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

DUE: as shown on D2L

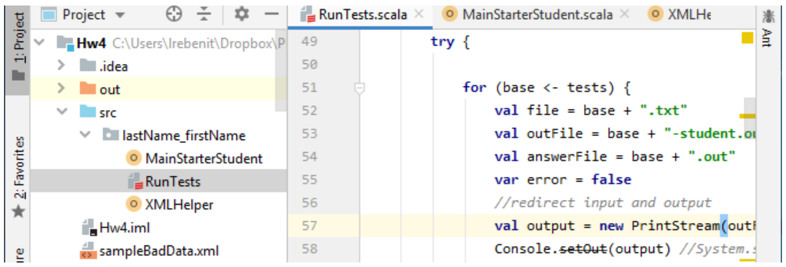
The purpose of this assignment is to give you practice with Scala, XML and the composite/RDP and chain of responsibility patterns.

This will be tested only on **Scala 2.12.14.**

Overview

You will be coding zoological taxonomy software (https://en.wikipedia.org/wiki/Taxonomic_rank). To keep the project size reasonable, there will only be a few levels and then summative data. 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 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** and a **MainStarterStudent.scala** file for built-in tests. The purpose of **MainStarterStudent** 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 do 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:



```
    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.
Aside: You may ignore deprecations. I have 2 deprecated calls in RunTests. The short explanation is that Scala is a young language. If you are curious about the *long* answer, please ask.

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 class
4) Load XML
5) Write XML
6) Find feature
7) Calculate species
0) Quit
Choice:>
```

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

Add Data

When you add data, ask for the current data's information. If it is found, continue to its components. If it is NOT found, add it to the system, and ask if there should be another level. Reprint the menu when the end is reached or when the user states that they are complete. To keep the size reasonable, summary information, 0 or more features, and 0 or more examples will only be added to the Family level in the menu, although the xml may have features at every level. For example, to add a new animal class with the assumption that no data has been loaded,

```
What class:> mammalia
Added class
Continue (y/n):> y

What order:> carnivora
Added order
Continue (y/n):> n
```

Then to add a new family to carnivora with summary information, update the genus and species counts (the current counts are in parenthesis), and add 2 examples, plus feature (this MUST be case **in**sensitive)

```
What class:> mammalia
What order:> Carnivora
What family:> canidae
Added family
Continue (y/n):> y
Add summary (y/n):> Y
Update genus count (0):> 12
Update species count (0):> 27
Add example (y/n):>
What example:> dog
Add example (y/n):>
What example:> wolf
Add example (y/n):> N
Add feature (y/n):> y
What feature:> tail
Add feature (y/n):> n
```

Genus and species count default to 0, and initially there are no examples.

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 whitespace as much as possible.

Display Data

Display the contents of the current data. For each nested level, preface another --. For example, after the command in the “Add Data” section completes, output

```
Class: mammalia
Feature:
--Order: carnivora
--Feature:
----Family: canidae
----Feature: tail
------Genus: 12  Species: 27  Examples: dog, wolf
```

Details:

- There is a double space after the genus and species count.
- RunTests should ignore empty lines.
- 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 hardcode in the number of “--”. 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(), but is some tricky code. **+5% bonus** to those that get it, without violating dependency inversion, and if you mark it with GRADING: BONUS. A later task will output a subtree with indentation. If that task can still use toString(), you probably have not violated dependency inversion.

Remove an Animal class

When you remove an animal call, ask for the class, and then output that the class was removed. The format must be

```
What class:> mammalia
Removed mammalia
```

The output should reflect this change. If the class is not found output

```
Class 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

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 taxonomic levels.

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 a taxonomy file. The following are errors that may occur and how you must handle them:

- 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 <taxonomy>
    - Output: `Invalid XML file. Needs to be a taxonomy 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

To write an XML file, first ask for the file name, and then save the current taxonomy 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, **one time**, in your code with GRADING: WRITE. This is likely in your menu.

Find feature

Find feature should ask for the feature and then find the animal class, order, or family that has that feature, case insensitive. Search with a preorder traversal. Then, output the sub tree, starting at the found item without earlier levels of indentation. Using the above example, searching for "tail," the format must be

```
Family: canidae
--Feature: tail
----Genus: 12  Species: 27  Examples: dog, wolf
```

Suppose that class carnivora has a feature "meat eater." If searching for that, you would have this:

```
Order: carnivora
Feature: Meat eater
--Family: canidae
--Feature: tail
----Genus: 12  Species: 27  Examples: dog, wolf
```

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

<feature> not found

substituting <feature> with the not found item.

Details

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

Sum Specie Counts

This should first ask for the animal class. Calculate the total number of species in that class using the information in the summaries. For example, assume we already have Canidae from above and then add Felidae that has 36 species. The result would be

```
What class: mammalia
Count: 63
```

Details

- This MUST be done in parallel. You may convert the data to parallel at any point you wish.
- Tag your parallelization, ONCE, with GRADING: PARALLEL.

XML Format

The information you will be working on includes the following:

- Animal classes have names (string), and features
- Orders have names (string), and features
- Families have names (string), a summary, and features
- Feature holds the feature as text
- Summaries hold the genus count and species as attributes, and the examples as text separated by commas and a space

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

```xml
<taxonomy>
  <class name="mammalia">
     <feature>Fur</feature>
     <order name="Carnivora">
        <feature>carnivore</feature>
        <family name="Canidae">
          <feature>Fangs</feature>
          <feature>Tail</feature>
          <summary species="37" genus="12">Dog, Wolf, Coyote</summary>
        </family>
        <family name="felidae">
          <feature>Fangs</feature>
          <feature>retractable claws</feature>
          <feature>Tail</feature>
          <summary species="36" genus="18">Lion, domestic cat</summary>
        </family>
    </order>
  </class>
</taxonomy>
```

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

If you are curious were I pulled most of my information: https://animaldiversity.org/ and https://www.itis.gov/

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 with the stay item codes before the food item.

Aside: attributes are case sensitive in XML, but tags are not necessarily case sensitive. If you were wondering about <summary> being separated from family, this shows how good xml is for future expansions. When coding this, I was, at first, going to go all the way to species. Having it in its own tag allowed for both with and without more information!

Additional restrictions

- You MUST put your code into a package named your lastName_firstName (lowercase with no prefixes or suffixes).
- You may not use xml string literals.
- 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.
- AddInfo(), loadXML, and saveXML, must be done with the RDP technique, and marked with tags.
- The species count must use the *"chain of responsibility" technique*, and be marked with a tag. Specifically, it must stop when the item is found (do not continue looking for more items).
- `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 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.

Submission instructions

1. Check the coding conventions before submission.
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	5
1. Add + Display*	34
a. Prompts correct (-1 pt each missed)	4
b. Adds a one of each item and displays (-2 pt each missed)	10
c. Adds multiples (-2 pt each missed)	10
d. Above displays correctly formatted (-2 pt each missed)	10
2. A) Remove + Display*	10
a. Prompts correct	2
b. Removes and displays correctly (-50% for missing one)	8
B) Add + XML save*	16
c. Console added items saved correctly (-3 for each missed)	12
d. Console added multiples is saved correctly	4
C) XML load + XML save*	16
e. 1 XML file loaded and saved correctly (-3 for each item missing)	12
f. 2+ XML file loaded and saved correctly	4
D) XML load + Display*	16
g. 1 XML file loaded and displays correctly (-3 for each item missing)	12
h. 2+ XML file loaded and displays correctly	4
E) XML+ Display with bad file handing	10
i. Calling display on empty, and after “messy” load works	2
j. Handles files not found correctly	2
k. Handle “not taxonomy” correctly	2
l. Handles extra tag, text, and attributes correctly	2
m. Handles missing attributes correctly	2
3. Stress test for above*	15
a. Loads in file, adds data, and displays/saves correctly (50% each)	4
b. Appends a file and displays/saves correctly (50% each)	4
c. Removes animal after edits, and displays/saves correctly (50% each)	4
4. Find food*	16
a. CoR format at least there	4
b. Prompts correct	2
c. First item found and output formatted correctly	5
d. Handles “not found case”	5
5. Total species count*	12
a. Prompts correct	2
b. Calculated correctly	5
c. Parallelized*	5
Total	150

* These have required tagging in the comments.