The due date for this programming assignment is Friday, November 14, 2014.

Acknowledgement: This assignment was adapted from one used by Professor Arland Richmond, who has taught CIT-239 many times.

Credit Card Verification Application

This programming assignment involves writing a Java application which allows the user to CREATE and VERIFY credit card accounts. The user types textual commands which operate as shown in the examples below. (This sample list is provided to aid your understanding. It is *not* an exhaustive list of test cases for the program.)

```
create v
```

In this example, the user is requesting to CREATE a new "Visa" credit card account. The program should create a new account and assign it a credit limit as described later in this document.

```
create AE
```

In this example, the user is requesting to CREATE a new "American Express" credit card account. The program should create a new account and assign it a credit limit as described later in this document.

```
verify 5439123412345678 250.00
```

In this example, the user is requesting to process a **purchase** of $250.00.
First, the software must VERIFY that account 5439123412345678 has at least $250.00 of available credit. If the account *does* have at least $250.00 of available credit, then the available credit is reduced by $250.00 and the program outputs a message indicating that the transaction was approved. This type of transaction is called a "debit".

```
verify 5439123412345678 -125.00
```

In this example, the user is requesting to process a **return** of $125.00.
First, the software must VERIFY that adding $125.00 to the available credit of account 5439123412345678 will not raise the available credit above the Maximum Credit Limit for this account. If the transaction is allowed, then the available credit is increased by $125.00, and the program outputs a message indicating that the transaction was approved. This type of transaction is called a "credit".

Credit Card Account Numbers and Issuer Codes

Each credit card account number should be represented in the program by a String object. The text we place in this String consists of 16 decimal digits: The first digit is assigned according to the table on the next page. The remaining 15 digits should be generated randomly. When initially creating each account, the maximum credit limit is based on the last digit of the credit card account number. That is, if the last digit is in the range of 0-4, then a maximum credit limit of $1000.00 should be assigned. Otherwise, the maximum credit limit of $500.00 should be assigned.

The table below lists several credit card issuers, the first digit of their account numbers and the symbol we will use on the program command-line to select that particular issuer.

Card Issuer	Symbol	First Digit
American Express	AE	3
Visa	V	4
MasterCard	MC	5
Discover	DIS	6
Diners Club	DINE	7

Data File Format

The credit card data should be stored in a text file. Each line of text in the file represents one credit card account, and the format of the data should be as follows:

account_Number **|** *current_available_credit* **|** *maximum_credit_limit*

The field delimiter character should be the vertical bar (“ **|** ”) character. (This character is sometimes referred to as the “pipe” character.) This is a good choice because it will never occur in the data fields, and also makes the data file easy to browse manually with a text editor.

Processing

The user enters text commands from the keyboard. The user transaction either creates a new data record (in memory) or modifies an existing data record. After the transaction has been completed, then all data must be written out to the data file before the program exits. This requirement can pose a challenge during the debug process: if the program writes bad data to the file during one test, then that could mean the data file becomes unusable for the next test. One simple way to deal with this is for the tester (you) to manually copy the data file to a different file name or different folder before executing each test. The following section describes an automated approach to handle this issue.

Programmatic Data File Management (optional, EXTRA CREDIT)

The approach described in this section is not required, but optional, for EXTRA CREDIT. This approach gives the programmer more control of the data file during the debug process. Specifically, if one debug session produces a “garbage” output file, then it is easy for the programmer to throw that particular file away, and revert to an earlier, hopefully correct, version of the data. The disadvantage of this approach is that it leaves lots of old data files lying around, which the programmer must eventually delete, by using an appropriate manual operating system command.

The program, as described in this document, saves the credit card data in a file named "**dataFile.txt**". If a pre-existing file with this same name already exists, then the program renames it to a filename which contains the current date and time, in the following format:

dataFile_YYYYMMDD_HHMMSS.txt

where YYYYMMDD is the current year, numeric month and day, and

HHMMSS is the current time-of-day in 24-hour format.

The "timestamped" file becomes the *input* file, and a **new** file named "**dateFile.txt**" becomes the *output* file. The result of this approach is that, after each user command, the *most recent* program output is always contained in "**dataFile.txt**", but older versions are preserved. Maintenance (cleanup) of old data files is NOT automatically done, and if your program crashes before creating the output file, then you will need to manually rename the old data file to the default "**dataFile.txt**" name.

Transaction Types

There are two user commands which must be handled (CREATE and VERIFY):

- A "**`create`**" command creates a new account, and includes its data in the output file.
- A "**`verify`**" command authorizes a "purchase" or a "return" transaction. For simplicity, any VERIFY command which specifies a positive number should be considered a purchase (also known as a "debit"), and any VERIFY command which specifies a negative number should be considered a return (also known as a "credit"). A purchase (debit) reduces the available credit, while a return increases the available credit.

Transaction Acceptance or Rejection

There are several possible scenarios which are possible when the user is running the program. The program outputs a message to the screen, indicating which of these results has actually occurred:

- AUTHORIZATION GRANTED -- if the account exists, and the VERIFY transaction does not cause the account to exceed the remaining credit line, then this should be reported to the user, along with the remaining available credit.
- AUTHORIZATION DENIED -- if the account exists, but the requested purchase would exceed the remaining available credit, then the purchase should be rejected. Also, a return transaction should not be allowed to increase the available credit above the maximum.
- ACCOUNT NOT ON FILE – if the account is not in the data file, then obviously, no purchase or return can be processed.

Design Suggestion: Two Classes

The discussion below describes how I implemented this solution. Please consider this a *suggestion* to help you think about the problem in an object oriented way, but ***not*** an absolute requirement. (If you have a different design in mind, feel free to implement that.) The two classes in my implementation are called "**Credit**" and "**CreditAccount**".

The "Credit" Class

The "Credit" class should be the one which includes the "main" method. It reads the input parameters from the command line and processes the two major user commands, CREATE and VERIFY. The "main" method creates a local variable which is an array of "CreditAccount" objects.

In addition to the "main" methods, the Credit class includes these methods:

- **loadData**: loads data from the input file.
- **writeData**: writes data to the output file.
- **createAccount**: creating a new account.
- **authorizeTransaction**: verifies a transaction and modifies the "current available credit" amount.

These major methods call other "utility" methods:

- **findAccount**: searches the data (in memory) for the account number specified by the user.
- **renameWithTimestamp**: renames the existing data file (dataFile.txt) with a timestamp in the file name.

The "CreditAccount" Class

The second class, called "CreditAccount" is where most of the detailed processing takes place. Some of its data members are described in the table below.

Private Data Member	Description
`private String accountNum;`	Credit Card account number
`private double available;`	Current available credit
`private double maxLimit;`	Maximum credit for the account. (At all times, the `available` amount must be less than or equal to the `maxLimit` amount.)
`private String issuerSymbol;`	Credit Card Issuer Symbol
`private boolean accountValid;`	Flag to identify a "valid" vs. "invalid" account.
`private static boolean verboseMode;`	Debug aid (default value `== false`)

I also chose to use define several constant values associated with the "CreditAccount" class:

Credit Card issuer Symbols (constants)	Description
`public static final String ISSUER_AMER_EXPRESS = "AE";`	American Express
`public static final String ISSUER_VISA = "V";`	Visa
`public static final String ISSUER_MASTER_CARD = "MC";`	MasterCard
`public static final String ISSUER_DISCOVER = "DIS";`	Discover
`public static final String ISSUER_DINERS_CLUB = "DINE";`	Diners Club

Credit Card issuer Codes – first digit of account number (constants)	Description
`public static final int ISSUER_CODE_AE = 3;`	American Express
`public static final int ISSUER_CODE_V = 4;`	Visa
`public static final int ISSUER_CODE_MC = 5;`	MasterCard
`public static final int ISSUER_CODE_DIS = 6;`	Discover
`public static final int ISSUER_CODE_DINE = 7;`	Diners Club

In addition to the normal "getters" and "setters", some of the methods I defined for the CreditAccount class are described below.

Method Name, parameters	Description
`public CreditAccount(String issuerSymbol)`	Constructor for creating a new account
`public CreditAccount(String recordText, int recordLength)`	Constructor for loading records from the data file
`public String assembleRecordText()`	Create a string in the format suitable for writing to the data file.
`public static int getIssuerCode(String issuerSymbol)`	Check issuer symbol and return corresponding issuer code
`public static String getIssuerSymbol(int issuerCode)`	Get Issuer Symbol from issuer code.

Project Deliverables:

The project should be submitted by **Moodle**. Submit ***only*** the "**`.java`**" file(s).
I will need to be able to compile the code, and test it myself.
Do NOT include the NetBeans project folders, or any binary files.

Grading Criteria

The following guidelines should be helpful, but feel free to reply with further questions, if you have any.

The source code should be:

1. Logically correct -- that is, the program should exhibit the behavior specified in this assignment document.
2. Clearly written, with appropriate comments so I can understand what the code is trying to do. At a minimum, each method should be preceded by a brief comment explaining what the method does.
3. Use descriptive variable and method names.
4. Do NOT place all the code in the "main" method. Break the code into logically appropriate Java methods.

It is difficult to say with precision, exactly what will constitute an "A" vs. a "B", etc. If you follow the guidelines above, then your grade will be good.