**Summary**

Build three classes (**Money**, **Date** and **Bill**) for use in managing a set of outstanding and paid bills.  Submit these **three classes and a test driver (4 files total)** that exercises all functionality for these classes.  (Submit only the .java source code files.)  In this assignment, we’ll explore class composition, overloading, accessors and mutators, privacy leaks, toString(), and equals().

**Introduction**

Just as in previous assignments, we will not only be focusing on code, but also taking a “bigger picture” view of the overarching design, using the software development techniques learned this week (specifically, class invariants, privacy leaks, and more on object composition).   In this assignment, the interfaces are already defined for you below.  Once you have classes implemented for Money, Date, and Bill, you will need to develop a test harness (or testing strategy) to exercise the functionality of your class.  I’ve provided a sample driver that may need additional testing as a starting point (the link to the driver file is in the driver class description section below), but this is simply a starting point and will not test every function in Bill, Date, or Money (and your final driver should). Since a Bill makes use of a Money and a Date object, we can start our development there, but first, a bit on the *class invariants* we will want to build into our system.

**Class Invariants**

Your software must enforce the following rules:

(1)   Money Class

1. Our internal cents should be between [0-99] at all times
2. Our internal Cents and Dollars should never be < 0, at all times

(2)   Bill Class

1. The internal paid date should be on or earlier than the due date.
2. Every Bill is owed someone, so the originator should never be empty.

(3)   Date Class

1. All days should be between [1-31]
2. All months should be between [1-12]
3. All years should be between [2001-2024]

(4)   Data privacy should be maintained at all times; verify there are no privacy leaks such as returning aliases to an private object instance variable when you really should be returning a copy of that object instead.  (Be sure to make prevent a similar type of privacy leak by making copies of all incoming objects as well in set methods.)

**The Money Class**

This class is used to track a USD amount consisting of two integers to manage dollars and cents.  All dollars and cents will be positive or 0, and cents will never exceed 99.  The data and methods below will be called from the test driver I’ll use, so be sure to adhere to the names of the functions, as well as the number, order, and type of arguments handed to each function, as indicated below.  Also, previous labs and lectures contains hints and details on how to build and test the Money or Date class.

**Data Members**

* You should define a pair of instance variables for use in tracking dollars and cents These should be private and avoid privacy leaks.

**Method Members**

* Money(int dol)
  + This constructor initializes using dollars input and assumes no cents
* Money(int dol, int cent)
  + This constructor initializes dollars and cents accordingly
* Money(Money other)
  + This constructor could redirect to the one above
* Getters/setters for dollars and cents
  + These should be careful to not invalidate class invariants
  + Also these functions should avoid privacy leaks
* double getMoney()
  + A getter for the total monetary amount, as a double
  + Returns 5.75, for example
* void setMoney(int dol, int cent)
  + Sets our dollars and cents, accordingly
  + This function should make use of setters for dollars and cents
* void add(int dol)
  + Adds the int passed into the function to dollars
  + Should accept positive and negative arguments.
* void add(int dol, int cents)
  + Adds to dollars and cents the two ints passed into the function.
  + Should accept positive and negative arguments (though both arguments have to be positive or negative).
* void add(Money other)
  + Adds to this object the dollars and cents stored in the other object.
* boolean equals(Object o)
  + Determines if this money object is equal to that Money object
* String toString()
  + Prints out a Money object as a String, such as “$3.75”

**The Date Class**

This should be similar to the Date classes we’ve covered in class and in the textbook.  Do **not** use any built-in Date classes for this assignment, and provide all of the common class components for this Date.  For example, you should add:

**Data Members**

* These data items should be private or final
* Constructors & copy constructors
* Getters/setters for month, day and year
* boolean equals(Object o)
* String toString()

**Method Members**

* Follow the Savitch examples for how to build the methods needed in a Date class.

**The Bill Class**

Construct a class for use as a bill that contains data related to an outstanding or paid bill of a specific amount of money.  The class should contain the amount of the bill as a Money object, the due date of the bill (a Date object) and a Date object to track the date paid (null if not yet paid).  Check to be sure that when paying the bill, the date isn’t past the due date – if so, either quit (System.exit()) with an error message or throw an exception. (Note: either build a Date class(more simple and direct) or use Java’s Date class.)

**Data as Instance Variables:**

* amount – a Money object
* dueDate – a Date object
* paidDate – a Date object, null if not yet paid
* originator – a String containing the company or institution who issued the bill

**Methods in the Public Interface:**

* Bill(Money amount, Date dueDate, String originator)
  + constructor (null for paidDate)
* Bill(Bill toCopy)
  + copy constructor
* Date getDueDate() – return the dueDate\*
* String getOriginator() – return the originator
* boolean isPaid() – return true if bill has been paid
* void setPaid(Date onDay) – make the bill paid, save the date\*
* void setUnpaid() – make the bill unpaid
* void setDueDate(Date nextDate) – set the due date of the bill\*
* void setAmount(Money amount) – change the amount\*
* Money getAmount() – provide the Money object for the bill that is the amount\*
* void setOriginator(String originator)\* - change the originator
* String toString() – build a string that reports the amount, when its due, to whom, whether paid, and if paid, the date paid.
* boolean equals(Bill toCompare) – determine if the two bills are identical

\*Note: The items starred above are to be considered critical sections in your code where it could be very easy to violate the *class invariants* that we’ve defined above.  Proceed carefully in these functions and be sure that, for example, calling setPaid() with a date that is after the due date will fail, or that calling setDueDate() with a date that is before the paid date will also fail.

**The Driver Class**

See the driver, linked below, as an example **starting point** for your final driver, which should test your Money, Bill and Date classes.

[BillMoneyDateDriver.javaPreview the document](https://canvas.uw.edu/courses/1331881/files/58017726/download?wrap=1)

Let me emphasize again, this driver is only a starting point.  **Your driver should be much more complete in its testing.**  At a minimum, it should test every method of every class at least once.  Most methods need more than one test because there is more than one use case of interest.

**Hints and Notes**

* Start with the smallest class first (Date or Money) and then build the larger classes (Bill).
* First, use the sample driver with your classes — then extend the driver to test each function in Bill, Date and Money.
* Be sure to document your code with comments, highlighting the class invariants where they are relevant.
* Again, **make sure you comment all methods and the class with javadoc comments.**  If you have a driver that has code someone else has written (e.g., is based on a skeleton), you have to javadoc comment the methods someone else wrote also.
* Your code should not depend on absolute path specifications or any other environment-specific specifications.  The program should **not** ask for user input from the console.
* Watch out for privacy leaks in your getters, setters, and constructors.
* As was mentioned earlier, for this assignment, you still can deal with error conditions by either exiting the program or throwing an exception.  But realize that, generally speaking, hard exits are not the way you want to deal with error conditions, as they don't give you any options to handling the error except termination.  You are strongly recommended to start utilizing exceptions for handling errors, even though in this particular assignment you will not be graded down if you do not do so.  For later assignments, utilizing exceptions may be required.

**What to Submit**

Please turn in the source code files for your classes and driver.  Your classes **have to be named** Money, Date, and Bill and your driver named the same as the sample driver.

Please **submit your files inside a zip archive called ClassesAndPrivacy.zip.**  If you are not sure how to create a zip archive, do a Google search on "create zip archive windows" (swap out your computer's operating system, as needed).  Make sure you give yourself enough time before the due time to learn about and create the archive!

**About This Document**

Original by Rob Nash. Minor edits by Johnny Lin, Spring 2017-October 2019.

**Rubric**

Class and Privacy Leaks Rubric (1)

| Class and Privacy Leaks Rubric (1) | | |
| --- | --- | --- |
| **Criteria** | **Ratings** | **Pts** |
| This criterion is linked to a Learning Outcome Money Class |  | 6.0 pts |
| This criterion is linked to a Learning Outcome Bill Class |  | 6.0 pts |
| This criterion is linked to a Learning Outcome Date Class |  | 6.0 pts |
| This criterion is linked to a Learning Outcome Comments in Code |  | 8.0 pts |
| This criterion is linked to a Learning Outcome Accomplishes Goals |  | 4.0 pts |
| This criterion is linked to a Learning Outcome Extended Driver |  | 6.0 pts |
| This criterion is linked to a Learning Outcome Code Executes |  | 2.0 pts |
| Total Points: 38.0 | | | |

**Submission**

Submitted!

Oct 24 at 11:58pm

[Submission Details](https://canvas.uw.edu/courses/1331881/assignments/4923142/submissions/3729452)

[Download ClassesAndPrivacyLeaks-3.zip](https://canvas.uw.edu/courses/1331881/assignments/4923142/submissions/3729452?download=59261897)

Grade: 34 (38 pts possible)

Graded Anonymously: no

[View Rubric Evaluation](https://canvas.uw.edu/courses/1331881/assignments/4923142/submissions/3729452#rubric)

**Comments:**

Your Money class is missing an add(int, int) overload. getMoney() returns an incorrect cent value. The equals() in your Bill class compares Object with == rather than .equals().