Lab 2, CSC/CPE 203
static methods v. instance methods
Due: 10/13

Orientation

This lab explores writing static methods versus instance methods in Java. Both are useful ways to get something done (a computation), but as we develop more complex code, there are many reasons to strongly associate a computation with a specific instance of an object (associate data and computation together in one object).

Objectives

- To write **overloaded** methods that operate on objects of different types
- To write **static methods** to perform operations on provided objects
- To write **instance methods** to perform similar operations
- To able to write your own test cases to check that the code you write does what you expect it to do

Resources

Same as the last two labs
You can get the base code from Canvas Week 2 activities.

Greenhouse gases

Emissions of gases that create a greenhouse effect have resulted in large-scale shifts in global weather patterns. A *greenhouse effect* occurs when the Earth absorbs sunlight and then radiates it as heat. Gases emitted largely due to human activities—like carbon dioxide and methane gas—have collected in the atmosphere, slowing the rate at which this heat leaves the atmosphere and escapes into space. This has resulted in large-scale changes in global weather patterns, i.e., climate change.

In this lab, we will use Object-oriented design principles to (1) model some data about global greenhouse gas emissions (obtained from the Emissions

Database for Global Atmospheric Research), and (2) use the resulting classes to help answer some research questions about greenhouse gas emissions.

Part 1: Classes and Overloaded Methods

In the provided **part1** subdirectory, implement the following classes with these general requirements.

- Each instance variable must be **private**
- Provide constructors for creating new instances of the class with values specified as parameters
- Provide the appropriate “accessor”/“getter” methods for the instance variables (see PartOneTestCases.java for names)
- You may add methods beyond those required, but any such additional method must be **private**

Automaton Warning: You are an intelligent person; you should not just mechanically apply these rules only to satisfy a style checker in order to complete the lab. Instead, consider the reasons for these rules, what violating them exposes to the user/client of the corresponding class, and what following them guarantees to you, the implementer. If you aren’t sure, ask your neighbor, they might have thoughts of their own.

Requirements specific to each class are described below.

Classes

`Emission.java`

The Emission class represents the emissions in kilo-tons of the greenhouse gases carbon dioxide (CO2), nitrous oxide (N2O), and methane gas (CH4). Each emission amount is represented as a Double.

So for example, an Emission might look like this:

```
co2: 288000.0,

n2o: 684000.0,

ch4: 4690000.0
```

Note: The Emission class should have a constructor that takes double values for the CO2, N2O, and CH4 emissions, in that order.

Country.java

The Country class represents a country and its greenhouse gas emissions over time. Each Country is represented by its name (String) and a Map from years (Integer) to the greenhouse gas Emission amounts for that year. So, the following…

```
name: "United States"
emissions:
  1970: Emission(
    co2: 288000.0,
    n2o: 684000.0,
    ch4: 4690000.0
  )
  1971: Emission(
    co2: 290000.0,
    n2o: 689000.0,
    ch4: 4560000.0
  )
```

…would indicate that the United States emitted 288,000 kilo-tons of carbon dioxide, 684,000 kilo-tons of nitrous oxide, and 4,690,000 kilo-tons of methane gas in 1970; and 290,000 kilo-tons of carbon-dioxide, 689,000 kilo-tons of nitrous oxide, and 4,560,000 kilo-tons of methane gas in 1971.

Sector.java

Sector.java represents a particular sector of industry and its ***global*** greenhouse gas emissions over time, measured in kilo-tons.[1] Each Sector is represented by a name (String) and a Map from years (Integer) to greenhouse gas emissions measured in kilo-tons during that year (Double). So, the following…

```
name: "Transport",
emissions:
  1970: 2278.8,
  1971: 2358.43
```

…would indicate that the Transport industry emitted 2278.8 kilo-tons of greenhouse gases globally in 1970, and the equivalent of 2358.43 kilo-tons of CO2 globally in 1971 globally.

Note that as your implementation will be tested with PartOneTestCases, you may want to read exactly what the expected "get" method names are!

Overloaded Methods

Define a **Util class** with two static methods to **getYearWithHighestEmissions**. Each such method takes, as its single parameter, one of the classes defined in the previous step (i.e., there is one **getYearWithHighestEmissions** method that takes a **Country**, and one **getYearWithHighestEmissions** that takes a **Sector**) and returns the year with the highest greenhouse gas emissions as an int.

To be more specific, do the following:

- For the method that takes an object of Sector, return the year with the highest emissions in its Map of years to emissions.
- For the method that takes an object of Country, return the year with the highest *total* emissions in its Map of years to CO2, N2O, and CH4 emissions.

A method (or methods) like getYearWithHighestEmissions is considered ***overloaded*** since there is a separate definition for different parameter types. This is also referred to as *ad-hoc polymorphism* because the (theoretically) single method is effectively defined to work with a small set of parameter types, not for all parameter types—that is, the method takes “many forms”, but only very specific forms are supported. You will likely hear "overloaded" much more often, but “ad-hoc polymorphism” sounds so much cooler (though, admittedly, it is a bit more intimidating).

Note that to call a static method like these, you can invoke:

```
Util.getYearWithHighestEmissions(new Country("United States", ...))
```

Deeper understanding: How does the Java implementation know which version of getYearWithHighestEmissions to use when the method is invoked (not-so-great hint at this point, the compiler determines the implementation under this scenario)? If the answer is not apparent, then think about the different method invocations and speak with those around you. If the answer seems obvious, then hold on to that belief for the next few weeks to see if it continues to hold.

Tests

Yes, of course you will want to test all of these operations. Add your tests to the provided PartOneTestCases.java file.

Helper: here is an example of testing the highest emissions method for the Transport sector.

```java
@Test
public void testYearWithHighestEmissions() {
    // Create the testable Sector object
    Map<Integer, Double> emissions = new HashMap<>();
    emissions.put(1970, 2278.8);
    emissions.put(1971, 2356.43);
    emissions.put(1972, 2243.3);
    Sector sector = new Sector("Transport", emissions);
    // Check that the method works as expected
    assertEquals(1971, Util.getYearWithHighestEmissions(sector));
}
```

Part 2: Methods

Copy your files from part1 into part2 (you will not need the test cases file from part1).

The definition of yearWithHighestEmissions in the first part of this lab does not follow an object-oriented style. This part of the lab asks that you make a few modifications to improve the code (further such improvements can come with later material). *Don't worry this part will be easier!*

From Util.java, move each getYearWithHighestEmissions method into the appropriate class (as a non-static method, i.e., instance method) corresponding to yearWithHigestEmissions's parameter. The goal is that each object "knows how" to compute its own year with the highest greenhouse gas emissions. As such, yearWithHighestEmissions will no longer need to take a parameter (it acts on this which refers to…well, there is no universally accepted term for the target of this because computer scientists are not very good at naming things or at agreeing on the meaning of names/terms. You might hear "calling object", "current object", "context object", "target", "callee", "referent", "object on which the method was invoked" (that last one is real, and accurate, but is a sign of giving up on naming)).

You can remove Util.java once this is done.

Take a moment…do you like one style over the other?

Tests

Add tests to the provided PartTwoTestCases.java file. This is where you will also be able to see the changes of having the methods defined within each class (instance methods) versus the static methods with which this lab started. **Again, there should be an added test for each of the types of getYearWithHighestEmissions.**

Again, reflect—do you like one style over the other?

Part 3: Answering some questions

Take a look at the Main.java file in the part3 directory. **Don't modify any of the existing methods in this file.** In the main method, notice that we have created two lists, countries and sectors. Both contain real data about greenhouse gas emissions from 1970 to 2012, sourced from the Emissions Database for Global Atmospheric Research. In this part, you will write methods to help make sense of this data by answering some questions. Copy the CVS files to your project.

Note: You can copy the Country, Sector, and Emission classes to the part3 or import these classes in Main.java from part2. You need to make sure you do not break the part2 test cases after adding the methods.

Write methods to do the following:

- Given a list of countries (List<Country>), identify the country with the highest CH4 emissions in a specified year. To do this, write a static method in the Country class with the following signature:

```
public static Country countryWithHighestCH4InYear(List<Country> countries, int year)
```

- Given a list of countries (List<Country>), identify the country with the highest change in *total greenhouse gas emissions* between two specified years. Write a static method in the Country class with the signature below. In addition to returning the country, print its name and the change in emissions you identified.

```
public static Country countryWithHighestChangeInEmissions(List<Country> countries, int
startYear, int endYear)
```

- Given a list of sectors (List<Sector>), identify the sector which has seen the highest average greenhouse gas emissions between two specified years. Write a static method in the Sector class with the following signature. Print the sector name and the average value you identified.

```
public static Sector sectorWithBiggestChangeInEmissions(List<Sector> sectors, int
startYear, int endYear)
```

Hint: Note that startYear and endYear provide you with a range of years. What do you use to compute something across a range of numbers?

Back in Main.java, use these methods to answer the following questions using the data in the countries and sectors lists. * Which country had the highest methane gas emissions in 2000? *

Which country had the highest increase in greenhouse gas emissions between 1988 (the year the Intergovernmental Panel on Climate Change was formed) and 2012? * Which sector had the highest change in greenhouse gas emissions between 1988 and 2012? Also print the highest value for each of the questions.

Submission and demo

Be sure to submit your code by 10:00pm on the due date. Your instructor will run additional tests on your code, which will account for the bulk of the available points for this lab.

Demonstrate **all** parts of your working program to your instructor at once. Be prepared to show and answer questions about your source code. You will need to demo your working code IN LAB on the day it is due. (Partial credit available at the instructor's discretion). You may not demo after your assigned lab period, unless arranged with the instructor beforehand.

Rubric

Part1 (each class 5 Points)	15 Points
Util	5
Test Cases	8
Part2	
Updated Country	5
Updated Sector	5
Test Cases	7
Part 3	
Adding methods	15
Part1 Extra Test Cases	15
Part2 Extra Test Cases	25

---

1. Actually, emissions of gases other than CO2 are measured in equivalent kilo-tons of CO2. Amounts of other gases are converted to the equivalent amount of CO2 with the same global warming potential.↩