Java Refresher

Module 5 - Class Composition and Iteration

In full-sized applications, it is common to have many classes with complex interrelationships between them. One very common relationship pattern involves a parent being composed of a list of children. Examples include: a customer has a list of bank accounts. A company has a list of employees. A word processor has a list of open windows. After completing this module, you will have written a simple class that is composed of a list of children.

Background Material

The Input class

To help you read input from the user, we provide you with a class called `Input`. This class is included in the Module 5 Skeleton Code and provides the following methods:

Javadoc

Class Input

The Input class provides a set of static methods for easily asking the user questions and reading their answers.

Method Summary

Modifier and Type	Method and Description
`static boolean`	**`askBoolean`**`(java.lang.String question)` Asks the user the given question, waits for the user to enter a boolean at the keyboard, and then returns this boolean.
`static char`	**`askChar`**`(java.lang.String question)` Asks the user the given question, waits for the user to enter a single character at the keyboard, and then returns this character.
`static double`	**`askDouble`**`(java.lang.String question)` Asks the user the given question, waits for the user to enter a double at the keyboard, and then returns this double.
`static int`	**`askInt`**`(java.lang.String question)` Asks the user the given question, waits for the user to enter a integer at the keyboard, and then returns this integer.
`static java.lang.String`	**`askString`**`(java.lang.String question)` Asks the user the given question, waits for the user to enter a string at the keyboard, and then returns this string.

Reference example

Consider the following definition for a simplified class of bank accounts:

```
public class BankAccount
{
    private String accountName;
    private String accountType;
    private double balance;

    public BankAccount(String accountName, String accountType, double balance) {
        this.accountName = accountName;
        this.accountType = accountType;
        this.balance = balance;
    }

    public String toString() {
        return accountName + "'s " + accountType + " account has $" + balance;
    }

    public void deposit(double amount) {
        balance += amount;
    }

    public void withdraw(double amount) {
        balance -= amount;
    }
}
```

In a banking application, we can imagine wanting to store information about each customer and what accounts they have:

```
public class Customer
{
    private String name;
    private BankAccount savingsAccount;
    private BankAccount creditAccount;
}
```

Read: *each customer has a name, a savings account and a credit account*. The main limitation with this design, however, is that it assumes that every customer always has two accounts, when in reality, some customers have only a savings account, and other customers may have 3 or more accounts, including a home loan account. To manage the unknown number of accounts that each customer may have, we will store them in a list. A list is a data structure quite similar in purpose to an array, except that while an array has a fixed length, the length of a list can grow and shrink as needed.

The `ArrayList` class which can be imported from the `java.util` package (see the documentation) provides the ability to store a list of objects of a particular type. Because the size of the list is not fixed, we are free to add and remove items from the list as needed. Open the linked documentation for `ArrayList` and read the documentation for the `add` method. The basic usage of `ArrayList` and the `add` method can be summarised by the following code example:

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
accounts.add(new BankAccount("Tom Smith", "savings", 1500.0));
accounts.add(new BankAccount("Tom Smith", "credit", -53.99));
accounts.add(new BankAccount("Tom Smith", "loan", -79000.0));
```

In older versions of Java, the first line would have been written more simply as `ArrayList accounts = new ArrayList();` but this older, simpler, way is no longer considered safe and the compiler will issue a warning for such code. The fuller, more complex version (highlighted in yellow above) includes the notation `<BankAccount>` after the `ArrayList` class name which *restricts* the type of items that are allowed to be stored into the list, in this case, to `BankAccount` objects. This prevents the programmer from accidentally storing the wrong type of item into the list. Whatever type you choose will become the required type for the parameter to the `add` method. In this case, the `add` method requires a `BankAccount` object as a parameter. You can verify in the above example that each time the `add` method is called, the parameter passed in is always an instance of the `BankAccount` class. For example, the first use of the `add` method above passes the parameter `new BankAccount("Tom Smith", "savings", 1500.0)` which is an instance of the `BankAccount` class.

Once a list has been created and items have been added to it, a programmer will usually want to iterate over that list and process each item sequentially in some way. This is achieved using Java's *for-each* loop, as shown in the following example:

```
System.out.println("List of all bank accounts:");
for (BankAccount account : accounts) {
    System.out.println(account);
}
```

This for-each loop should be read: *For each `account` (of type `BankAccount`) in the list `accounts`, print that `account`.*

Here is another example:

```
System.out.println("Depositing $10 into each account");
for (BankAccount account : accounts) {
    account.deposit(10.0);
}
```

Read: *For each `account` (of type `BankAccount`) in the list `accounts`, deposit `10.0` dollars into that `account`.* The `account` variable highlighted in yellow is the key variable in this loop. If the list being iterated over contains 3 accounts, then this loop will iterate 3 times, and each time the next account will be automatically loaded into this `account` variable. The code you write inside the body of the loop (in this case `account.deposit(10.0);`) will be executed 3 times, and each time, the variable `account` will automatically be loaded with the next item from the list in sequence.

Putting what we've learnt together, it is now possible to write a `Customer` class that maintains a list of bank accounts, allows you to add more accounts, and shows all accounts:

```
import java.util.ArrayList;

public class Customer
{
    private String name;
    private ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();

    public Customer(String name) {
        this.name = name;
    }

    public void showAllAccounts() {
        System.out.println("List of all bank accounts for " + name + ":");
        for (BankAccount account : accounts) {
            System.out.println(account);
```

```
        }
    }

    public void addAccount(BankAccount account) {
        accounts.add(account);
    }
}
```

And the following code demonstrates how this `Customer` class can be used in an actual program:

```
Customer customer = new Customer("Tom Smith");
customer.addAccount(new BankAccount("Tom Smith", "savings", 1500.0));
customer.addAccount(new BankAccount("Tom Smith", "credit", -53.99));
customer.addAccount(new BankAccount("Tom Smith", "loan", -79000.0));
customer.showAllAccounts();
```

This demo program is included in the Module 5 skeleton code project in the class named `CustomerDemoProgram` and you are encouraged to run it and view the program's output.

Exercises

In the provided skeleton code for this module, you will find a `Contact` class containing the following definition:

```
public class Contact {
    private String name;
    private String phone;

    public Contact(String name, String phone) {
        this.name = name;
        this.phone = phone;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public String toString() {
        return name + ": " + phone;
    }
}
```

In the following exercises, you will build a `Phonebook` class that contains a list of contacts. It is perfectly acceptable to copy code from the reference example above, since most of the code will be very similar.

Exercise 1.

Learning objective: Define a class that is composed of a list of smaller objects.

Class to edit: `Phonebook`

Since no `Phonebook` class is included in the skeleton code, your first step is to create a new class named `Phonebook`. In BlueJ, you can accomplish this by clicking on the "New Class..." button. **Don't forget:** class names should begin with an uppercase letter, so make sure you type the class name as `Phonebook`, *not* `phonebook`. Also note that `Phonebook` is one word, and the `b` in `book` is a lowercase `b`.

After creating this class, double click on it to open the code editor. Select all code in this file and DELETE it. Write the following class declaration:

```
import java.util.*;

public class Phonebook
{
    private String owner;
    private ArrayList<Contact> contacts =   « complete the rest of this line »

    public Phonebook(String owner) {
        this.owner = owner;
    }
}
```

Read: *A phonebook has an owner and a list of contacts. When creating a phonebook, you must supply the name of the owner.*

Compile your code and if there are no errors, export your project to a jar file being sure to "Include the source files", and then submit this to PLATE to confirm whether your solution is correct.

If your code is incorrect, you may have unlimited attempts to edit your code, re-export to a jar file and resubmit.

Passed? Excellent! As you progress through each exercise below, be sure to continually submit to PLATE to confirm whether your solution is correct. If your solution is not correct, let PLATE's feedback report inform you on what you need to fix.

Exercise 2.

Learning objective: Learn how to add an object to a list.

Class to edit: `Phonebook`

Define a method for class `Phonebook` called `addContact` which takes a `Contact` as a parameter and adds this contact to the `contacts` list.

Hint: base your solution heavily on the `addAccount` method in the Background Material section.

Example of usage:

```
Phonebook phonebook = new Phonebook("Kylie");
phonebook.addContact(new Contact("Jenny Li", "(02) 88776655"));
phonebook.addContact(new Contact("Sam Tanaka", "(02) 11112222"));
```

Exercise 3.

Learning objective: Learn how to iterate over a list and perform an action on each item.

Class to edit: `Phonebook`

Define a method for class `Phonebook` called `show` with no parameters which iterates over each contact in the `contacts` list and prints the string representation of each contact. Before printing out each of the contacts, also print the header line `"Kylie's phonebook"` where `Kylie` should be replaced by the actual owner of the phonebook.

Hint: base your solution heavily on the `showAllAccounts` method in the Background Material section.

Your method should behave according to the following sample I/O

```
Kylie's phonebook
Jenny Li: (02) 88776655
Sam Tanaka: (02) 11112222
```

Exercise 4.

Learning objective: Put the `Phonebook` class to use.

Class to edit: `ShowPhonebook`

Create a new class called `ShowPhonebook` defining a standard `main` method. That is, a method called `main` that is `public`, `static`, has a return type of `void` and takes an array of strings as its parameter.

Inside the `main` method, write a program that uses your `Phonebook` class, according to the following steps:

1. Create a new phonebook with "Sam Johnson" as the owner.
2. Add a new contact to the phonebook with name "Kelly Wong" and phone number "(02) 12345678".
3. Add a new contact to the phonebook with name "Richard Jackson" and phone number "(02) 87654321".
4. Show the phonebook.

Exercise 5.

Learning objective: Implement a linear search over a list of child objects.

Class to edit: `Phonebook`

Double click on the `Phonebook` class and within it, define a new method called `findContactByName` which takes a contact name as a parameter, searches through the `contacts` list for a matching contact with that name, and then returns the found contact. When no matching contact is found, this method should instead return null. Please read the entire exercise description before attempting your solution.

This method should iterate over each contact in the contacts list and if that contact's name is equal to the contact name given as a parameter, then return that contact. We don't need an

"else" statement, because if we have not found the contact we are looking for, we want to iterate to the next entry in the contact list. Let's consider how this method will work using two examples. Consider a phonebook with the following contacts:

```
Kelly   Wong:   (02)    12345678
Richard Jackson:        (02)    87654321
Sue     Carey:  (02)    11223344
Simon   Jones:  (02)    11112222
```

In the first scenario, let's suppose that your method findContactByName is called with parameter "Sue Carey". The loop iterations will proceed as follows:

- Iterate to the first contact Kelly Wong
- Check if "Sue Carey" equals the name of this contact (FALSE)
- Iterate to the next contact Richard Jackson
- Check if "Sue Carey" equals the name of this contact (FALSE)
- Iterate to the next contact Sue Carey
- Check if "Sue Carey" equals the name of this contact (TRUE)
- THEN return this contact.

Now let's consider a second scenario. You may ask, what would happen if the method findContactByName is called with parameter "Bill Forsythe" which does not correspond to any contact in the phonebook. What will be returned? In this case, your method will loop through every contact testing FALSE every time. ONLY after testing every contact, the method should then return the special value of null, indicating no contact could be found:

- Iterate to the first contact Kelly Wong
- If "Sue Carey" equals the name of this contact (FALSE)
- Iterate to the next contact Richard Jackson
- If "Sue Carey" equals the name of this contact (FALSE)
- Iterate to the next contact Sue Carey
- If "Sue Carey" equals the name of this contact (FALSE)
- Iterate to the next contact Simon Jones
- If "Sue Carey" equals the name of this contact (FALSE)
- We try to iterate to the next contact, but we have reached the end of the list.
- Therefore the Iteration completes without finding any match
- Return null.

In other words, the code you need to insert to return `null` should be inserted *after* the for-each loop has completed. That is, after it has exhaustively searched through every item.

Without the final `"return null;"` statement, the compiler will report that you have a missing return statement. This is because the compiler can see that without it, there is a neglected case in which you have not specified what your method will return.

Hint: The return type of your method should be `Contact`

Exercise 6.

Learning objective: Learn how to test for a null result.

Class to edit: `LookupPhonebook`

Create a copy of the `ShowPhonebook` class called `LookupPhonebook`. Keep all of the existing code, but add the following steps to the main method:

1. Ask the user `"Enter a contact name:"` using the `Input` (see the Background Material for documentation of this class).

2. Use the `findContactByName` to search for the contact in the phonebook that has the name chosen by the user in the previous step, and store this result into an appropriately named and typed local variable.
3. If the contact returned in step 2 is not null, print `"Phone number is xxx"` where `xxx` is the phone number of the contact that was found.
4. Otherwise, print `"xxx not found"` where `xxx` is the contact name that the user chose to search for.