The CSC461 Scala, Composite, RDP, and Chain of Responsibility Patterns

DUE: as shown on D2L

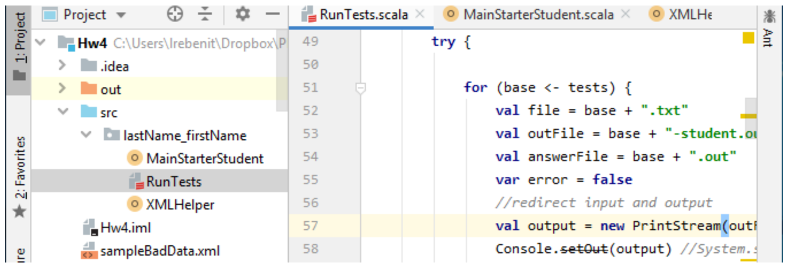
The purpose of this assignment is to give you practice with Scala, XML and the composite/RDP and chain of responsibility patterns.

***There is a checklist and* this will be tested only on Scala 3.3.# with an IntelliJ project (not SBT).**

Overview

You will be coding car insurance estimation software. The first part of the assignment will be to set up your class structure with basic add/remove and display tasks. The second part is to then load and save the information in XML files. See section "XML Format" for format of the XML file and the information that will be stored (I suggest reading this part first). The third part is to add/remove data. This third part will be performing a few pieces of functionality.

I will be posting a **RunTests.scala** file and a **MainStarter.scala** file for built-in tests. The purpose of **MainStarter** is to give a set entry point. This must remain as the entry point **in the root of the lastName_firstName** folder. While I will be requiring the PrettyPrinter class to output your XML files, I expect there may be differences. Therefore, I will check your XML work by reformatting (Ctrl+Alt+L) and then doing a diff check with IntelliJ's built-in diff checker if there is a mismatch noted. Below is the structure you should have when starting:



```
Hw4 C:\Users\lrebenit\Dropbox\P
  .idea
  out
  src
    lastName_firstName
      MainStarterStudent
      RunTests
      XMLHelper
  Hw4.iml
  sampleBadData.xml
```

```
try {

    for (base <- tests) {
        val file = base + ".txt"
        val outFile = base + "-student.ou
        val answerFile = base + ".out"
        var error = false
        //redirect input and output
        val output = new PrintStream(outF
        Console.~~setOut~~(output) //System.s
```

Tip: Review the collection functions. Your code will likely be 30% less in size if you use them.

Required Functionality

Initially, the dataset will be empty. The menu (given) must be in the following format:

```
1) Add Data
2) Display Data
3) Remove Zip
4) Load XML
5) Write XML
6) Find a Cars of Make in Zip
7) Find a Service
8) Total Value Insured
9) Insurance For
0) Quit

Choice:>
```

You may assume all proper console input this time, but the XML files may be faulty.

Add Data (Menu Option 1)

This should add a new zip code. Ask for the zip code in the following format:

```
What Zip Code:>
```

If the zip code is found, tell the user it already exists and return to the menu. If it is NOT found, add the zip code to the system, and ask for its children's element(s). Some children elements may be nested. Loop until the user states there are no more elements, and then return to the menu. To decrease the project size, assume that a zip code is completed when returning to the main menu. It will not be edited afterwards. These are the available elements:

1. Zip code: the zip code of an area. This has 0+ owners and car shops.
2. Owner: a person. Each person has a name, 0+ vehicles and 0+ claims.
3. Owned vehicle: the insured vehicle with its details.
4. Claims: an insurance claim with a few details.
5. Car shops: a car repair shop. Each shop has a name and 0+ services.
6. Services: the car service options. Each service option has a code, a description and a price.

See the XML Format section for more details.

To ask for what type of zip code element, prompt the user with

```
What Element (Owner, Car Shop ):>
```

To ask for what type of owner element, prompt the user with

```
What Element (Vehicle, Claim):>
```

There is a new line before "what element". When completing an insurance element, output `Added <type>`.

You must be able to select the type with both the full name and the first letter, case-insensitive. For example, to add a new owner with the assumption that it is currently not in the insurance database,

```
Choice:> 1
What Zip Code:> 57701

What Element (Owner, Car Shop ):> Owner
Name:> Harold

What Element (Vehicle, Claim):> vehicle
Make:> Ford
Model:> Taurus
Year:> 2015
Value:> 15000
Added Vehicle
Add another Owner element (y/n):> y

What Element (Vehicle, Claim):> c
Date:> 12-12-2020
```

```
Added Claim
Add another Owner element (y/n):> n
Added Owner
Add another Zip Code element (y/n):> y

What Element (Owner, Car Shop ):> c
Name:> Easy Fix
Code:> 1
Description:> Oil Change
Add another element (y/n):> y
Code:> 2
Description:> Tire Rotation
Add another element (y/n):> n
Added Car Shop
Add another Zip Code element (y/n):> n
```

What is printed after the above is shown in section "Display Data (Menu Option 2)".

While I am not testing for it, my solution code had "Element format not found" output in the case of an unknown element for debugging reasons. You may wish to do the same.

Details

- Names and y/n must be case **in**sensitive.
- New examples are appended to the list.
- This must be done with the RDP technique. Tag the starting RDP the **one time** (right above the function *call*) that it is used for the RDP in your code with GRADING: ADD. This is likely in your menu.

Reminder, redirected IO strips the new line from the user's "enters." This limits the length of the line that has to be matched. RunTests ignores *new* line whitespace as much as possible.

Display Data (Menu Option 2)

Display the contents of the current data, *prefaced by a new line*. For each nested level, preface another 2 spaces. For example, after the command in the "Add Data" section completes, output

```
Zip Code: 57701
======================================================
  *****************************************************
  Harold
  Vehicle(s)
    Vehicle: Make: Ford      Model: Taurus    Year: 2015      Value: $15000
  Claim(s)
    Claim: 12-12-2020
  *****************************************************
  .....................................................
  Car Shop: Easy Fix
    (1) Oil Change
    (2) Tire Rotation
  .....................................................
```

Details:

- The vehicle’s individual data components have a string of width 10.
- Car services have their code inside parenthesis.
- After each zip code there is a “==” line (copy and paste this)
- Put “`*************************************************`” after the owner’s name and again at the end of the entry. This is indented to the same level.
- Put “`.................................................`” after the car shop’s name and again at the end of the entry. This is indented to the same level.
- This MUST be called in your menu similarly to `println(data.getInfo(…)).` This may NOT print inside the classes (which is actually code smell since it decreases testability). This must return a string. This may NOT have the number of indentation spaces hardcoded. This is a major violation of dependency inversion. Some help on this:
    - Use the collection map() function with `mkString(…).`
    - You can multiply strings in Scala just like you can in Python. Therefore `"   " * 2` will output `"      ."` You just need to know the level.
- This must be done with the RDP technique. Tag the starting RDP function, **one time**, for string creation with GRADING: PRINT. This is likely in your menu.

If you are wondering why I’m not requiring the toString() function to be overridden, toString() typically works within a single class, and this is altered with the level. It *is possible* to handle it with toString() alone, but that is some tricky code.

Remove a Zip Code (Menu Option 3)

When removing a zip code, ask for the number, and then output that the zip code was removed. The format must be

```
What Zip Code:> <zip code>
Removed <zip code>
```

The output should reflect this change. If the zip code is not found, output

```
Zip Code not found
```

If you are wondering why this does not have the level of detail "adding" does, it is to decrease the size of the project.

Load XML (Menu Option 4)

To load an XML file, first ask for the file name, and then load it. The format must be

```
File name:> <insertFileNameHere>
```

The menu should then reprint after loading. If a second file is loaded, simply append, rather than merge by trying to find matching zip codes.

Files also include car shops. See the XML Format section for more details.

Details

- This must be done with the RDP technique. Tag the start of the read RDP chain, **one time**, with GRADING: READ. This is likely in your menu.
- Store items in the same order as read in.

XML Loading Errors

The XML file is not guaranteed to exist or to be an insurance file. The following are errors that may occur and how the errors are to be handled:

- Calling display before any contents are loaded
    - Should repeat the menu with no warning
- The file cannot be opened
    - Output: `Could not open file: errorMessage`
    - You can get the errorMessage with `e.getMessage`
- The topmost tag is not <InsuranceData>
    - Output: `Invalid XML file. Needs to be an InsuranceData XML file`
- There may be additional tags throughout
    - These are to be ignored/skipped
- There may be additional text throughout
    - These are to be ignored/skipped
- There may be additional attributes throughout
    - These are to be ignored/skipped
- There may be missing attributes throughout
    - These are to be substituted with their default values

Write XML (Menu Option 5)

To write an XML file, first ask for the file name, and then save the current item contents in the named file. The format must be

```
File name:> <insertFileNameHere>
```

To make the output easier to read, you must use the PrettyPrinter class with 80 columns and 2 space indentation. The code should be very close to this:

```
val prettyPrinter = new scala.xml.PrettyPrinter(80, 2)
val prettyXml = prettyPrinter.format( xmlTree)
val write = new FileWriter( filename )
write.write( prettyXml)
write.close()
```

Details

- This must be done with the RDP technique. Tag the starting RDP write, **once**, in your code with GRADING: WRITE. This is likely in your menu.

XML Format

The information you will be working on includes the following:

- Zip codes have code (int)
- Owners, which have a name, 0+ cars, and 0+ claims
- A vehicle, which has make, model, year, and value (integer)
- A claim, which has a date (no required format)
    - A claim is independent of a vehicle
- A car shop, which has a name and 0+ services
- A service, which has a code and a description with its price

The outer most tag must be InsuranceData. An example of the location of the data will be as follows:

```
<InsuranceData>
  <ZipCode code="57701">
    <Owner name="Harold">
      <Vehicle value="1000" year="2015" make="Taurus" model="Ford" />
      <Claim>
        <date>12-16-2016</date>
      </Claim>
    </Owner>
    <CarShop name="Easy Fix">
      <CarService code="1">Oil change: $36</CarService>
    </CarShop>
  </ZipCode>
</InsuranceData>
```

Tip: Ctrl+Alt+L will reformat the XML in IntelliJ for you.

Details

- There is NO guarantee of order of nested tags. You MUST process these in the order of the file using the technique shown in class. This is so that converting to streaming data would be readily doable in the "future."
- You MUST output the XML tags in the order they are added to the system.

Reminder: Attributes are case sensitive in XML, but tags are *not* normally case sensitive.

Find Vehicles of Make in Zip (Menu Option 6)

Print out all the vehicles of a given make in a given zip code. Print the vehicle the same way as the display tasks, only without the indentation (you are not printing the whole tree after all). The purpose of this task is to get used to processing down the tree. You should be able to reuse the output function you made in the display task. As an example, suppose there are two Ford vehicles in zip code 57701. An example run would be

```
Zip Code:> 57701
Vehicle:> Ford
Vehicle: Make: Ford      Model: Taurus    Year: 2005      Value: $1500
Vehicle: Make: Ford      Model: F150      Year: 2009      Value: $22500
```

If nothing is found, print nothing.

Details

This must be done with the "recursive descent parsing" technique. Tag the start of the search with GRADING: VEHICLE. This is likely in your menu, but may be in the top of the composite object.

Find a Service (Menu Option 7)

This should find the first zip code that has a given service code, case insensitive. Search with a preorder traversal. Then, output the first zip code that contains the service in the following format:

```
Car Service:> <code>
<code> found in <zip>
```

substituting <code> with the service code, and <zip> with the zip code where the service is available.

You must be able to handle the instance of not locating a feature by outputting

```
<service code> not found
```

substituting <service code> with the not found item.

Details

This must be done with the chain of responsibility technique. Tag the start of the search with GRADING: SERVICE. This is likely in your menu, but may be in the top of the composite object.

Total Value Insured (Menu Option 8)

This should first ask for the zip code to sum the car values. The value is formatted to two decimal places with commas. For example, for a zip code with nothing in it,

```
What Zip Code:> 57700
Value: $0.00
```

For a zip code with $5,000 total dollars,

```
What Zip Code:> 57701
Value: $5,000.00
```

Details

- Use the DecimalFormat that was used earlier in the semester for formatting.
- The *summation* MUST be done in ***parallel.*** You may convert the data to parallel at any point you wish.
- Tag your parallelization, ONCE, with GRADING: PARALLEL.

Insurance For (Menu Option 9)

To get the monthly payment for a person, first ask for their zip code, and then the name to find the owner (case insensitive). The monthly payment is calculated as the car's total value times 0.1% plus the number of cars times 25 plus the number of claims times the value times 0.02%. In other words: carsTotalValue * 0.001 + vehicleCount * 25 + claimCount * carsTotalValue * 0.002. The output format should be

```
What Zip Code:> 57702
What Owner:> bob
Monthly payment: $122.01
```

Details

- Use the DecimalFormat that was used earlier in the semester for formatting.
- The *search* MUST be done in ***parallel.*** You may convert the data to parallel at any point you wish.
- Tag your parallelization, ONCE, with GRADING: INSURANCE.

Additional restrictions

- You MUST put your code into a package named your lastName_firstName (lowercase with no prefixes or suffixes).
- You must follow the class developed OOP diagram. Minor changes are expected. Moderate changes must be approved.
- Variable access rules are still in place.
- While some ***minor*** differences in spacing and newlines are expected in XML (you do not write this library after all!), the case, wording, error messages, and the output from displaying the content must be exact. Points will be docked otherwise.
- You may not use synchronized(…). The reason is that I see this used improperly on a regular basis to the extent that it undoes the parallelization.
- `data.getInfo(…))` must return a string.
- You may not use instanceof or similar.

Grading Tiers

These tiers start with the simplest tasks, and go to the most involved. Please refer to the rubric for the tiers. You must “reasonably” complete the lower tiers before gaining points for the later tiers. If a tier has an additional bolded line, it means that tier has multiple tests that can be passed in any order. By “reasonably,” I can reasonably assume you honestly thought it was complete. For partial credit on a tier, you **must** put a comment in the main file header comment on how to test for it.

For tiers with sub parts (e.g., 2a, 2b, etc.), the sub tests may be completed in any order. ***However***, the tests are run in order and stop on the first failed line. There is a line in RunTests that starts with “tests =” that you may rearrange to change the order. Since the tests are overwritten on our side, if you complete a higher number sub tier and do not complete a lower one, **you must let the grader know** in the checklist in order to have those graded.

Submission Instructions

1. Check the coding conventions before submission (that includes the checklist).
2. All of your Scala files must use a package named your lastName_firstName (lowercase with no prefixes or suffixes).
3. Delete the out folders, then zip your **ENTIRE PROJECT** into ***ONE*** zip folder named your lastName_firstName (lowercase with no prefixes or suffixes). Make sure this matches your package name!

   If you are on Linux, make sure you see "hidden folders" and that you grab the ".idea" folder. That folder is what actually lists the files included in the project!

4. Submit to D2L. The dropbox will be under the associated topic's content page.
5. *Check* that your submission uploaded properly. No receipt, no submission.

You may upload as many times as you please, but only the last submission will be graded and stored.

If you have any trouble with D2L, please email me (with your submission if necessary).

Rubric

Item	Points
0. Got it running	6
1. Add + Display*	36
Prompts correct (-50% per error)	6
Each item added (-33% per error)	15
Display correctly formatted (-33% per error)	15
2. A) Remove + Display*	10
Prompts correct	2
Removes and displays correctly (-50% per error)	8
B) Add + XML Save*	14
Console added items saved correctly (-50% per error)	10
Console added multiples are saved correctly	4
C) XML Load + XML Save*	14
1 XML file loaded and saved correctly (-50% per error)	10
2+ XML files loaded and saved correctly	4
D) XML Load + Display*	12
1 XML file loaded and displayed correctly (-50% per error)	8
2+ XML files loaded and displayed correctly	4
E) XML+ Display with Bad File Handing	10
All errors handled (-20% for each missed)	10
3. Stress Test for Above*	9
Loads in file, adds data, and displays/saves correctly (-50% per error)	3
Appends a file and displays/saves correctly (-50% per error)	3
Removes elements after edits, and displays/saves correctly (-50% per error)	3
4. Find Cars*	9
RDP format at least there *	3
Lists all cars	3
Formatted correctly	3
5. Find Service*	14
CoR format at least there	4
First item found and output formatted correctly	5
Handles “not found case”	5
6. A) Total Insured	7
Correct without claims	2
Correct with claims	2
Parallelized*	3
B) Insurance for	9
Correct without claims	3
Correct with claims	3
Parallelized*	3
Total	150

* These have required tagging in the comments.