DRAFT SKELETON OF FINAL

The evaluation criteria for every question are below, where “somewhat satisfied” on a criterion = C (75%), “entirely satisfied” = A (95%), and between these = B (85%). Refinements to each of these three grades are available to graders.

**Technical Mastery**—response demonstrates mastery of the technical aspects of the question.

**Thoroughness**—explains the relevant aspects, and how deeply.

**Relevance**—avoids irrelevant material.

**Clarity**—is clear.

Regarding the use in your term project of **abstract classes and interfaces**, explain, using specifics from your particular project, whether this helped in your project(s) at the time and how; or, if they did not help, why not.

* **Account abstract class (used)**, perfect example of what to use given the project, multiple accounts that have shared methods like crediting and debiting accounts
* Budget abstract class – used to expand the budgeting section, ie total monthly budget vs rent vs groceries vs gas vs entertainment budgets(mandate setting some up like the main vs the discretionary), each can have different preferences like alerts when approaching or going over
  + Discretionary (entertainment, restaurants
  + Essentials: mandate setting up (Total, rent/mortgage(can be 0), food, utilities)
* Alert: tied with the budget, users will need some form of notifications, such as push notifications, emails, texts, etc. when they are overspending or saving plenty each month. (some negative to say too close to budget this month as well as monthly congratulatory alerts based on savings, increase in net worth/decrease in debt etc).
* Transaction abstract class – used for tracking spending, each bill paid, each paycheck cashed, each credit card purchase, and each stock investment, each affect different accounts, some add to debt, some to cash, some to net worth, some need to be monitored constantly (CC), some need to be checked several times a day (bank), and some daily or weekly (investment), will help to further modularize
  + Purchase
  + Deposit
  + Investment
  + BillPay
* Eventually, will need to pair budgets with transactions, trans will be categorized so we know how spending is being done and keep automatic track of the budgets.
* Note: we will never want to make an instance of these base classes, no generic account, only bank accounts, no generic transaction, needs to be specific like a purchase, and budgets need to be discretionary or nondiscretionary spending

My project was called "FinFree", and it helped users keep track of all of their financial information from banks, credit card companies, and investment firms. In my project, I utilized an abstract class to develop an "Account" class, which was later implemented by "BankAccount", "CreditCardAccount", and "InvestmentAccount". I found that the Account abstract class was a great foundation for the rest of my project.

All of the various types of accounts had similar methods, some of which needed separate implementation. For instance, a user may pay off a balance on a credit card, but they could have multiple credit cards on one account. Therefore, the process for crediting that account is different than say, a deposit to a savings account. While the effect on the account is overall the same -- that being the increase in value -- the credit card account must also specifically target which card's balance is specifically targeted, whereas the value of the bank account would simply increase. This type of polymorphism demanded an abstract class. To go further, if needed in future releases, the program could add functionality where savings accounts track the number of transactions a month to avoid overage fees or investment accounts help prevent users from violating day trading rules.

Additionally, the program never should instantiate an Account object specifically, as anytime the program or a method is dealing with an account, it would be with a specific type of account. Despite the fact that I later delegated some of the responsibilities of the Account subclasses to other classes in an effort to provide more modularity to the project overall (and to avoid having all of the work be completed by a handful of classes), the eventual skeleton of the Account abstract class laid an important foundation for expanding on the rest of the project. This gave me a great "jumping off point" to later figure out how to develop the menu process as well as the user classes, along with a few user defined exceptions.

The Accounts class became the skeleton of the entire project, despite other classes such as Menu, User, and FinFree performing much of the user interaction and data collection, and most of the user queries must go through and access or alter an account or accounts in some capacity. The streamlined way everything was setup because it was an abstract class helped provide a framework for developing future aspects of the project.

In terms of scale and future releases, there are several additional abstract classes that the program will eventually need, such as Transaction(subclasses include Purchase, Deposit, Withdrawal, Investment, BillPay -- the reason for all of these is to aid in categorization down the line for budgeting), Budget(Discretionary vs Nondiscretionary), as well as more general ones like Alert to notify users of issues with their account (push notification vs email or budget alert vs monthly savings notification). None of these should ever be instantiated as the base abstract class but rather implemented by a subclass. While on the surface it seems they should be able to be instantiated on their own – and it is the case that many will contain concrete methods -- these will each have abstract methods that need their children to implement to fully become a concrete class. For instance, the different alert types will be sent out very differently or with very different texts (warning versus a congratulations), and FinFree shall mandate users setup their nondiscretionary budgets have more alerts associated with them. The thinking there is every user ought to know how much they spend on groceries and rent a month, but not everyone goes to bars every week or sees a movie or goes out to eat, etc. (especially given the pandemic!) Finally, transactions will target different types of accounts, sometimes multiple, like BillPay which debits a bank account and credits a credit card account if it is a credit card bill.

Ultimately, I found the use abstract classes extremely useful to my project, and I will assuredly being expanding their usage in future releases.

2.

Regarding the use in your term project **of exception handling (preferably user-defined**), explain, using specifics from your project, how you could have used this more effectively and/or additionally in your project—or, if you could not have used them more effectively and/or additionally, explain why.

FinFree, in its most recent release, contains only one user-defined exception: NoCreditCardException. This is a useful exception that makes logical sense, but could use some expanding. The exception is thrown whenever a user is trying to save a CreditCardAccount object that has no CreditCard objects stored within it. With hindsight, most of FinFree's exception handling was simply notifying user's there was an exception. There are several additional steps that FinFree can and will take to further develop the skills needed for successful usage of exceptions as well as handling.

First, as Prof. Braude suggested in lecture, I want to create a hierarchy for a number of different exceptions that I will implement in the future. Almost all have to do with bad user input. Upon thinking of several exceptions to add, the main one that jumped out to me as being not dependent on user input is an "UncategorizedTransactionException". As mentioned in Q1, future releases will implement a Transaction abstract class that helps users monitor their money movement and general spending habits. The purpose of this exception is when users login, FinFree will query all of their accounts for transactions since their last login in order to update all their information, specifically for budgeting purposes. For instance, large biweekly deposits could be categorized as paychecks, purchases at a Whole Foods or Stop & Shop as groceries, and a withdrawal from a checking account directed towards a credit card company as a bill being paid. However, when the system is unaware or unable to pull sufficient information from the relevant transaction, the user should be notified so that it is neither miscategorized nor left uncategorized. (For instance, a large purchase at a bar should be noted as making a dent in a user's discretionary budget).

Continuing with the idea of the hierarchy of exceptions, all other ideas that I have had relate to bad user input. So, these all would be subclasses of the IOException. Furthermore, FinFree's exception handling should do just that: actually handle the exceptions in a meaningful way. To that end, with a description of each exception, I will explain broadly how the program will handle it. For the sake of readability, I will demonstrate the exception hierarchy with bullet points (the exceptions which would be thrown are in italics, those there for the sake of categorization and meant to be abstract are in bold):

* IOException
  + **AccountCreationException**: these are thrown whenever a user gives bad input relating to the addition or creation of accounts to their profile
    - *DuplicateAccountException*: This is used when users are attempting to add the same account twice -- this is a feature, shockingly, that similar apps to FinFree do not support and users can have several instances of the same account. This is also very important for overhead and scalability. If every user had multiples of every account, it would take up significantly more storage space and slow the program down since many transactions would be accounted for twice, account lists would be twice as long, etc.
    - *NoCreditCardException*: the details are mentioned above. However, rather than simply notifying users this is the case, FinFree will redirect them and confirm if they want to add a new CreditCardAccount object to their profile, and if so, they will go through the process again, otherwise, it will return them to the main menu.
  + *InvalidMenuInputException*: this is thrown whenever users provide input in an unacceptable format. For example, if a user provides a String when asked for an integer for their account number. This is handled by asking the user repeatedly to enter the information in the correct format.
  + **ProfileException**: these are thrown whenever a user provides bad input at the time of their login attempts or setting up/making changes to their user profile
    - *BadExternalLoginException*: As mentioned in my assignment submissions, I would like to eventually add the functionality of Plaid so that users can securely login to their actual bank accounts to provide FinFree with their information. This is thrown whenever they provide an incorrect username and/or password for such an account. To handle this, users will be asked twice more to attempt to login before exiting this process
    - **PasswordException:**these are thrown anytime a user enters incorrect or somehow invalid information related to their password.
      * *OldPasswordException:*this is thrown when a user is changing their password to one of their previous passwords. For the sake of scalability and storage space, FinFree would store the three most recent passwords. I added this as it is a function in many applications' security policy. To handle this, the user would be asked to change their password to something else.
      * *PasswordFormatException:* this is thrown when a user's password fails to meet the strength requirements (minimum number of characters, a number, and a symbol, etc). To handle this, they are asked to enter the password again, but one that meets the necessary requirements.
      * *WrongPasswordException:*this is thrown when a user attempts to login with the incorrect password. To handle this, they are asked to enter the password two more times before the program exits.
    - **UsernameException:** these are thrown anytime a user enters incorrect or somehow invalid information related to their username.
      * *DuplicateUsernameExcception:* this is thrown when a user is attempting to create a new profile with a username that is already in use. To handle this, the user is notified this username is in use, and they are asked to try another. This is, in my opinion, one of the most critical exceptions I failed to implement thus far, and it is crucial for handling scalability as well as any potential errors by mixing up users' information, potentially giving bad actors access to vulnerable people.
        + If email is also eventually attached, a similar exception would be needed to handle duplicate email addresses, too
      * *NoUsernameFoundException:*this is thrown when a user attempts to login with a username that is not stored in the FinFree logins SQL database. To handle this, they are simply notified that no user with that name exists and are asked again.

All of these exceptions would help develop a clearer path than the one that currently exists in my project in terms of how to handle bad user input or bad data generally. Currently, the project has very little true exception handling, and the modularization that all of these could provide would help make the project significantly more readable as well as more efficient and certainly more scalable in the long term.

3.

Regarding the use in your term project of classes or methods that use **generics**, explain, using specifics from your project, how scaling to a truly usable version of your application would affect the use of this feature (or these features) in development and maintenance.

* Current program limits users to one account of each type (that is, one bank account one credit card account, and one investment),, would need to be able to write all of the accounts of a certain type out to the database, which will output a generic type given the generic input
  + Have a system setup with faulty generics (not much use) right now, with array-lists outside of that, the generic should be the list
  + Need to be able to have multiple accounts for a large scale project!
* Can also have a list generic for budgets so all the budgets are together
  + Could even be the same generic?
  + Would store all the budgets of a each type then store the object to write to the database and the values associated with it (ie groceries value is 87$ and the object would have 100$ so when it is processed we know 13$ left)
* Leverage all the properties of the generic class for all the objects that instantiate it, not in a hierarchy but each shares some form, would be able to take all the objects within that generic list and return their values: for bank accounts its cash, investment NW, CC is debt, budgets are budgets… this would very likely be a method instead of a class, too
  + Could have a get, set, and add to the list, in addition to the getValue()

In FinFree, the current generic class is called AccountFileandValue. It's purpose, first and foremost, was to take the responsibility of storing the string version of any account objects as well as handle writing that to a file or to the SQL database. Secondarily, it stored the most up-to-date value of the account. User objects would keep an array list of this generic type for all of the accounts. This class and related methods could certainly be worked -- and need to be, in my opinion -- to cement down the proper functionality of the generics as well as aid in the long term scalability and ease of maintenance for the project. For instance, I did not think about the fact that the String and value variables are essentially being stored twice because of this.

To begin, rather than have individual accounts with their strings and values, I would change this class to be collection that stores generics. I would also have it implement Comparable so that all of the accounts were grouped by type. (B< C< I) Firstly, there is little to distinguish the generic class from an accounts class other than the method to write its data to a file/database. Secondly, this generic collection could be used in a number of ways. It could hold all of a user's accounts, all by type (so three: one for Bank, CC, Investment), as well as the new abstract class types specifically the Budget classes (same idea that could have an overall budget as well as two, one discretionary and one for the essentials like rent/food/utilities). Thirdly, the program, given this setup, only allows users to have one account of each type, so making the generic a collection would enable users to store all of their accounts in one place. Now, all of these accounts data can be streamed to the SQL database.

Returning to the second point, having the functionality to work for both Account and Budget classes that outputs these generic objects would make this truly a generic class with actual use cases. Furthermore, this class would still store the value of all the accounts or the amount of money spent thus far in the budgets, so that aspect would not go away and would apply to both types, too. For instance, we would need to know the balance of all credit card accounts to know how much credit card debt a user has, and we would need to know how much as user has spent in a given category this month to know how much left they have budgeted. For Budget, the actual object will store the limit, whereas this value will store how much has been spent in this category for the month. To avoid issues with taking up too much storage as I mentioned earlier, the actual Account will keep its value until the user logs out, so that all transactions can be cross checked to verify the actual change in a user's account balances. Additionally, instead of outputing the account as a string, I plan to use streams/lambdas and write the objects using binary I/O.

Ultimately, unlike abstract classes and exceptions, I did not find an overabundance of use cases for generics. However, learning more about them has me interested in altering and optimizing FinFree for future releases.

4.

Regarding the use in your term project **of reading and writing objects using binary I/O and streams/lambdas**, explain, using specifics from your project, how scaling to a truly usable version of your application would affect the use of this feature (or these features) in development and maintenance.

* Need further explanation on the read and write and how it is all done
* Should also reread this stuff
* Have a very rudimentary version of DataOutputStream writing to a file in an earlier iteration for storing user information for in between sessions.
  + Need to add serializable information, straight forward as the data is always the same order (a char B/C/I for the account, then firm/bank, account number, balance, then the account type specific stuff which almost exclusively Strings)
* Can store user’s files on their computer locally while they are using the app, and when they log off, the file will delete itself and send that information to the SQL database, this can be done by reading in each line, making lists of each type of account, then streaming that list to update SQL queries using lambdas.
* Actual bytes much more cost effect so efficient at scale

FinFree in an earlier iteration used a very rudimentary version of DataOutputStream for writing the toString output of account objects to a File for storage. However, given the implementation, the manner in which the data was serializable was heavily manual, difficult to manage, and is not in need of a rewrite. This however, is much simpler than previously anticipated. To serialize this more easily, the toString functions need to be arranged such that it outputs as: {char type[B/C/I], String bank, String accountType(CC will always be credit), int accountNumber, double balance, double interestRate/retirementContributions (always 0 for CC), ArrayList<CreditCard> cards (null for bank/invest)}. Note: the cards list has a toString() function of its own, so it shall be read in streams as a String that will need to be parsed back to be serialized since banks and investment accounts can follow the other format identically. Once FinFree is able to successfully implement this type of serialization for the Account classes, it will make writing these objects significantly easier.

As I go on to mention in Q5, one potential future release use case is instead of constantly querying the database when account changes are made, all changes are logged on a locally encrypted text file, which is then sent over to update the database when the user logs out and promptly deleted. To ensure security and prevent bad actors from interfering, this data shall be encrypted, hashed, salted etc and written in binary to the file. Writing in binary also will minimize the size of the file on the user's machine, improve the efficiency of the program and the servers as they will have a lower query load, and further modularize the program and aid in the overall scalability of the project. The program will naturally need to read in these bytes when sending them to the SQL database. Additionally, when necessary, SQL databases allow users to store data as binary bytes, which are considerably more cost effect and efficient at scale.

Additionally, the way all this data would be written would be through a Stream object and a lambda function. FinFree currently has a very primitive stream/lambda setup in the main menu which allows users to print the values of all of their accounts. For writing to these files, FinFree shall create a generic Stream of objects from all the user's accounts, go through the collection of attributes related to the account as described above, then using a DataOutputStream convert them all to bytes, then encrypt all of that, finally ending with writing that data to the locally stored file.

Though I am largely inexperienced with lambdas and streams, I am excited to continue tinkering with them, learning more about this interesting feature of java, and continue developing with them in future releases of this project.

5.

Regarding the use in your term project of **concurrency**, explain, using specifics from your project, how you could have used this used this more effectively and/or additionally in your project—or, if you could not have used them more effectively and/or additionally, explain why.

* Need to clarify the difference between thread and database concurrency
* Implemented SQL database concurrency on my end, not thread concurrency, which is useful but not necessarily what was needed
* When syncing all user transactions, I would want to add a method categorize which would take all of these transactions (potentially hundreds or thousands depending on the last time the user opened the application) and categorize each one, should be done concurrently so it can be done quickly and efficiently at scale
* Could have user class implement runnable and each user login instantiates a new thread
  + Check if inactive for 5 minutes, auto log out

FinFree certainly could have benefited from some additional work on concurrency. The concurrency that was implemented related to having a user query all their accounts for new data every thirty seconds, something that most applications would do when you are browsing. Additionally, FinFree utilized several Thread locks, but these were all for maintaining database concurrency, not thread concurrency. Therefore, I will examine additional ways FinFree could effectively implement Thread concurrency for future iterations.

First and foremost, as this is a program in which users login and are constantly querying the internet and affecting FinFree's databases, each User at login shall become its own Thread. This currently exists as the User class does implement Runnable. However, as I mentioned, the currently implementation of the run() function is rather simple and serves little purpose. Although it could still do that, FinFree should add more here. One additional aspect I would add to this function is the program should start a timer after every interaction the user has with FinFree, and after five minutes of inactivity, for the user's security, they would be automatically logged off. Again, this is mostly simple, so I would like to go futher.

As mentioned in Q1, I would like to eventually incorporate Transaction objects which demonstrate how users are moving their money around and spending it. Everytime a user logs in, FinFree shall automatically query all accounts associated with this user's profile, and pull every transaction since their last login. From there, each of these will create a certain type of Transaction object. For instance, banks will create Deposit and Withdrawal objects, credit cards will create Purchases and BillPay objects, and so on. All of these transactions will be stored in a collection of general Transactions, and Transaction will also implement Runnable. This will be implemented in the actual Transaction abstract class since the implementation should not differ for each type of transaction.

After a user logs in and all of their transactions are instantiated and collected, each Transaction will create a thread for a new list collection. From there, when each of these transactions are started, they will then be processed by FinFree, meaning the program will adjust the relevant accounts' values, add the amount to the relevant budget category, then end the thread. Luckily, since the data within each transaction is not fully dependent on the other data, these all can be performed concurrently, which would drastically perform more efficiently if a user has not logged in in months or if many users log in simultaneously, leaving thousands of transactions to be processed in an instant. This usage of concurrency will also aid in readability, scalability, and general ease of maintenance.

One other potential use case I see for future implementations would reduce overhead for the server side. Instead of updating the database after each alteration to a user profile, FinFree shall create a small encrypted and hashed text file locally on their machine that stores all of this information, and when a user logs out of the application, all of the finalized data is then moved to the database and that file deleted. The file would be so small that it would not affect users machines, but a reduction of millions of queries to a database every minute would assuredly assist in scalability. In terms of Thread concurrency, anytime a change is being added to the file, the ability to write to this file should be locked and synchronized so the file does not become corrupted.

All in all, I am somewhat disappointed that I failed to connect the dots that database concurrency != thread concurrency in my first implementation of concurrency in FinFree, however, I see many opportunities for the implementation of concurrency the more I dwell on and the more this project scales.