**ICS 22 / CSE 22 Fall 2012  
Project #2: *You Won't Find Me There***

**Due date and time:** *Monday, October 22, 11:59pm*

*This project is to be done in pairs using the "pair programming" technique*

***Introduction***

Most of us have lived in more than one home in our lives; it's common for people to move from time to time. Moving poses a number of logistical challenges, not the least of which is getting all of your belongings to your new home. But even discounting the complexity of the physical move, there are other issues to be dealt with. One of them is making sure that people and companies that you correspond with — friends, employers, banks, utility companies, and so on — have your new address, so your correspondence can continue even after your move.

Try as we might to give everyone our new address when we move, though, there's always someone we forget — some friend we haven't heard from in years, some company that we do business with only rarely — or someone who doesn't make the appropriate change in their records. Fortunately, the U.S. Postal Service provides a service called *mail forwarding*. Any mail addressed to you at your old address will automatically be sent to your new address instead. They even put a yellow sticker on the envelope to remind you that you should notify the sender of your new address. Very handy!

In this project, we'll explore the technological side of mail forwarding a little bit, by writing a program that determines whether individual pieces of mail should be forwarded and, if so, the address to which they should be forwarded. Along the way, you'll gain experience implementing your own data structure called a *singly-linked list*. You'll also learn how to implement *generic classes*, which are generic in the same way that ArrayList is generic — ArrayLists take a "type parameter" that configures what kinds of objects they store (e.g., ArrayList<Student>).

***Choosing a partner***

Pair programming is again required on this project, so you'll need to choose a partner from among the people enrolled in your lab section, different from the people with whom you've partnered previously. Remember, too, that you are required to partner with someone who is enrolled in your lab section.

If you're having trouble finding a partner, notify your TA, so that you can be assisted in finding one. If you have not found a partner and notified your TA of the pairing by the end of your lab section meeting on Wednesday, October 10, we will assign you a partner, in which case we will notify you via email of your partnership arrangement. Once your TA has selected a partner for you in this fashion, we will not allow you to switch to another one, so the best way to control your destiny is to choose a partner yourself.

*You will not receive credit on this assignment if you work on it alone without the prior consent of the instructor.* (Please note that "prior consent" does not include approaching us the day the project is due, having completed it on your own, and telling us that you haven't been able to find a partner.)

As with the previous assignment, if you or your partner are retaking the course, you are not permitted to reuse code from a previous quarter, as we expect partnerships to do all of their own work; if one partner brings a partial or complete solution to the table, the other partner is deprived of the journey of solving the problem (which is the objective of these projects).

***The program***

Your program will maintain a list of *forwarding entries*, each consisting of a name, an old address, and a new address. The program consists of commands that allow the user to add a new forwarding entry, remove an existing forwarding entry, or report on the correct address to which a piece of mail should be sent, based on the address on the envelope.

The program should support the following commands:

|  |  |  |  |
| --- | --- | --- | --- |
| ***Command*** | ***Format*** | ***Description*** | ***Output*** |
| ADD | ADD *name* *old address* *new address* | Adds a new forwarding entry to the forwarding list, given the name, old address, and new address. Subsequent mailings to the named person at the old address will be forwarded to the new address. If an entry with the same name and old address exists already, adding should fail with an error message. | If a forwarding entry is added, the output should consist of the word **Added** on a line by itself. If not (because an entry for the same name and original address exists), the output should consist of the phrase **Entry already exists**on a line by itself. |
| REMOVE | REMOVE *name* *old address* *new address* | Removes the forwarding entry from the forwarding list with the given name, old address, and new address, if there is one. Subsequent mailings to the named person at the old address will not be forwarded. | If a forwarding entry is removed, the output should consist of the word **Removed** on a line by itself. If not, the output should consist of the phrase **No such entry** on a line by itself. |
| MAIL | MAIL *name* *address* | A piece of mail is ready to be sent; it is addressed to the given name at the given address. This command checks to see if the mail should be forwarded to a different address. | This command should output the phrase **Send to**, followed by the address that this piece of mail should be sent to, on a line by itself. (If the mail is to be forwarded, its forwarding address should be printed; if not, its original address should be printed.) |
| QUIT | QUIT | Quits the program. | The output should consist of the word **Goodbye** on a line by itself. |

Your program should read a sequence of commands from the console (System.in) and write its output to the console (System.out). It should print no prompts or other output, other than the output required in response to each command, as specified above.

**A few minor, but important, notes**

When you first start your program up, the list of forwarding entries should be empty.

The mail forwarding supported by your program should be recursive. (Note that you do not have to write your code recursively; you can use a loop to solve this problem instead.) For example, suppose that the following two forwarding entries are in the program's forwarding list:

* name: Alex Thornton  
  old address: 123 Main St., Irvine, CA 92697  
  new address: 234 Main St., Irvine, CA 92697
* name: Alex Thornton  
  old address: 234 Main St., Irvine, CA 92697  
  new address: 111 Beach Dr., Kihei, HI 96753

If a piece of mail is sent to Alex Thornton at 123 Main St., Irvine, CA 92697, the program should determine that it should be forwarded not to 234 Main St., Irvine, CA 92697, but to 111 Beach Dr., Kihei, HI 96753.

Your program is not required to parse or understand names or addresses; it's fine if they're stored as strings, and it's also fine if your program only considers a piece of mail to match a forwarding entry if the name and old address match exactly.

**An example of the program's execution**

The following is an example of the program's execution, as it should be. Boldfaced, italicized text indicates input, while normal text indicates output.

***ADD***

***Alex Thornton***

***123 Main St., Irvine, CA 92697***

***234 Main St., Irvine, CA 92697***

Added

***MAIL***

***Alex Thornton***

***123 Main St., Irvine, CA 92697***

Send to 234 Main St., Irvine, CA 92697

***MAIL***

***Jane Doe***

***123 Main St., Irvine, CA 92697***

Send to 123 Main St., Irvine, CA 92697

***ADD***

***Alex Thornton***

***234 Main St., Irvine, CA 92697***

***111 Beach Dr., Kihei, HI 96753***

Added

***MAIL***

***Alex Thornton***

***123 Main St., Irvine, CA 92697***

Send to 111 Beach Dr., Kihei, HI 96753

***QUIT***

Goodbye

Notice, again, that there are no prompts or other output, other than the output that is required as a response to each command. This may seem strange, but there's a good reason for it, which is described a little bit later in the write-up.

**Fair assumptions**

It's fair to assume that your program will only receive valid input. We will not test your program with non-existent commands, nor with existing commands in the wrong format. This is not to say, of course, that error handling is unimportant in real programs, but it adds a level of complexity to this program that's more than I'd like you to be faced with. (You're free to implement error checking if you'd like, but it's not something that extra credit will be offered for.) In the event that your program receives input that doesn't follow the specifications above, it's fine for your program to ignore it, print an error message, or even crash; we won't be testing your program in these circumstances.

It's also fair to assume that there will be no "cycles" among the forwarding entries. In other words, you can assume that it will never be the case that two or more forwarding entries will exist like these, where mail is forwarded back to an address it originally came from:

* Alex — 123 Main St., Irvine, CA 92697 → 234 Main St., Irvine, CA 92697
* Alex — 234 Main St., Irvine, CA 92697 → 123 Main St., Irvine, CA 92697

Consider what would happen if we allowed entries like these to exist simultaneously. If someone sends mail to Alex at 123 Main St., Irvine, CA 92697, where should it be forwarded? To 234 Main St.? Back to 123 Main St. again? Then on to 234 Main St. again? Your program need not check for this case. You can simply assume that this case will never come up, and we won't test your program in this case. It's fine for your program runs infinitely or even crashes in this case. (An algorithm for solving this kind of problem, which is called a *cycle* in a data structure called a *graph*, will be covered in ICS 23 / CSE 23.)

***Storing your forwarding list***

One of the central tasks that this program will perform is to store and access a list of forwarding entries. There are two primary ways we store lists of elements in most programming languages:

* An array (or, more commonly in Java, an ArrayList)
* A linked list

Since we explored the use of ArrayLists in the last project, you are required to use a linked list to store your list of forwarding entries in this project. It's fine to use a *singly-linked list with a head reference*, which I'll be describing in more detail during lecture; in this project, using a more complex variant of linked list provides little benefit, though we'll learn about situations later this quarter where a little extra complexity makes a big, positive difference.

***Starting point***

As a means of getting you started, I'm providing some code as a starting point. In particular, note that I've provided the main() method, which you should use without modification, as well as a skeleton for your LinkedList<E> class, which you are required to complete, rather than starting your own from scratch.

The starting point is available as a [Zip archive](http://www.ics.uci.edu/~thornton/ics22/LabManual/FindMe/FindMe.zip).

There is less code provided in this project than the previous one, because we'd like you to begin thinking more carefully about how to design a program. It is important to realize that the go() method in the Forwarder class — which is called from the main() method in ForwardingProgram — is where the user interface begins, but the entirety of your user interface shouldn't be written in this one method. Instead, you should consider ways to split this large problem into smaller ones (e.g., separate methods for each of the user interface's commands).

***Wait... what kind of crazy user interface is this?***

Unlike the program you wrote in your last project, this program has no graphical user interface, or even an attempt at a "user-friendly" console interface. It's a fair question to wonder why there isn't one. Not all programs require the user-friendly interfaces that are important in application software like Microsoft Word or iTunes, for the simple reason that humans aren't the primary users of all software. For example, consider what happens when you used your web browser to load this page:

* You clicked a link or typed the address of this page into your browser. That part required a user interface.
* Your browser connected, via the Internet, to a machine located in the ICS building on campus, asking to converse with a program called a *web server*. A web server's job is to listen for requests for web pages, responding with the requested pages (or an error message if no such page exists). There's no user interface activity during this process, except that the browser may display some kind of icon that provides the user with the feeling that something is happening; otherwise, all of this activity is invisible until the page has been downloaded and can be displayed.
* Your browser created a request in the format expected by the web server. Web servers expect requests to be formatted using a standardized format called an *HTTP request*, which looks something like this:
* GET /~thornton/ics22/LabManual/FindMe HTTP/1.1
* Accept: \*/\*
* Accept-Language: en-us
* User-Agent: Mozilla/4.0
* Host: www.ics.uci.edu
* Connection: Keep-Alive

It's important to point out that, though there's a [complex, detailed standard](http://www.w3.org/Protocols/rfc2616/rfc2616.html) that specifies what HTTP requests are supposed to look like, it's not necessary for the vast majority of human users of the web to know anything about them. The requests are composed by browsers and consumed by web servers, with people uninvolved in the process.

* The web server responds by sending the browser an *HTTP response*, in another standardized format. Again, the format is carefully defined in a standard, the details of which are unimportant to almost everyone who uses the web; the responses are composed by a web server and consumed by a browser, with people again uninvolved.
* The browser, given the information in the response, draws the web page for you to see.

The details of how the web works are not the point of this assignment, but this example serves to suggest that not all software needs a clean, "user-friendly" interface. A web server is intended to simply run quietly for months at a time with no human intervention required. It may write some information into a log once in a while, especially when something goes wrong, but otherwise it does nothing but listen for requests (which are generated and formatted by browsers, in response to user activity) and respond to them appropriately.

Now consider again the requirements of the program you're being asked to write for this project. It waits for requests to be sent in via the console — though they could almost as easily be sent across the Internet, if we preferred — in a predefined format, then responds using another predefined format. Our program, in essence, can be thought of in the same way as a web sever; it's the engine on top of which lots of other interesting software could be built. When a new address change is added to the system, that could be coming from a web form filled out by a user. When an address is sent to the system to see if there is a forwarding address, that address could be typed by a human user in the post office, or even scanned (e.g., using optical character recognition, or OCR) automatically from the envelope with little or no human intervention. When a piece of mail is being processed and our program says to send it anywhere other than the original address, that could cause a yellow label to be printed and automatically placed on the envelope by a machine.

While we won't be building these other interesting parts, suffice it to say that there's a place for software with no meaningful user interface; it can serve as the foundation upon which other software can be built. You can think of your program as that foundation.

***Testing***

The previous project required you to write a test plan, detailing what specific actions you would perform to test your complete program when you were done with it. Testing a program as a whole is an important part of making sure that the program works, but relying solely on whole-program testing leads to at least a couple of problems:

* It can't realistically be done until the program is done, or at least until individual program features are done. Bugs are harder to fix when a program or a feature is complete, because making changes has a greater probability of introducing unintended changes, breaking other features that were working fine.
* It's often difficult to find the source of a problem when you're testing a large program or a complex feature in the whole, because there are so many possible causes of the problem.

We'll be discussing *unit testing* in lecture, a technique which supplements whole-program testing like you did in the previous project, by allowing you to get feedback about whether individual methods and classes are working correctly. A necessary — but not sufficient — condition for a large program to work correctly is that all of its individual pieces are working correctly. Of course, showing that the pieces are correct doesn't necessarily show that the program as a whole is correct, as there may also be problems with the way the pieces interact with one another. But if the pieces themselves don't work, the whole program certainly can't.

For this project, I'm requiring you to use JUnit to write unit tests for your LinkedList<E> class. Your unit tests should test the class as completely as possible, which means you will likely need multiple test methods for each of the methods in your LinkedList<E> class in order to cover all of the interesting cases. Like the test plan in the previous project, there is no predefined number of test cases that must be included; your tester is complete when all of the important cases are included. Pay attention to normal cases, boundary cases, and error cases. Note that the iterator is part of the linked list, so it's necessary also to test the iterator.

Aside from your LinkedList<E> class, you are not required to test other parts of your program, though you're welcome to do so if you wish. (Note that testing the console-mode input and ouptut can be problematic, requiring skills that we don't have under our belts yet, so this is probably best avoided for the time being.)

**Using JUnit from within Eclipse**

JUnit is not officially a part of Java; it's an add-on library that was developed by Kent Beck and Erich Gamma, two well-known members of the object-oriented software development community. Fortunately, it's integrated with Eclipse, meaning that Eclipse knows what JUnit tests are, knows how to execute them for you easily, and can show you the results of your tests within the Eclipse environment. Very handy indeed!

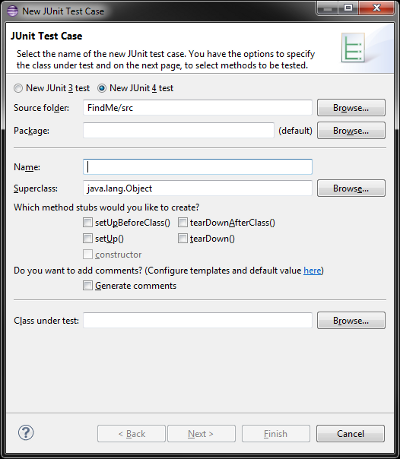
In order to use JUnit from within Eclipse, you'll need to know how to do some things:

* Create a new "JUnit Test Case" class. JUnit Test Case classes are where you write test methods.
* Add JUnit 4 to the "build path" of your project.
* Execute your JUnit tests.

**Creating a new JUnit Test Case class and adding JUnit 4 to your build path**

Normally, when you want to create a new class from within Eclipse, you right-click your project in Package Explorer, select **New** and then **Class** from the menu. (You can also do this from the **File** menu at the top of the window, but it boils down to the same thing: **New** and then **Class**.) To create a new JUnit Test Case class instead, select **New** and then **JUnit Test Case**.

You'll then be confronted with a dialog that looks a lot like the one you need to fill in when creating a new class. However, it will ask you for slightly different information. Let's go over the things you'll need to fill in. Here's a screenshot of the dialog:

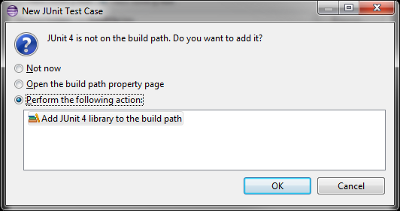


First, near the top of the dialog, it's important that you select **New JUnit 4 test**. (JUnit 4 and JUnit 3 differ in some fairly substantial ways, so it's important that you make the right choice here. On some machines, the other choice is the default, so be sure to check it before moving on.)

Next, fill in the name of your test class. If you're developing a class to test your LinkedList, a good name might be LinkedList.

Leave the other selections empty, as they are above, then click **Finish**. (These options are ways to ask Eclipse to generate some code for you, but it's been my experience that I never want to keep any of the code that's generated; Eclipse simply doesn't know what tests I want to write.)

The first time you use JUnit 4 within a project, you'll be asked one more thing after clicking **Finish**: to set up the "build path" for JUnit 4, which tells Eclipse where to find JUnit 4. Fortunately, JUnit 4 is already part of the Eclipse installation, so all you need to do is select "Perform the following action (Add JUnit 4 library to the build path)," then click **OK**, as shown below:



Note that, once JUnit 4 is in the build path of your project, you won't see this warning again within the project. If you decide to write more than one JUnit test case class in this project — which is not required, but you are welcome to do, if you'd like — you won't see the warning on the second or subsequent classes that you create.

**Writing your JUnit tests**

For more information about writing your JUnit tests, check out the JUnit example from lecture in the [Code Examples](http://www.ics.uci.edu/~thornton/ics22/CodeExamples) section of this web site.

**Executing your JUnit tests**

To execute all of your JUnit tests, right-click on your project in Package Explorer, then select **Run As** and **JUnit Test**. This will execute all of your tests, and show you a report of the results.

**Where can I find more information about JUnit and unit testing?**

That's the spirit! If you want to know more, a great place to start is [www.junit.org](http://www.junit.org/).

***Deliverables***

It is only necessary for one of the two partners to submit the project; the TAs are aware of the partnerships and will figure out which project submissions belong to which pairing. Put the names and student IDs of both partners in a comment at the top of each of your **.java** files, then submit all of the .java files, including the ones that we provided to you, to Checkmate. Please do not turn in the **.class** files, or other files generated by Eclipse. Afterward, take a moment to be sure that you submitted all of the files; if you missed one, we won't be able to compile and run your program, which can result in a substantial penalty, since we won't be able to evaluate your program's correctness.

Follow [this link](http://www.ics.uci.edu/~thornton/ics22/LabManual/SubmittingProjects.html) for a discussion of how to submit your project via Checkmate. Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want graded. We won't regrade a project simply because you submitted the wrong version by accident.

***Limitations***

The Java library contains a LinkedList class (java.util.LinkedList), but you are not permitted to use it in this project. It's a nice class to use in a practical context, but since one of the skills I'd like you to gain in this project is learning how to build your own linked list, the prebuilt version in the Java library is strictly off-limits.

* Some of the requirements charged, including the introduction of JUnit, by Alex Thornton, Winter 2012.
* Originally written by Alex Thornton, Winter 2007.