MadLibs: Hours of Linguistic Fun!

Today we're going to implement a Mad Libs system. If you don't know what Mad Libs are, please tell us what rock you've been living under! (We're always looking for real estate investments and peaceful, secluded rocks command high prices in today's insecure world.) After you've informed us, visit madlibs.org to try out some Mad Libs.

The main idea is to create “funny” stories from a story template with key words missing. The missing words are filled in by asking the player for words of a certain type — such as a noun, number, or adjective. We call the missing words *slots*. In the old days, you'd walk uphill both ways in the snow to your friend's house to play with her Mad Libs book. Since we're at MIT, we'll dispense with the book and design a computer system instead. A computerized MadLibs system needs to permit creation and maintenance of Mad Libs, as well as permit users to actually do them.

A Java Mad Lib System and Suggested Representation

Getting the files

Today, we're going to have you make a new Java project in eclipse, and then copy the provided .java files into your src directory. To do this, open eclipse (using add eclipse-sdk, then elcipse). In Eclipse, select File->New->Java Project.

A dialogue box will open. Select Java Project and click Next. Name the project "Lab 2". Keep the radio button on "Create project in workspace" (unlike last time where the project remained in your projects folder), but change the next radio button group to select "Create separate source and output folders". The last option will create a src directory for the .java files, and a bin directory for the class files. Click next.

We need to change the Java Settings to import JUnit.jar. This jar file allows you to use JUnit to run tests. You already have JUnit.jar in your lab1 project. Select the Libraries tab, and then select Add External Jar. Use the browser to find ...6.092/projects/lab1_project/junit.jar -- select the jar file and click ok. Then click Finish.

This whole process sets up a .classpath and .project file at the root of your Lab 2 project in your workspace folder. You can check out these text files if you are curious in the Labs section.

Now that you have a project, simply copy the .java files from /mit/iapjava into the src folder of the project you created. Type the following commands at the athena prompt:

```
add iapjava
cp -r /mit/iapjava/www/day2/lab2_project/* ~/6.092/workspace/Lab\ 2/src/
```

Note: after you have copied the .java files to the src directory, right click on the src folder in the package explorer of Eclipse and select "Refresh". Only then will you see the Java files in the src folder.

If you have any problems, let a staff member know.

Task 0: Understanding the MadLibs ADT in Java

A reasonably sane specification for a MadLibs abstract data type is available from the labs section as MadLibTemplate.java. For example, a “Hello <noun>!” Mad Lib would be programmed in our system with the following code:

```
 ml.addString("Hello "); ml.addSlot("noun"); ml.addString("!");

// now go and do it!
ml.doLib(ui);
```

Your first task is to understand our ADT by reading the MadLibTemplate.java file, looking at the specifications (in comments), and seeing how the code snippet above works.

Note that parts of the specification are actually captured as an `abstract class` in the code; any object that sub-classes MadLibTemplate must contain implementations for the abstract methods in the class. (Abstract methods are easy to pick out - they are the ones with the keyword `abstract` in their signatures.)

Task 1: Implementing the ADT in Java

Write a Java class `MadLib` that extends `MadLibTemplate` and implements all the required methods. In the code we've given you — available from the labs section — we've provided the `MadLibTemplate.java` that you'll be subclassing, a starter `MadLib.java`, and a suite of JUnit test cases (`MadLibTest.java`) to help you out.

Our test code only exercises the high-level `MadLib` code, you can choose to implement the functions in the `MadLib` class any way you wish. However, we suggest that you try to make maximal use of the Java type system and use some of the polymorphism that you just learned about. Two starts are provided below:

```
//Option 1
abstract class MadLibEntry {

  /** @return the value of this entry in a template **/
  abstract public String templateString();
  /** @return the value of this entry in an actual madlib **/
  abstract public String madLibString();

  /**
   * Collect any relevant information from the user for this entry
   **/
  abstract public void doLib(UserInterface ui);
}

class TextEntry extends MadLibEntry {
  //What needs to go here?
}

class Slot extends MadLibEntry {
  //What needs to go here?
}
```

```
//Option 2
interface class MadLibEntry {

  /** @return the value of this entry in a template **/
  abstract public String templateString();
  /** @return the value of this entry in an actual madlib **/
  abstract public String madLibString();

  /**
   * Collect any relevant information from the user for this entry
   **/
  abstract public void doLib(UserInterface ui);
}

class TextEntry implements MadLibEntry {
  //Local state goes here
  //Make sure to implement the methods in the MadLibEntry interface
}

class Slot implements MadLibEntry {
  //Local state goes here
  //Make sure to implement the methods in the MadLibEntry interface
}
```

In both cases, we abstract away the differences between a text entry and a slot but require both of them to implement a common interface implementing a single set of methods for getting their contents for printing purposes and for prompting users for input. This allows the code in the `MadLib` class to simply iterate through all of the entries in a particular Lib and know that the `TextEntry`s will do what they need to do and the `Slot`s will do what they need to do.

If you're not sure what this means, review today's lecture notes or ask for help.

We suggest looking at some of the Collections classes in the `java.util` package of the Java API. It includes utility classes like ArrayLists and HashSets that are more useful than arrays for holding items.

Finally, in order to ensure that your MadLib class works with out test harness, please do the following:

- Prompt the user for a word type with the following string template: "`Please enter a` *`type`*`: `". That is to say, the words "Please enter a", a space, the kind of word, a colon, and another space.
- Don't add any extra spaces when you print out templates or the MabLib itself.
- When printing the template, print out slots with in the following format: "`<type>`". That is, a less than sign, the type, and greater than sign.

With these conventions, a transcript of our "Hello!" example from above looks like this:

```
%java MadHello
Please enter a noun:
Cheese
Hello Cheese!
```

with a template that looks like:

```
Hello <noun>!
```

For the curious...

We use the `UserInterface` interface to abstract away how our Mad Libs program deals with the user. For the curious, `StdStreamUI.java` implements a `UserInterface` that uses the terminal window. It was a nice way for us to hide some of the difficulties of getting user input from you, but also made our `MadLib` class code much simpler. We also use some `UserInterface` sub-classes in our tests; here the sub-classes just give back known answers, making our test code repeatable!

Task 2: Making Some Mad Libs

Play with your program; do some MadLibs. We particularly recommend transcriptions of President Bush's recent speeches (some of which read like MadLibs all on their own); Shakespeare and the text from Walter Rudin's ``Principles of Mathematical Analysis" aren't half bad either (both in the original and appropriately mad-libbed).

We've given you MadHello.java and MadHamlet.java as starting points.

Task 3: Extending Your System (Optional)

The Java system admits some natural extensions. If you happened to enjoy the exercise, we recommend experimenting with some or all of the following:

- Interface your system with a dictionary so you can verify whether or not a piece of user input is of the appropriate form.
- Attempt to characterize a simple set of character-based tests you can perform on user input to (approximately) judge whether or not it is the appropriate kind of word.
- Build a system which reads the MadLibs available on the internet or from some other source and imports them into your system. We can point you to Scheme procedures and/or Java classes you may find useful as a part of this effort.
- Develop an Artificial Intelligence. Teach it to judge whether or not particular mad libs are funny (hint: the answer is probably "no"). Run it on your program and report the results.