a zoned time*

|  |  |
| --- | --- |
| *ZonedDateTime currentMeeting = ZonedDateTime.of(1985, Month.****AUGUST****.getValue(), 28,* | *// year/month/day* |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| *10, 0, 0, 0,* |  |  |  |  | *// h/mn/s/nano* |

|  |  |  |
| --- | --- | --- |
| *ZoneId.of("Europe/London"));* |  | *// Zone* |

* + *I want to schedule our next meeting one month later, so I just need to add a period of one month to our currentMeeting so the pattern is really very easy to understand just by reading it. But this meeting also involved people in the U.S., so I want to invite them and I want to send to them the proper time within their time zone, so I will convert the nextMeeting ZonedDateTime inside their own time zone by just calling the withZoneSameInstant.*
  + *So it is the same Instant of the time line, because of course, if it's not, we'll have problem to meet, but it is not the same hours since the time zone is not the same.*
  + *So I just call this withZoneSameInstant and passing it the ZoneId of here "US/Central"*

*ZonedDateTime nextMeeting = currentMeeting.plus(period.ofMonths(1));*

*ZonedDateTime nextMeetingUS = nextMeeting.withZoneSameInstant(ZoneId.of("US/Central"));*

***Printing Dates and Times: The DateTimeFormatter***

*System.****out****.println("Date Time: "+DateTimeFormatter.****ISO\_DATE\_TIME****.format(nextMeetingUS));*

*System.****out****.println("Date: "+DateTimeFormatter.****ISO\_DATE****.format(nextMeetingUS));*

*System.****out****.println("Date: "+DateTimeFormatter.****RFC\_1123\_DATE\_TIME****.format(nextMeetingUS));*

***Bridges Between the APIs***

Machine generated alternative text:
Bridges Between the APIs 
How to interoperate with the legacy Date API? 
Instant & Date: 
Date date = Date.from(instant); 
// API - > legacy 
= date. tolnstant(); 
// legacy -Y new API 
Instant instant 
Instant & TimeStamp: 
TimeStamp time = TimeStamp.from(instant); 
Instant instant = time.tolnstant(); 
LocalDate & Date : 
Date date = Date. from(10ca1Date); 
LocalDate localDate = date.toLoca1Date(); 
LocalTime & Time 
Time time = Time. from(10ca1Time); 
Local Time local Time = time.toLoca1Time(); 
// API 
legacy 
// API -Y 
legacy 
API -Y 
legacy 
legacy 
-Y new API 
legacy 
-Y new API 
legacy 
-Y new API 

***Strings IO***

*Introduction and Outline*

* + *String class.*
  + *several new methods on Number that will allow much more efficient code.*
  + *browse through the many new methods on the map interface. Now we can do many new things on maps like merging maps, building bitmaps in just one line of code,*
  + *notion of repeatable annotations. Repeatable annotations are new things of Java 8 that we'll present here.*

***Strings and StringJoiner***

***Creating a stream on string***

*String s = "Hello world!";*

*IntStream stream = s.chars(); // creates a Stream on the letters of s*

*stream.mapToObj(letter -> (char)letter)*

*.map(Character::toUpperCase)*

*.forEach(System.out::print);*

*String s1 = "Hello";*

*String s2 = "World!";*

*String s3 = s1+s2;*

*some people will tell, don't use the concatenation with the + sign, it's not efficient because of the multiple creations and deletions of intermediary strings.*

*So better to use the StringBuffer class and append method can append nearly everything on the String, another string. call the toString method to build the string itself. But StringBuffer is synchronized*

*StringBuffer sb1 = new StringBuffer();*

*sb1.append("Hello");*

*sb1.append(" ").append("world"); // can be chained*

*String s = sb1.toString();*

*StringBuilder. It is basically the same class as the StringBuffer. but not synchronized, it is supposed to execute much quicker than the StringBuffer.*

*but JVM is smart enough to realize if a StringBuffer is not called in a multi-threaded environment, it will remove the synchronization mechanisms, so in fact, most of the time the StringBuffer is as efficient as the StringBuilder.*

*from JDK 7, the concatenations of String used with the + sign already compiles with the StringBuilder so you don't have to use a StringBuilder at all*

***StringJoiner***

*in JDK 8, it becomes even simpler with this magic class called StringJoiner.*

*StringJoiner sj = new StringJoiner(", ");*

*sj.add("one").add("two").add("three");*

*String s = sj.toString();*

*System.out.println(s);*

*Output:*  one, two , three

*The StringJoiner joins Strings with built-in separator.*

I can also pass a prefix and a postfix as arguments to a StringJoiner.

StringJoiner sj= **new** StringJoiner(", ","{","}");

String s = sj.toString();

System.***out***.println(s);

Output: {}

StringJoiner sj= **new** StringJoiner(", ","{","}");

sj.add("one");

String s = sj.toString();

System.***out***.println(s);

Output: {one}

StringJoiner sj= **new** StringJoiner(", ","{","}");

sj.add("one").add("two").add("three");

String s = sj.toString();

System.***out***.println(s);

Output: {one, two, three}

StringJoiner can be used directly from the String class.

String s = String.*join*(", ","one", "two", "three");

System.***out***.println(s);

new static method in the String class, join

***Java I/O***

***Reading Text Files***

|  |  |
| --- | --- |
| ***Without Path***  **try**(BufferedReader reader = **new** BufferedReader(**new** FileReader(**new** File("d:/temp/debug.log")));) {  Stream<String> stream = reader.lines();  stream.filter(line -> line.contains("ERROR"))  .findFirst()  .ifPresent(System.***out***::println);  } **catch** (IOException e1) {  e1.printStackTrace();  } | ***With Path***    Path path = Paths.*get*("d:" , "tmp", "debug.log");  **try**(Stream<String> stream = Files.*lines*(path)){  stream.filter(line -> line.contains("ERROR"))  .findFirst()  .ifPresent(System.***out***::println);  } **catch** (IOException e1) {  e1.printStackTrace();  } |

***New Methods on Iterable, Collection, and List***

bridges that have been made between the Collection API and the Stream API.

* + methods on the Collection interface were the stream() and parallelStream()
  + spliterator()
    - spliterator as in Iterator to a stream.
    - Iterator is to Collection likewise spliterator is to the Stream.

What does this split stuff mean?

spliterator is able to split a Collection or a Stream to be able to parallelize the processing of that Stream or or that Collection.

Note: A method reference is strictly equivalent to a Lambda expression.

***removeIf()***

Collection<String> strings = Arrays.*asList*("one", "two", "Three", "four");

//will not work if list is unmodifiable

Collection<String> list = **new** ArrayList<>(strings);

//returns true if the list has been modified

**boolean** b = strings.removeIf(s -> s.length() > 4);

System.***out***.println(strings.stream().collect(Collectors.*joining*(",")));

* + This removeIf method takes a predicate, the predicate just takes an object and returns a Boolean.
  + the object this predicate will have as a parameter is an object from the Collection, and if the predicate returns true, then this element will just be removed.
  + So what does this code do? It will just remove a String from the list if its length is strictly greater than 4, that is if its length is 5 or more.
  + So what we can see is that the 3 string has been removed and the other one has been kept.

***replaceAll ()***

List<String> strings = Arrays.*asList*("one", "two", "Three", "four");

//will not work if list is unmodifiable

List<String> list = **new** ArrayList<>(strings);

//returns true if the list has been modified

list.replaceAll(String::toUpperCase);

System.***out***.println(list.stream().collect(Collectors.*joining*(",")));

* + It takes a function as a parameter, which is a special type of function called a unary operator.
  + That takes an element from that list and that returns another element of the same type.
  + And then all the elements of that list will be processed by the separator one by one

***Sort()*** It just sorts the content of a list by providing a Comparator.

List<String> strings = Arrays.*asList*("one", "two", "Three", "four");

//will not work if list is unmodifiable

List<String> list = **new** ArrayList<>(strings);

list.sort(Comparator.*naturalOrder*());

System.***out***.println(list.stream().collect(Collectors.*joining*(",")));

Output# four, one, three, two

***Reading Directory Entries***

|  |  |
| --- | --- |
| ***Ex1:***  *Path path = Paths.get("d:", "tmp ", "debug.log");*  ***try*** *(Stream<String> stream = Files.lines(path)) {*    *stream.filter(line -> line.contains("ERROR"))*  *.findFirst()*  *.ifPresent(System.****out****::println);*  *}* ***catch*** *(IOException ioe) {*    *}* | ***Ex2:***  *Path path = Paths.get("c:", "Users", "hvyww", "Downloads");*  ***try*** *(Stream<Path> stream = Files.walk(path, 2)) { //2 is level of depth to get the files*    *stream.filter(line -> path.toFile().isDirectory())*  *.forEach(System.****out****::println);*  *}* ***catch*** *(IOException ioe) {*    *}* |

***Comparator: Patterns and Utilities***

***Comparator***

*// comparison using the last name*

*Comparator<Person> compareLastName =* ***new*** *Comparator<Person>() {*

*@Override*

***public int*** *compare(Person p1, Person p2) {*

***return*** *p1.getLastName().compareTo(p2.getLastName());*

*}*

*};*

*this code most of the time will not work because I will get NullPointerExceptions, either Person p1 or Person p2 is null*

*so we need to add NullPointerException handler.*

***Chain Comparators***

*Ex: first to compare using the last name and if the last names are equal, then compare the first name*

*// comparison using the last name then the first name*

*Comparator<Person> compareLastNameThenFirstName =* ***new*** *Comparator<Person>() {*

*@Override*

***public int*** *compare(Person p1, Person p2) {*

***int*** *lastNameComparison= p1.getLastName().compareTo(p2.getLastName());*

***return*** *lastNameComparison== 0 ? p2.getName().compareTo(p2.getName());*

*}*

*};*

* + *JDK 8 got this new method in the Comparator interface, new static method* ***comparing()***
  + *what do I give to comparing I just give what is called a key extractor.*
  + *A key extractor is a regular function, it takes an instance of the object the comparator will compare, and extracts a given field of that object.*
  + *here, Person:: is just a regular Lambda expression written with a method reference. What does it take? This method reference takes a Person as a parameter and returns the last name of that person.*

*Comparator<Person> compareLastName= Comparator.comparing(Person::getLastName);*

* + *So what does this Comparator do?*

*It just compares Persons in the last name sense,*

* + *which is very natural. When I read it, Comparator.comparing.LastName, I probably immediately know that what I'm comparing are persons using their last name.*
  + *And if I want to chain those Comparators, I can use the* ***thenComparing()*** *default method defined on the Comparator interface. This* ***thenComparing()*** *takes the same kind of arguments as the Comparing method and it will just chain the comparison of the last name and if the last names are equal, then compare the persons using their first name.*

***reversedComparator()***

*if I want to sort a list of persons with a reversed Comparator.*

*Well, I have another default method,* ***reversed()*** *that will just take the same Comparator, but in the reversed order.*

*Comparator<Person> reverseComp = Comparator.reverseOrder();*

***naturalOrder()*** *-> natural Comparator, which will compare comparable objects. Suppose I want to compare strings, well I can compare them in their naturalOrder, which is, of course, the alphabetical order with this static method from the Comparator interface.*

*Comparator<Person> compareLastNameThenFirstName = Comparator.comparing(Person::getLastName).thenComparing(Person::getName);*

*Comparator<Person> reverseComp = Comparator.reverseOrder();*

*Comparator<String> c = Comparator.naturalOrder();*

*Comparator<String> c = Comparator.nullsFirst(Comparator.naturalOrder());*

*Comparator<String> c = Comparator.nullsLast(Comparator.naturalOrder());*

*Comparator<Person> compareLastName= Comparator.comparing(Person::getLastName);*

***Numbers, Method References, and Hashcodes***

* + *new useful methods defined on the Number types.*
  + *three methods are sum, max, and min, so they are available on long, integer, float, and double.*
  + *Long.max is just the max of two longs passed as parameters to this new static method.*
  + *It can be used also to create reduction operations*

*BinaryOperator<Long> sum = (l1, l2) -> l1 + l2;*

*= (l1, l2) -> Long.sum(l1, l2);*

*= Long::sum;*

* + *I can also write it with a method reference.*

*HashCode Computation*

* + *Now the hashCode of an integer is the integer itself, so it doesn't make a lot of sense to compute thehashCode of an integer by calling a method,*
  + *but it is not the case for a long, where there is a special algorithm to compute the hashCode of a Long.*
  + *I create a Long on this Long primitive type, so this is creating a wrapping object on the long primitive type, and I call the hashCode method.*

*// JDK 7*

***long*** *l = 3141592653589793238L;*

***int*** *hash = new Long(l).hashCode(); // -1985256439*

* + *Now this has a cost because I need to convert this long primitive type in a wrapping object, so there is a cost to box and unbox this primitive type each time I need to compute this hashCode.*
  + *Thus, the creation of a static method hashCode on all the number wrapping type, this time I can compute the hashCode of a Long without having to pay the cost of boxing and unboxing this long primitive type.*

*// JDK 8*

***long*** *l = 3141592653589793238L;*

***int*** *hash = Long.hashCode(l); // -1985256439*

***Map: Enhancements of Existing Methods***

*New methods have also been added to the Map interface.*

***forEach()***

*Map<String, Person> map= ...;*

*map.forEach((key, person) ->*

*System.out.println(key + " "+ person);*

* + *It is basically the same kind of method as the one defined on the Iterable interface, and it takes a BiConsumer as a parameter.*
  + *This BiConsumer is just a regular Consumer that takes two parameters instead of one, and those two parameters are, of course, the key and the value pairs of the Map.*

***get()***

*Person p = map.get(key); // p can be null!*

* + *We had this method on the very first version of the Map interface, but it has a little problem. when I call the get(key) method it can return me null, which is fine, but what does it mean for the get method to return null. Does it mean that the key is absent from the Map or does it mean that the key is associated to a null value.*

***getOrDefault()***

*Person defaultPerson= Person.DEFAULT\_PERSON;*

*Person p = map.getOrDefault(key, defaultPerson); // JDK 8*

* + *Now in Java 8, we have this new method called getOrDefault.*
  + *we still pass the key as a parameter, but this time we pass the value that should be returned in case the key is not present in the map,*

***put()***

* + *java 8 have a modification to the put value.*
  + *When we call the put(key) value method, it automatically erases the existing value associated with that key.*

*map.put(key, person); // will erase an existing person // JDK 7.0*

* + *Now we have a new version of the put method called putIfAbsent.*

*map.putIfAbsent(key, person); // JDK8*

* + *If the key is not present in the Map, then I will associate that key to the given value here, the object person. This is new in JDK 8, and very useful too.*

***Replace()***

*map.replace(key, person);*

* + *The replace method replace the value associated with the given key,*
  + *so if I call replace(key, person), then no key will be added to the Map, but merely the old person will be erased and the new person will replace it.*
  + *Now, I can also call replace(key) with the expected old value and the new value that should replace that value.*

*map.replace(key, oldPerson, newPerson);*

***replaceAll()***

*map.replaceAll((key, oldPerson) -> newPerson);*

* + *This replaceAll method takes a Lambda expression.*
  + *This Lambda expression is called the remapping function and it's, in fact, a BiFunction that takes a key and a value, and that will produce a new value following this key value pare.*

***remove()***

*map.remove(key); // JDK 7*

*map.remove(key, person); // JDK 8*

* + *If I call remove(key), it just removes the key or at least try to do that. If the key is not present, it will not do anything.*
  + *Now I've got a new version of the remove method that will take both the key and the expected value as a parameter. So if that key is not associated with the provided value, then nothing will be removed from the Map.*

***Map: The Compute and Merge Methods***

* + *I also have a new family of methods on the interface map.*
  + *Those methods are* 
    - *Compute ()*
    - *computeIfPresent ()*
    - *computeIfAbsent ()*
  + *Now what happens when I call compute?*

*Map<String, Person> map= ...;*

*map.compute(key, person, (key, oldPerson) -> newPerson);*

* + *I need to provide a key, a given value, and a BiFunction that takes a key and a value, and that will compute a new value from this key and value.*
  + *Now this value does not need to be associated with the key in the Map and the key does not need to be present in the Map.*
  + *What happens if it is not?* 
    - *the BiFunction will be called with the key value and with the null value, since there is no existing value associated with this key.*
  + *new value will be computed after the given key and the null value.*

***computeIfPresent()***

*map.computeIfPresent(key, person, (key, oldPerson) -> newPerson);*

* + *the computeIfPresent method works is almost the same, but this time the key and the Person value associated with it should be present on the Map.*
  + *If it is not the case, the BiFunction passed as a parameter will not be evaluated and the Person value will not be changed.*
  + *the BiFunction doesn't have to assume that the oldPerson parameter cannot be null since the person is always present on the Map when this BiFunction is called.*

***computeIfAbsent()***

*map.computeIfAbsent(key, key -> newPerson);*

* + *computeIfAbsent this time, instead of IfPresent. This method works the same, we just provide a key. If this key is not present on the Map, then we call the function passed as a parameter that will compute a new value associated with that key.*

* + *all three methods really return the computed value*
  + *In the case of the computeIfAbsent, it allows a very simple pattern to compute bimaps.*

*Map<String, Map<Integer, Person>> bimap= ...;*

*Person p = ...;*

*bimap.computeIfAbsent(key1, key -> newHashMap<>()).put(key2, p);*

* + *What is a bimap?* 
    - *It is a map in which the value is itself another map.*
    - *So suppose we have such a bimap, a Map in which the key are Strings and the value another Map with an integer as a key and a person as a parameter.*
  + *And suppose we want to associate this person to this kind of pair of double key? We just need to call bimap.computeIfAbsent, the given key, and then a number expression that will just create a new HashMap out of that key.*
  + *In fact, that key value is not used to compute that HashMap, it is just a brand-new empty HashMap.*
  + *And since this HashMap is returned in all the cases, we can directly put the key value pair in that value map.*

*Map<String, Map<Integer, Person>> bimap= ...;*

*Person p = ...;*

*bimap.computeIfAbsent(key1, key -> newHashMap<>()).put(key2, p);*

*The first case #*

* + *if key 1 is already present in the Map, then the function expressed as a Lambda expression and passed as a parameter is not evaluated at all, but we return the existing HashMap that has been put as a value associated with key1.*
  + *So, we can directly call put on this existing HashMap and associated key2 with the Person p.*
  + *Now if key1 is not present as the map, then the Lambda expression will be executed, a new HashMap will be built and associated to the key1 key in the first Map and then this new empty HashMap will also be written by the computeIfAbsent method.*
  + *Then this time we could put on this new HashMap and can associate person to the new key, key2.*

*Merge()*

*Map<String, Person> map= ...;*

*map.merge(key, person, (key, person) -> newPerson);*

* + *Merge assumes that the key is present in the map or associated to a null value. So when we call merge, we pass a key and a person as a parameter,*
  + *the key should be present on the map, and the person should match the person already associated to the key in the Map, and*
  + *then we compute a newPerson with the Lambda expression passed as a parameter. This Lambda expression is just an implementation of a BiFunction that takes the key and the value of the Map as a parameter, and builds a new value out of those two parameters.*

***Annotations***

* + Java 8 brings the concept of "multiple annotations".

**what was the problem?**

**Java 7**

* + Suppose want to test this case with more than one parameter
  + I would like to do is to add, in fact, this same annotation more than once to this even method.
  + The problem is I cannot apply more than once, the same annotation on a given element.
  + Java 7 solution is the following: I need to wrap this annotation in another annotation, and this other annotation will take an array of the annotation I want to add as an attribute.
  + wrapping annotation called TestCases, and this TestCases take an array of TestCase as an attribute.

JAVA 7

@TestCases({

@TestCase(param=1, expected=false),

@TestCase(param=2, expected=true)

})

Public Boolean even (int param) {

Return param% 2 == 0;

}

* + This pattern has been found all around the JDK user in Java SE or in Java EE, for instance, in the JPA API.

**Java 8**

* + can write same annotations more than once on any element.
  + wrapping annotation is automatically added for us by the compiler.

Java 8

@TestCase(param=1, expected=false)

@TestCase(param=2, expected=true)

Public Boolean even(int param) {

Return param% 2 == 0;

}

* + How can I do that
    - I just need to create the annotations as usual, so the first annotation is the annotation I want to use.Here is TestCase, and the second annotation is the wrapping annotation TestCase's.
  + Then what should I do?
  + just add this repeatable annotation on the TestCase annotation to tell that this annotation will be repeatable.
  + Now the Compiler needs to know which wrapping annotation it will use, so I just provide this wrapping annotation as an attribute to the repeatable annotation.
  + In our example, it is the TestCases.Class.

@NonNull

To declare that a variable should be null

Private @NonNull List<Person> persons= ... ;

To declare that list should not be null and should not contain null values

Private @NonNull List<@NonNull Person> persons= ... ;