

Learning Outcomes

Upon completing this assignment, students should be able to

- **Implement** the *reusable* Menu class that can easily be put into future projects.
- Maintain a collection (`List`) of a *hidden inner* class type inside a public class.
- Read a *file by line*, constructing `MenuItem` objects from the data and adding them to a `List`.
- **Document** a thorough test plan for a *reusable* object.

Introduction

The previous programming assignment demonstrated how to use multiple Java *packages* where one of them was a precompiled jar file. The provided class implemented `menu.Menu`. This assignment is about your replacing the instructor's menu class with your own.

You will be expected to use your menu class through out the programs in the rest of this course. This means it is worth your investment in making sure the class performs *correctly*.

Method

This project begins with several artifacts from the previous project. Create a new assignment directory and initialize git in it.

Copy over the following files from `pUsingMenu`:

.gitignore All projects will have one of these to filter “noise” files out of the repository.

data/* The menu description file(s); probably only `calculator.menu` to start.

driver/Calculator.java Your working calculator program will require a couple of lines changed to work with your new menu class.

menu/Menu.java This is the interface your menu class will implement.

README Whatever format you wrote. You can probably put all of your documentation at the bottom of this file, just writing about the changes.

You can add and commit these to git as the initial commit for the new project.

Getting Started

Testing

Before writing any code for your menu class, open up `menu.Menu.java` and your `README`.

Document, in the `README`, how you would *test* an object that claimed to implement the given interface. You should, after brainstorming and getting some ideas for testing, order your tests in the document in the order that you would run them so that later tests can count on anything the earlier tests checked.

Also make note of any tests that *can* be automated (conditions can be set up in code, the menu method can be called, and code can check the performance of the menu) and those that *cannot*.

If the method test *can* be automated, sketch (in pseudocode) how you would test it and what different values you'd need to use to be confident that the method meets the specifications in the interface.

For the rest of the methods, sketch how you would, interactively, test the method. Imagine that you had a brand new program, not `Calculator`. What sorts of interactive tests would you have it run to be confident that your menu implementation works.

Add and commit your changed README to git. This is *explicitly* necessary before you write any code. Remember that git time stamps all commits and keeps a log of what files are added with each commit. It does this for efficiency and software engineering reasons; I can use the information to assess how well you follow directions.

Menu205

You will implement `menu.Menu` in the `menu.Menu205` class. Create the appropriate file and try to compile it:

```
javac --class-path . menu/Menu205.java
```

While the class is missing any interface functions, you will get appropriate errors. Iron them out (as in the class *test-driven design* discussion: put in **stub** procedures to get it to compile without worrying about doing the *real* work).

`Menu`, an interface, has no *constructors*. `Menu205` needs a *primary* constructor that takes a `Scanner`, open on a source of menu data (typically a file). The constructor assumes that the input is some multiple of three lines long and reads the input by line,

```
<command>
<description of command>
<help text for the command/description>
```

Trim the leading and trailing white space off each line (check the `String` documentation) before adding it to the menu.

`Menu205` contains an *inner class* to hold the three parts of a menu entry. What a great name! `MenuEntry` should be defined inside `Menu205` with three `String` fields; as with most inner classes, `MenuEntry` should not be returned from or passed in to any public method (it is *encapsulated* in `Menu205`).

`Menu205` needs a `List` of menu entries. You should instantiate the list field before reading anything from the `Scanner` in the primary constructor. After reading three lines, instantiate a `MenuEntry` and add it to the list.

There should be a secondary constructor that takes an `InputStream` and wraps it in a `Scanner` before calling the primary constructor, something like this:

```
public Menu205(InputStream in) {
    this(new Scanner(in));
}
```

(Remember that this, treated as a function, as the *first* line of a constructor, calls another constructor. So, the one taking an `InputStream` does no work except make a `Scanner` for the primary constructor.)

Testing and Scanner

A `Scanner` can be constructed from an `InputStream` as above. But it can also be constructed from a `String`. The `Scanner` will then *scan* the contents of the string.

So, the following code snippet:

```
Scanner strScan = new Scanner("a b c")
while (strScan.hasNext())
    System.out.println(strScan.next());
```

Would print one letter per line and stop after the third line.

It is also possible, using triple quotes, to make strings that contain multiple lines:

```
String menuEntry = ""
print
Print out the contents of the BST
Print, in order, elements of the binary search tree.
"";
```

A Testing Executable

For testing, create a new *main* class, *driver.TestMenu*. In it *main* constructs a new *TestMenu* object to call *run* on. That function runs test functions and exits with code 0 if all tests pass, non-zero if some tests fail.

In *run*, if all the test methods you write pass, do not print anything. If, instead, at least one test fails, print a failure message and end the program with a non-zero return code.

You will want to modify *build.gradle*, letting *mainClass* be set to a different value from the *gradle run* command:

```
application {
    mainClassName = project.getProperties().getOrDefault("mainClass", 'driver.Calculator')
}
```

When you want to run *TestMenu* rather than *Calculator*, use the *-P* switch to define *mainClass*:

```
gradle run -q --console=plain -PmainClass='driver.TestMenu'
```

(**Note** do not use *mainClassName* for the name of the property in *build.gradle*; Groovy/Gradle will lose their mind.)

Design Considerations

Look at *Menu*. *size* is straight-forward: how many entries does this menu have? *containsCommand* traverses the list of entries, comparing the given string and each entry's command for an equal value. *display* prints each item in the list in the form:

```
<command> - <description>
```

while *help* prints each item in the list along with its help text:

```
<command> - <description>
             <help>
```

That leaves *match*:

String match(String prompt, Scanner keyboard) does the following:

```
display all items
prompt user for an item
while (command is not in the menu)
    read keyboard into command
    if (command is "?")
        display help for menu
    else if command is in the menu
        return command
```

```
print an error message
prompt user for an item
```

You should use `bclMenu` from the previous assignment as a *reference implementation* to answer any questions about the user interface.

File Formats

As discussed above, an entry for a menu is three lines:

```
<command>
<description of command>
<help text for the command/description>
```

So a data file is a sequence of menu entries. See the data file `calculator.menu` for an example.

Testing

`TestMenu` should have functions to automate constructing menus with different numbers of entries (you can pass in `String` based `Scanners`). You can check the resulting size and what commands are contained (if you make good use of `containsCommand` in `match`, this should give you confidence that your code works).

You can add manual tests to `TestMenu` to handle any other test cases you needed in your initial list. One thing to consider: can you test `match` on an empty menu?

Better test *automation* coming soon.

Documentation

README

Must document (in an early commit) what you need to test. Feel free to add more to this as you discover things that need testing. Remember to see what you can automate and what is manual.

For manual tests, give the reader what input they are to provide and the expected on-screen results.

Must include instructions on how to **compile** and how to **run** the program as submitted.

Deliverables

Submission medium: git to Gitea at `cs-devel.potsdam.edu`. `pMakingMenu` is the name of the repo. So, the repo, in the `drbcladd` repo would be: `S24-205-drbcladd/pMakingMenu.git`; the naming is important so that I can have a program download your work.