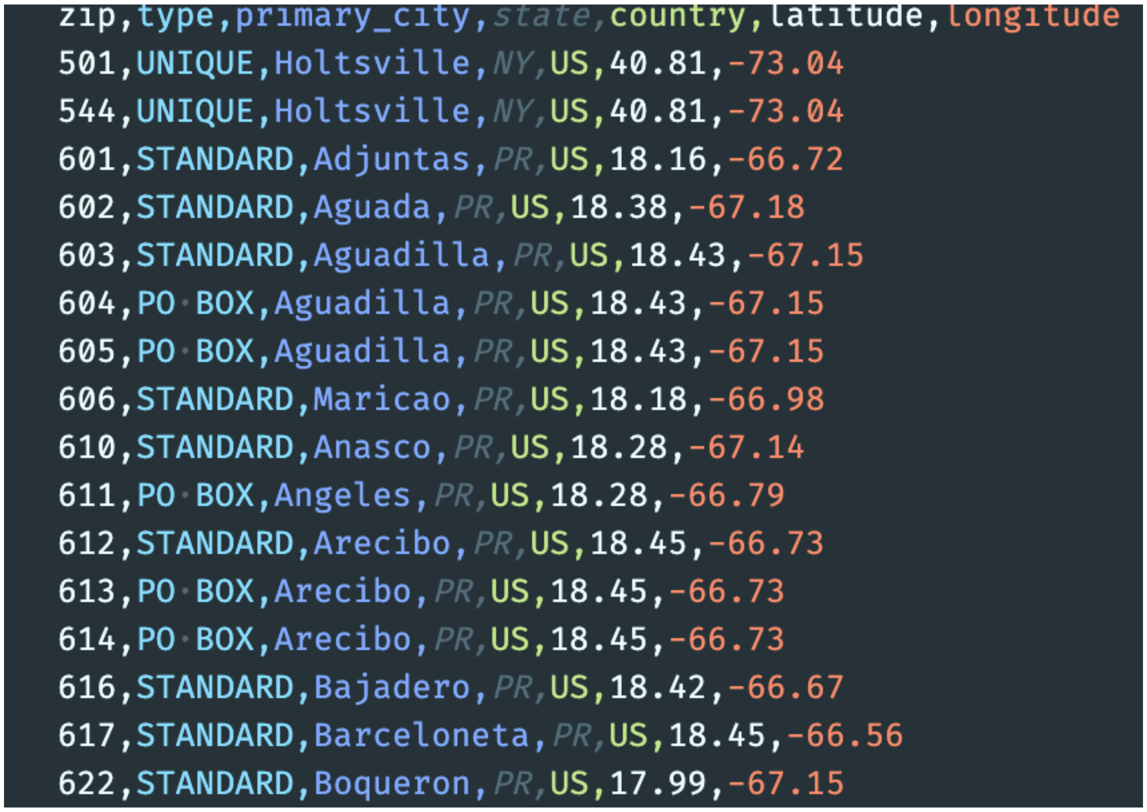
CSCI165 Computer Science II
Object Oriented Programming: Composition
Lab Assignment

This is part two of module five's project. All privacy leaks must be prevented!

In this lab assignment we will be extending the composition of the Customer, Address work we completed in the discussion. We will now use the composition pattern to include a local list of city, state and zip data inside of the Address class. This data will need to be handled a bit differently though. The details of this implementation follow.

Task One: In this task we will accomplish the automatic generation of the city and state from a zip code. You have been provided with a CSV file that contains records of zip codes and the cities and states that they reference. Here is a snapshot of the file.

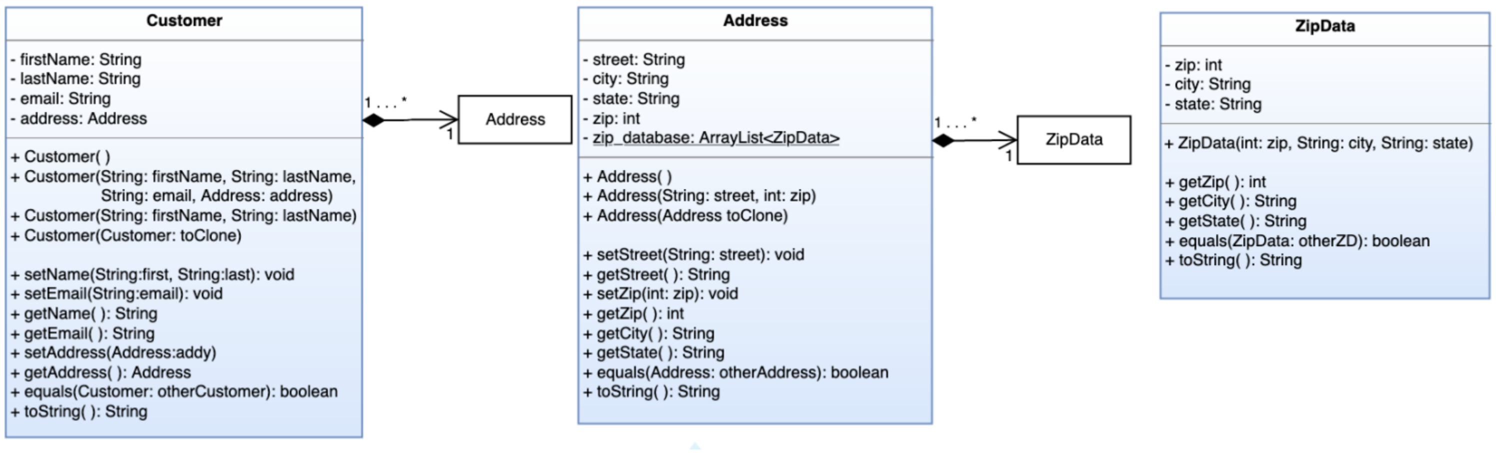


```
zip,type,primary_city,state,country,latitude,longitude
501,UNIQUE,Holtsville,NY,US,40.81,-73.04
544,UNIQUE,Holtsville,NY,US,40.81,-73.04
601,STANDARD,Adjuntas,PR,US,18.16,-66.72
602,STANDARD,Aguada,PR,US,18.38,-67.18
603,STANDARD,Aguadilla,PR,US,18.43,-67.15
604,PO BOX,Aguadilla,PR,US,18.43,-67.15
605,PO BOX,Aguadilla,PR,US,18.43,-67.15
606,STANDARD,Maricao,PR,US,18.18,-66.98
610,STANDARD,Anasco,PR,US,18.28,-67.14
611,PO BOX,Angeles,PR,US,18.28,-66.79
612,STANDARD,Arecibo,PR,US,18.45,-66.73
613,PO BOX,Arecibo,PR,US,18.45,-66.73
614,PO BOX,Arecibo,PR,US,18.45,-66.73
616,STANDARD,Bajadero,PR,US,18.42,-66.67
617,STANDARD,Barceloneta,PR,US,18.45,-66.56
622,STANDARD,Boqueron,PR,US,17.99,-67.15
```

In an attempt to enforce that an Address' city, state and zip are always in sync we will use a local database. Notice that the constructor for the Address class only accepts the zip, but there are variables for city and state. Our task now is to write code to allow for the city and state to be automatically chosen from the database when the zip code is provided. Because we have the privacy leaks handled from earlier, we are well on our way to enforcing the relationship between city, state and zip.

Here is the UML diagram for this task. Notice the composition pattern demonstrated by the inclusion of the class ZipData as the basis for a local database inside the Address class.

Address **has a** collection of ZipData objects


Customer
- firstName: String
- lastName: String
- email: String
- address: Address
+ Customer( )
+ Customer(String: firstName, String: lastName, String: email, Address: address)
+ Customer(String: firstName, String: lastName)
+ Customer(Customer: toClone)
+ setName(String:first, String:last): void
+ setEmail(String:email): void
+ getName( ): String
+ getEmail( ): String
+ setAddress(Address:addy)
+ getAddress( ): Address
+ equals(Customer: otherCustomer): boolean
+ toString( ): String
1 . . . *
1
Address
Address
- street: String
- city: String
- state: String
- zip: int
- zip_database: ArrayList<ZipData>
+ Address( )
+ Address(String: street, int: zip)
+ Address(Address toClone)
+ setStreet(String: street): void
+ getStreet( ): String
+ setZip(int: zip): void
+ getZip( ): int
+ getCity( ): String
+ getState( ): String
+ equals(Address: otherAddress): boolean
+ toString( ): String
1 . . . *
1
ZipData
ZipData
- zip: int
- city: String
- state: String
+ ZipData(int: zip, String: city, String: state)
+ getZip( ): int
+ getCity( ): String
+ getState( ): String
+ equals(ZipData: otherZD): boolean
+ toString( ): String

ZipData Class: This class will encapsulate relevant data from the provided CSV file. There are 7 fields in the file, but we are only interested in the ZIP, City and State. This class file will encapsulate those data for us and allow us the ability to build a list of ZipData objects for easy retrieval. Define the ZipData class according to the UML diagram above. Notice that all setting will happen via a call to the constructor. Do not include sets for City, State or Zip.

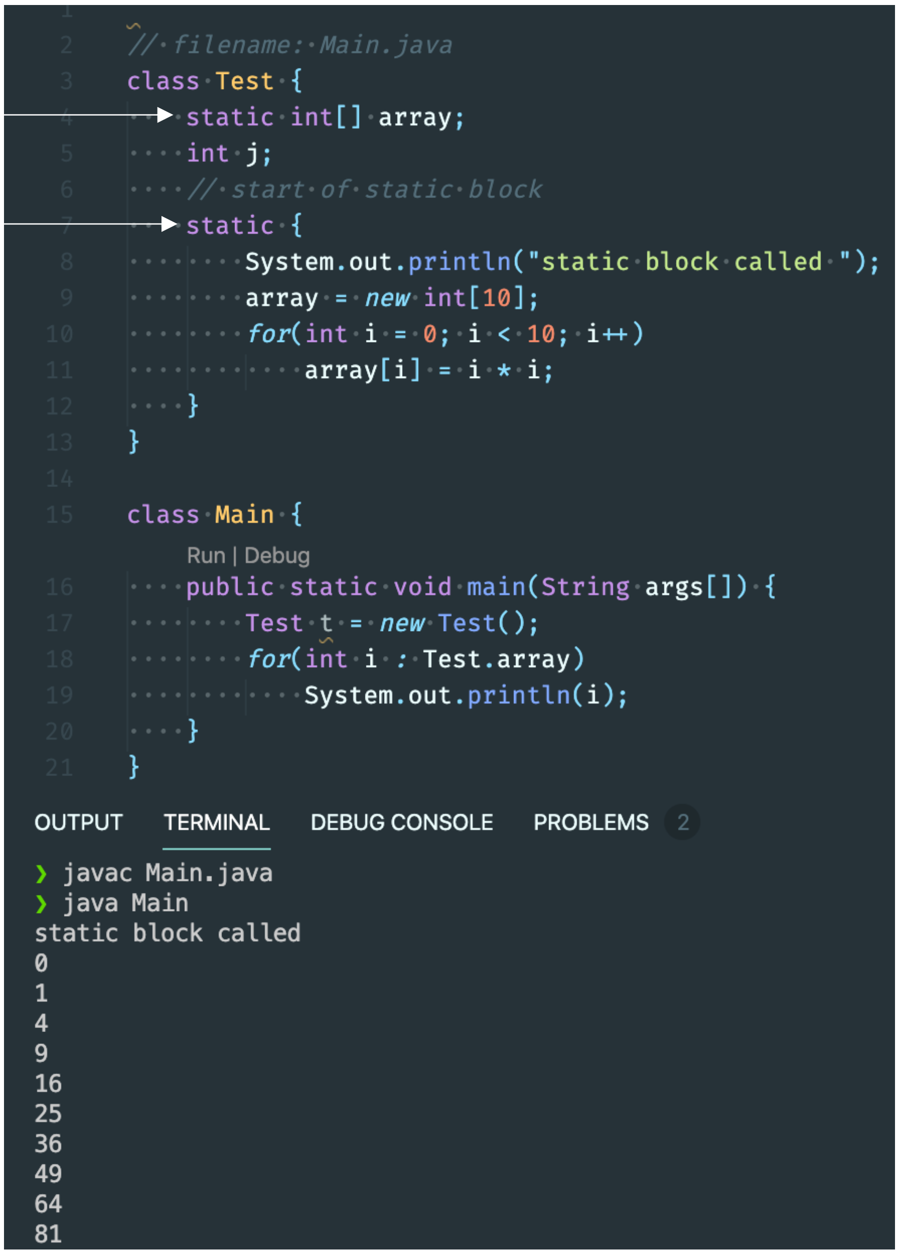
Address Class Additions: We need to come up with a way for us to store the zip code database in memory, so that it can be quickly accessed. Having to continually read the file every time we need to grab a city and state **will be very slow,** so we need a way to keep this data in RAM. This will obviate the need to continually open and read the file.

- In the Address class add a ***public static ArrayList*** that will hold ZipData objects. It is important that this feature is static so that ***it only exists once*** no matter how many instances of the Address class exist in memory**.** If we made this a ***non-static instance feature***, then every instance of the Address class would get its own copy of this data. That would be redundant and wasteful of system resources.
- Define the method ***private static void fillList.*** This method will
    - Open ***zip_code_database.csv***
    - With each line in the file:
        - Parse the fields and extract the zip, city and state
        - Create a ZipData instance with this data
        - Add the ZipData instance to the ArrayList

- How do we have this code run automatically when the class is loaded? Java supports a special block, called a ***static block*** (also called a static clause) which can be used for static initializations of a class' static features. The code inside a static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

Notice the static array on line 4 of class Test.

Notice the static block of code beginning on line 7. This code is outside of any method, which is usually not allowed. Making the block of code static allows it to run **when the class is loaded.** This will only happen once. You can put whatever you like in here **BUT** a static block can **only** reference static features. If I tried to access instance variable **j** in this block the compiler would complain.



```
 1
 2  // filename: Main.java
 3  class Test {
 4      static int[] array;
 5      int j;
 6      // start of static block
 7      static {
 8          System.out.println("static block called ");
 9          array = new int[10];
10          for(int i = 0; i < 10; i++)
11              array[i] = i * i;
12      }
13  }
14
15  class Main {
        Run | Debug
16      public static void main(String args[]) {
17          Test t = new Test();
18          for(int i : Test.array)
19              System.out.println(i);
20      }
21  }
```

```
❯ javac Main.java
❯ java Main
static block called
0
1
4
9
16
25
36
49
64
81
```

- In your Address class add a static block of code (above the constructors, but below the variables) to call the ***fillList*** method.

- Write a JUnit Test to ensure that the list is filled properly. Test the following entries from the ArrayList. Use the **get(int: index)**

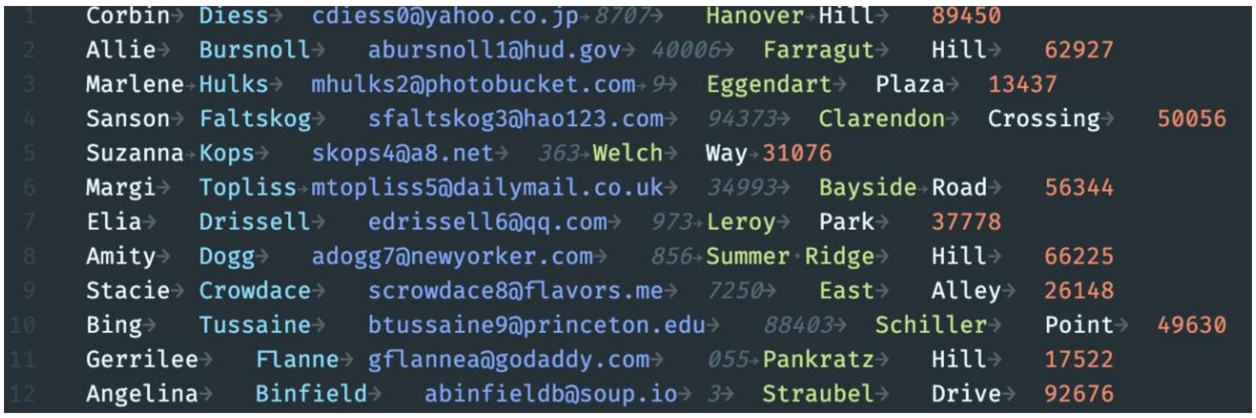
  - Index 0 => [zip: 501, city: Holstville, state: NY]
  - Index 10656 => [zip: 25361, city: Charleston, state: WV]
  - Index 21314 => [zip: 48604, city: Saginaw, state: MI]
  - Index 42631 => [zip: 99949, city: Ketchikan, state: AK]
  - I triple checked these values but do yourself a favor and check for yourself. Don't rely on me to verify your results.

- Write the method **public static ZipData locate(int zip)** This method will accept a zip code and return the appropriate ZipData instance from the zip database. Write a Unit Test to prove this works. Include 5 individual assertions.

Task Two: You have been provided with a file (***customers.txt***) containing Customer information. This file contains the following data

- First name, last name
- Email
- Street Address: {number, street, street type}
- Zip Code

Here is a snapshot of this file. These fields are tab delimited, there are regular spaces in the street names. If you set the Scanner up to use tabs **"\t"** you will be able to easily parse the individual fields

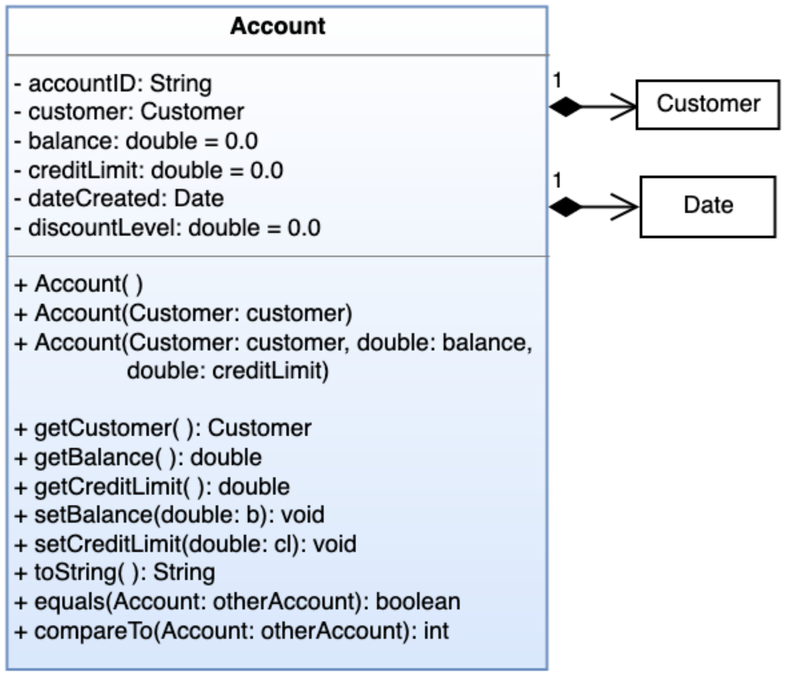


```
Corbin	Diess	cdiess0@yahoo.co.jp 8707	Hanover Hill	89450
Allie	Bursnoll	abursnoll1@hud.gov	40006	Farragut	Hill	62927
Marlene Hulks	mhulks2@photobucket.com 9	Eggendart	Plaza	13437
Sanson	Faltskog	sfaltskog3@hao123.com	94373	Clarendon	Crossing	50056
Suzanna Kops	skops4@a8.net	363 Welch	Way 31076
Margi	Topliss mtopliss5@dailymail.co.uk	34993	Bayside Road	56344
Elia	Drissell	edrissell6@qq.com	973 Leroy	Park	37778
Amity	Dogg	adogg7@newyorker.com	856 Summer Ridge	Hill	66225
Stacie	Crowdace	scrowdace8@flavors.me	7250	East	Alley	26148
Bing	Tussaine	btussaine9@princeton.edu	88403	Schiller	Point	49630
Gerrilee	Flanne	gflannea@godaddy.com	055 Pankratz	Hill	17522
Angelina	Binfield	abinfieldb@soup.io	3	Straubel	Drive	92676
```

Create a Driver class that accomplishes the following

- Defines an array of 1000 Customer objects . . . that is the size of the file.
- Reads the data from ***customers.txt***
- With each line of the file:
  - Create an Address instance. Auto-populate the city and state by grabbing the zip code and passing it into the ***locate*** method. Pull the city and state out of the returned ZipData instance and assign them to the Address.
  - Using the Address instance created above, build a Customer instance using the remaining data from the line.
  - Add this instance to the array
- When the array has been filled iterate through the it and print the toString for each instance. Pause at each print and require a key press to move to the next instance. You can accomplish this easily with a Scanner and a junk variable. Include some way for me to be able to stop this loop when I have met my fill of viewing Customer records.

Task Three: Extending the concept of object-oriented composition let's introduce the Account class. The Account class will contain a Customer instance and a Date instance. The Customer will be the **owner** of the Account and the Date will be the **date the account was created.** Notice how the complexity of the applications composition is growing. We now have an Account that requires a Customer. The Customer has an Address which requires ZipData. These levels of composition can extend indefinitely. Feel free to use the Date class from previous work. You can find it in the repository. Make sure that all of your source files are in the same directory.


Account
- accountID: String
- customer: Customer
- balance: double = 0.0
- creditLimit: double = 0.0
- dateCreated: Date
- discountLevel: double = 0.0
+ Account( )
+ Account(Customer: customer)
+ Account(Customer: customer, double: balance, double: creditLimit)
+ getCustomer( ): Customer
+ getBalance( ): double
+ getCreditLimit( ): double
+ setBalance(double: b): void
+ setCreditLimit(double: cl): void
+ toString( ): String
+ equals(Account: otherAccount): boolean
+ compareTo(Account: otherAccount): int
1
Customer
1
Date

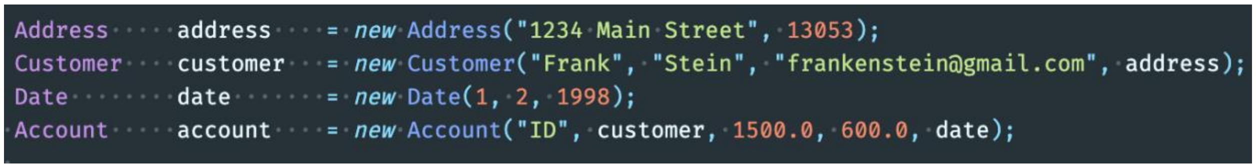
toString and equals: Hopefully you understand what needs to be done here. Call the toString and equals methods of the inner objects and write new code to handle the unique account fields.

compareTo: Build this method around the ***balance*** field.

When you are building an Account instance you will need to fulfill the following dependencies . . . ie. You will need instances of the following classes

- Date
- Customer
- Address

Notice in the screen shot below, that Customer and Date must be built before building an Account and an Address has to be built before a Customer can be built. Implicit in the building of the Address is the population of the city and state with the passed in zip code.



```
Address    address   = new Address("1234 Main Street", 13053);
Customer   customer  = new Customer("Frank", "Stein", "frankenstein@gmail.com", address);
Date       date      = new Date(1, 2, 1998);
Account    account   = new Account("ID", customer, 1500.0, 600.0, date);
```

Driver: Using the same Driver as before, add the following code

- Create an array of type Account with a size of 1000. You will be creating an account for each of the 1000 customers
- For each Customer create a random date within the last 20 years to serve as the account created date. You can accomplish this with three random numbers
  - https://www.geeksforgeeks.org/generating-random-numbers-in-java/
  - One for the month: 1 – 12
  - One for the day: 1 – 28 (make it easy on yourself and don't go above 28)
  - One for the year: 1 – 20 (then you can subtract this number from 2020 to get the actual year)
  - Take these values and create a Date instance.
- The Account ID should be created in the following fashion. This would be a good opportunity to create a private helper method in the Account class.
  - Customer name with vowels removed, all upper case + current date with slashes removed
  - **Example:** Ken Whitener 2/22/2019
  - **Account ID:** KNWHTNR2222019
- The balance should be set to a random number and the credit limit should be set to 50% of the balance. Discount level can remain zero
- Add the account to the array.

- Repeat the iterative process of showing the Account toString and requiring a button press to show the next one.

Submission: Push Account, Customer, Address, ZipDate and Driver to your repo