This problem set is due **Friday, March 10 at 11:59pm, KST**. Note that the due date that you see on Blackboard is not accurate since it shows the time in EST. You should go by the due date in this handout.

- Follow the specification *exactly*. It makes grading much easier.
- Each method that you write must be properly commented and attractively formatted. If you format your functions the way I do in my examples or the way it is done in the text, it would be considered attractive.
- Be sure to include a comment at the top of each file submitted that gives your name, your student ID number, and your email address.

### What to Submit

Your submission should include the following. Please do not hand in `.class` files or any other that I did not ask for.

```
Warmup.java
ArrayTools.java
Name.java
Customer.java
Account.java
CheckingAccount.java
SavingsAccount.java
Bank.java
(others if you added more)
```

Hand in all the source files that make up your solutions as a **single zip or tar file** on **Blackboard**. Your submitted files should be self-sufficient in that I won’t have to add any additional file to run your programs. Multiple submissions are allowed within the time limit. Please do not submit `.class` files or any Eclipse-specific project files that I did not ask for.

### Introduction

The main goal of this problem set is to give you a chance to bring yourself up the level of Java programming skills that we will need as we start the semester. Problem 1 is a warm up exercise. The level of sophistication in Java and programming skills needed to solve Problem 2 below may be a little higher than you might have expected, but let’s see how much of that we can handle. If you are not 100% comfortable with Problem 2, I am not that concerned as long as you can catch up within a few weeks beyond the due date. You should hand in whatever you have by the due date and continue working on it to get it done. While we study the next topic, algorithm analysis, we won’t introduce any new Java so you will have some extra time to catch up on Java during that time. If you
find yourself struggling with Problem 2 whether it is because of the level of Java or because of the complexity of
the problem, please let me know as soon as possible so that we can help you. Suggestion: read this handout all
the way through before you try to solve any of the problems in the set.

Warning on incomplete work: I would rather see a partial solution that runs without any errors (both compile-
time and run-time errors) than an almost complete program that crashes. Any program that you hand in that
stores will not be worth much—it may earn about 25% of the available points at maximum. This will apply to
any program that you hand in throughout the semester.

Problem 1

Create a class named Warmup in a file named Warmup.java. Your program must be error-free. This problem
must satisfy the following:

1. The class Warmup should have a static method named readNumber that takes no argument and reads
a number of type int from the standard input device (i.e., keyboard) and returns the number read in to
the caller, the main in this case. Obviously, main will have a local variable that remembers the number
returned from readNumber and use it to call the method that I will describe next.

2. I am making a class named ArrayTools available and you will use it here. If you need to modify that
class to satisfy the specification here, feel free to do so. The main this time calls the static method
named randomArray using the number that you read in in the previous step. That number will be used
as the size of the array that you create. Populate the array with random integers in the range of 0 and 99,
inclusive. Make sure that there are no duplicates in the array. That is, the numbers in the array must be
unique. The method returns the array built. Here again you will have a local variable in main that holds
the array so that you can refer to it as you need below.

3. In main print the contents of the array using an appropriate method found in ArrayTools. If you don’t
find one that is readily usable there, define one there and use it.

4. In Warmup, introduce a static method named isEven that takes an integer and returns true if the
actual argument is even and false if odd. Use this method in the next few steps.

5. In Warmup, introduce a static method named countEvenOrOdd that takes two arguments: one inte-
ger and an array of integers. If the integer argument passed in is an even number it will count only the even
numbers in the array and returns the count. If an odd number is passed in, count only the odd elements and
returns it.

6. In Warmup, introduce a static method named sumEvenOrOdd that takes two arguments: one integer
and an array of integers. If the integer argument passed in is an even number, it will sum only the even
numbers in the array and returns the sum. If an odd number is passed in, sum only the odd elements and
returns it.

7. In main, print the average of even elements and the average of the odd elements of the array. To get more
accurate results, print them as double numbers. If the result must be 6.78, your program should produce
6.78, not 6.0.

8. In Warmup, introduce a static method named iSort that takes an array of integers and sorts the numbers
in the array using the insertion sort algorithm. If it needs some auxiliary method(s), feel free to add it (them)
too. Call the sorting method from main to sort the array.

9. In main, print the sorted array.

10. In Warmup, introduce a static method named search that looks for an item passed in as argument
along with another argument which is an array of numbers. If the item is found in the array, it returns the
index of the array where the item was found. If the item is not found, it should return -1. Print the index
found. Your search method should assume that the array is already sorted and it must take advantage of the
fact that it is sorted. And add a piece of code in main to test your search method. In the search method,
add a precondition if appropriate.
11. In Warmup, introduce a static method named binSearch that looks for an item much like search does. binSearch however must perform binary search, iteratively. And add a piece of code in main to test your binSearch.

12. In Warmup, introduce a static method named recSearch that looks for an item much like binSearch does. recSearch, however, must do binary search, recursively. And add a piece of code in main to test your recSearch.

Hand in Warmup.java and ArrayTools.java.

Problem 2

In this problem we will design a set of classes to model a bank. We will design at least six classes: Name to model names of customers, Customer to model customers, Account to model bank accounts, CheckingAccount to model checking accounts, SavingsAccount to model savings account, and Bank to model banks. If you want to add more classes, feel free to do so. (Be sure to submit them all.) Using this program you should be able to represent any one of the banks, e.g., "Kookmin Bank", "Bank of America", etc., each as an instance of the Bank class.

Design these classes that satisfy the following requirements. We will look at classes that will make up the program bottom-up, i.e., from simpler to more complex classes in order.

- A customer contains the customer’s name (first name, middle initial, and last name), address, phone number, social security number, and mother’s maiden name. You get to decide the type of each of these. You should represent the name of a person as a separate class. This is where the concept of object composition will be used. That is, the type of the name field in the Customer class will be Name. I strongly suggest that you test your implementation of Name and Customer classes before you write the next class.

- Each account in a bank has the following features:
  - When an account is created, the beginning balance is set to a random number between 0 and 100. This is a special that is being offered only if the account number generated for the account happens to be a number that is a multiple of 7. Otherwise, the initial balance is set to 0. If you decided to add the initial balance to be passed into the constructor, you can add it to the 0 or the random number that was generated whichever case it may be.
  - An account has an account number which is unique. See below to see how this would be generated by whom.
  - An account contains the customer’s (account owner’s) information. You will be using object composition here as well. That is, the type of the customer field in this class will be Customer.
  - An account knows which bank it belongs to by remembering the object representing the bank as one of its fields. That is, using this object reference if you want to find the name of the bank that the account belongs to, you will have to follow the object reference to the bank object associated with the account object and ask for the name of the bank from the referenced bank object. This means that you are not allowed to keep the bank’s name, e.g., "Bank of America” as a string, explicitly in the account object. This is another instance of object composition.
  - It supports the usual methods such as deposit, withdraw, and whatever else you want to add.

- Each checking account in a bank has the following features:
  - It must be represented as a subclass of Account class.
  - It must add features that are unique to checking accounts. That is, we are modeling checking accounts and savings accounts separately so find features that are unique to checking account and include them here. You don’t need to add very many features, but at least one.
  - Would you add at least one constructor in this class? Most likely yes.

- Each savings account in a bank has the following features:
  - It must be represented as a subclass of Account class.
– It must add features that are unique to savings accounts. Here again, you don’t need to add very many features, but at least one.
– Would you add at least one constructor in this class, too?

• A bank object has the following features:

  – A bank has a name, e.g., "Bank of America".
  – A bank remembers all of its accounts (i.e., account objects). You may assume a reasonable number as the upper limit on the number of accounts that a bank can handle in your program, e.g., 100. I am assuming that you will be using an array to hold these objects, not an ArrayList or any other collections class. An interesting question to ask: Would you ever actually create any instance of the Account class explicitly by calling one of its constructors outside the class? Should you keep the checking accounts and savings accounts separately? Given an account object, how would you tell a checking account from a savings account at runtime? Would you add a separate field in the account class to remember the type of accounts, checking vs. savings vs. any other type if you choose to add other kinds? Or, would you use the Java keyword instanceof to find it out? Would you ever see the need to tell them apart at runtime as part of your code even using instanceof? By considering these questions, your design may come out differently and it is acceptable.
  – A bank can create a new account. Name the method createAccount. Think about what you would pass as arguments to this method. You would most likely want to pass a customer object that represents the owner of the new account being created. You would most likely have to pass a name object as you create a customer object, right? The customer object that was created this way will be passed as you create an account object, whether it is a checking account or a savings account. The createAccount method then would create an account object by calling the constructor of CheckingAccount or SavingsAccount. Wait! Do we want two separate methods: one called createCheckingAccount and the other called createSavingsAccount? Or, just use createAccount with another parameter telling which kind? You get to decide! In either case when you call the constructor, you would want to pass a unique account number, the bank object to which the account object will belong to (you may find the ‘this’ object reference useful here), and the customer object. Once you create the account, you will then add it to the array that is keeping track of all the accounts that have been created for the bank. This depends on whether you are using one array to remember all the account objects or multiple separate arrays to keep the different kinds of accounts separately (Again, your decision). You will of course want to update the count of the accounts also here - again one count, multiple separate counts, or three separate counts?
  – Add the capability of deleting an account. Name this method deleteAccount. Deleting an account means that you need to remove it (the object reference referencing the account object being deleted) from the array(s) that remember(s) the accounts for the bank. If you delete an object from an array, you are creating a hole and that hole must be filled in. One easy way to fill it in would be to move the last one in the array into that hole. I am assuming that for each array you are keeping track of the number of account objects that are actually in the array, which would be different from the size of the entire array. That number (count) will be useful as you try to fill the gap. Here again you should ask yourself whether to use deleteAccount or two separate methods (deleteCheckingAccount and deleteSavingsAccount). Similar comments to the ones that I added for createAccount would apply here as well.
  – A bank can return the number of customers (account objects) it actually has. Name the method getNumCustomers. If a customer has a checking account and a savings account, should we count him/her as one customer or two? You decide!
  – A bank can return the average balance of all the accounts. Name the method getAverageBalance. How about average balance of savings accounts and that of checking accounts? You decide!
  – A bank can return the name of the customer with the highest balance. Name the method getNameOfHighestBalance. Should we separate different kinds of accounts here? You decide!
  – The Bank class supports a method named sortAccounts that sorts the account objects according to the account balances. Note that the Account class would have to implement the Comparable interface for this to work. Here again would you do it in one array if you are using only one array to keep all the accounts regardless of the kind of accounts? Or, in two or three arrays? Two if you are
only separating checking accounts from savings accounts. Three if you are keeping all the accounts in
a separate array in addition to the two arrays that remember checking accounts and savings accounts
respectively. You decide!

- Your program must have the ability to generate a unique account number for each account being generated
  in a bank. You can use a static field and an associated static method to handle this issue. Let's add
  this capability to the Bank class, not to the Account class. This depends on whether to use static
  fields and methods as you design a scheme to generate them or non-static fields and methods to do so.
  Think about it!

- Add a test that demonstrates your program is working correctly. You may use a main method in the Bank
  class or create a new class such as UseBank in a separate class. Whichever you choose, add enough to
demonstrate that your program has all the capabilities required in the specification, without being verbose.

If you want to add any other features, you are welcome to do so. If you do, please document clearly at the top of
the Bank.java file so that we can see clearly what extra features are added. I am not offering any extra credit
for extra features that you may be adding though.

One thing that you may want to think about is whether it would make sense to keep Account as an abstract
class or not. You will be using at least one interface, namely Comparable. Finding a way to incorporate
an abstract class would be great if you see an appropriate place to use it.

Be very careful on what should be static or non-static. What should be public, protected, or
private. Also, include appropriate constructors, getters, and setters in each class where appropriate.

Suggestion: As you develop your code, try to exercise incremental development: start with a simpler class, in
this case it would be Name. Introduce a main method in that class to test that class alone. Then, continue on
to Customer class with a main in it to test it. Note that in this particular problem set, you will have to have
a Bank object created before any kind of an account object can be created because the constructor of one of the
account classes will need an instance of Bank as its argument. So, here is an example where linear incremental
development isn’t quite going to work, but the general idea of incremental development is still a good idea to
practice whenever possible.

Hand in all the classes (only .java files) that make up your solution. If there are other files that we would need
to run your program as we grade your work, include them as well. We don’t need .class files or the Eclipse
project files though.

Naming conventions in Java and programming style in general

Please use these conventions as you establish your programming style. Programming professionals use them, too.

- **Names:** Choose informative names, e.g., hourlyRate rather than hr if it is to be a variable name.
- **Class names:** We start a class name with a upper-case letter as I have in my examples: Hello not hello
  or HELLO. For multi-word names do as I do here: UndergraduateStudent, GraduateStudent,
  CheckingAccount, etc.
- **Variable names:** We start a variable name with a lower-case letter, e.g., count, hourlyRate, etc.
- **Method names:** Same as that for variable names.
- **Use of white space:** Adopt a good indentation style using white spaces and be consistent throughout to
  make your program more readable.

Please pay attention to these seemingly trivial yet important things as you establish your programming style. You
are establishing a style and it is important to build a good one. We will deduct points when we grade your work if
you deviate from these suggestions for no apparent good reasons.