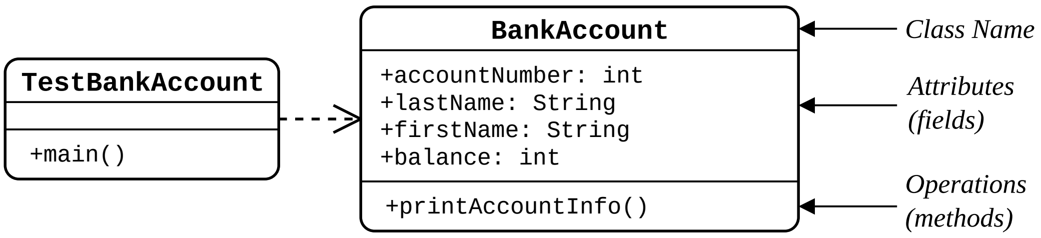
English name: ____________________

Java Project: `class BankAccount`

1. Understand the following UML class diagram representing two classes. Note the arrow with the dotted line represents a *dependency*: the `TestBankAccount` class is dependent on the `BankAccount` class. This is because the `TestBankAccount` class will create *instances* of objects of the type `BankAccount`, and call methods that are in the class.



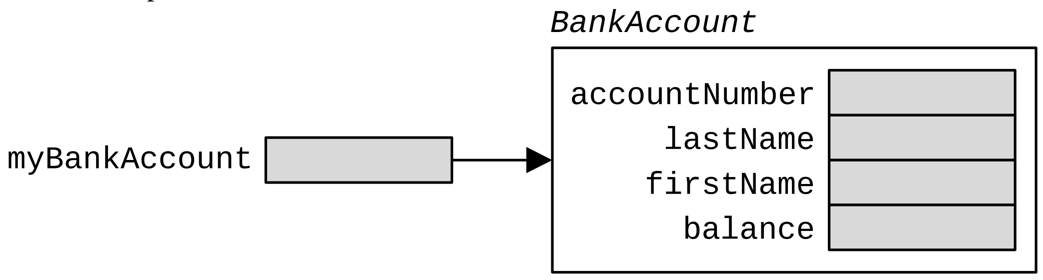
Begin with the following code for these two classes, `BankAccount` and `TestBankAccount` defined together in one package. Read through the code and try to understand what it does. You will soon be modifying the code, so change the second author to your name.

```
/**
 * This class defines BankAccount objects
 *
 * @author christophernielsen
 * @author 你的中文姓名 Your English Name
 *
 */
public class BankAccount {

   // Instance Fields
   public int accountNumber;
   public String lastName;
   public String firstName;
   public int balance;

   public void printAccountInfo() {
      System.out.println("Account Number: " + accountNumber);
      System.out.println("Owner:          " + firstName + " " + lastName);
      System.out.println("Balance:        " + balance);
   }
}
```

```
/**
 * The purpose of this class is to test the functionality
 * of the class BankAccount.
 *
 * @author christophernielsen
 * @author 你的中文姓名 Your English Name
 *
 */
public class TestBankAccount {

   public static void main(String[] args) {
      BankAccount myBankAccount = new BankAccount();
      BankAccount yourBankAccount = new BankAccount();

      // Add initial code here

      myBankAccount.printAccountInfo();
      System.out.println();
      yourBankAccount.printAccountInfo();
   }
}
```

In the BankAccount class, note the code below the comment `//Instance Fields` on line `10`. ***Fields*** are basically variables that at the top level of a class. In UML diagrams, they are referred to as ***attributes***.

Examine line `12` of `TestBankAccount`. The latter part, `new BankAccount()`, ***instantiates*** an object (creates an ***instance***) of type `BankAccount`. This means that memory is allocated to store a copy of the instance fields. The values that all the instance fields are set to is sometimes called the ***state*** of the object. The early part of line `12`, `BankAccount myBankAccount`, creates a place to store a ***reference*** to the object of type `BankAccount`, and labels it `myBankAccount`. You can think of this reference as a number that gives the address, or place, in memory where the object is stored. The diagram below may help you to understand. The shaded boxes represent memory locations. The labels (for example, `myBankAccount` and `balance`) for these memory locations are used while coding in Java, but they do not remain in the final, compiled program that the computer will run.



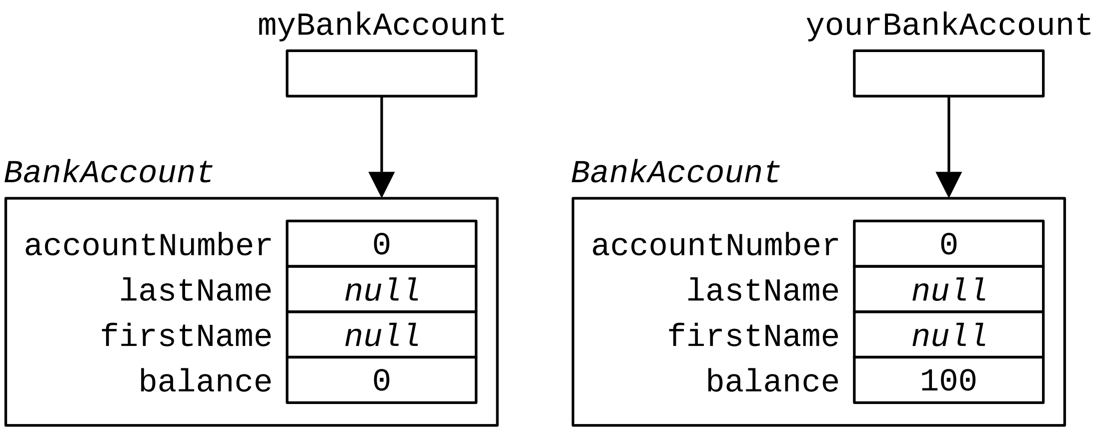
Run the code and verify the output is what you expected. Use the box below to take note of the output:

```
Account Number:
Owner:
Balance:

Account Number:
Owner:
Balance:
```

2. Add code below line 15 in `TestBankAccount` to change the value of the field named `balance` of `yourBankAccount` to equal `100`. Recall that when we accessed `static` class fields, for example to access the value for π from the `Math` class, we use the class name, a period (`.`), and then the name of the field, `PI`, like this: `Math.PI`. When we have *instances* of an object, we access the instance (non-`static`) fields of the object by using the instance variable name rather than the class name:

```
yourBankAccount.balance = 100;
```

Run the program and compare the new output values to the output you took note of in the box above. You should see that the value of `balance` in `myBankAccount` remained zero while the value of `balance` in `yourBankAccount` was updated to `100`. The values of the instance fields within an instance of an object are independent of each other.



3. As it is, the code can only store balance as an integer. In the final version of our `BankAccount` class, we want to store the currency to two decimal places. But we are not going to use a `float` or `double` primitive type to store the value. Instead, we will use the primitive type `int` to store the number of cents (or number of 分 if the currency is Chinese yuan), then divide by `100` whenever we want to get the balance in dollars (or yuan).
Also, it is often safer to disallow other methods to modify the fields of a class directly. Let's start by preventing access to the *field* `balance`. This is done by changing the access modifier of `balance` from `public` to `private`:

```
private int balance;
```

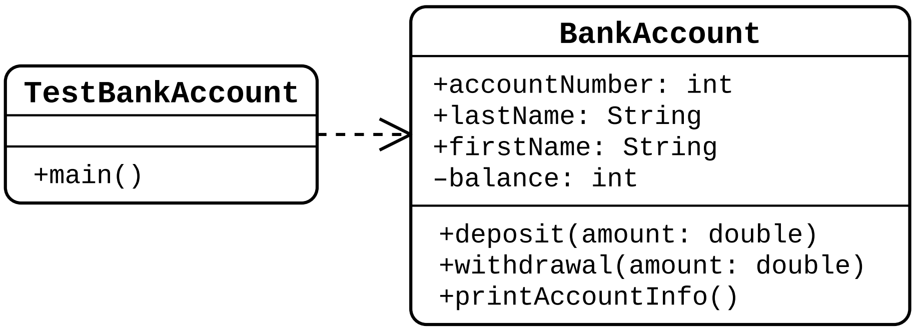
Try to run the code and see the result. The compiler should output an error that contains details similar to the following:

```
The field BankAccount.balance is not visible
```

So if we cannot access the `balance` *field* directly, how are we to change the balance when we withdraw or deposit currency? The solution is to write methods that are called to indirectly modify the instance fields. There are a number of advantages to this that you will learn over time. One example of how we might use this for a bank account object is perhaps you might wish to enforce a maximum withdraw amount. The `BankAccount` class itself would not be able to do this if other classes were able to modify the value of the `balance` field directly.
So let's write methods to perform the operations `deposit` and `withdraw` on an instance. These methods will take in the amount as type `double`, round to the nearest hundredth, and store as an integer number of cents (分).
Any method that will access or modify instance (non-`static`) variables cannot have the `static` modifier applied. Thus, our method declarations will be:

```
public void deposit(double amount)
public void withdraw(double amount)
```

Complete the definitions of these two methods within the `BankAccount` class, and change the `TestBankAccount` class to call the `deposit` method to deposit `100`, rather than attempt to access the balance field directly. Our updated UML diagram will now look like this:

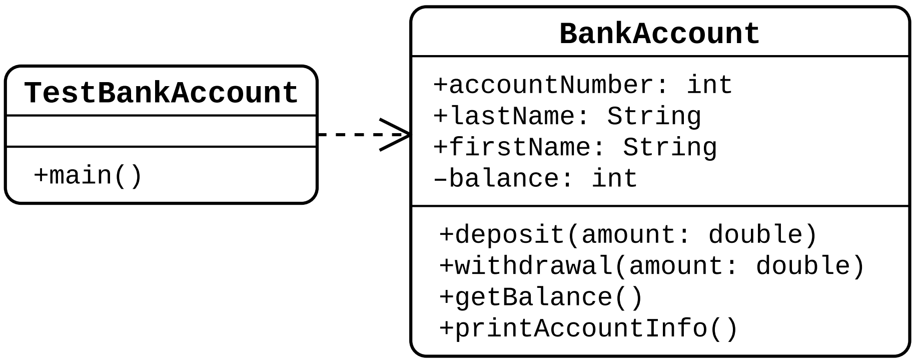


We've added the two new methods to the list of *operations*, and where the `balance` field previously had a plus sign (`+`) in front of it to show it was `public`, there is now a minus sign (`–`) in front of it to show it is `private`.

When you have completed the changes to the code, run it and note the output. If your code is error free and conforms to the specifications above, the output should look like this:

```
Account Number: 0
Owner:          null null
Balance:        0

Account Number: 0
Owner:          null null
Balance:        10000
```

4. We're storing the value as specified, but when we call the `printAccountInfo` method, we're printing the value stored in the `balance` field directly without converting it back to dollars (or yuan). It's time to fix that. However, thinking ahead, our banking software will probably also want to query the balance of the account, not only deposit, withdraw, and print. So first write a public method called `getBalance` that will return the dollar (yuan) balance of the account as type `double`. The updated UML diagram looks like this:



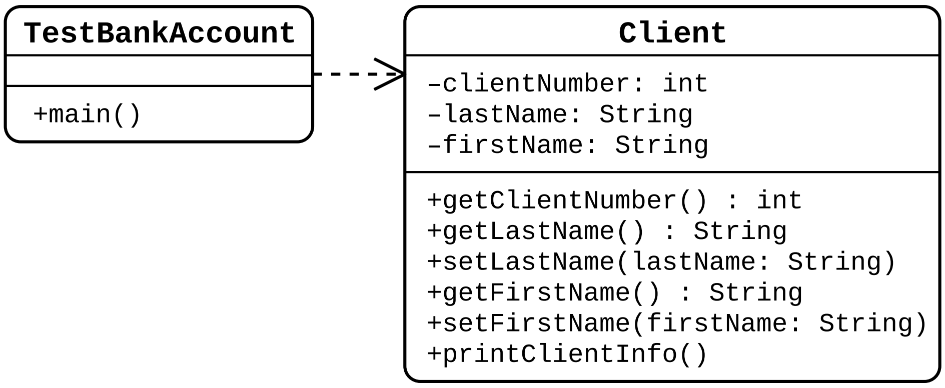
Once you have the method written, modify `printAccountInfo` to call the method `getBalance` to obtain the balance for printing. The need for this type of method that just returns the value of a field is so common in object-oriented programming that there is a special name. They are referred to as "*getters*". It is standard practice in Java to name getter methods, as we have done here, by appending "`get`" to the name of the field, changing the first letter of the field name to upper case to maintain camel case in the new method name.

Once you have completed `getBalance`, and updated `printAccountInfo`, run and verify that the balance is printed as you expect.

5. Now to update the account owner information. We will make not only *getter* methods, but also a *setter* methods. You can probably guess that while a *getter* method is used to retrieve the value of a field, a *setter* method is used to modify the field value. But before we add these methods, let's think more about how this bank software might be used. We might not only want a first and last name to identify our client. We will probably want information about the identification they used to create the account. Perhaps we should additionally store their name using Chinese characters if they have a Chinese name. And usually we will want to record their address and phone number. If we store all this info in our `BankAccount` class, then if a client wishes to open two separate bank accounts, we will have all the client information duplicated in the two instances of `BankAccount` that belongs to the one client. Also, perhaps there may be reasons to maintain client information even if they don't have an account (for example if they wish to close their current account, but may wish to open one at a later date). While each bank account requires an owner, it seems that the information associated with the owner doesn't really fit into this `BankAccount` class.

The solution to this problem is actually very easy. In the real world, the client and the bank account are two separate entities. In object-oriented software we make two separate classes.

Create a new class named `Client`, and follow the UML diagram below to write the code for it. Notice that all the fields within the class have been made `private`, and access to them is provided by setter and getter methods. Also note that the `clientNumber` field has no setter method. We will write code to set the client number later. When writing the `printClientInfo`, use the getter methods to retrieve values for the fields rather than accessing them directly.



When writing your `setLastName` and `setFirstName` methods, if you follow the UML diagram above accurately, you will notice that you want to set the instance field `lastName`, but the parameter of the method is also called `lastName`. It is not only acceptable to have parameters with the same name as the instance field name, it is quite common in practice. When we have this type of ambiguity, the instance field name is appended to the prefix "`this.`" to resolve the ambiguity, so your assignment of the parameter `lastName` to the instance field `lastName` will look like this:

```
this.lastName = lastName;
```

Test your `Client` class by adding a few lines of code to your `TestBankAccount` class. Below is some example code that you may use as a template to test your code (you are encouraged to do more testing than these few lines!). Replace the names given your own family name (姓) in pinyin for the last name and your given Chinese name (名) in pinyin, or your English name, for the first name.

```
Client client1 = new Client();
client1.setFirstName("Chris");
client1.setLastName("Nielsen");
client1.printClientInfo();
```

The output of this code should look similar to:

```
Client Number:  0
Last Name:      Nielsen
First Name:     Chris
```

6. Change the `setLastName` and `setFirstName` methods so that the length of either name is greater than ten characters, the length with be truncated to a maximum of ten characters. This choice of ten characters is arbitrary, and when we hard code a number as a literal, as we must do here, the number may be referred to as a ***magic number***. When another software developer reads code with a magic number in it, the reasons for choosing the number may not be obvious. Maintaining and updating the code becomes more difficult. If a magic number is required, the better way to do it is declare a ***constant***, and use the label for that constant in the code. Earlier, we discussed the constant π in the `Math` class that is stored in the constant field `Math.PI`.

   In Java, constants are created by adding the modifier `final` to the field declaration. If any field (or variable) is declared as `final`, once it is initialized to a value, that value cannot be changed. The naming convention of constants in Java is to use all capital letters, with words separated by an underscore (_).

   At the top of the `Client` class, create two `final` fields, named `LAST_NAME_MAX_LENGTH` and `FIRST_NAME_MAX_LENGTH`, of type `int` and set the value of each to `10`. Then use these two fields when you write the code to truncate the length of the names passed to methods `setLastName` and `setFirstName`. The field declarations should look like this:

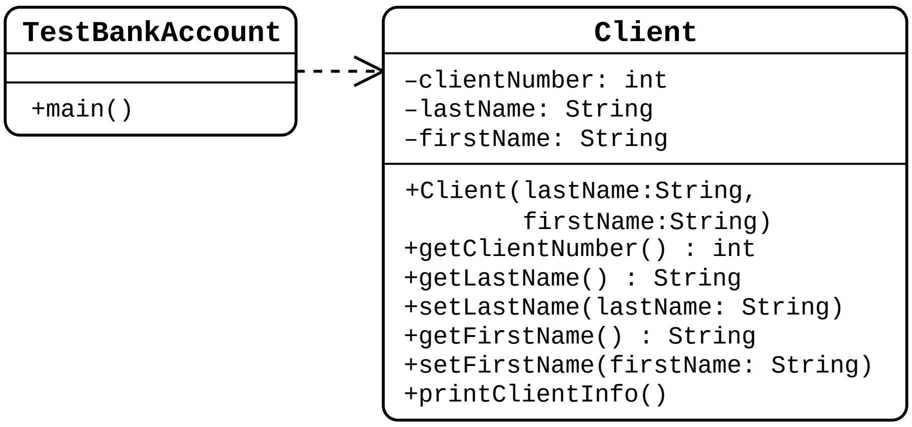
```
private static final int LAST_NAME_MAX_LENGTH = 10;
private static final int FIRST_NAME_MAX_LENGTH = 10;
```

The `static` modifier is better, but not essential; we will discuss its purpose soon.
Add some code to the `TestBankAccount` class to test that your `Client` class properly truncates names to ten characters.

7. We will now write a ***constructor*** for `Client` objects. The code we wrote above to test our `Client` class first created a new instance of type `Client`, then initialized values in that object. The purpose of a constructor is to initialize the object when we create it. This is especially useful when we want to require that some fields be initialized to a value.
   To write a constructor in Java, we write a method with the name <u>exactly</u> the same as the class name that does not specify a return value (because constructors always return an instance of the class). In this case, the declaration of the constructor for the `Client` class, which takes in two parameters of type `String` that will be used to set the fields `lastName` and `firstName`, will look like this:

   ```
   public Client(String lastName, String firstName)
   ```

   Our updated UML diagram will look like this:



   Write the remainder of the code for this constructor. Rather than setting the instance fields directly, use the setter methods to set values for `lastName` and `firstName`. Doing this allows reuse (as opposed to duplication) of the code we wrote in the setters to truncate the length of names.
   If you wrote your constructor correctly and did not introduce some other error, upon running your code, you should get an error similar to:

   ```
   The constructor Client() is undefined
   ```

   This is because our test code in `TestBankAccount` still calls the constructor for the `Client` class without any parameters with this line:

   ```
   Client client1 = new Client();
   ```

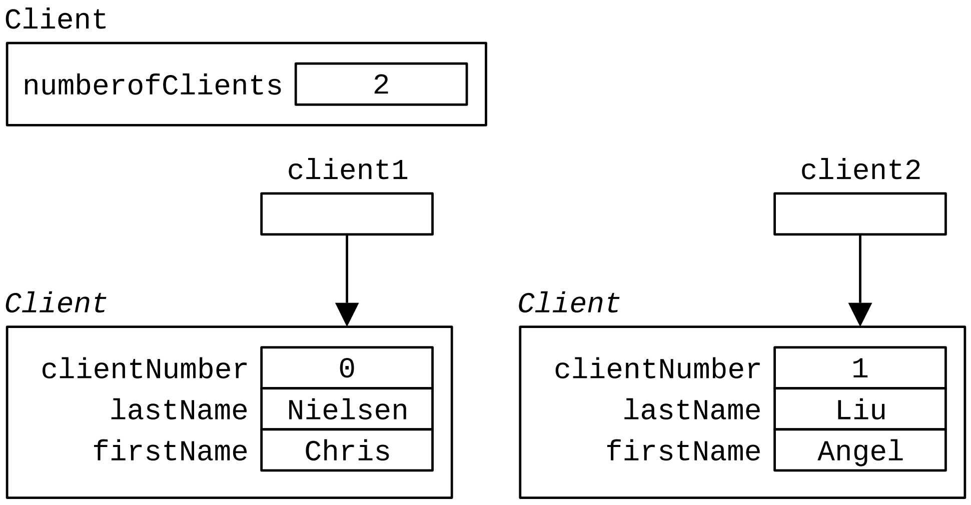
   In Java, if you do not explicitly write a constructor for a class, a "default" constructor that does not take any parameters and does not initialize any field values will be inferred. Once any constructor is explicitly defined, no default constructor will be inferred. If you wish there to be a constructor that accepts no parameters, you will need to explicitly define one. For our `Client` class, it doesn't really make sense to have a nameless client, so we will not create a constructor with no parameters. The only way to create a `Client` is to provide parameters with values for the fields `lastName` and `firstName`.
   Modify the test code in the `TestBankAccount` class so that the values for the fields `lastName` and `firstName` are passed as parameters to the constructor call for `Client`.

8. We will assign a unique sequential `clientNumber` starting from client number zero (`0`) by keeping a counter that increments each time we instantiate an instance of a client. Such a counter could be stored in a class that creates the new instances, but this would complicate things if we want to instantiate instances of the `Client` class from multiple classes. It would be most logical to store the counter inside the Client class; however the instance fields we've been created have a different value for each instance of the class we create. We require a field that has a single value that is shared amongst all the instances. This is what the `static` modifier is used for.

Add a field named `numberOfClients` of type `int` that is modified to be both `private` and `static` to the `Client` class:

```
private static int numberOfClients = 0;
```

Below is a diagrammatic representation of example data using the current `Client` data structure.



For this paragraph, assume all fields in `Client` were public. For non-`static` fields, we must access them through an instance, such as: `client2.lastName`. An attempt to access a non-`static` field using the class name, such as `Client.lastName`, has no way of working because we do not know which instance of the class that we want to get `lastName` from. Do we want `client1.lastName` or `client2.lastName`? However, for a `static` field, such as `numberOfClients`, we are able to access it it either of two ways – either from the class with `Client.numberOfClients`, or from any instance such as this: `client2.numberOfClients`. The former way is better practice, since it makes it obvious to readers that the field is `static`, so do **not** use an instance of a class to access `static` fields or methods.

Now back to coding our project. The `static` field `numberOfClients` will store a count of the number of clients (i.e.: the number of instances of the type `Client` we have instantiated). Since the constructor for the `Client` class is called in order to instantiate an object of type `Client`, if we increment `numberOfClients` each time the `Client` constructor is called, it should correctly keep track of the number of clients we have.

Add code in the `Client` constructor to increment the `numberOfClients` counter by one when the constructor is called. Then, also in the constructor, add code to assign a value to the field `clientNumber`. It should be assigned the value that is in `numberOfClients` before it is incremented so that our `clientNumber` will be zero-based.

Also, implement a getter method `getNumberOfClients`. As this method will only access the `static` field `numberOfClients` and will not access or modify any instance (non-`static`) fields, the method declaration should include the `static` modifier. That way an instance of type `Client` is not required in order to call the method. The method can simply be accessed using the class name: `Client.getNumberOfClients()`. As mentioned, although `static` methods can also be called using an instance variable for the class (for example `client2.getNumberOfClients()`), this is considered bad practice and should be avoided.

Write code that will instantiate at least three clients to ensure the `clientNumber` field for each new instance is assigned the next increasing value, and that your `getNumberOfClients` method is working properly.

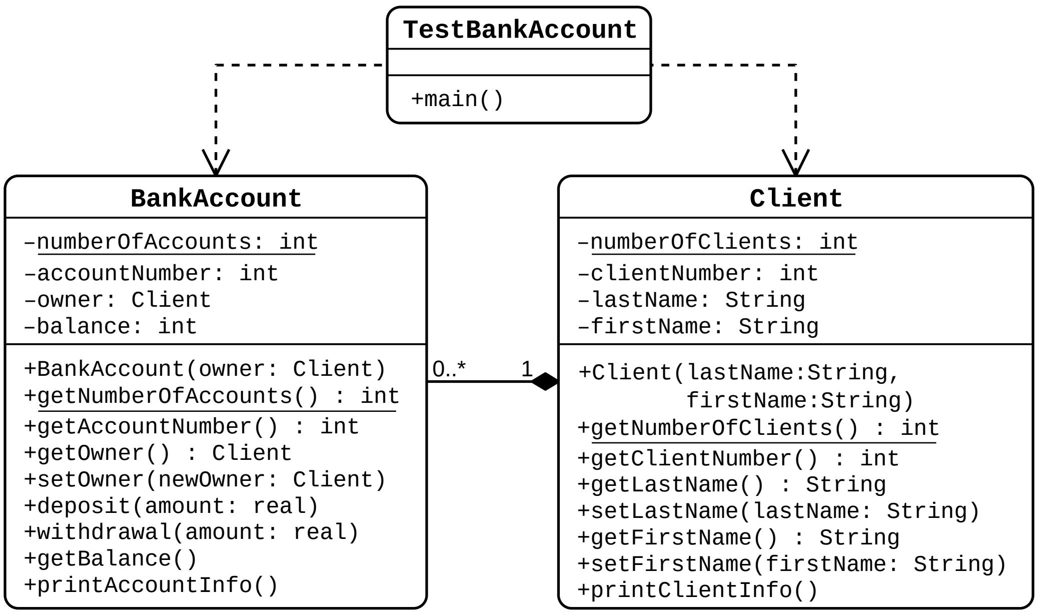
9. Modify the `BankAccount` class to add a constructor for the class, assign sequential values to the `accountNumber` field, and use an instance of the `Client` class to store the information for the owner of the account.

   Start this by adding a `private static` variable of type `int` called `numberOfAccounts` and initialize it to zero. Increment this with each new `BankAccount` that is instantiated, and use it to set the value of the `accountNumber` field for each instance. Write a getter for this field following the common naming convention for a getter.

Remove the `firstName` and `lastName` fields, and add a `private` instance (non-`static`) field named `owner` of type `Client`. Write setter and getter methods for this new field.

Write a constructor for the `BankAccount` class that takes a single parameter of type `Client` that contains the data for the owner of the account. Remember to increment the `numberOfAccounts` field when a new instance is created.

The UML diagram for the most recent update to our code is given below. In UML, `static` attributes (fields) and operations (methods) are underlined.



Update your TestBankAccount code to test the newly implemented functionality.

10. Currently, in both the `BankAccount` class and the `Client` class there are methods that print out the *state* of the instance (the values stored in the fields). This is coupling the functionality of these classes with functionality of a user interface. We will not be implementing a sophisticated user interface, however, let us remove the functionality related to the user interface from each of these classes. (In actual fact, an additional reason to do this is to minimize coupling between these two classes when we implement additional functionality in the next step.)

    Remove method `printAccountInfo` from the `BankAccount` class, and remove method `printClientInfo` from the `Client` class. Add code to the `TestBankAccount` class to implement similar functionality.