Lab 9

1. Short Answer:
   1. What happens when the following code is executed?

public static void main(String[] args) {

IntStream ones = IntStream.*generate*(() -> 1).distinct();

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

}

Explain. What would be a quick way to fix this?

It will always loop, can not stop, we can use limit() to refer the loop number.  
  
b. You have a Stream of Strings called stringStream consisting of the values “Bill”,   
 “Thomas”, and “Mary”. Write the (one line of) code necessary to print this stream to the   
 console so that the output looks like this:

Bill, Thomas, Mary

public class prob1b {  
 static public void main(String[] args){  
 List<String> names=Arrays.*asList*("Bill","Thomas", "Mary");  
 System.*out*.println(names.stream().collect(Collectors.*joining*(",")));  
 }  
}

* 1. You have a Stream of Integers called myIntStream and you need to output both the  
     maximum and minimum values. Write code that efficiently accomplishes this.

IntStream.*range*(1,100).forEach(System.*out*::println);

1. Implement a method -- done in class

public static void printSquares(int num)  
which creates an IntStream using the iterate method. The method prints to the console the first num squares. For instance, if num = 4, then your method would output 1, 4, 9, 16. Note: You will need to come up with a function to be used in the second argument of iterate.

public static void printSquares(int num){  
 IntStream.*iterate*(1,n->n+1).limit(num).map(n->n\*n).forEach(System.*out*::print);  
}

1. Done in class --Create a method   
    Stream<String> streamSection(Stream<String> stream, int m, int n) which extracts a substream from the input stream stream consisting of all elements from position m to position n, inclusive; you must use only Stream operations to do this. You can assume 0 <= m <= n. A Java class has been provided for you in the lab folder for this lesson; implement the method streamSection given in that class, and test using the main method provided.

Stream<String> streamSection(Stream<String> stream, int m, int n){  
 stream.filter(s->s.length()>n).map(s->s.substring(m,n));   
}

1. Implement a method

**public** Set<String> union(List<Set<String>> sets)

by creating a stream pipeline that transforms a list of sets (of type String) into the union of those sets. Make use of the reduce method for streams.   
  
*Example*: The union method should transform the list [{“A”, “B”}, {“D”}, {“1”, “3”, “5”}] to the set {“A”, “B”, “D”, “1”, “3”, “5”}.

public Set<String> union(List<Set<String>> sets){  
 Set<String> ret=new HashSet<>();  
 sets.stream().forEach(set->set.stream().forEach(s->ret.add(s)));  
 return ret;  
}

1. In the package lesson9.labs.prob7a, there is an Employee class and a Main class, which has a main method that loads up a Stream of Employee instances.
   1. In the final line of the main method, write a stream pipeline (using filters and maps) which prints, *in sorted order (comma-separated, on a single line),* the full names (first name + “ “ + last name) of all Employees in the list whose salary is greater than $100,000 and whose last name begins with any of the letters in the alphabet *past* the letter ‘M’ (so, any letters in the range ‘N’--‘Z’).   
        
      For the main method provided in your lab folder, expected output is:   
       Alice Richards, Joe Stevens, John Sims, Steven Walters

public static void main(String[] args) {  
 List<Employee> list = Arrays.*asList*(new Employee("Joe", "Davis", 120000), new Employee("John", "Sims", 110000),  
 new Employee("Joe", "Stevens", 200000), new Employee("Andrew", "Reardon", 80000),  
 new Employee("Joe", "Cummings", 760000), new Employee("Steven", "Walters", 135000),  
 new Employee("Thomas", "Blake", 111000), new Employee("Alice", "Richards", 101000),  
 new Employee("Donald", "Trump", 100000));  
  
 // your stream pipeline here  
 String msg=list.stream()  
 .filter(e->e.salary>100000)  
 .filter(e->{char c=e.lastName.charAt(0);return c>='M'&&c<='Z';})  
 .map(e->e.firstName+" "+e.lastName)  
 .sorted()  
 .collect(Collectors.*joining*(","));  
 System.*out*.println(msg);  
}

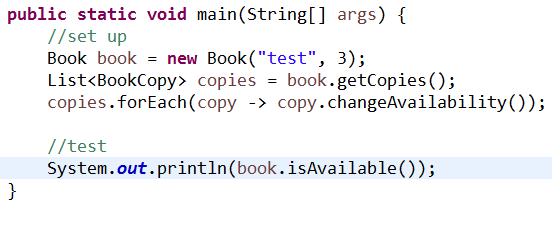
* 1. Turn your lambda/stream pipeline from part (a) into a Lambda Library element, following the steps in the slides. First, create a class LambdaLibrary; this class will contain only public static final lambda expressions. Then, identify the parameters that need to be passed in so that your lambda/stream pipeline can operate properly. Finally, think of a function-style interface (Function, BiFunction, TriFunction, etc) that can be used to accommodate your parameters and then name your pipeline, with the function-type interface as its type (as in the slide example). Call your Library element in the main method instead of creating the pipeline there, as you did in part (a).

public class LambdaLibrary {  
 public static final TriFunction<List<Employee>,Integer,Character,String> *FindEmplyee*=  
 (list,minSalary,paseLetter)-> list.stream()  
 .filter(e->e.salary>minSalary)  
 .filter(e->{char c=e.lastName.charAt(0);return c>=paseLetter&&c<='Z';})  
 .map(e->e.firstName+" "+e.lastName)  
 .sorted()  
 .collect(Collectors.*joining*(","));  
  
 public static void main(String[] args) {  
 List<Employee> list = Arrays.*asList*(new Employee("Joe", "Davis", 120000), new Employee("John", "Sims", 110000),  
 new Employee("Joe", "Stevens", 200000), new Employee("Andrew", "Reardon", 80000),  
 new Employee("Joe", "Cummings", 760000), new Employee("Steven", "Walters", 135000),  
 new Employee("Thomas", "Blake", 111000), new Employee("Alice", "Richards", 101000),  
 new Employee("Donald", "Trump", 100000));  
 System.*out*.println(*FindEmplyee*.apply(list,10000,'N'));  
 }  
}

1. In the package lesson9.labs.prob8, a Main class is provided that is essentially the same as the one used in Problem 7. Comments appear in the main method that indicate two queries that need to be executed. As in Problem 7, create a class LambdaLibrary that will store implementations of these queries as lambda pipeline expressions. Then call these expressions in the main method to verify they produce the expected results.

public class Main {  
  
 public static void main(String[] args) {  
 List<Employee> list = Arrays.*asList*(new Employee("Joe", "Davis", 120000), new Employee("John", "Sims", 110000),  
 new Employee("Joe", "Stevens", 200000), new Employee("Andrew", "Reardon", 80000),  
 new Employee("Joe", "Cummings", 760000), new Employee("Steven", "Walters", 135000),  
 new Employee("Thomas", "Blake", 111000), new Employee("Alice", "Richards", 101000),  
 new Employee("Donald", "Trump", 100000));  
  
 // print the number of Employees in list whose salary > 100000 and whose  
 // last name begins  
 // with a letter that comes after the letter 'E'  
 System.*out*.println(*FindEmplyee*.apply(list,100000,'E'));  
 // print a list of sorted full names - all upper case -- of Employees  
 // with  
 // salary > 85000 and whose first name begins with a letter that comes  
 // before the letter 'R'  
 System.*out*.println(*FindEmplyee*.apply(list,85000,'R'));  
 }  
  
}

1. In the folder lesson9.labs.prob9 there are classes Book and BookCopy, as in the Library project. Use a lambda/stream pipeline to implement an isAvailable() method in Book that uses the stream operation reduce (Hint: a Book is available if copy1 is available OR copy2 is available OR…). To test your code, add a Main class to the package and run the following main method:



public boolean isAvailable(){  
 return copies.stream().filter(c->c.isAvailable()).findFirst().isPresent();  
}