The theme of this assignment is to write code to for a bookkeeper to manage assets and income. The purpose is to give you practice writing classes, fields, methods, conditional statements, and constructors. You will also be introduced to polymorphism.

**II. Code Readability (20% of your project grade)**

* All variables (fields, parameters, local variables) must be given appropriate and descriptive names.
* All variable and method names must start with a lowercase letter. All class names must start with an uppercase letter.
* The class body should be organized so that all the fields are at the top of the file, the constructors are next, and then the rest of the methods.
* Every statement of the program should be on it's own line and not sharing a line with another statement.
* All code must be properly indented (see page 689 of the Lewis book for an example of good style). The amount of indentation is up to you, but it should be at least 2 spaces, and it must be used consistently throughout the code.
* You must be consistent in your use of {, }. The closing } must be on its own line and indented the same amount as the line containing the opening {.
* There must be an empty line between each method.
* There must be a space separating each operator from its operands as well as a space after each comma.
* There must be a comment at the top of the file that includes both your name and a description of what the class represents.
* There must be a comment directly above each method that, in one or two lines, states *what* task the method is doing, not how it is doing it. Do not directly copy the homework instructions.
* There must be a comment directly above each field that, in one line, states what the field is storing.
* There must be a comment either above or to the right of each non-field variable indicating what the variable is storing. Any comments placed to the right should be aligned so they start on the same column.

**III. Program Testing Document (20% of your project grade)**

**To receive full testing marks, you must write a testing report that shows that you thoroughly tested every method of the program.** The report should be a short English description for each test (what you are testing and what the expected result of the test is) followed by the actual result of the test. If you are using DrJava, you can enter the test into the interactions pane and then copy and paste the test code plus the result to your report. If you fail to complete the program, your report should indicate how you would go about testing the incomplete methods.

**Your grade on the testing report is how thoroughly you test your code, not how correctly your code runs.** If your code is not 100% correct then your report *should* show an incorrect result to some test. Testing methods that do not have conditional statements should be pretty straightforward, but you need to put thought into testing methods with conditional statements so that each branch of the if-statement is tested.

*Hint 1*: You can test multiple methods with one test. For example, you can test each setter/getter method pair together or you can test constructors and getter methods together.

***Hint 2: Do not put off testing to the end!*** Test each method after you complete it. Many methods depend on other methods. Delaying testing could mean cascading errors that cause your whole project to collapse. Since you need to test anyway, copy the tests you do into a document, and you are most of the way to completing your report.

If you are not using DrJava, you are allowed (but not required) create a separate class that tests your program. You must still write a testing report that documents the tests you do in this class. Do not place testing code into a main method of the classes below. That is not the purpose of a main method.

**IV. Java Programming (60% of your grade)**

**Guidelines for the program**:

* Unless specifically indicated, the listed methods must be public.
* You will need to create several instance fields to store data, and every field must be private.
* All fields must be initialized to an appropriate value. They can be initialized either as part if the field declaration or in the constructor. Even if you feel that the default value provided by Java is appropriate, you still must give an explicit initialization.
* Any method whose name begins with set should only assign a value to an appropriately named field. The method should do no other processing. Any processing described in a set method description below is for information only. That actual processing will be done by other methods.
* Any method whose name begins with get should only return the appropriate value. No other processing should occur in these methods.
* Your class must include only the methods listed. You may not write any other methods.
* The behavior of your methods must match the descriptions below.
* You should not write any loops in your program (though loops are allowed in the testing code).

1. Asset  
   The Asset represents any tangible property that has a value. The Asset class will need instance fields to keep track of the name of the asset, a description of the asset, the cost basis of the asset (how much you spent to acquire it), the current price someone is willing to buy the asset for, and the capital gains for the asset (how much profit you made when/if you sold it). The class will have the following methods:
   1. getName  
      Takes no input and returns a String. Returns the name of the asset.
   2. setName  
      Takes a single String as input and returns nothing. The name of the asset is changed to the input value.
   3. getDescription  
      Takes no input and returns a String. Returns a description of what the asset is.
   4. setDescription  
      Takes a single String value as input and returns nothing. The description of the asset is changed to the input value.
   5. getCostBasis  
      Takes no input and returns a double. Returns the cost basis of the asset (how much you paid to acquire it).
   6. setCostBasis  
      A *protected* method that takes a single double as input and returns nothing. Changes the cost basis of the asset.
   7. getCurrentPrice  
      Takes no input and returns a double. Returns the current price of the asset.
   8. setCurrentPrice  
      Takes a single double value and returns nothing. The input value is the current price of the asset.
   9. getCapitalGains  
      Takes no input and returns a double. Returns the capital gains from the asset (how much profit you made when selling all or part of it).
   10. setCapitalGains  
       A *protected* method that takes a single double as input and returns nothing. Changes the capital gains of the asset.

The Asset class will have one constructor:

* 1. The constructor takes two inputs: a String that is the asset name and a double that is the cost basis for the asset.

The asset should be created with an initial capital gains of 0.

1. Bond  
   A Bond instance represents an asset that is a loan from a government or corporation. The Bond class should extend the Asset class. The Bond class has the same features as the Asset class with the following additions. The Bond class should include additional fields to keep track of the principal of the bond, the interest rate of the bond, and the number of bonds owned. The Bond class should have the following additonal methods:
   1. getPrincipal takes no input and returns an int. Returns the principal of the bond. (The principal does not change.)
   2. getInterestRate takes no input and returns a double. Returns the interest rate of the bond.
   3. setInterestRate takes a single double as input and returns nothing. Changes the interest rate for the bond to the input value.
   4. payInterest takes no input and returns a double. Returns the product of the interest rate and the principal.
   5. buy  
      Takes no inputs and returns a double. The method purchased a bond. The cost basis is increased by the current price, the current price is returned, and the number of bonds owned is increased by 1.
   6. sell  
      Takes no inputs and returns a double. The method sells a bond. If there are no bonds owned, the method returns 0. Otherwise, the cost basis is reduced by (*cost basis* / *number of bonds owned*), the capital gains is increased by the difference between the current price and the amount the cost basis was reduced, and the number of bonds owned is decreased by 1. The current price is returned.
   7. getNumberOwned  
      Takes no input and returns an int that is the number of bonds owned.

The Bond class should have one constructor:

* 1. the constructor takes three inputs: a String that is the name of the bond, an int that is the principal, and double that is the interest rate. The current price should be set to equal the principal. The initial cost basis should be 0.

1. Equity  
   An Equity represents an asset where you can own shares of the asset. The Equity class should extend the Asset class. The Equity class has the same features as the Assetclass plus the following additions. The Equity class should have additional instance fields to keep track of the symbol of the equity and the number of shares owned of the equity. For the purposes of this homework, all equities will have single character symbols. The Equity class will have the following additional methods:
   1. getSymbol  
      takes no input and returns a char. Returns the single character symbol of the equity. (The equity's symbol does not change.)
   2. getNumberShares  
      Takes no input and returns a double. Returns the number of shares of the equity.
   3. setNumberShares  
      A *protected* method that takes a single double value as input and returns nothing. The input value is the number of shares of the equity.

The Equity class has one constructor.

* 1. the constructor takes three inputs: a String that is the name of the equity, a char that is the symbol, and a double that is the current price. The equity should be created with an initial cost basis of 0.

1. MutualFund  
   A MutualFund instance represents an equity that is the shares of a mutual fund. The MutualFund class should extend the Equity class. The MutualFund class has the same features as the Equity class with the following additions. The MutualFund should have an additional field that stores the "load" of the fund. The MutualFund class should have the following additonal methods:
   1. getLoad  
      Takes no inputs and returns a double. Returns the current load of the mutual fund. (The load is a percentage that is charged on all sales of the fund.)
   2. setLoad  
      Takes a single double as input and returns nothing. Changes the load of the mutual fund to be the input value.
   3. buy  
      Takes a double as input that the amount of money you are investing in the mutual fund. The method returns a double. If the money amount is not positive, the method returns 0 and does nothing. Otherwise, the method increases the number of shares owned in the mutual fund by *input amount x (100% - load) /current price*. The cost basis is increased by the input amount, and the input amount is returned.
   4. sell  
      Takes one input: an double that is amount of money you are withdrawing from the mutual fund. The method returns a double. If the input number is not positive or is larger than the current value of the mutual fund, the method should return 0 and do nothing. Otherwise, the number of shares owned is decreased by *amount withdrawn / current price*. The cost basis is decreased by the ratio of the number of shares sold to the number of shares owned prior to this sale. (For example, if you sell 1/3 of your shares, the cost basis should decrease by 1/3.) The capital gains is increased by the difference between the amount withdrawn and the amount that the cost basis decreased. The method returns the amount withdrawn.

The MutualFund class should have one constructor:

* 1. the constructor takes three inputs: a String that is the name of the mutual fund, a char that is the symbol, and a double that is the current price.

1. Stock  
   A Stock instance represents an equity that is the shares of stock in a company. The Stock class should extend the Equity class. The Stock class has the same features as the Equity class with the following additions. The Stock class should have the following additonal methods:
   1. buy  
      Takes two inputs: an int that is the number of shares and a double that is the commission. The method returns a double. The method increases the number of shares owned in the stock by the input value. The method computes the value *z = n x p + c* where *n* is the input number of shares, *p* is the current price per share, and c is the input commission. The method should increase the cost basis for the stock by *z* and return *z*.
   2. sell  
      Takes two inputs: an int that is the number of shares and a double that is the commission. The method returns a double. If the input number of shares is greater than the number currently owned, the method returns 0 and makes no other changes. Otherwise, the number of shares owned is decreased by the input number of shares. The cost basis is decreased by the ratio of the input number of shares to the number of shares owned prior to this sale. (For example, if you sell 1/3 of your shares, the cost basis should decrease by 1/3.) The capital gains is increased by the difference between *(the number of shares sold x the current price) - the commission* and the amount that the cost basis decreased. The method returns *(the number of shares sold x the current price) - the commission*.
   3. split  
      Takes two inputs, a int that is the ratio numerator and an int that is the ratio denominator. The method should return a double. If either input value is zero, nothing is done and 0 is returned. Otherwise, the number of owned shares changes by the ratio *numerator / denominator*. (For example, split(2,1) should double the number of shares while split(1,2) should result in one half the number of shares.) If the resulting number of shares contains a fractional share (not a whole number), then the fractional share must be sold so that the final number of shares is a whole number. Follow the same logic as for the sell method except that no commission is charged, and the value returned should be the same as what would be returned by the sell method if the commission were 0. If there is no fractional share (the initial result of the split is a whole number), then 0 should be returned.

The Stock class should have one constructor:

* 1. the constructor takes three inputs: a String that is the name of the stock, a char that is the symbol, and a double that is the current price.

1. CashAsset  
   The CashAsset class represents money reserves. The CashAsset class will need fields to keep track of the current balance, the interest rate paid for savings, the interest rate changed for loans, the limit that can be loaned, an overdraft penalty, the amount of interest accrued this month (but not paid), and whether the account has been overdrafted. The class should have the following methods.
   1. getBalance  
      Takes no input and returns a double. Returns the current balance in the account.
   2. getSavingsRate  
      Takes no input and returns a double. Returns the interest rate applied to positive balances.
   3. setSavingsRate  
      Takes a single double as input and returns nothing. Changes the interest rate applied to positive balances.
   4. getLoanRate  
      Takes no input and returns a double. Returns the interest rate applied to negative balances.
   5. setLoanRate  
      Takes a single double as input and returns nothing. Changes the interest rate applied to negative balances.
   6. getLoanLimit  
      Takes no input and returns a double. Returns the loan limit for the account.
   7. setLoanLimit  
      Takes a single double as input and returns nothing. Changes the loan limit for the account.
   8. getOverdraftPenalty  
      Takes no input and returns a double. Returns the overdraft penalty for the account.
   9. setOverdraftPenalty  
      Takes a single double as input and returns nothing. Changes the overdraft penalty for the account.
   10. deposit  
       Takes a single double as input and returns nothing. Increases the balance by amount.
   11. withdraw  
       Takes a double and a boolean as input and returns a boolean. If the input boolean is false and if the input double is greater than the account balance, the method returns false and makes no changes to the account balance. Otherwise, the method returns true and reduces the balance by the input amount.
   12. processDay  
       Takes no input and returns nothing. If the account balance is positive, multiplies the balance by the savings rate (divided by 365) and adds the amount to the accrued monthly interest. If the account balance is negative, multiplies the balance by the loan rate (divided by 365) and adds the (negative) amount to the accrued monthly interest.
   13. processMonth  
       Takes no input and returns nothing. Adds the accrued monthly interest to the balance and resets the accrued monthly interest to zero. If the account balance plus the accrued monthly interest was negative and exceeded the (negation) of the loan limit at any time that processDay was called since the last time processMonth was called, the overdraft penalty is subtracted from the balance.
   14. *(Optional)* getAccruedInterest  
       Takes no input and returns nothing. Returns the current accrued monthly interest amount. This method is optional because it was not included in the original assignment posting.
   15. *(Optional)* setAccruedInterest  
       A *protected* method that takes a single double as input and returns no value. Changes the accrued interest for the month to the input value.

The Cash class should have a single constructor:

* 1. the constructor takes four inputs: a double that is the interest rate for savings, a double that is the interest rate for loans, a double that is the loan limit, and a double that is the overdraft penalty.

1. Customer  
   The Customer class represents a customer account. The class should have fields to keep track of a stock instance, a bond instance, a mutual fund instance, a cash account, and a commission rate. The class should have the following methods.
   1. getBond  
      Takes no input and returns a value of type Bond. Returns a bond instance associated with this account.
   2. setBond  
      Takes an input of type Bond and returns nothing. Changes the bond instance associated with this account.
   3. getMutualFund  
      Takes no input and returns a value of type MutualFund. Returns a mutual fund instance associated with this account.
   4. setMutualFund  
      Takes an input of type MutualFund and returns nothing. Changes the mutual fund instance associated with this account.
   5. getStock  
      Takes no input and returns a value of type Stock. Returns a stock instance associated with this account.
   6. setStock  
      Takes an input of type Stock and returns nothing. Changes the stock instance associated with this account.
   7. getCashAccount  
      Takes no input and returns a value of type CashAsset. Returns the cash account associated with this account.
   8. setCashAccount  
      Takes an input of type CashAsset and returns nothing. Changes the cash account associated with this account.
   9. getCommissionAmount  
      Takes no input and returns a double. Returns the commission amount for the account.
   10. setCommissionAmount  
       Takes a single double as input and returns nothing. Changes the commission amount for the account.
   11. currentValue  
       Takes no input and returns a double. Returns the cash account balance plus the current price of the bond plus the number of shares times the current price for both the stock and the mutual fund (if these all exist).
   12. getCapitalGains  
       Takes no input and returns a double. Returns the sum of the capital gains of the bond, mutual fund, and stocks, if they exist.
   13. deposit  
       Takes a double and returns no value. Adds the input amount to the cash account balance.
   14. withdraw  
       Takes a double as input and returns a boolean. Calls the withdraw method of the cash account with the boolean set to false in order to withdraw the desired amount (if it exists in the acccount). The method returns the value returned by the cash account withdraw method.
   15. sellBond  
       Takes no input and returns nothing. Calls the bond's sell method and deposits the value returned into the cash account's balance.
   16. buyBond  
       Takes no input and returns a boolean. If the current price of the bond is larger than the customer's current value, the method returns false and does nothing else. Otherwise, the bond's buy method is called and the amount returned is subtracted from the cash account's balance. (Call the cash account's withdrawmethod with the boolean set to true to force the withdrawal even if the funds are not there.)
   17. payBondInterest  
       Takes no input and does nothing. Call's the bond's payInterest method and deposits the value returned into the cash account.
   18. withdrawMutualFund  
       Takes a single double as input that is the amount to withdraw and returns nothing. Calls the mutual fund's sell method with this number and adds the amount returned to the cash balance.
   19. buyMutualFund  
       Takes a single double as input and returns a boolean. If the input value is larger than the current value of the customer, the method returns false and does nothing else. Otherwise, the method call's the mutual fund's buy method with the input value, the cash account balance is reduced by the amount returned from the buy method and true is returned.
   20. sellStockShares  
       Takes a single int input that is a number of shares and returns nothing. Calls the Stock sell method with this number of shares and the account commission value and adds the amount returned to the cash balance.
   21. buyStockShares  
       Takes an int as input that is the number of shares to sell and returns a boolean. If the result of calling the stock buy method with given number of shares and the account commission amount would cost more than the customer's current value, the method returns false and does nothing else. Otherwise, the stock buymethod is called with the appropriate values, the cash balance is reduced by the amount returned from the buy method, and true is returned.

The Customer class should have two constructors:

* 1. The first constructor takes two inputs: a CashAsset that represents the cash account and a double that is the commission.
  2. The second constructor takes five inputs: a CashAsset that is the cash account, a Bond, a MutualFund, a Stock and a double that is the commission.