**Functional (Programming)**

The functional programming (FP) paradigm, in a general sense, involves operating on data using functions. Contrast this with operating on data given the exact steps to accomplish some task (and modifying state along the way), which is *imperative* / procedural programming, i.e. normal Java.

FP generally describes the logic of computation, rather than the exact order of the steps of computation. For more info on how FP fits into the Java ecosystem, see [here](https://winterbe.com/posts/2014/03/16/java-8-tutorial/). For an even more in-depth exploration of Java 8 and its FP elements, see [here](https://www.mkyong.com/tutorials/java-8-tutorials/).

1. The java.util.Arrays.toList method will return some number of comma-separated elements (known as "var args") as a List; very useful for test purposes. Examples:
   1. List<Integer> nums = new ArrayList<>(Arrays.asList(1, 2, 3));
   2. List<String> list = new ArrayList<>(Arrays.asList("hello", "and", "goodbye"));

**Note:** The value Arrays.toList returns must be passed to ArrayList's constructor as shown above, as it returns an immutable list and couldn't be changed if saved directly into a List variable.

1. Print each element in a List on a separate line.
   1. forEach - Called on a list, this method performs an action for each element in the ***stream*** (a sequence of elements). The action (a function) is what is supplied as an argument. Examples:
      1. (n -> System.out.println(n))
         1. Read as: for each element n in the stream of elements in the list, perform a System.out.println on parameter n.
            1. The type of n is inferred by the compiler.
            2. The -> operator separates the "parameter list" from the "method body".
         2. System.out.println(n) is the body of this "anonymous" method.
      2. Alternative syntax: (System.out::println)
         1. Invoke println on System.out (for each element in the stream).
         2. Can be used when there is no other transformation logic.
2. Remove all even numbers from a list.
   1. removeIf - This method expects a function that returns true when an element should be removed. Think of it expanding like this:

list.removeIf(

boolean removeIf(int **a**) {

return **a % 2 == 0**;

});

The portion in red is all that is needed for a complete lambda function. The parameter type and the return keyword can be omitted, as the compiler can infer them from the context of the method call. The -> operator separates the parameter list from the method body.

1. Print the number of odd numbers in a List<Integer>.
   1. stream - Converts the list into a stream of elements.
   2. filter - Returns a stream consisting of the elements of this stream that match the given condition. **Note:** filter has the opposite logic of removeIf.
   3. count - Returns the number of elements in the stream.
   4. You can improve clarity by lining up successive method calls as such:

nums.stream()

.filter(<args>)

.count();

1. Print all elements in a List<String> that begin with "a" on a separate line.
2. Double the value of all numbers in a List<Integer>.
   1. map - Applies the given function to list elements to transform them in some way. Returns a stream of the transformed elements.
   2. collect - Collect the stream elements, put back into a data structure.
      1. Note that map returns a Stream, it must be "collected" to turn it back into a List.
      2. Collectors.toList - Utility method to accumulate the elements of a stream into a List. Passed as an argument to collect.
   3. **Note:** List also has a replaceAll method that can accomplish the above in less code, however it will change the list calling the method (rather than making a new list). It's used like the removeIf method.
3. [Complete this CodingBat problem](https://codingbat.com/prob/p132748) using what you learned in the previous problem.
4. Apply 12% tax to all prices (values) in a List<Double>.
5. **In one line**, create a List<Integer>, sum the values in the list, and output the result.
   1. reduce - Reduces the elements in the stream using the supplied operation. The reducing process is commonly called a ***fold***, as all stream elements are folded into a single element.
      1. More accurately, reduce applies a binary operator to each element in the stream, where the first argument to the operator is the return value of the previous step and the second argument is the current element.
      2. Does this sound like recursion? (It's recursion.) Think of it like this (assuming the addition (+) operator is used):

if (list empty)

return 0;

else

return listElement + fold(list..1)

…resulting in 1 + (2 + (3 + (4 + (5 + 0))))

* + 1. Multiple parameters should be grouped with round brackets and separated with a comma (e.g. (a, b) -> a + b).
    2. **Note:** reduce returns an Optional, an object that represents something that may or may not be present (you don't always know if reducing the stream will produce a value). You can use the get method to return a Double (rather than the Optional).

1. Given a List<Integer> representing item costs, apply 12% tax and calculate the total cost of everything in the list.
2. Return the largest value in a List<Integer>.
   1. max - expects a Comparator, an interface that describes how to compare two values. The interface has a single method: int compareTo(T obj1, T obj2). You are creating a Comparator instance and overriding the compareTo method in-line, with a lambda function. Here is the older way of creating an anonymous class that overrides compareTo:

        Comparator<Integer> maxComparator = new Comparator<>() {

            @Override

            public int compare(Integer n1, Integer n2) {

                return n1.compareTo(n2);

            }

        };

1. Create a JButton that will print "Button clicked!" each time it is clicked.
   1. Some basic graphics code to get you started:

JButton button = new JButton("Click here");

JFrame frame = new JFrame("Button test"); //window to contain the button

//add your code here

frame.setSize(200, 200); //set the size of the window

frame.add(button); //add button to the window

frame.setVisible(true); //make frame visible

* 1. You are doing with a lambda what used to be done with an anonymous class:

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e){

System.out.println("Button clicked!");

}

});

Which could also have been done by adding another class that implemented the ActionListener interface (and using an instance of that class):

class ButtonListener implements ActionListener

{

@Override

public void actionPerformed(ActionEvent e){

System.out.println("Button clicked!");

}

}

//client code

button.addActionListener(new ButtonListener());

1. Get the age of the oldest Person in a List<Person> without implementing Comparable.
   1. mapToInt - Returns a stream of integers using the supplied function to map each element to an integer value.

Free code! Fresh from the code factory (feel free to use for testing):

List<Person> users = new ArrayList<>();

users.add(new Person("Sarah", 40));

users.add(new Person("Peter", 50));

users.add(new Person("Lucy", 60));

users.add(new Person("Albert", 20));

users.add(new Person("Charlie", 30));

class Person

{

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return this.name + ", " + this.age;

}

int getAge() { return this.age; }

}