# FINFREE: A System for Financial Planning, Monitoring, and Budgeting

e.g., PANTRACK: A System for Tracking Pandemics in Iceland

# Aidan Duffy

# Assignment 1 --1/12/2021 7:08 AM

For this assignment, you will describe and implement release 1 of your term project. You will incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*. You are free to choose a project that interests you but if you prefer, the instructor and your facilitator will be happy to suggest a topic. If you are already an experienced developer, this is an opportunity to build a challenging application (check with your facilitator if it requires significant API’s) or discuss research with the instructor. It is OK to name a project with much more scope than you can accomplish in the course (as in the example above): we will not hold you to completing everything associated with it. What we do expect is that you specify and implement a set of do-able requirements within such scope.

Submit this completed Word document. Replace as indicated. Please observe and retain the gray text. Your materials—in black 12-point Times New Roman—should not exceed 5 pages excluding references, figures, and appendices. Use the Appendix sections for additional material if you need to. These will be read only on an as-needed basis.

We want you to develop in Eclipse preferably or else IntelliJ (talk to your facilitator about exceptions). As you code, use JUnit tests whenever possible but certainly by week 2—package-by-package, class-by-class, and method-by-method, except for trivial methods and those requiring I/O. Use non-Junit classes for testing the latter. Keep the evaluation criteria in mind, listed at the end.

For this assignment, you do not need to read data from a file—you can build all data into the code.

Include a ReadMe file describing where to run the application from, and including necessary execution notes. All JUnit tests will be assumed runnable.

# 1.1 SUMMARY DESCRIPTION *EVALUATION CRITERION (i) APPLIES*

One- or two-paragraph overall description of your proposed term project—half-page (12-point Times New Roman) limit. By the end, term projects will incorporate most of the techniques discussed in the course. To do this, you may need to alter the direction of your project or introduce an additional project in future. You may alter this or even replace it as the semester progresses. You will probably find it useful to use your project acronym.

This project concerns an application for younger, less financially savvy adults that would enable them to develop their own financial freedom through effective budgeting and planning. The application, called FinFree, incorporates information and data from the user’s bank, brokerage, and credit card accounts in order to help give them a brief overview of their current financial position. The earliest iterations will ask the user to manually input this information, then later releases will ask for login information. Eventually, FinFree will incorporate Plaid as a login mechanism. From there, to use FinFree, users can preset budgeting goals, like saving for a big trip or just generally increase the size of their savings account or IRA. FinFree will also ask the user to set predetermined spending limits in a variety of core categories, such as necessities like groceries as well as discretionary purchases like coffee or alcohol, in an effort to help their budgeting. Alerts would be sent out if they overspend in any category.

## 1.2 I/O EXAMPLE FROM *PROJECTED* COMPLETED PROJECT *EVALUATION CRITERION (i) APPLIES*

Provide an example of projected *concrete* output for designated input. You will not be held to fulfilling exactly this—it is just explanatory at this point, to indicate where your project is going. We recognize that project direction and details will change as the term progress. This section refers to the project as a whole, not just to what you will produce this week, so we can gain an idea of what you have in mind overall.

FinFree would start at a menu with a number of options for the user to navigate through to see their current financial posture.

Here are two examples of a simple walkthrough in FinFree.

**FinFree:**

Welcome to FinFree, Aidan! Please select from one of the options below:

1. Check monthly net income
2. Check net income year to date
3. Check monthly spending
4. Check net worth
5. Check bank account balance
6. Check brokerage account value
7. Check credit card account balance

Ex 1:

**User:** 1

**System:**

You have been paid $1700 since the start of the month. You have spent $900 since the start of the month.

Your monthly net income is $800.

Ex 2:

**User:** 7

**System:**

You have 3 credit cards. Please select which card whose balance you are checking.

1. Chase Freedom
2. American Express Gold
3. Bank of America Cash Rewards

**User:** 2

**System:**

You have spent $452.81 on your American Express Gold card so far this month, and you have paid $0.00 off this balance. Your current balance on this card is $452.81.

## 1.3 REQUIREMENTS IMPLEMENTED IN THIS RELEASE *EVALUATION CRITERION (ii) APPLIES*

Supply [functional requirements](https://docs.google.com/document/d/1eU7eINLDxmrf793D4OF2yGT4ry_SW3GQGoVDYzecGHc/edit?usp=sharing) statement that you accomplished for this assignment, i.e., functionality that the application provides for the user. Please state requirement in declarative form, as illustrated in the examples, because here we want to know the functionality intended (*what*, not *how*). For example, the following is *not* a proper functional requirement: *TicTac will have a class for O’s and a class for X’s.* It is common to mistake design elements like this for requirements. To get started, state what the application will accept as input, like requirement 1.3.1 below.

Keep in mind that the implementation of your requirements will incorporate *an abstract class*, *inheritance*, *upcasting or downcasting*, and *polymorphism*; that will probably influence the requirements you choose to implement in this assignment. The example material supplied should be deleted before you submit.

### 1.3.1 Create Accounts

FinFree shall accept a variety of accounts from the user (bank, credit card, brokerage, etc.) as an input.

### 1.3.2 Report Account Balances

FinFree shall report the value of the assets contained in all of the accounts.

## 1.4 ILLUSTRATIVE OUTPUT FROM IMPLEMENTATION *EVALUATION CRITERION (ii) APPLIES*

### Provide illustrative output from your implemented application (so far) showing that the requirements have been met. Explain what class.method(s) produce it.

The following is produced by *AccountBalanceTest.displayCreditValueAndLimitTest().* *CreditCardAccount* and *CreditCard* objects are hard-coded data.

Card Balances | Card Limits

For all AMEX cards

Gold

Balance: 5000.0 | Remaining Limit: 2200.0

Platinum

Balance: 1000.0 | Remaining Limit: 2400.0

The Gold card limit started at 7200, then the user spent 5000, lowering the remaining limit to 2200. For the Platinum card, it started at 3400, the owner spent 1000, so only 2400 remains on their limit.

## 1.5 YOUR DIRECTORY

### Show a screenshot of your directory. This should include a parallel directory of JUnit tests where possible—package-by-package, class-by-class, and method-by-method, except for trivial ones.

All non-trivial methods in the Accounts are tested, otherwise they are ignored.

Graphical user interface, text, application, chat or text message

Description automatically generated

## 1.6 TECHNIQUES IMPLEMENTED *EVALUATION CRITERION (iii) APPLIES*

Your implementation should include *inheritance*, *polymorphism*, and *either an abstract class or interface* at least once, and in a manner that is useful to your application. Explain where and how you applied these, using the headings below.

### 1.6.1 Class model and Sequence Diagram

Identify where you included *inheritance*, *polymorphism*, and *abstract classes* or *interfaces* in your class model. Make classes and members *static* or not as per their intended usage. To do this use tools (e.g., Visio and Lucidchart), PowerPoint, or a combine models as in [this example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to cut and paste from). Insert indications in red (as in [this example](https://docs.google.com/spreadsheets/d/1ZvkerE9FkWHWwVGdzuy7YMBU6oBMFGZbA4sotFETs8Y/edit?usp=sharing)) to show where the three features below apply.

The UML class model for the accounts is [here](https://lucid.app/lucidchart/invitations/accept/10326096-486b-4e00-b852-09bbf57adea2). *Account* is abstract. Polymorphism occurs in the versions of *credit*() and *debit*(). This is the case specifically for CreditCardAccounts, because one of these accounts can contain multiple cards, and the system must credit or debit the correct credit card. The figure includes inheritance. Here is an image of the UML:

Diagram, table

Description automatically generated

### 1.6.2 Code showing an abstract class or interface

Show the relevant code (only) and explain why an abstract class or interface is appropriate here. It should be clear where the code is located (class and method).

There are only three kinds of accounts, so there is no need for the *Account* base class to be concrete since there will be no generic Fragments. Additionally, these three types will have different implementations of these methods.

Text

Description automatically generated

### 1.6.3 Code showing polymorphism

Show the relevant code (only) and explain why *polymorphism* is appropriate here. Recall that polymorphism is implemented in one of two ways – overriding methods in subclasses or overloading methods in the same class where the method signatures are different – and allowing the language runtime to dynamically invoke the correct method. It should be clear where the code is located (class and method).

The method *creidt*() and *debit()* for the CreditCardAccount must also apply the value changes to the individual credit card tied to the specific purpose. Since a Chase or AMEX account can have multiple credit cards, whereas a user only has one savings or brokerage account, these two methods are demonstrative of polymorphism. See a screenshot of the relevant code below:

### Text Description automatically generated

### s1.6.4 Code showing upcasting or downcasting

Show the relevant code (only) and explain why upcasting or downcasting is appropriate here. It should be clear where the code is located (class and method).

I was unable to find a suitable use or location for either of these. I would like to see if you had any suggestions here? Thanks!

## 1.7 YOUR CODE

Unless your facilitator requests another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request an alternative means.

Please find my code attached in the submitted zip file.

## 1.8 EVALUATION OF ASSIGNMENT 1



## Appendix 1 (if needed; should be referenced above, and will be read as-needed only)

## Appendix 2 (if needed; should be referenced above, and will be read as-needed only)

# Assignment 2

Implement the next release of your term project. You will incorporate exception handling and file I/O, as specified below.

Submit this completed Word document, including your name within the file name, observing and retaining the gray text like this. Retain the headings or edit them as indicated. Your Assignment 2 materials—in black 12-point Times New Roman—should not exceed 5 pages excluding references, figures, and appendices. Use the Appendix sections for additional material if you need to. These will be read on an as-needed basis.

As you code, use JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones and ones requiring I/O that cannot be unit tested. Use non-Junit classes for testing the latter, as in the worked example.

Include a ReadMe file describing where to run the application from, and including notes as necessary (not more).

Keep the evaluation criteria in mind, listed at the end and referred to in the headings.

## 2.1 SUMMARY DESCRIPTION, UPDATED AS APPLICABLE

## *EVALUATION CRITERION (i) APPLIES*

One- or two-paragraph overall description of your whole proposed term project. Edit your last description as needed.

This project concerns an application for younger, less financially savvy adults that would enable them to develop their own financial freedom through effective budgeting and planning. The application, called FinFree, incorporates information and data from the user’s bank, brokerage, and credit card accounts in order to help give them a brief overview of their current financial position. As users can have multiple cards through one issuer, like Chase or American Express, each credit card account contains all the cards associated with that account, as opposed to having separate classes for each individual card. The earliest iterations will ask the user to manually input this information, then later releases will ask for login information. Eventually, FinFree will incorporate an outside service such as Plaid as a login mechanism.

Once all the accounts are setup, to use FinFree, users can preset budgeting goals, like saving for a big trip or just generally increase the size of their savings account or IRA. FinFree will also ask the user to set predetermined spending limits in a variety of core categories, such as necessities like groceries as well as discretionary purchases like coffee or alcohol, in an effort to help their budgeting. Alerts would be sent out if they overspend in any category. Eventually, given a user’s habits and goals, the program will have sign up offers for credit cards as well as information for high interest savings accounts or low fee brokerage firms.

## 2.2 I/O EXAMPLE FROM PROJECTED COMPLETED PROJECT, UPDATED AS APPLICABLE *EVALUATION CRITERION (i) APPLIES*

Provide an example of projected *concrete* output for example input, indicating how users will interact with your application. You will not be held to fulfilling exactly this—it intended to help us understand the probable direction of your application as a whole.

FinFree would start at a menu with a number of options for the user to navigate through to see their current financial posture.

Here are two examples of a simple walkthrough in FinFree.

**FinFree:**

Welcome to FinFree, Aidan! Please select from one of the options below:

1. Check monthly net income
2. Check net income year to date
3. Check monthly spending
4. Check net worth
5. Check bank account balance
6. Check brokerage account value
7. Check credit card account balance

Ex 1:

**User:** 1

**System:**

You have been paid $1700 since the start of the month. You have spent $900 since the start of the month.

Your monthly net income is $800.

Ex 2:

**User:** 7

**System:**

You have 3 credit cards. Please select which card whose balance you are checking.

1. Chase Freedom
2. American Express Gold
3. Bank of America Cash Rewards

**User:** 2

**System:**

You have spent $452.81 on your American Express Gold card so far this month, and you have paid $0.00 off this balance. Your current balance on this card is $452.81.

## 2.3 REQUIREMENTS IMPLEMENTED IN THIS RELEASE NOT IMPLEMENTED BEFORE *EVALUATION CRITERION (i) APPLIES*

### 2.3.1 Create toString() for Account, CreditCard

FinFree shall output information from account classes into a readable, proper format for the accounts file as well as information related to each credit card.

### 2.3.2 Create Accounts File

FinFree shall create a file from all of the toString outputs of all of a user’s accounts, and it shall handle any errors that may arise from this process.

### 2.3.3 Read Accounts File

### FinFree shall read the accounts file at the start of the program to take in all of the user’s account information, allowing a user to store this in between sessions.

### 2.3.4 Accept Main Menu Data

### FinFree shall display the main menu and accept any user input when they are prompted.

### 2.3.5 Handle Bad Main Menu Input

### FinFree shall properly handle if a user attempts to enter invalid integers or non-integers when prompted for input(s) during the main menu phase.

## 2.4 I/O EVIDENCE THAT THE ABOVE FUNCTIONALITY WAS ACHIEVED

## *EVALUATION CRITERION (ii) APPLIES*

## This typically consists of screen shots of input and output, together with text explaining their context.

The following are produced by *AccountToStringTest.testCreditCardString(), AccountToStringTest.testBankString(),* and *AccountToStringTest.testInvestmentString(), respectively.* *BankAccount, InvestmentAccount, CreditCardAccount,* and *CreditCard* objects are hard-coded data. All data points are separated by commas, but for credit cards within a credit card account, those pieces of information are separated by semicolons, as seen below.

C{AMEX,Gold;1111 2222 3333 4444;012;10;25;0.0;7200.0,Platinum;5555 6666 7777 8888;345;11;25;0.0;3400.0}

B{Savings,123456789,Bank of America,1586.32,1.5}

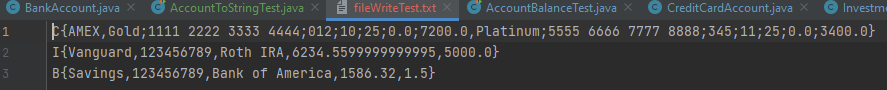
I{Vanguard,123456789,Roth IRA,6234.5599999999995,5000.0}

Below, find a screenshot of the code from *BankAccount.writeToFile(File file)*, which is where this information will be written to an output file:

Text

Description automatically generated

And, here is an example file output with the same data used in the above tests:



Additionally, in the main program, here is the relevant code that handles user input on the main menu.

Text

Description automatically generated

## 2.5 YOUR DIRECTORY

Show a screenshot of your directory. This should include a parallel directory of JUnit tests—package-by-package, class-by-class, and method-by-method, except for trivial ones.

Graphical user interface

Description automatically generated with medium confidence

## 2.6 TECHNIQUES IMPLEMENTED *EVALUATION CRITERION (iii) APPLIES*

Your implementation should exploit *file IO* and *exceptions* at least once, in as natural a manner as possible. Using the headings below, explain where and how you applied these.

### 2.6.1 Class model and Sequence Diagram

Indicate clearly in your class model where you applied file IO and exception handling, including a user-defined exception if possible. “Enforce what you intend.” For example, make classes and members *static* or not as per their intended usage. To do this use tools, PowerPoint, or combine models as in [this RUML example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to copy, cut and paste from). Insert indications in red (as in the example) to show where the three features below apply.

All of the writeToFile methods in all of the account types implement both file I/O as they read then write to a file, as well as exception handling that is necessary when completing these tasks, in case there is an IOException. The writeToFile method for the CreditCardAccount class will throw a NoCreditCardException when the user attempts to write this account to the user account text file before populating the account with any credit cards.

Diagram

Description automatically generated

### 2.6.2 Code showing *file I/O*

Show the relevant code (only). It should be clear where the code is located (class and method). Specify nontrivial methods with pre- and postconditions (and examples if this clarifies).

Here is the actual code written to complete the file I/O. This method will also be called on each account once the user opts to exit the program from the main menu. Note: the write method always appends, so the main program will erase the contents of the file after reading them upon startup.

Text

Description automatically generated

Here is the code from the JUnit testing that actually calls this method. Please see the screenshot from section 2.4 to see the output to the text file.

Text

Description automatically generated

Here is where the file input reading is done in the main program (Not shown: exception catches and the deletion of this file once done reading it).

Text

Description automatically generated

### 2.6.3 Code showing *exception*, preferably a user-defined exception

Show the relevant code (only) and explain why *exceptions* are appropriate here. It should be clear where the code is located (class and method). Specify nontrivial methods with pre- and postconditions (and examples if this clarifies).

FinFree contains a user-defined exception called NoCreditCardException, which is thrown by *CreditCardAccount.writeToFile(File file)*. This exception is thrown when the user sets up a credit card account, but failed to populate that account with any of the associated credit cards. Therefore, the program will not write that account down to the accounts text file. Here is the actual exception code:

A screenshot of a computer

Description automatically generated with medium confidence

Here is the exception being thrown:

Text

Description automatically generated

## 2.7 YOUR CODE

Unless your facilitator requests or allows another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request an exception. Specify nontrivial methods with pre- and postconditions (and examples if this clarifies). For excellent work, specify the class invariants.

Please find my code attached in the submitted zip file.

## 2.8 EVALUATION OF ASSIGNMENT 2



## Appendix 1 (will be read as-needed only—add more as necessary)

1/28/2021 7:03 AM

# Assignment 3

Implement the next release of your term project (preferably, or start a new one if you have to). You will incorporate generics. The same instructions as in Assignment 2 apply to this completed Word document, the gray text, the 5 page limit, appendices, JUnit tests, and a ReadMe file.

## 3.1 SUMMARY DESCRIPTION, UPDATED AS NECESSARY

*EVALUATION CRITERION (i) APPLIES*

One- or two-paragraph overall description of your proposed term project. Color in red the parts different from Assignment 2.

This project concerns an application for younger, less financially savvy adults that would enable them to develop their own financial freedom through effective budgeting and planning. The application, called FinFree, incorporates information and data from the user’s bank, brokerage, and credit card accounts in order to help give them a brief overview of their current financial position. The earliest iterations will ask the user to manually input this information, then later releases will ask for login information. Eventually, FinFree will incorporate Plaid as a login mechanism. From there, to use FinFree, users can preset budgeting goals, like saving for a big trip or just generally increase the size of their savings account or IRA. FinFree will also ask the user to set predetermined spending limits in a variety of core categories, such as necessities like groceries as well as discretionary purchases like coffee or alcohol, in an effort to help their budgeting. Alerts would be sent out if they overspend in any category.

## 3.2 ADDITIONAL REQUIREMENTS (FEATURES) IMPLEMENTED IN THIS RELEASE

*EVALUATION CRITERION (i) APPLIES*

Title and one or two sentences per requirement. Don’t repeat requirements implemented for prior assignments unless they are necessary to provide context—in which case, make it clear which are new vs. old.

### 3.2.1 Create Accounts File (OLD, Reworked)

FinFree shall create a file from all of the data of all of a user’s accounts, and it shall handle any errors that may arise from this process.

### 3.2.2 Read Accounts File (OLD, Reworked)

### FinFree shall read the accounts file at the start of the program to take in all of the user’s account information, allowing a user to store this in between sessions.

### 3.2.3 Differentiate Between Users (NEW)

FinFree shall store and process numerous users’ account information assigned to unique usernames.

### 3.2.4 Display Net Worth, Debt, and Cash Values (NEW)

FinFree shall display to the user constantly updated figures representing their net worth, credit card debt, and their available cash balances.

### 3.2.5 Accept Main Menu Data (OLD Requirement, NEW Features)

### FinFree shall display the main menu and relay the requested information back to the user.

## 3.3 I/O EVIDENCE OF ACCOMPLISHING THE REQUIREMENTS LISTED ABOVE

*EVALUATION CRITERION (ii) APPLIES*

Provide an example of actual input / output corresponding to the requirements above

Your response replaces this.

### Input / Output

Console I/O:

**FinFree:**

Welcome to FinFree, User! Please select from one of the options below:

1. Check monthly net income
2. Check net income year to date
3. Check monthly spending
4. Check net worth
5. Check bank account balance
6. Check brokerage account value
7. Check credit card account balance

Ex 1:

**User:** 5

**System:**

Your cash balance in your bank accounts is currently $15,000.

Ex 2:

**User:** 4

**System:**

Your net worth is $13,978.

### Output File(s)

SavedAccounts.txt outputs same information with same data as week 2, but it is now completed in a different object and function.

Ex:

C{AMEX,Gold;1111 2222 3333 4444;012;10;25;0.0;7200.0,Platinum;5555 6666 7777 8888;345;11;25;0.0;3400.0}  
I{Vanguard,123456789,Roth IRA,6234.5599999999995,5000.0}  
B{Savings,123456789,Bank of America,1586.32,1.5}

## 3.4 YOUR DIRECTORY

Show a screenshot of your directory. This should include a parallel directory of JUnit tests where applicable—package-by-package, class-by-class, and method-by-method, except for trivial and inapplicable ones.

Graphical user interface, text, application, chat or text message

Description automatically generated

## 3.5 YOUR UPDATED CLASS MODEL AND CLARIFICATION OF HOW THE EXECUTION WORKS

*EVALUATION CRITERION (i) APPLIES*

Supply a main use case, the class model, and the sequence diagram corresponding to the use case. These should be consistent and clear. Indicate clearly in your class model where you applied generics. To do this use tools, PowerPoint, or a combine models as in [this example](https://docs.google.com/spreadsheets/d/1vBmDVtWWh3EX0oehFFLRU0P6eR-fn4d0qVg1-XOUooM/edit?usp=sharing) (which you are free to cut and paste from). Insert indications in red to show where generics apply.

I have now moved the functionality for updating the value of accounts as well as writing account information to files to a generic file so as to further modularize my repository. The Generic class is the AccountFileAndValue and this used by the User class functions.

Diagram

Description automatically generated

## 3.6 WHERE GENERICS ARE IMPLEMENTED

*EVALUATION CRITERION (iii) APPLIES*

### 3.6.1 Class model fragment showing generic class

Explain where and how you applied *generic classes* in your class model.

FinFree implements a generic class in the new AccountFileAndValue class, whose attributes include a generic account as well as that account’s value and string representation. This class also now takes control of writing the account information to the text files that stores a user’s information in between sessions.

Additionally, FinFree implements generic methods in the new User class. The User class is what ties all of the accounts to a specific user. It contains a list collection of the type AccountFileAndValue to remove the need for any casting. This is also used for a number of functions the user class uses, like calculating net worth or debt. “Update” will be performed after every operation, so it is used here especially for the sake of efficiency and scalability. (Future updates will have a dashboard that will display this information[worth, debt,cash] at all times). Further, the User class contains a method where all of the accounts of a given type (bank, credit, or investment) are returned based on some input, which returns a List of a generic type.

### Diagram Description automatically generated

### 3.6.2 Code (including test code), input (if applicable), and output showing generics

Explain why the use of *generics* is appropriate here.

This object for the generic class below will be used, though it will be mostly used a collection of these objects, in the User class. The usage of generics is due to the fact that while each object will always have on associated integer and one associated string value, the actual main object will be one of three different possible account types and therefore must be generic. The reason to keep the object with the integer and String values is because they will be updated after every update operation.

The User class has three functions pictured below for retrieving net worth, debt, and cash of the user by checking what kind of account object is in the generic. Additionally, the User has a collection of these generic objects for all the user’s accounts, and it has a function for adding accounts to this list.

Text

Description automatically generatedText

Description automatically generatedText

Description automatically generated

## 3.7 YOUR CODE

*EVALUATION CRITERION (iii) APPLIES*

Unless your facilitator arranges another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request another transmission process (e.g., github).

Please find my code in the attached zip file.

## 3.8 INSTRUCTOR’S EVALUATION

