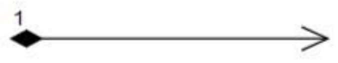
CSCI165 Computer Science II
Object Oriented Programming : Composition
Discussion

This is part one of module five's project.

In UML, Composition is represented by: 

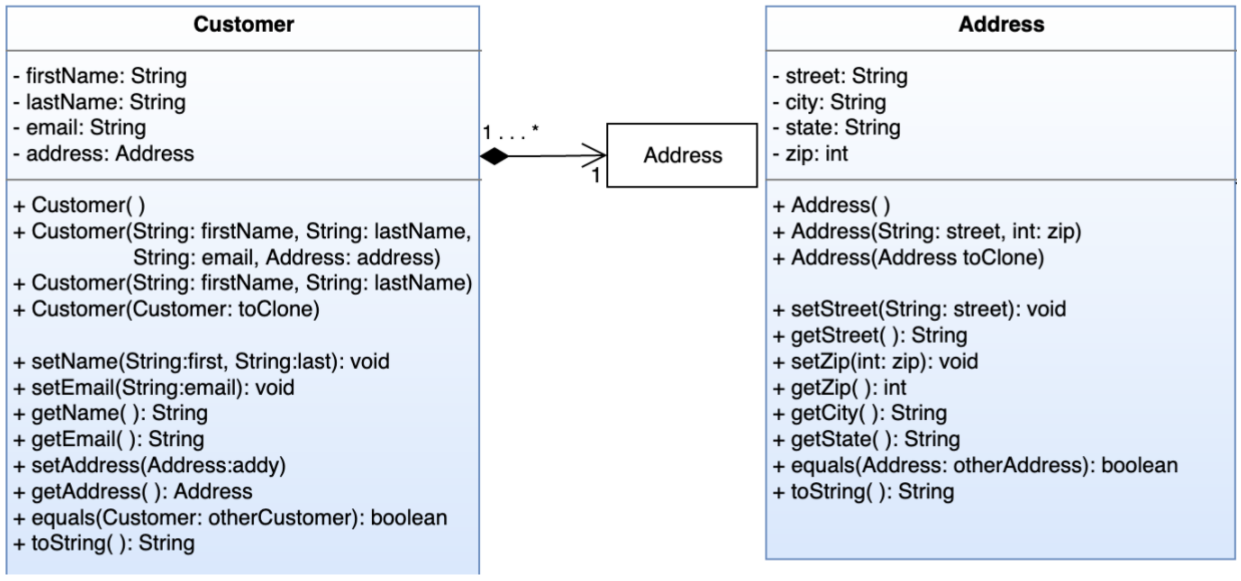
https://www.uml-diagrams.org/composition.html

Task One: Your task is to define the Customer and Address classes shown in in the following UML diagram. The composition pattern is illustrated in the design of the Customer class which contains an instance of the Address class as part of its features. Composition satisfies a special type of relationship colloquially referred to as "**has a**".

A Customer has an Address. If you can answer this question about two objects when you are designing an application, you can be relatively certain that you will use composition. A Customer has an address so add an instance of the Address class as a feature of the Customer class.

The multiplicity notation on the connector is stating the following

- A Customer has exactly one Address (1)
- An Address may be related to one or many Customers (1 . . . *)


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
+ 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

toString and equals: Another interesting aspect of composition is the creation of methods that invoke methods of the composition objects. There are two good examples of this concept here: toString and equals.

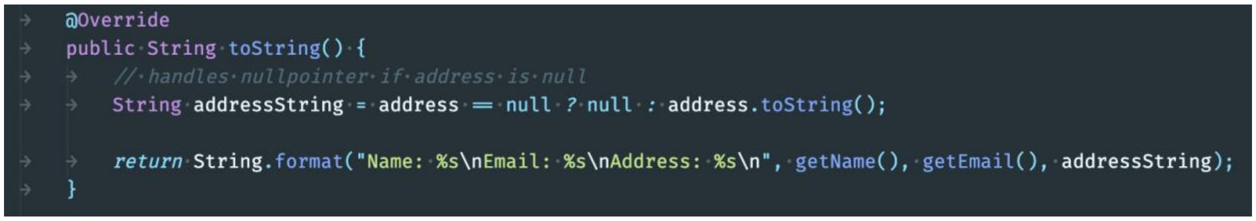
toString: The definition of the Customer's toString should include and invocationof the Address toSring. This is a prime example of code re-use. We have already defined the toString in Address, so when we define the toString in Customer we can simply call

this.address.toString()

equals: Two customers are only considered equal if they contain the same address. We already have the ability to determine address equals in **Address.equals** so we just need to include a call to this method in the Customer class.

this.address.equals(otherCustomer.address)

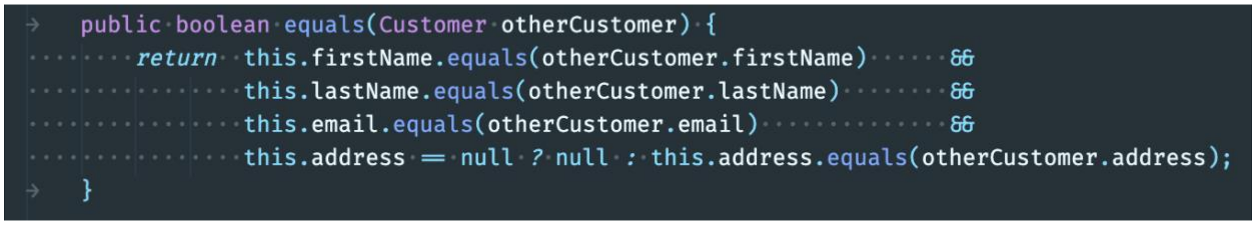
Here is an example of a Customer's toString method. Notice the use of the ternary operator to control inclusion of the Address. This is to prevent null pointer exceptions. If we happen to have a Customer instance that has a null Address, a null pointer exception would be thrown when referencing it's toString. Null pointer exceptions are of **major concern** when dealing with composition



```java
@Override
public String toString() {
    // handles nullpointer if address is null
    String addressString = address == null ? null : address.toString();

    return String.format("Name: %s\nEmail: %s\nAddress: %s\n", getName(), getEmail(), addressString);
}
```

Here is an example of an equals method in a class with composition. Notice all of the other calls to equals in the different classes. **This is an important concept!!**



```java
public boolean equals(Customer otherCustomer) {
       return  this.firstName.equals(otherCustomer.firstName)      &&
               this.lastName.equals(otherCustomer.lastName)        &&
               this.email.equals(otherCustomer.email)              &&
               this.address == null ? null : this.address.equals(otherCustomer.address);
}
```

This type of method chaining is crucial to understanding composition and will also be important in the implementation of inheritance. Refer to the readings for further clarification. Let's dissect this issue in the discussion. I want you all to fully understand it.

Privacy Leaks: I want you to purposefully leave a privacy leak in the Customer class. As demonstrated in the readings for this module, privacy can be leaked out of an encapsulated class when using composition. I want you to prove that this is true in task two.

Note: Don't worry about the lack of City and State in the constructors and set methods. We will be handling this in the lab, and the city and state will be populated automatically from a zip code database. **You don't have to worry about city and state at this point.** What we do want to do is ensure that our privacy is protected so that the relationship between the zip code and the city/state is not compromised.

Task Two: At this point you should have the Customer and Address classes built, with a purposeful privacy leak in the Customer class when accepting and returning an Address. Prove that this is true by demonstrating that you can manipulate the private Address variable. Exploit the privacy leak in the following features

1. public Customer(String, String, String, Address)
2. public Address getAddress()
3. public void setAddress(Address)

Create a Driver class with a main method. Using the approaches shown in the readings, demonstrate

- That you maintain access to a private Address variable via an Address instance passed to the constructor
- That you can maintain access to a private variable when calling setAddress and passing an Address instance
- That you can be granted access to a private variable when calling getAddress
- That you can directly manipulate the zip code of a private Address from outside the class.
- Write a JUnit test that proves that a private address is leaked from the Customer class.
- Create a post in the Discussion area describing how you accomplished this, along with a description of the issue and a screen shot of your terminal window showing the details of the privacy leak and exploit. Also post a screen shot of your JUnit test that proves the privacy leak. Talk to each other about it, help your classmates understand what is going on. **Do not post your actual code**

Task Three: Demonstrate that you can fix the privacy leak by plugging in the copy constructor in appropriate areas. Run the same Driver file without modification and analyze whether or not the issue has been fixed. Create a post describing what is going here along with a screen shot of your terminal window showing that the privacy leak was fixed. Show the update run of the JUnit test. **Do not post your actual code**

Submission: Push your source code files to the discussion folder in your repo.