

APS 105 – Computer Fundamentals

Lab #7: Objects and Classes

Fall 1999

(To be completed *before* your lab period, week of October 25)

Objective: To learn the fundamentals of object-oriented programming.

1 A Simple Object

Using the program skeleton provided below, you are to implement a class which represents a calendar year. All that you need to do is to fill in the method stubs which are provided since no additional variables or methods are needed. Once you have completed the Year class, create a Demo class with a main method. Your main should prompt the user to enter an integer value and use it to instantiate a new Year object. Then you will use the `isLeapYear` and `century` methods to display some properties of the object. Print out appropriate messages like: 1997 is not a Leap year . and 1997 is in century 20.

A year is a leap year if it is before 1582 and it is evenly divisible by 4 (eg. 4, 8, 1000, 1580). After 1582, years which are evenly divisible by 4 are leap years, except for the centuries (eg. 1800, 1900) which are not, unless the century is evenly divisible by 400 (eg. 2000) which is a leap year. It should be an error for your class to represent years before the year 0. Years 0 to 99 are considered as the first century, 100 to 199 as the second century, and so on.

```
class Year
{
    private int year;

    Year (int y) {
        // PUT CODE HERE
    }

    public boolean isLeapYear () {
        // PUT CODE HERE
    }

    public int century () {
        // PUT CODE HERE
    }
} // end of class Year
```

1

2 Bank Account Management – Revisited

For this lab, you are to modify your solution to lab #4 to use objects. Your program will simulate a bank with a limited number of accounts.

2.1 Bank Class

The bank itself will be represented by a class containing the method `main` as well as a class variable to hold an array of account objects. Since your bank only has one employee, who can only manage a fixed number of accounts, you should use an account array of length 100. Not all of the accounts will be created at once (see below). Each bank account will have an account number representing its location in the bank array.

The `main` method will do like before, prompting the user for a transaction to perform (deposit, withdrawal, etc). For each transaction, a proper account number must also be entered. You should add three more transaction types:

1. **Open a new account** and ask whether it is to be a savings or checking account. Once opened, the bank guarantees that the interest rate on your account will never change.
2. **Set interest rates** to be used for any *newly* opened accounts — you must ask for two rates, one for savings and one for checking.
3. **Transfer** to move funds from one account to another.
4. **Balance Inquiry** to check the funds in an account.

Other Changes

To increase the security at the bank, all transactions will now require the entry of a password, which will now be a `String` value. When interest is applied, it should be applied to all open accounts. When asking the user for a dollar value, ask for dollars and cents separately as integers.

2.2 Account Class

Each account must keep track of its account number, interest rate, the current balance (in cents), and the minimum balance since interest was last applied. As well, they should have a character variable containing either 'S' indicating this is a savings account or 'C' indicating a checking account. You should select appropriate visibility modifiers for the variables contained in the account class, and you may add additional variables if you desire. To properly apply the principles of object oriented programming, each account will supply at least the following methods: `deposit`, `withdraw`, `transferFrom`, `getBalance`, and `payInterest`. You must decide on the parameters required for these methods. The constructor for this class will have as parameters any values necessary for initializing the instance variables of the class.

2.3 Optional Part

1. Investigate the use of the API's `Vector` class which permits dynamically sized arrays to be used. This would remove the 100 account limit you currently have.
2. Improve the program by adding a `Customer` class. Modify the `Account` class to store an index into an array of customers. Each customer object should also have an array of references to the accounts that they 'own'. You will need to add an option to add customers to the customer array.
3. Create an array of `Teller` objects, each with their own password and name. Have the system require a teller to log on prior to performing any transactions.

2