**Java Functional Mapping**

Please note your grade for this lab goes in BOTH test and programs categories.

Java functional/lambda code is kind of magical, able to solve certain problem types with just 1 or 2 lines of code.

**replaceAll() Method**

In general, "mapping" is a strategy that starts with a collection of items, and runs a function on each item individually to compute a new value for that item. The Java replaceAll() method is an easy way to do mapping.

As a first example, we'll use the "doubling" problem: given a list of integers, double each integer in the list. Here is a mapping solution:

public List<Integer> doubling(List<Integer> nums) {

**nums.replaceAll(n -> n \* 2);**

return nums;

}

**Lambda Syntax**

How does the above code work? First look at this snippet of code: **n -> n \* 2**

Mapping uses a little function that takes in one item and computes the new value for that item. For the doubling problem, we want a function that takes in a single Integer and returns double its value.

The code snippet n -> n \* 2 is that function in "lambda" syntax: take in a number and return it multiplied by 2.

The first "n" is the parameter name, followed by "->" and then the expression for the new value. Lambda’s work is great for simple little functions like this example, but { … } can be used to include multiple statements allowing the expression to accomplish more complex requirements as will be seen later in this assignment ☺. The data type, Integer in this case, is determined by the lambda's context.

**replaceAll() + Lambda**

The method list.replaceAll(lambda) calls the provided lambda function once for each item in the collection to compute each item's new value which is stored back into the collection. replaceAll() changes the original collection in-place and does not return anything. Therefore, after the collection is changed, we need the final return nums;

The API for the replaceAll method is on the next page.



Note: UnaryOperator is an Interface

With this functional/mapping solution, we don't have to write any sort of loop to iterate over the collection, and we don't mess with index numbers. We provide the lambda of what we want done to the items, and replaceAll() does all the bookkeeping for us. Nice!

**replaceAll() Limitations**

Limitation 1: since replaceAll() stores the new value back into the original collection, it only works if the new value is the same type as the original, e.g. int → int, or String → String. Fortunately, that's the common case. To change type, use streams (below).

Limitation 2: mapping works best where each item is handled on its own. It's not easy to make a relative reference like an element "to the left" of another. For a relative problem like that, a traditional loop is probably better.

**Loop vs. Lambda Comparison**

Here's a non-lambda solution to the doubling problem, it's a lot longer!

public List<Integer> doubling(List<Integer> nums) {

List<Integer> result = new ArrayList<>();

for (Integer n:nums) {

result.add(n \* 2);

}

return result;

}

Note: Both implementations are NOT equivalent. Is it obvious the Mapping method changes nums, while the Loop solution does not modify nums.

Lambdas don't work for all cases, but for the pattern of transforming each element in a collection, lambdas work great.

**Looking ahead: Java Functional Stream**

The Java "stream" classes provide another, more flexible way to do mapping. Here is the solution to the doubling problem written with the stream system:

public List<Integer> doubling(List<Integer> nums) {

return nums.stream()

.map(n -> n \* 2)

.collect(Collectors.toList());

}

The stream() call sets up the list for mapping. That call is followed by any number of map(lambda) calls to map (or filter) the item values as we have done. Finally the collect() call collapses the stream down to a list or set or whatever, which is returned. The stream system returns a new collection rather than changing the original.

**Remember this, it will be useful in the third and final part of this assignment.**

☺, ☺, ☺: It should also be noted that the filter, map and/or collect methods (magically?) determine from content what type of List should be return allowing the Stream to process a List<Integer> and return a List<String>.

**Remember this, it will be useful in the third and final part of this assignment.**

**replaceAll() - Your Assignment**

The doubling problem required just 1 line of code to do the work, and a similar style will work for any problem where we want to replace each item with a computed value. Each of these problems puts the main work in 1 line of code:

See Java8ReplaceAllPractice class for your assignment. Each method in this assignment can be implemented with a single replaceAll(lambda) immediately followed by a return.

\* hint: for the removePhrase method (the last method in this class), I did use the String method: replace(CharSequence target, CharSequence replacement). Realize CharSequence is an interface that String implements allowing you to pass it two Strings as parameters.

\* All three parts of this program are to be submitted to the folder: java8 All programs

**Continue for more Java 8 fun**

**Next Step: Filtering**

**Java Functional Filtering**

"Functional" programming in Java (and other languages) is a great technique that can solve certain problem types with just 1 or 2 lines of code. This section looks at "filtering" added to mapping.

**Filtering - removeIf()**

Filtering uses a lambda to decide which items should be kept or removed from a collection.

In Java, the removeIf( *lambda* ) takes a lambda function which takes in a single item, and returns boolean true if it should be removed, and false otherwise.

As a first example, we'll use the "noNeg" problem: given a list of integers, returns a list of the integers, omitting any which are less than 0. Here is the noNeg solution code:

public List<Integer> noNeg(List<Integer> nums) {

**nums.removeIf(n -> n < 0);**

return nums;

}

The **list.removeIf( *lambda* );** method does all the work, removing any items where the lambda function returns true:

The lambda function n -> n < 0 has the parameter n and boolean expression n < 0 and the boolean expression evaluates to true n < 0, and n will be removed.

**Mapping and Filtering Together**

The obvious next step is mapping the items in a list first, and then filtering the resulting values. This can be done with 2 lines of code: nums.replaceAll(...) to modify the list, and then nums.removeIf(...) to filter (omit or delete) the resulting values.

**replaceAll() && removeIf() - Your Assignment**

See Java8FilteringPractice class for your assignment. Each method in this assignment can/should be implemented with a call to removeIf(lambda). The last two methods can/should use replaceAll(lambda) before using removeIf(lambda) (filtering).

\* All three parts of this program are to be submitted to the folder: java8 All programs

**Java Stream Filter**

The Java stream system provides more complicated mapping and filtering. Here is the solution to noNeg() using Java streams. Note that the filter() function has the opposite boolean logic as removeIf(): true means keep the element, false means remove it. A .map() stage can be inserted before or after the .filter() to combine mapping and filtering.

Remember: The stream system returns a new collection rather than changing the original.

public List<Integer> noNeg(List<Integer> nums) {

return nums.stream()

.filter(n -> n >= 0)

.collect(Collectors.toList());

}

An implementation of shortenString() method uses a List of Strings and returns a new (and different) List with the first and last letter removed from each String in the List while removing all Strings with a length shorter than (not equal) the parameter minLength after the first and last letter have been removed. Here is the solution to shortenString() using Java streams.

public static List<String> shortenString(List<String> words, int minLength)

{

List<String> ans = words.stream()

.map(x -> x.substring(1, x.length()-1))

.filter(x -> x.length() >= minLength)

.collect(Collectors.toList());

return ans

}

In the above solution, note:

* .map(x -> x.substring(1, x.length()-1)) is then used to create a List with the first and last character removed from each string.
* .filter(x -> x.length() >= minLength) is used to remove each String with length less than minLength. (i.e., keeps all Strings with required length)
* If the order of the map and filter methods is reversed (filter first then map), the Strings with an original length less than minlength are removed, then the first and last element of each String is removed.

You should also notice that the above solution encounters a runtime error when a String with length less than two is encountered. To avoid the runtime error, the map method is modified to handle Strings with length < 2 differently than Strings with length >= 2. As mentioned on the first page, { and } are used to modified the shortenString method. The modified version of shortenString method follows:

public static List<String> shortenString(List<String> words, int minLength)

{

List<String> ans = words.stream()

.map(x -> {

if(x.length() <2)

return "";

else

return x.substring(1,x.length()-1);

} )

.filter(x -> x.length()>=minLength)

.collect(Collectors.toList());

return ans;

}

**Map() && filter() - Your Assignment**

The Java stream system API has many other methods you may find useful including concat, distinct, forEach, generate, limit, max, min, reduce, skip, sorted and toArray. You can look here for the API: [Java Stream API](https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html). You will find some of these methods useful in completing the last couple methods in your final assignment.

See Java8StreamMapCollect class for your assignment.

In completing the final method, you may find the following useful ☺.

public boolean isAssignableFrom([Class](https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html)<?> cls)

**Parameters:**

cls – the Class object to be checked

**Returns:**

This method returns the boolean value indicating whether objects of the type cls can be assigned to objects of this class

**Sample code:**

Given: List<BetterTurner> rBot, Robot r, BetterTurned b, SuperBot s;

rBot.get(0).getClass().isAssignableFrom( r.getClass() ) returns false

rBot.get(0).getClass().isAssignableFrom( b.getClass() ) returns true

rBot.get(0).getClass().isAssignableFrom( s.getClass() ) returns true

**note:** SuperBot extends BetterTurned extends Robot

For more about streams, see the [Java Stream Docs](https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html)

\* All three parts of this program are to be submitted to the folder: java8 All programs