BSM 262: Object Oriented Programming-II
Programming Project 1

Due date: 11:59PM, Wednesday, April 12

1 Overview

In this assignment, you will be implementing code for investment account for a customer. This class will keep track of owned assets of a customer such as bond, equity, and mutual fund. The real purpose of the assignment is to give you practice writing classes, fields, methods, conditional statements, and constructors. You will also be introduced to polymorphism.

Please submit your assignment to the ubs as a single zipfile containing: Asset.java, Bond.java, Equity.java, MutualFund.java, Customer.java, and your testing report.

NOTE: There are technical terms. You do not have to understand the meaning of the terms. The project instructions will tell you where to set a value and where to use a value so that you can code this assignment without knowing anything about these terms. This is a fairly common situation in the real world. A software engineer will work to understand the customer's needs, and design the instructions (or project specifications) so the software developers can write the program without necessarily understanding everything about the business.

2 Code Readability (20% of your project grade)

The comments above the class and above each method must be written in **JavaDoc** format. You are already introduced to JavaDoc style commenting in the lectures. You can also find a description in the Java in a Nutshell textbook.

JavaDoc is a tool used to generate documentation from Java source code. The JavaDoc tool reads specially formatted comments in Java source code and generates HTML pages that document the classes, interfaces, methods, and fields in the source code. Here are a couple of examples:

Example 1: Class and a method
```
/**
* This is a sample class that demonstrates how to use JavaDoc.
* @Author:  John DOE
*/
public class SampleClass {
     /**
     * This is a sample method that adds two numbers together.
     * @param a the first number to add
     * @param b the second number to add
     * @return the sum of a and b
     */
     public int add(int a, int b) {
```

Example 2: Method with return type and inputs
```
/**
* Return true of the given date/time is daylight savings.
* Daylight savings time begins 2am the second Sunday of March and ends 2am the first
Sunday of November.
*
* @param month - represents the month with 1 = January, 12 = December
* @param date - represents the day of the month, between 1 and 31
* @param day - represents the day of the week with 1 = Sunday, 7 = Saturday
* @param hour - represents the hour of the day with 0 = midnight, 12 = noon
* @return true/false - if the given date is daylight savings
*
* Precondition:  the month is between 1 and 12, the date is between 1 and 31, the day
is between 1 and 7 and the hour is between 0 and 23.
*/
public static boolean isDayLightSavings(int month, int date, int day, int hour) {
```

Example 3: Method with no return type and no inputs
```
/**
* Displays a welcome message on the console.
*/
public void displayWelcomeMessage() {
```

Example 4: Method with a return type and no inputs
```
/**
* Generates a random number between 1 and 10 (inclusive).
*
* @return the random number generated
*/
public static int generateRandomNumber() {
```

Example 5: Constructor
```
/**
* Constructs a new Employee object with the given name and salary.
*
* @param name the name of the employee
* @param salary the salary of the employee
*/
public Employee(String name, double salary) {
```

To receive the full readability scores, your code must follow the following guideline:
• 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. 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 is in proper **JavaDoc** format and includes both your name and a description of what the class represents. The comment should include tags for the author.
• There must be a comment directly above each method (including constructors) that is in proper **JavaDoc** format and states what task the method is doing, not how it is doing it. The comment should include tags for any parameters, return values and exceptions, and the tags should include appropriate comments that indicate the purpose of the inputs, the value returned, and the meaning of the exceptions.
• 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.

Here is the readability rubric: **General Program Structure**

Points	Metric
10	Excellent, readable code. Correct comments at top of class file, above each method and above each field. Each comment at most one or two sentences and is a good description. Class has fields first, then constructors, then the remaining methods. All code follows the proper indentation and naming style.
9	Very readable code. There are only a few places (at most 5) where there is inconsistent or missing indentation or poorly named fields/methods, or missing comments, or comments that are not good descriptions, or fields or constructors buried inside the code or missing whitespaces.
7	Reasonably readable code. Overall good commenting, naming, and indentation. However, there are more than 5 places with missing or inconsistent indentation, poor variable names, or missing, unhelpful, or very long comments OR many comments that are a direct copy of the homework description.
6	Poorly readable code. A significant number of missing comments or inconsistent and/or missing indentation or a large number of poorly named variables. However, at least half of the places that need comments have them.
4	Some commenting done (close to half), but missing majority of the comments.
3	There is at least five useful comment in the code OR there is some reasonable indentation of the code.
0	Code very unreadable. No useful comments in the code AND/OR no indentation in the code.

JavaDoc Comments:

Points	Metric
10	All comments above the class and methods follow the JavaDoc comment style. Correct tags are added for the author, all method parameters and return types and any explicitly thrown exceptions.
9	All comments above the class and methods follow the JavaDoc comment style, and most of the comments contain and have correct all required tags. (Possibly a few misspelled tags.)
7	Most comments above the class, fields, and methods follow the JavaDoc comment style, but a significant number of places are missing tags or the tags are misspelled so the comments are not formatted properly on the webpage.
5	Most comments in JavaDoc style of /** but missing most tags OR has most tags but the comments do not start with /** and so do not appear on the webpage.
0	No JavaDoc comments provided.

3 Program Testing Document (20% of your project grade)

The current softwares are very complex and one minor error might even cause a death. The software glitches starting to become a major issue and destroying companies. Now, standard practice is that all code must be thoroughly verified before a company is willing to release it. Even some companies are requiring the programmers to write the test cases before even start programming. In this class, we will not be that strict, but you will need to test your code.

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 or Turkish 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 (this will not result in point deduction on your testing or readability), 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! est 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.
Here is the testing rubric:

Points	Metric
20	Excellent demonstration of how to test the program. Completed tests for each working method of the student's submission, either on its own or as part of a related "unit" of the program. Good descriptions of how to test the parts of the submission that are not working. Tests of conditional statements cover the different possible execution paths. Each test gives the true output of the student's submission. A note on any test that gives an incorrect result.
18	Very good: Good descriptions, tests covers most of the program but a few tests were missed.
15	Reasonable demonstration of how to test the program. The tests performed and the testing description covers a majority of the program; however, multiple obvious cases were missed.
12	Poor demonstration of how to test the program. The testing report shows how to test some of the methods and conditional statements, but at least half of the needed tests are missing.
10	Some testing done (at least half), but nothing that demonstrates the proper way to test a conditional statement.
5	There is at least five proper testing of the methods/constructors in the code.
0	The testing report does not match the behavior of the student's code.

4 Programming (60% of your grade)

Guidelines for the programming part:

• Unless specifically indicated, the listed methods must be public instance methods.
• 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).

For the programming part, create the following classes that will keep track of a customer's assets:

4.1 Asset class:

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:

Constructor: The Asset class will have one constructor. 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.

Methods: The Date class should have the following methods:
• **getName:** takes no input and returns a **String**. Returns the name of the asset.
• **setName:** takes a single **String** as input and returns nothing. The name of the asset is changed to the input value.
• **getDescription:** takes no input and returns a **String**. Returns a description of what the asset is.
• **setDescription:** takes a single **String** value as input and returns nothing. The description of the asset is changed to the input value.
• **getCostBasis:** takes no input and returns a **double**. Returns the cost basis of the asset (how much you paid to acquire it).
• **setCostBasis:** takes a single **double** value as input and returns nothing. Changes the cost basis of the asset.
• **getCurrentPrice:** takes no input and returns a **double**. Returns the current price of the asset.
• **setCurrentPrice:** takes a single **double** value and returns nothing. The input value is the current price of the asset.
• **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).
• **setCapitalGains:** takes a single double as input and returns nothing. Changes the capital gains of the asset.

4.2 Bond class:

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:

Constructor: The Bond class should have a single constructor that takes three inputs: a **String** that is the name of the bond, an **double** that is the principal, and a **double** that is the interest rate. The current price should be set to equal the principal. The initial cost basis should be 0. (**Hint:how can you use super()?**)

Methods: The Bond class should have the following methods:
• **getPrincipal:** takes no input and returns an **double**. Returns the principal of the bond. (The principal does not change.)
• **getNumberOwned:** takes no input and returns an **int** that is the number of bonds owned.
• **setNumberOwned:** takes a single **int** as input and returns nothing. Changes the number of bond to the input value.
• **getInterestRate:** takes no input and returns a **double**. Returns the interest rate of the bond.
• **setInterestRate:** takes a single **double** as input and returns nothing. Changes the interest rate for the bond to the input value.
• **payInterest:** takes no input and returns a **double**. Returns the product of the interest rate and the principal.
• **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.
• **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.

4.3 Equity class:

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 Asset class 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:

Constructor: The Equity class should have a single constructor that 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. (**Hint:how can you use super()?**)

Methods: The class should have the following methods:
• **getSymbol:** takes no input and returns a **char**. Returns the single character symbol of the equity. (The equity's symbol does not change.)
• **getNumberShares:** takes no input and returns a **double**. Returns the number of shares of the equity.
• **setNumberShares:** takes a single **double** value as input and returns nothing. The input value is the number of shares of the equity.

4.4 MutualFund class:

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:

Constructor: The MutualFund class should have a single constructor that 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.

Methods: The class should have the following methods:
• **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.)
• **setLoad:** takes a single double as input and returns nothing. Changes the load of the mutual fund to be the input value.
• **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.
• **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 (current price times number of shares), 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.

4.5 Customer class:

The **Customer** class represents a customer account. The class should have fields to keep track of a bond instance, a mutual fund instance, and total cash for the customer. The class should have the following methods.
Constructor: The Customer class should have one constructors takes five inputs: a Bond, a MutualFund, a String that is the first name of the customer, a String that is the last name of the customer, and a double that is the total cash of the customer.
Methods: The class should have the following methods:
• **getFirstName:** takes no input and returns a String that is the first name associated with the account.
• **setFirstName:** takes a String as input and returns nothing. Changes the first name associated with the account.
• **getLastName:** takes no input and returns String the last name associated with the account.
• **setLastName:** takes a String as input and returns nothing. Changes the last name associated with the account.
• **getBond:** takes no input and returns a value of type Bond. Returns a bond instance associated with this account.
• **setBond:** takes an input of type Bond and returns nothing. Changes the bond instance associated with this account.
• **getMutualFund:** takes no input and returns a value of type MutualFund. Returns a mutual fund instance associated with this account.
• **setMutualFund:** takes an input of type MutualFund and returns nothing. Changes the mutual fund instance associated with this account.
• **currentValue:** takes no input and returns a double. Returns the current price of the bond times the

number of bonds owned plus the number of shares times the current price for the mutual fund (if these all exist).
• **getCapitalGains:** takes no input and returns a double. Returns the sum of the capital gains of the bond and mutual fund, if they exist.
• **sellBond:** takes no input and returns nothing. Calls the bond's sell method and deposits the value returned into the customer's total cash.
• **buyBond:** takes no input and returns a boolean. If the current price of the bond is larger than the customer's total cash, the method returns false and does nothing else. Otherwise, the bond's buy method is called and the amount returned is subtracted from the customer's total cash, the method returns true.
• **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 total cash.
• **buyMutualFund:** takes a single double as input and returns a boolean. If the input value is larger than the total cash 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 total cash is reduced by the amount returned from the buy method and true is returned.

Programming part of the assignment will have 4 main parts including Fields/Getter/Setter methods, Constructors, Object-Oriented coding, and Non-Getter/Setter methods. Here is the programming rubric:

Fields/Getter/Setter Methods: 20 points

Points	Metric
20	Perfect design and implementation of fields and getter/setter methods. All necessary fields and getter/setter methods provided. All fields are private and the getter/setter methods are public. All fields are given explicit initial values either at the declaration or in the constructor. All getter/setter methods only set or retrieve the field values. There are no extraneous fields. There are no static fields or methods.
18	Very good design and implementation of fields and getter/setter methods. In almost all cases, the above features are present in the fields and getter/setter methods. However, there are a couple places where the above features are missing (for example a couple missing methods, an extraneous field, or a couple fields that do not get initial values) OR there is some single error type that shows up in the code (for example, most fields and getter/setter methods are correct, but a significant number of fields are missing explicit initial values).
15	Reasonable design and implementation of fields and getter/setter methods. A majority of the fields and getter/setter methods meet the above criteria, but a significant number have errors and there are multiple types of errors OR all the fields and getter/setter pairs are correct except that most or all have the same type of error (for example, everything is correct except that most of the fields and methods are static).
10	Poor design and implementation of fields and getter/setter methods. A majority of the fields and getter/setter methods have errors, but there are a significant number that are correctly implemented. Or, most of the fields and getter/setter methods are correctly implemented except for two or three consistent errors (for example, all fields are public, static, and not given initial values).
5	There is at least one case of a private, non-static required field that has an explicit initial value and proper non-static getter/setter methods for the field.
0	Code does not demonstrate proper use of fields and getter/setter methods. There is no case of a private, non-static, required field that has an explicit initial value and proper non-static getter/setter methods for that field.

Constructors: 10 points

Points	Metric
10	Perfect design and implementation of constructors.
9	Very good design and implementation of constructors. The constructors work correctly but have extra unnecessary operations or duplicate computation.
7	Reasonable design and implementation of constructors. At least one class has a correct constructor.
5	Poor design and implementation of constructors. All constructors fail to properly set field values OR the methods are not really constructors because they are incorrectly given return values or incorrect names.
0	No constructors provided.

Object-Oriented Coding: 10 points

Points	Metric
10	Perfect object-oriented design. Classes correctly use inherited methods and correctly override methods (if needed). There are no extraneous methods. There are no unnecessary fields storing duplicate data. Correct use of this and super. The code (other than the constructor) uses getter/setter methods when available and not the fields directly.
9	Very good object-oriented design. Classes correctly use inherited methods and correctly override methods (if needed). There are no extraneous methods. There are no unnecessary fields storing duplicate data. Correct use of this and super. The code may incorrectly use fields directly instead of the available getter/setter methods. The code may incorrectly use public getter/setter methods in the constructors.
7	Reasonable object-oriented design. In multiple places, the code uses inherited methods. There are some places where the inherited and/or overridden methods are not done correctly leading to a few extra fields or unnecessary methods.
5	Poor object-oriented design. There is some use of inherited methods, but the code does not properly use the inherited methods or properly override the methods (if needed) so there are many unnecessary extra fields or methods.
0	No object-oriented design. There is no use of inherited methods or overriding of inherited methods.

Non-getter/setter Methods: 20 points

Points	Metric
20	Perfect logic and coding. All methods present and correct. There is no unnecessary operations or a significant amount of unnecessary duplicated code.
18	Very good logic and coding. All methods that require conditional statements have well-formed statements for the problem. All the methods that do not require conditional statements are correct or have have correct logic but only minor errors.
15	Reasonable logic and coding. At least half of the methods that require conditional statements have well-formed statements, possibly with minor errors. At least half of the methods that do not require conditional statements have the correct mathematical logic, possibly with minor errors.
10	Poor logic and coding. There are at least two cases of methods that require conditional statements that have well-formed statements, possibly with minor errors.
5	At least one non-getter/setter method is either correct or at least has a correctly formed conditional statement for the problem.
0	No non-getter setter method is correct and no method that requires a conditional statement has a correctly formed conditional statement.

Penalties

Deduction	Cause
-10	The submitted code does not compile.
-10	Code that contradicts the rules of the assignment (example: using loops in the non-testing portion of the assignment).

5 Course Honor Policy

See the university policy on academic integrity for general rules that you should follow on tests and quizzes. Rules specific to programming in this class are listed here.

Programming is a collaborative enterprise. However, you cannot be an equal collaborator until you first build up your own programming skill set. Because of that, it is essential that you do programming project coding on your own. That does not mean that you can not seek help or give help to other students, but the assistance must be at a high level - for example English descriptions of how to solve a problem and not coding descriptions. The most important skill you will develop in this class is to translate a solution description into the Java commands needed to implement the solution. If you copy code from other sources, you are not developing this needed skill for yourself.

Here is a short list of things you may and may not do:

• **DO NOT** send an electronic copy of your code to another student. (Exception: You may share with your lab partner a copy of the code you wrote together during the recitation section.)

• **DO NOT** look at another person's code for the purpose of writing your own.

• **DO NOT** tell another student the Java code needed for an assignment.

• **DO** sit down with a student and go over the student's code together in order to help the student find a bug.

• **DO** describe in English (not code!) the steps needed to solve an assignment problem.

• **DO** show students how to do certain Java coding tasks as long as the task does not directly relate to a homework question.

• **DO** use coding examples from the lecture and textbook as a guide in your programming.

• **DO NOT** search the internet to find the exact solutions to assignment problems.

• **DO NOT** post code you write for this class onto an internet site such as StackExchange.

• **DO NOT** post the homework specific problem descriptions onto internet sites such as StackExchange.

• **DO NOT** use any online artificial intelligence tools.