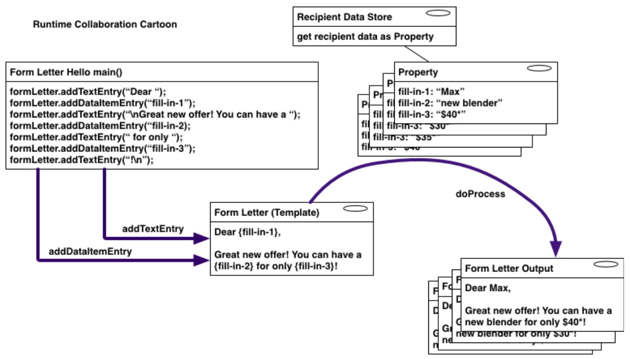
ICT-4361 Homework 5a

Purpose

This exercise will familiarize you with using polymorphic containers and text processing in Java.

We will use the idea of "form letter processing" to convey these ideas.

Form letters are a way of combining a form letter template (text and placeholders for substitution) with a data collection (say, names and addresses) and some computed variables (such as the date) to create a useful result (say, a personalized business letter).



```
formLetter.addTextEntry("Dear ");
formLetter.addDataItemEntry("fill-in-1");
formLetter.addTextEntry("\nGreat new offer! You can have a ");
formLetter.addDataItemEntry("fill-in-2);
formLetter.addTextEntry(" for only ");
formLetter.addDataItemEntry("fill-in-3");
formLetter.addTextEntry("!\n");
```

Visualization of Form Letter Formation: Template applies property files to produce form letters

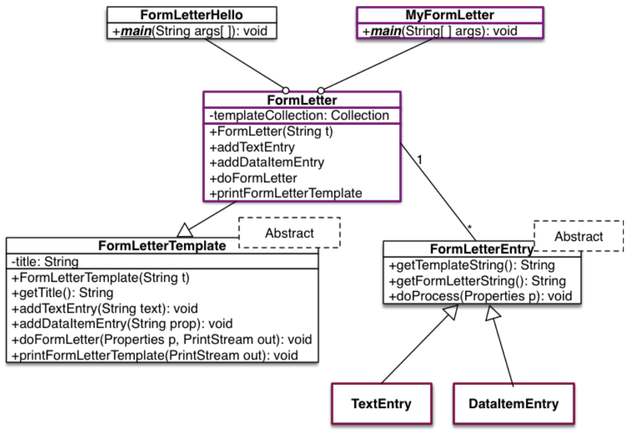
As you can see from the class diagram, a `FormLetter` is a concrete class derived from the abstract class `FormLetterTemplate`. Each instance of `FormLetter` can be used to create one new form letter per `Properties` object.

The `FormLetterTemplate` class has four public methods (other than constructors). Two build the form letter, and two are used for producing output.

- `addDataItemEntry` is used to add one `DataItemEntry` to the form letter
- `addTextEntry` is used to add one `TextEntry` to the form letter
- `printFormLetterTemplate` outputs the form letter template itself to the provided output stream. This method will show the placeholders.
- `doFormLetter` creates and outputs a form letter, using the `Properties` object to provide values for all the placeholders.

In this exercise we will complete the classes for the form letter framework, and process a form letter to produce the appropriate results.

In the followup exercise for next week, the data collections and output will use files rather than internal storage.


FormLetterHello
+main(String args[ ]): void
MyFormLetter
+main(String[ ] args): void
FormLetter
-templateCollection: Collection
+FormLetter(String t)
+addTextEntry
+addDataItemEntry
+doFormLetter
+printFormLetterTemplate
1
*
Abstract
FormLetterTemplate
-title: String
+FormLetterTemplate(String t)
+getTitle(): String
+addTextEntry(String text): void
+addDataItemEntry(String prop): void
+doFormLetter(Properties p, PrintStream out): void
+printFormLetterTemplate(PrintStream out): void
Abstract
FormLetterEntry
+getTemplateString(): String
+getFormLetterString(): String
+doProcess(Properties p): void
TextEntry
DataItemEntry

HW5 Form Letter Class Diagram

What to Hand In

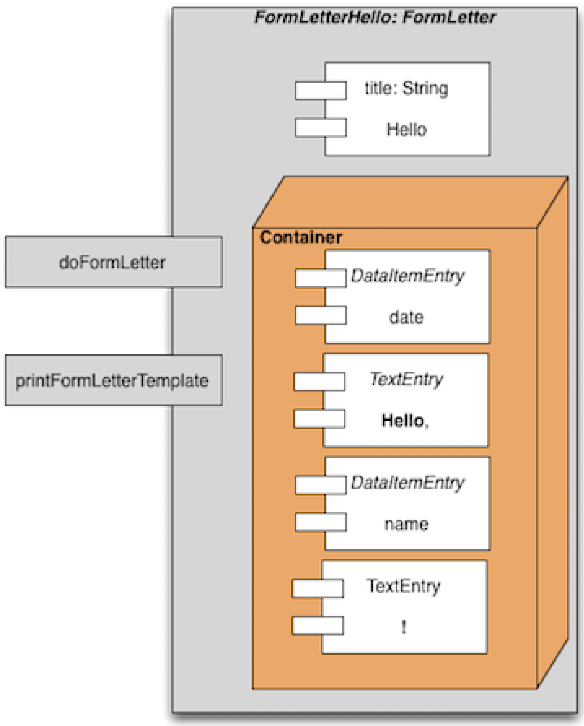
Please hand in a listing for each program requested, formatted in an easy-to-read style.
Ensure your name, and the name of the file is available in a comment at the top of the file.
You do not need to submit files from the homework starter files that are unchanged.
Also, ensure that you have a sample of the output from the program.
If your program fails to compile, hand in your error listing as your output.
For electronic submission, "zip" your submission together into a single file, to ensure nothing is missing;
For each question asked, provide one or two sentences

summarizing your answer. Please be both complete and succinct.

Problems

I. Create and Test a Form Letter
   1. Start with the homework starter files. These will give you a good framework for creating your `FormLetterTemplate`.
   2. Examine the `FormLetterEntry` abstract class, and create the two derived classes `TextEntry` and `DataItemEntry`. Be sure to begin to implement (or let your IDE do it) all the abstract methods in each derived class.
   3. Add fields and methods to the `FormLetter` class. These should include a container for `FormLetterEntry`s, appropriate constructors for `FormLetter`, and the methods outlined in the class starter.
   4. Compile and run the `FormLetterHello` program. When you have implemented the above, it should compile and run.
   5. Capture the output of your test run for your submission.
   6. Create a new class called `MyFormLetter`, which represents another example of a form letter. It must contain at least one `TextEntry` and at least one `DataItemEntry`. This letter should be different than `FormLetterHello`, and use different `Property` entries. Compile it, run it, and same the output of your test run for your submission.
   7. Optional: Run the JUnit tests and make them all pass (this might take minor changes to your classes).
      You will need to ensure the JUnit libraries are on your classpath.
      If you submit this optional part, be sure to capture the output showing the JUnit tests passed and include with your submission.

II. Question: Describe why these files will not compile as they are. Be specific, and display some confirmation for your description. The answer may be stored in a text file, or incorporated as comments in your code.



HW5 Form Letter Composition

Notes

- You can put the sample classes into your NetBeans environment by putting all the files except `FormLetterTest.java` into the source directory, and this remaining file into the test directory. Similar instructions for Eclipse. Or, you can also simply use ant with the provided `build.xml` file.
- While you could create the `TextEntry` and `DataItemEntry` at the bottom of the `FormLetterEntry.java`, please instead create them as separate, public classes.
- Note that while doing input and output in the `DataItemEntry`, `TextEntry`, and `FormLetter` code, you should always use the `out` instance passed in as a parameter, rather than coding `System.out` into these methods. This will ensure that all your output happens properly, and jUnit tests will succeed.
  In a production scenario, output from `doFormLetter` would go to a mass mailer, or printer, or such device.
- Note that the `FormLetter` class must store an ordered list of `FormLetterEntry`s. This is best handled by using one of the Collections classes in the `java.util` package. Various methods in your `FormLetter`

class will iterate through the collection. For example, a `List` of `FormLetterEntry` would be a reasonable collection, instantiated by, say, a `LinkedList` or `ArrayList` of `FormLetterEntry`. Because your ordered list stores `FormLetterEntry` instances, it will be able to store `TextEntry` objects and `DataItemEntry` objects (through inheritance).

- Note that the `Properties` object can contain things that might change for each letter (e.g., name) as well as computable things (e.g., date).
- Your `addTextEntry` method creates a new `TextEntry` using the provided text and adds it to your list
- Your `addDataItemEntry` method creates a new `DataItemEntry` using the provided name and adds it to your list
- Your `doFormLetter` method should first go through each item on your list, and invoke its `doProcess` method, providing the `Properties` object it needs for substitution. Afterward, it should invoke `formLetterString()` to place the result on the provided output stream.
- `FormLetterHello` is the simplest test program using a `FormLetter` and this set of classes, so try running it first.
- You must also be sure not to add extra spaces when printing the templates or `FormLetter`, and finally when printing out the template, you must print out `DataItemEntry`s as <name>. That is, a less-than, the name of the data item, and then a greater-than.

Evaluation

Criteria	Weight
Answer for the question, including justification	10%
TextEntry derived class and test	20%
DataItemEntry derived class and test	20%
FormLetter class completion	20%
FormLetterHello test program and output	15%
MyFormLetter test program and output	15%
JUnit tests completed and output	+10%