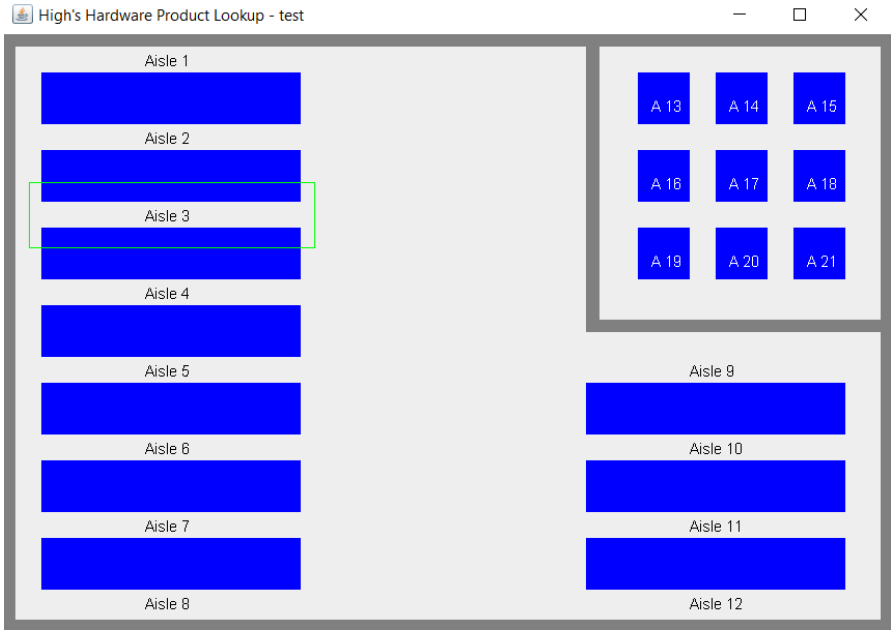
For your final project, you are going to need to use what you've learned this semester to create a store management system for High's Hardware and Gardening that will solve some of their logistical problems, including sawing planks to certain lengths, finding products in the store, and generating schedules for staff shifts. The program works almost entirely via I/O .txt files, which are read in at program start, modified after each command (if necessary), and are used to create output files.

The project is meant to be completed modularly, meaning you'll be working on several self-contained pieces that can be individually tested. In particular, when we as graders (and you as "testers" of your own work) go to evaluate your projects, we'll be using input files that run all the parts of the program and check for the correctness of the changes to the files and contents of the output files.

The goal of this project is for your team of 2-3 students to collaborate to produce an impressive object-oriented program that you'd be proud to show off to demonstrate your skills you've been honing over the semester. Like a real information system implementation, you'll be asked to create solutions that work inside of existing business infrastructure, such as predefined file formats. Unlike the kinds of information systems that would be used to solve this problem in the real-world, there's no incorporation of databases or external/internal application-programming interfaces (APIs).

Here are some demonstrations of what your program may look like in practice:



TODO

Starter files

Download the starter files from the Canvas assignment page.
The starter files are minimal for the project, however they do contain a directory/package structure that you must follow in your solutions. The provided files, StoreMap and FileUtils, are

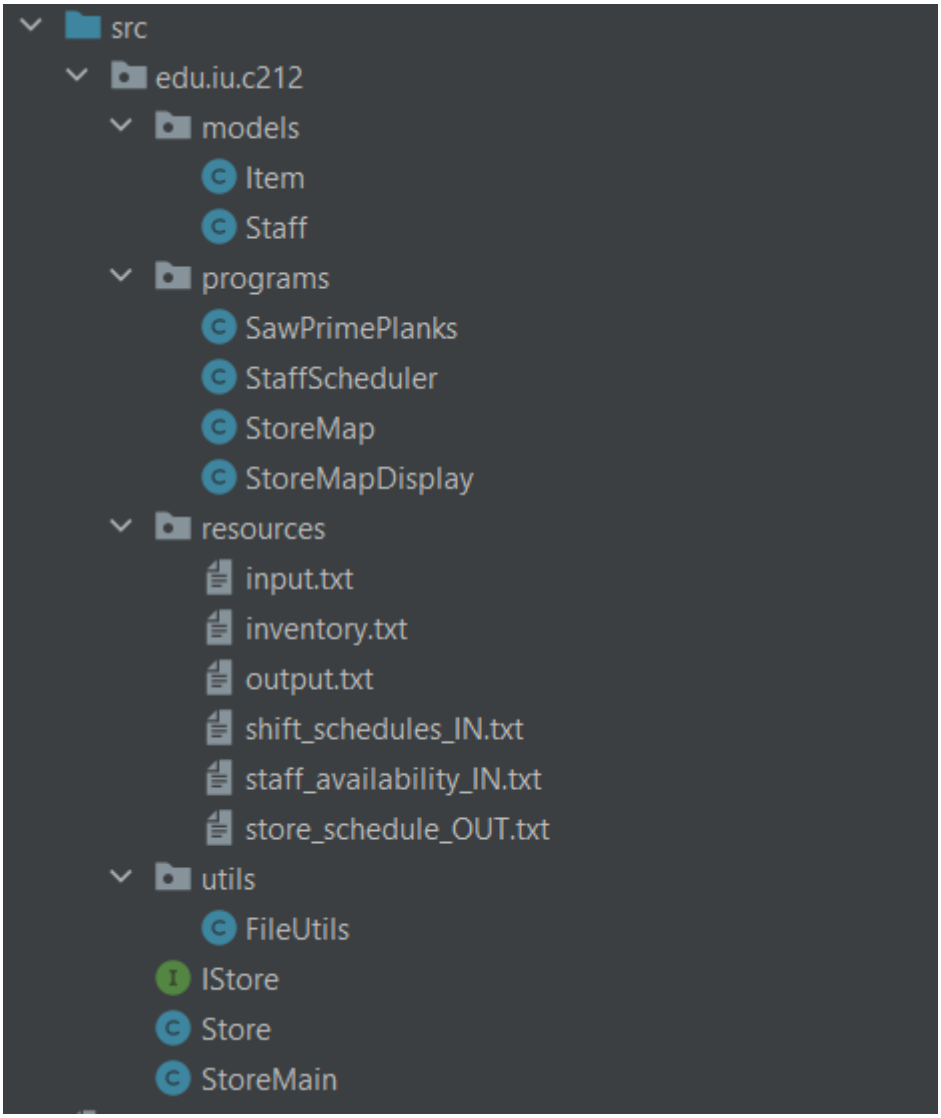
NOT complete and have methods and fields that must be completed on your own. Other classes, interfaces, etc. that you must complete from scratch are listed in this document.

A note on using external libraries

The starter code uses no external libraries like dependency managers one might use in modern Java development. These go beyond the scope of the course and as such aren't focused on in the project. If you are familiar with these or want to be and can think of some practical use for them in aiding your development process, you are encouraged to research them more with your group. Please ask your TAs for permission before incorporating them into your project, though. The project, however, should be fully completable using the standard Java library and material you've seen in class. Your JDK actually includes quite a bit of functionality we haven't seen in the course, too!

File structure

You will end up with a file structure similar to the following:



High's Hardware

High's Hardware and Gardening is a locally owned hardware store with its only location here in Bloomington, Indiana. Its offerings are comparable to competitor hardware stores, including the sale of bulk lumber, tools, furniture, and other items, as well as plants in its gardening section. It has a team of staff with different roles who are scheduled to work different shifts across the week.
As demand for the business grows, the store's owners are now interested in finding ways to use technology to automate its time-consuming tasks and provide a superior customer experience.

The initial program state

- **Important Notes:**
  - Throughout this document, file names like inventory.txt are referenced. Once the program begins, all the information from these files must be loaded into some data structures in the program, such as ArrayLists.

        - When any command is performed that mutates (deletes from, adds to, or modifies) any information, it should happen both in this List (or other data structure) and the relevant file. At each step, the data contained in the files and in the program memory should be equivalent. This means that, for example, when we use the QUANTITY command to find the quantity of an item in inventory, we do not need to search through the entire inventory.txt to find it, but rather can just search the List.
        - However, the text of this document is sometimes ambiguous with respect to which is being referred to. If you are unsure whether a section of the document is referring to the actual inventory.txt or just the list of items in memory, for example, ask!

public class StoreMain (edu.iu.c212.StoreMain)

- main(String[] args)
    - This will instantiate the Store object, which will handle everything about the Store.

public interface IStore (edu.iu.c212.IStore)

This interface defines the store functionality that the Store class will be implementing.

- List<Item> getItemsFromFile()
    - This will call FileUtils.readInventoryFromFile that reads in all the items from inventory.txt, and uses System.exit to exit the program if an exception is thrown.
    - Each item will be its own line in the file, and will be of the form "<itemName> <itemQuantity> <itemCost> <itemAisle>"
        - The inventory.txt is delimited by a ~~space~~ comma
- List<Staff> getStaffFromFile()
    - This will call FileUtils.readStaffFromFile that reads in all the staff from staff_availability_IN.txt, and use System.exit to exit the program if an exception is thrown.
- void saveItemsFromFile()
    - This will call FileUtils.writeInventoryToFile that saves all the items from inventory.txt, and uses System.exit to exit the program if an exception is thrown.
- void saveStaffFromFile()
    - This will call FileUtils.writeStaffToFile that saves all the staff to staff_availability_IN.txt, and use System.exit to exit the program if an exception is thrown.
- void takeAction()

public class Store implements IStore (edu.iu.c212.Store)

This class concretely implements IStore, including all the interface methods above.

- void takeAction()
    - This loads in the inventory and staff information, reads from the input file and takes the correct actions, then finally asks the user to hit Enter to end the program when they're finished.
        - See the rest of the document for possible actions and how they're formatted in the input file
- public Store()
    - It should also call takeAction() and run through the input file

Models

public class Item (edu.iu.c212.models.Item)

These are the items the store has in stock, with members:

- private String name;
- private double price;
    - How much the item costs a customer to buy
- private int quantity;
    - How many of this item are in stock
- private int aisleNum;
    - Which aisle in the store this item can be found in
- public Item(String name, double price, int quantity, int aisleNum)
- public String getName()
- public double getPrice()
- public int getQuantity()
- public int getAisle()

public class Staff (edu.iu.c212.models.Staff)
This represents a staff member, with members:

- private String fullName;
- private int age;
- private String role;
    - Their role as a full word or phrase, e.g. "Manager" instead of "M"
- private String availability;
- public Staff(String name, int age, String role, String availability)
- public String getName()
- public int getAge()
- public String getRole()
- public String getAvailability()

Input files, commands, and other files

Commands

The store management system's input files have a few possible command types:

- Commands with no arguments, such as "EXIT" and "SAW"
- Commands with simply String or numeric arguments, such as "FIND 'AA Batteries'"
    - Commands that use item or person names will have those names wrapped in apostrophes because they can be longer than one word. A reminder is given for each such command below that these quotes should not be in the ~~inventory~~, staff, or output files.

Every command in the input file should have exactly one line in the output file it corresponds to. The complete list of user actions that you are required to recognize is given alphabetically as follows:

- **ADD‘<itemName>’ <itemCost> <itemQuantity> <itemAisle>:** This command will allow you to add an item to the store’s inventory (**inventory.txt**). itemName will be surrounded by single quotes, since it could be multiple words. itemCost will be an integer. After this command is completed, the text “**<itemName> was added to inventory**” should be written to your output file (without the quotation marks). Notice that the item's name is surrounded by apostrophes in the command, but not in the ~~file or~~ output.

- **COST ‘<itemName>’** : This command will find the cost of the specified item and write the name and the cost to the output file with a dollar sign in front of it, like so (without quotation marks): "**<itemName>: $<itemCost>**". Notice that the item's name is surrounded by apostrophes in the command, but not in the ~~file or~~ output.

- **EXIT**: This command will have no arguments, write the text “**Thank you for visiting High’s Hardware and Gardening!**” to the output file, display to the console "**Press enter to continue…**" and then end the program when the user presses enter in the console. **This will always be the final command in the input file.**

- **FIND ‘<itemName>’**: This command will display a store map with a hollow box around the aisle in which that product can be found, as detailed in Sub-programs below. If the item does not exist in the inventory.txt file, write "**ERROR: <itemName> cannot be found**", and otherwise simply write "**Performing store lookup for <itemName>**". Notice that the item name is surrounded by apostrophes in the command, but not in the ~~file or~~ output.

- **FIRE ‘<staffName>’**: This command will remove the specified staff member from the staff_availability_IN.txt file and then write “**<staffName> was fired**” to a new line in the output file. If the staff member does not exist in the staff_availability_IN.txt file, write “**ERROR: <staffName> cannot be found**” instead. Notice that their name is surrounded by apostrophes in the command, but not in the file or output.

- **HIRE '<staffName>' <age> <role> <availability>**: This command will add the specified staff member to the staff_availability_IN.txt file. It will write the staff member onto a new line of the file, in the form “**<staffName> <age> <role> <availability>**” (i.e. separated by commas). The “role” attribute in the input file will either be M, C, or G, meaning Manager, Cashier, or Gardening Expert, respectively. It will then write “**<staffName> has been hired as a <role>**” to the output.txt file (where <role> is the full name of the role, not just the letter). Notice that their name is surrounded by apostrophes in the command, but not in the file or output.

- **PROMOTE '<staffName>' <role>**: This command will be used to promote (or demote) staff members to new roles. After changing the staff member’s entry in the staff_availability_IN.txt file to reflect the change in role, your code should write “**<staffName> was promoted to <role>**” on a new line in the output file. Again, in the output file, <role> should be the full name of the role, not just the single letter representation. Notice that their name is surrounded by apostrophes in the command, but not in the file or output.

- **SAW**: This command will use input from the files (**inventory.txt**) and replace certain planks in inventory with sawed-down planks in (**inventory.txt**), as detailed in Sub-programs below. The output file should display "**Planks sawed.**"

- **SCHEDULE**: This command will use input from the files (**staff_availability_IN.txt**) and (**shift_schedules_IN.txt**) and generate an optimal schedule as an output file called (**store_schedule_OUT.txt**), as detailed in Sub-programs below. The output file should display "**Schedule created.**"

- **SELL '<itemName>' <quantity>**: This command will remove the specified quantity of the specified item from the inventory file and then write "**<quantity> <itemName> was sold**" (i.e. "1 Hammer was sold", or "3 Paint Can was sold"). If the item does not exist, or the item has less than the quantity specified and thus the command cannot be completed, you should instead write "**ERROR: <itemName> could not be sold**". Notice that the item's name is surrounded by apostrophes in the command, but not in the ~~file or~~ output.

- **QUANTITY '<itemName>'**: Like the COST command, this command will find the quantity of the given item and write the integer value to a new line in the output file.

Interpreting commands

Commands can be interpreted within Store's takeAction() method. Use the method from FileUtils you write to get a List of Strings, where each String is one line/command from the input. Then, for each String, determine which command it is attempting to use and its arguments and call the appropriate method in your code.

Reading and writing with files

You must implement the members that allow FileUtils to read and write from the many files. Starter code has been provided for this purpose – each method and field you need included there.

Sub-programs

The store management system has the following sub-programs, which can be run both in the interactive interface and via commands in input files:

- ## Find Item in Store
  - This subprogram, when given an Item, will display a map to the user with a hollow box around where the item is located. We provide the map for you in starter code, you must simply create the JFrame it is displayed in and define boundaries of where each aisle should have its highlight. The box highlight can

simply be done with drawRect(). Notice that we can have as many maps displayed as we try to display in the input file – this means the user will just get a different popup for each display request. Since we don't end the program until the user hits Enter in the console when prompted, they are free to view these maps until they do so.
- In order to do this, you'll have to hard-code boundaries for each aisle in the canvas. The code will somehow lookup the bounds necessary to display the highlight box with appropriate dimensions.

public class StoreMapDisplay (edu.iu.c212.programs.StoreMapDisplay)

- public static void display(Item product)
  - This will create a frame, title it with "High's Hardware Product Lookup: <product name>", set the frame's size to 700 by 500, create an object of the StoreMap class by giving it the aisle number as argument, adding this map to the frame, displaying to the user, and not ending the program when the frame is closed.
  - This class, and its relationship with StoreMap, is analogous to StackedChartDisplay and StackedChart from Lab 9.
  - Input Files
    - None specifically, however the item name from the command must correspond to an existing item in **inventory.txt**. The program will find which Item the command refers to and pass it as an argument to StoreMapDisplay.display().
  - Output Files
    - None.

- Schedule Staff

public class StaffScheduler (edu.iu.c212.programs.StaffScheduler)

- public void scheduleStaff()

- This program reads in staff availability and shifts of the week. It will create schedules such that no one employee is working every day while others are working 0. It will do this by calculating the total hours from shifts of the week input file and evenly distributing the hours for days worked (as best as possible) to the staff availability list.
  - Input Files
    - staff_availability_IN.txt
      - Delimiter is a space character.
      - Each new line indicates a new staff member.
      - **Format is first_name last_name age role days_available.**
        - Days_available is a list of M,T,W,TR,F separated by periods.
        - Day dictations are:
          - M = monday
          - T = tuesday
          - W = wednesday
          - TR = thursday
          - F = friday
          - SAT = saturday
          - SUN = sunday
        - Example file:

Will Boland 22 M M.T.W.TR.F.SAT.SUN
Joshua Filler 25 C M.T.F.SAT.SUN

- So Will Boland would be available all days of the week, while Joshua Filler would only be available Monday, Tuesday, Friday, Saturday, and Sunday.

- shift_schedules_IN.txt
  - Delimiter is a space character.
  - Day dictations are:
    - M = monday
    - T = tuesday
    - W = wednesday
    - TR = thursday
    - F = friday
    - SAT = saturday
    - SUN = sunday
  - **Format is day_of_week open_military_time close_military_time**
  - Example file:

    M 0800 1700
    T 0800 1700
    W 0800 1700
    TR 0800 1700
    F 0800 1700
    SAT 0900 2130
    SUN 0900 2130
  - So they are open Monday to Friday from 8am until 5pm. Open Saturday and Sunday 9am until 9:30pm.

- Output Files
  - store_schedule_OUT.txt
    - The store's employees work schedule.
    - Delimiter is space. Every new line should be a new day in order M-SUN.
    - Sorted alphabetically by last initial.
    - First line should be "Created on DATE at TIME"
    - The format is day_of_week (first_name last_name_initial)
    - Example file:

      Created on 4/12/2022 at 1727
      M (Will B) (Josh F) (Ethan M)
      T (Will B) (Josh F)
      W (Will B) (Ethan M) (Jack R)
      TR (Ethan M) (Jack R)
      F (Will B) (Ethan M) (Jack R)
      SAT (Josh F) (Jack R)
      SUN (Josh F) (Jack R)

Saw Prime Planks

public class SawPrimePlanks (edu.iu.c212.programs.SawPrimePlanks)

- public static List<Integer> getPlankLengths(int longPlankLength)

    - This method header must match exactly what is above
- public static int sawPlank(int plankLength)
    - Previously on this document sawPlanks was listed as being the recursive method.This method may not be necessary in your solution. Either this or the above method must be recursive, but can also contain a loop in addition to recursive calls. You may also change the return type and parameter type of sawPlank to fit your needs – it is simply a helper method for the recursion.

- High's Hardware imports really long planks from its suppliers but does not sell them in the same form as they're received. Instead, it saws them into planks which must have a length of some prime number out of the "long planks" which must have a length of some composite number. You will saw planks recursively into a prime number of planks at each step until no more composite-number length planks remain. We want the resulting planks to be as long as possible, so we start by trying to cut them in half, then in thirds, then in fifths, and so on through the prime numbers. Return a Collection (such as an ArrayList) of prime plank lengths using recursion.
        - For example: a 616-foot plank will be sawed as follows:
            - Into 2 308-foot planks, where 308 is composite
                - Each into 2 154-foot planks, 154 is composite
                    - Each into 2 77-foot planks, 7 is composite
                        - Each into 7 11-foot planks, 11 is prime
                            - And we end up with 2*2*2*7=56 11-foot planks!
        - As another example, a 195-foot plank will be sawed as follows:
            - Into 3 65-foot planks, 65 is composite
                - Each into 5 13-foot planks, 13 is prime
                    - And we end up with 3*5=15 13-foot planks!
        - Your solution should return a Collection (such as ArrayList) of planks, and the recursive calls may resemble, in pseudocode (as one approach):
            - saw(195) → [65, 65, 65] → return list of (saw(65), saw(65), and saw(65))
                - saw(65) → [13, 13, 13, 13, 13] → return list of (saw(13), saw(13), saw(13), saw(13), and saw(13))
                    - saw(13) → [13] → return the element 13
            - Finally return the list of [13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
    - Input Files
        - None, other than **inventory.txt**
        - The price of the long plank is irrelevant here, since we don't intend to sell it until it has been sawed down
        - There could be many long planks in inventory, each with a quantity of more than one. Every long plank should be sawed after this command is run.
        - Example lines from inventory.txt that this subprogram should find, remove from inventory, saw prime planks out of, and add back to inventory:
            - 'Plank-616',379456,1,1
            - 'Plank-195',38025,4,1
    - Output Files
        - Remember to include in **output.txt** the text "**Planks sawed.**"

- The output file in this case is also **inventory.txt**, which should be modified to have all its long planks replaced with prime planks.
- The price of a plank is the square of its length.
- Example lines from inventory.txt that this subprogram would display instead of the above after sawing is finished:
    - 'Plank-11,121,56,1
    - 'Plank-13',169,60,1

Rubric

Rubric (out of 100 points)

- Models (10 points)
    - Item (5 points)
        - (2) across all variables, including appropriate access modifier and type
        - (1) for constructor
        - (2) across remaining methods, including appropriate access modifier, return type, and implementation
    - Staff (5 points)
        - (2) across all variables, including appropriate access modifier and type
        - (1) for constructor
        - (2) across remaining methods, including appropriate access modifier, return type, implementation (e.g. storing "Manager" instead of "M" in role)
- Programs (50 points)
    - Saw Prime Planks (20 points)
    - Staff Scheduler (20 points)
    - StoreMap (5 points)
    - StoreMapDisplay (5 points)
- FileUtils Method Implementations (16 points)
    - (2) for using only relative paths
    - (14) across all methods, 2 points per method, including correct return types and parameters. No penalty for changing method names to match past PDF versions
- Store Classes (24 points)
    - IStore (3 points)
        - (3) across all method headers, including appropriate access modifiers and return types
    - StoreMain (1 point)
    - Store (20 points)
        - (8) across all interface method implementations
        - (4) for constructor
        - (8) for takeAction