# CS1020: Data Structures and Algorithms I

## Tutorial 2 – Advanced Object Oriented Concepts

(3rd February 2012)

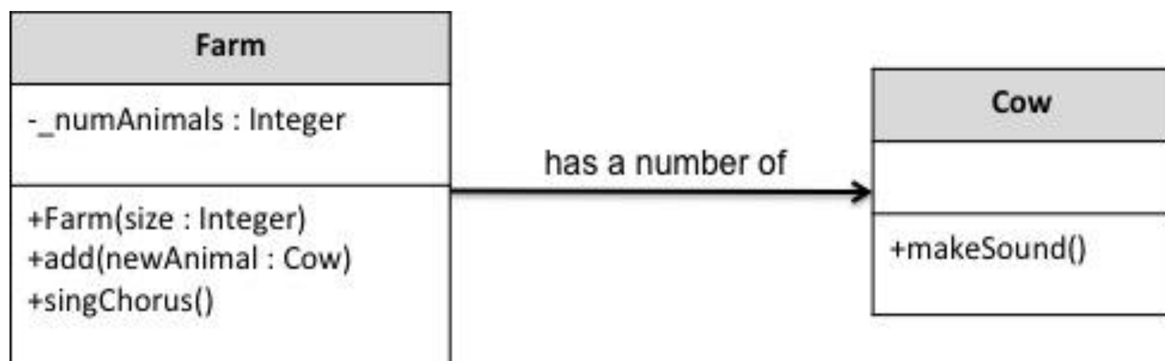
**Suggested Solutions**

1. **[Polymorphism]** There is a nursery rhyme "Old McDonald" to teach young kids about animals and the sound they make:

   **Old McDonald had a farm, E-Ya-E-Ya-O**
   **And, on his farm he has a Cow, E-Ya-E-Ya-O**
   **There is a "Moo Moo" here, and a "Moo Moo" there**
   …….

   Mr. Kidd attempts to model the nursery rhyme using the UML Class Diagram as shown below:

   > Notes:
   > - The symbols to the left of the variables and methods denote their visibility. A '-' sign denotes Private, a '#' sign denotes Protected, and a '+' sign denotes Public.
   > - You may follow this link to read up more on access control: http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html
   > - The 'has a number of' arrow between the two classes is an association, not an inheritance.
   > - You may also read more about UML Class Diagram in the following link: http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/




   His idea is to allow a Farm object to store a number of Cow objects. Whenever the **`singChorus()`** method is invoked, all cows on the farm will make the "Moo" sound. His implementation can be found in **`OldMcDonaldCow.java`**. Below is a sample run (user input in **bold**):

```
Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 1
Cow added!

Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 2
All farm animals start to sing:                 (Only 1 cow on farm)
Mooo Mooo

Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 1
Cow added!

Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 1
Cow added!

Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 2
All farm animals start to sing:                 (A total of 3 cows now)
Mooo Mooo
```

```
Mooo Mooo
Mooo Mooo

Choose 1. Add more cow, 2. Farm chorus, 3. exit -> 3
Bye bye!
```

Mr. Kidd realized later that the farm is supposed to have several types of animals instead of just cows! He would like to expand his program to handle at least two more types of animals, the chicken (which goes "Pook Pook") and the pig (which goes "Oink Oink"). Here's a possible sample run of the updated program (user input in **bold**):

```
Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 1
        Add: 1. Cow, 2. Chicken, 3. Pig -> 2
Animal added!

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 1
        Add: 1. Cow, 2. Chicken, 3. Pig -> 3
Animal added!

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 1
        Add: 1. Cow, 2. Chicken, 3. Pig -> 1
Animal added!

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 1
        Add: 1. Cow, 2. Chicken, 3. Pig -> 2
Animal added!

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 2
All farm animals start to sing:
Pook Pook
Oink Oink
Mooo Mooo
Pook Pook

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 1
        Add: 1. Cow, 2. Chicken, 3. Pig -> 3
Animal added!

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 2
All farm animals start to sing:
Pook Pook
Oink Oink
Mooo Mooo
Pook Pook
Oink Oink

Choose 1. Add more animal, 2. Farm chorus, 3. exit -> 3
Bye bye!
```

Your task:

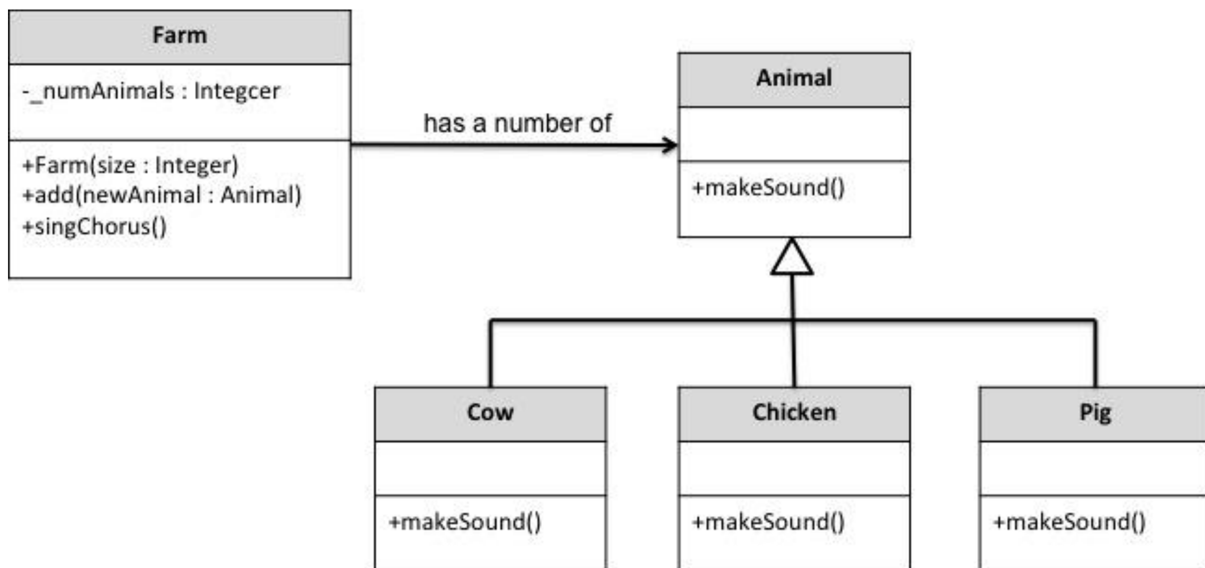
a) Modify the UML class diagram to include 3 more classes. The first class is the **Animal** class which consists of the generic *makeSound* method. The next two classes are a **Chicken** class and **Pig** class that have their own unique *makeSound* method. Please take note that the Cow, Chicken and Pig classes should be subclasses of the Animal class.

b) Write a Java source code to accommodate the new types of animals as shown. You are encouraged to apply inheritance and polymorphism in your solution.

# CS1020: Data Structures and Algorithms I

***Solution:***

A possible design is shown below:




Important observations:

- As the farm is associated with the reference of class ***`Animal`***, it can actually store reference to either ***`Cow`***, ***`Chicken`*** or ***`Pig`*** object during runtime due to **subclass substitution principle**.
- The method **`makeSound()`** in class ***`Animal`*** is overridden in each of the subclasses. This allows us to utilize the **polymorphism** effect:

```
Animal animalRef = new Chicken();     // can be Cow / Pig too

animalRef.makeSound();      // animalRef refers to a Chicken object, the
                            //  makeSound() in Chicken class is invoked.
                            // Similarly for the other animal subclasses
```

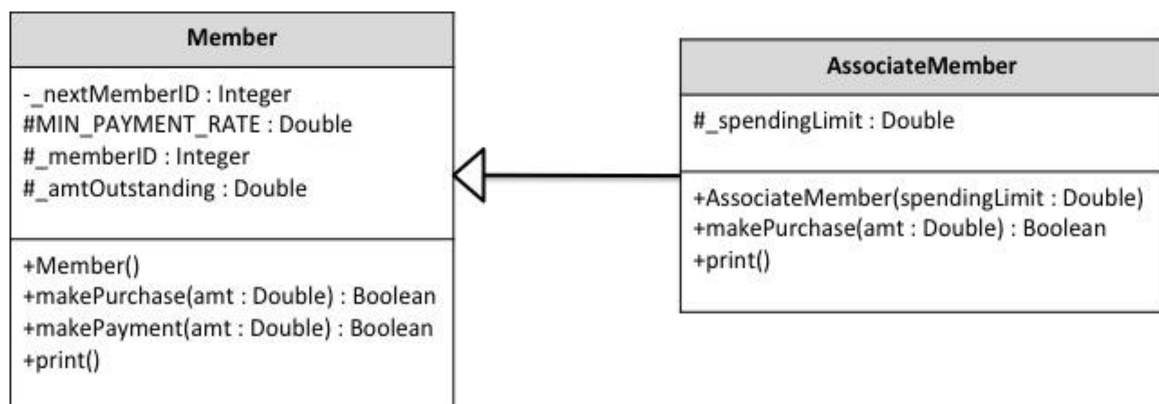
- You can even add a new animal subclass, and the existing code in Farm class can work without modification. Hence, you should try to write code for the least specific class whenever possible. This allows the same code to work for future descendant classes. For example, code that works with the **`Object`** class (the superclass of all Java classes) can work with **all classes** automatically.
- [Out of scope: for your own exploration] You can see that the **`Animal`** class serves as a "place holder" in the design, i.e. it is not meaningful to have **`Animal`** object directly. Java allows you to define an **abstract class**, which is a type of class that disallows object instantiation.

A sample implementation can be found in **`OldMcDonald.java.`**

2. **[Inheritance]** ABC Departmental Store is introducing a new Membership System to reward their customers. Similar to a credit card, a *member* is allowed to make purchases with this membership card, and the total outstanding amount will be billed to the member at the end of the month. All members have to pay a minimum of 20% of the amount outstanding to maintain their membership.

   There are three types of memberships. The first type is an ***Ordinary Membership****,* which is the most basic membership. The second type is an ***Associate Membership***, which has a spending limit. The third type is a ***Platinum Membership*** that does not have such restriction. In addition, a *Platinum Member* will enjoy a discount off the outstanding balance.

   The following is part of a UML Class Diagram for the Membership System.




   An ordinary member, simply known as **Member**, has the following information:
   - Member Identification Number (generated by Member class)
   - Amount Outstanding

   The functionalities provided by a **Member** class are:
   - Make Purchase
     - Ensure amount is positive.
     - Increases the amount outstanding.
   - Make Payment
     - Minimum payment is 20% of amount outstanding.
     - Decreases the amount outstanding.
   - Print
     - Print out the Member ID and Amount Outstanding.

The second type of members, an **Associate Member**, has the following information:

- Member Identification Number
- Amount Outstanding
- Spending Limit (Different Associate Members may have different limits)

The functionalities provided by an **Associate Member** class are:

- Make Purchase
    - Ensure amount is positive.
    - Ensure that the amount outstanding doesn't exceed the spending limit.
    - Increases the amount outstanding.
- Make Payment
    - Similar to its super class.
- Print
    - Print out the Member ID, Amount Outstanding and Spending Limit.

**Your task**:

Suppose we want to model the third type of membership: a **Platinum Member** that has the following information:

- Member Identification Number
- Amount Outstanding
- Fixed Discount Rate (It will be the same for all Platinum Members)

The functionalities provided by a **Platinum Member** class are similar to a Member class, with minor differences in the behavior:

- Make Purchase
    - Similar to its super class.
- Make Payment
    - Discount will be applied before calculating the minimum 20% payoff.
    - Decreases the amount outstanding.
- Print
    - Print out the Member ID, Amount Outstanding and Discount Rate.

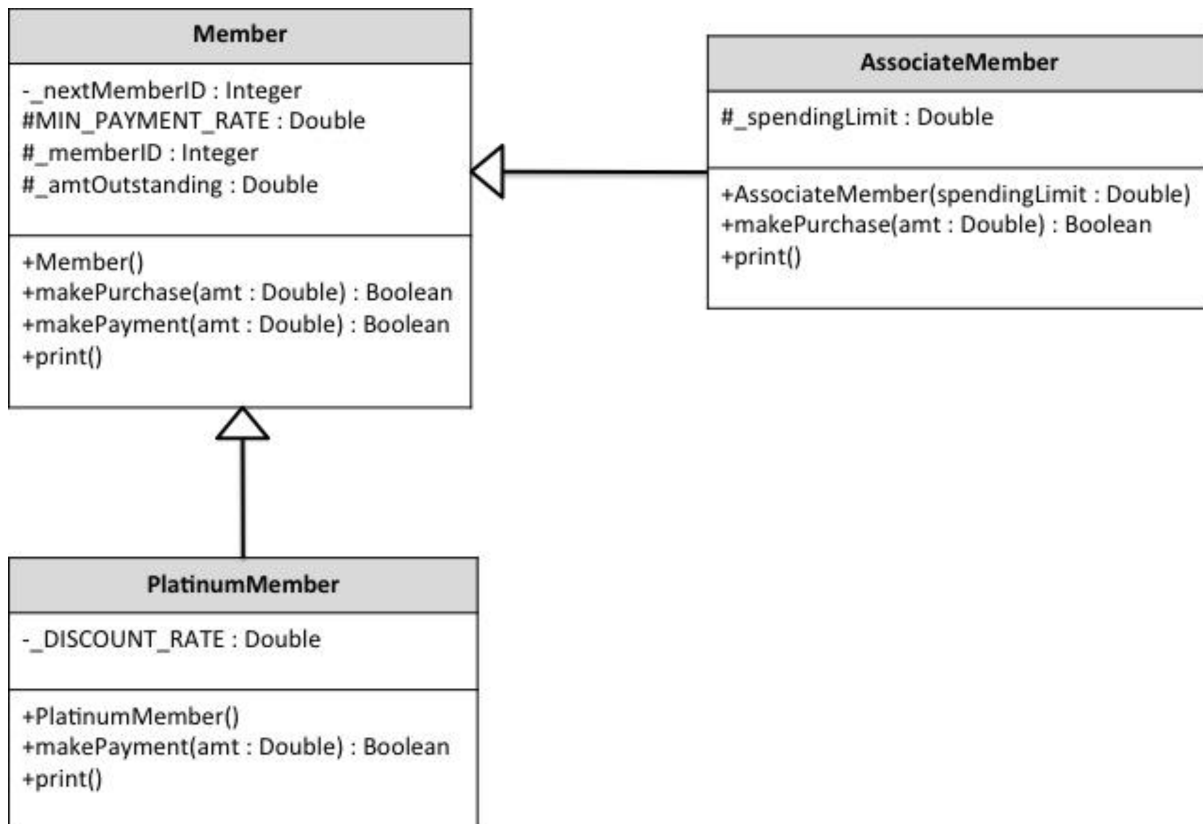
a) Modify the UML class diagram to include the **`PlatinumMember`** class.

b) Use inheritance to implement a **`PlatinumMember`** class. You are free to write a main method to test the new class.

# CS1020: Data Structures and Algorithms I

*Solution:*

The new **PlatinumMember** class should be a subclass of the **Member** class as shown in the class diagram below.




Important points to note:

- **AssociateMember** overrides the inherited **makePurchase**() method, as the method needs to ensure that the amount outstanding doesn't exceed the spending. On the other hand, the inherited **makePayment**() method can be reused without any change.
- **PlatinumMember** overrides the inherited **makePayment**() method, as they are entitled to a fix discount rate. On the other hand, the inherited **makePurchase**() method can be reused without any change.

See **Member.java and MemberTest.java** for a sample implementation of the **AssociateMember** class.

3. **[Object Oriented Modeling]** In *Question 2*, we have designed the UML class diagram for the Membership System. With these user-defined data types, we are now ready to build a *simplified version* of the Membership System Application for ABC Departmental Store which contains only the Member class from *Question 2*. We need to write a program to serve as a "Membership System" which provides the following services:

    a. Create new members.
    b. Make Purchase/Payment.
    c. Print information of all existing members.

    Assumptions:
    - The user will create no more than 100 members in a single test run.
    - The account number will start from 1000, incremented by 10 for each subsequent new member.

    An example test run session is given below (user input in **bold**):

```
**************************************
*     Membership System of CS1020    *
**************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 1

Result: Member added successfully!
Member's Identification Number: 1000
Amount Outstanding: $0.00

**************************************
*     Membership System of CS1020    *
**************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 1

Result: Member added successfully!
Member's Identification Number: 1010
Amount Outstanding: $0.00

**************************************
*     Membership System of CS1020    *
**************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 2

Member's Identification Number: 1010
Purchase Amount: 52.7

Result: Purchase successful!
Member's Identification Number: 1010
Amount Outstanding: $52.70
```

```
***************************************
*     Membership System of CS1020    *
***************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 3

Member's Identification Number: 1010
Payment Amount: 2.7

Result: Payment failed because the amount paid is less than the minimum amount
required!
Member's Identification Number: 1010
Amount Outstanding: $52.70

***************************************
*     Membership System of CS1020    *
***************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 3

Member's Identification Number: 1010
Payment Amount: 12.7

Result: Payment successful!
Member's Identification Number: 1010
Amount Outstanding: $40.00

***************************************
*     Membership System of CS1020    *
***************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 4

Result: 2 existing members.

Member's Identification Number: 1000
Amount Outstanding: $0.00

Member's Identification Number: 1010
Amount Outstanding: $40.00

***************************************
*     Membership System of CS1020    *
***************************************
Service available:
1. New Member, 2. Make Purchase, 3. Make Payment, 4. Print All, 5. Exit

Your choice: 5

Bye bye!
```

You are free to decide what message to print if an operation is unsuccessful, e.g. member number not found, payment amount is less than 20% of the amount outstanding and trying to create more than 100 members, etc.

The program implementation should be split into two parts:

---

**Part A.** (Simple Class) Skeleton of **`Member`** class:

```
class Member {

    //Constant Variable
    protected static final double MIN_PAYMENT_RATE = 0.2;

    //For a controlled memberID
    private static int _nextMemberID = 1000;

    //Instance Variables
    protected int _memberID;
    protected double _amtOutstanding;

    //Constructor
    public Member() {
        //* Your Code Here *//
    }

    //Public Methods
    public boolean makePurchase(double amt) {
        //* Your Code Here *//
    }

    public boolean makePayment(double amt) {
        //* Your Code Here *//
    }

    //Tutorial 2, Q3, Part A
    public int getMemberID() {
        //* Your Code Here *//
    }

    public void print() {
        //* Your Code Here *//
        // You may choose to write a toString() method instead
    }
}
```

Question to ponder: How do we decide what public methods to provide for a class?

---

**Part B**. (Driver Program) With the **`Member`** class in Part A, we can now write the driver code to provide the services.

<u>Suggestion</u>: Use an array to store the **`Member`** object references.

Let us modularize the program to make it more manageable. Write the following methods in a driver class **`MembershipSystemApp`**:

i. Write a static method

**`int findMemberID(Member[] arr_Members, int size, int targetMemberID);`**

Search through the members in *arr_Member* array to locate the **index** of Member object with the target member identification number, where the parameter *size* is the number of members in the system.

ii. Write a static method

```
void printAllMembers(Member[] arr_Members, int size);
```

Print out the members' information in the *arr_Member* array.

iii. Write a main method to:

- Print available services. Read user input.
- Perform the appropriate service from the user input.

The main method will mostly make use of the instance and static method specified in Parts A and B.

***Solution:***

A sample solution can be found in **`MembershipSystemApp.java`**.

Notes:

- This question provides practice for the object oriented modeling approach discussed in lecture 3.
- Instead of taking this as a simple coding question, you should take just the program specification (the description of the problem) and attempt to design a complete Java program. Compare with the design proposed in this question and learn from the differences.
- The class **`Member`** is fleshed out based on the **capabilities needed for the solution**. For example, we need to know the member's identification number in order to locate the **`Member`** object indicated by the user. So, an accessor method like **`getMemberID()`** should be provided by the **`Member`** class.
  - When you design a class, instead of trying to imagine all possible general usages, you should concentrate first on the minimum required capabilities.
  - In large-scale software project, we will use the specific usages (use cases) to determine the class design.
- The static methods in Part B are designed to modularize the driver class. In general, we will follow the top-down design approach to determine the major steps in the driver code. Each major step is further broken into simpler steps if needed. If a step is complicated enough or useful enough, it is packaged into a method. For example, the **`findMemberID()`** static method capture the useful logic to look for a **`Member`** object in an array.
- Some will notice the fact that the member's identification number is in sequential order, which allows us to locate a **`Member`** object in the array without searching. This is definitely a correct observation. This solution is designed to be more general so that the ordering of the **`Member`** does not matter.

# CS1020: Data Structures and Algorithms I

**Source Code for Q1**

***OldMcDonald*** Class

```
import java.util.Scanner;

class Animal {
    public void makeSound() {
        System.out.println("Aaaaanimaaal");
    }
}

class Cow extends Animal {
    public void makeSound() {
        System.out.println("Mooo Mooo");
    }
}

class Chicken extends Animal {
    public void makeSound() {
        System.out.println("Pook Pook");
    }
}

class Pig extends Animal {
    public void makeSound() {
        System.out.println("Oink Oink");
    }
}


class Farm {
//An array of animals references
    private Animal[] animalsOnFarm;
//Number of animals on farm so far
    private int _numAnimal;

    public Farm(int size) {
        animalsOnFarm = new Animal[size];

        //Initialize all references to null
        for (int i = 0; i < size; i++) {
            animalsOnFarm[i] = null;
        }

        _numAnimal = 0;
    }

    public boolean add(Animal newAnimal) {
        //Can we take in another animal?
        if (_numAnimal == animalsOnFarm.length) {
            return false;
        }

        //Add animal to farm
        animalsOnFarm[_numAnimal] = newAnimal;
        _numAnimal++;

        return true;
    }
```

```
    public void singChorus() {

        for (int i = 0; i < _numAnimal; i++) {
            animalsOnFarm[i].makeSound();
            // polymorphism is invoked here. Which makeSound method is called
            // depends on the object(a chicken, cow, or pig) referred by
            // animalsOnFarm[i] at the runtime.
        }
    }
}


class OldMcDonald {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);
        int userChoice, animalType;
        Farm myFarm;
        Animal newAnimal;

        myFarm = new Farm(10);        //10 is just an example
        newAnimal = null;               //Initialize a reference as null
        // if you don't have any valid
        // initialization

        do {
            System.out.print("Choose 1. Add more animal, 2. Farm chorus, 3. exit ->
");
            userChoice = sc.nextInt();

            switch (userChoice) {
                case 1:
                    System.out.print("\tAdd: 1. Cow, 2. Chicken, 3. Pig -> ");
                    animalType = sc.nextInt();

                    //Check animal type range
                    if (animalType >= 1 && animalType <= 3) {
                        switch (animalType) {
                            case 1:
                                newAnimal = new Cow();
                                break;
                            case 2:
                                newAnimal = new Chicken();
                                break;
                            case 3:
                                newAnimal = new Pig();
                                break;
                        }
                        if (myFarm.add(newAnimal)) {
                            //You can enhance this part by printing
                            // out the actual animal type added
                            //Possible approaches:
                            //   - use String (lecture 4)
                            //   - define toString for the animal subclasses
                            System.out.println("Animal added!");

                        } else {
                            System.out.println("Farm full!");
                        }
                    } else {        //Animal type is not valid
                        System.out.println("Invalid Animal Type!");
                    }
                    break;
```

```
                case 2:
                    System.out.println("All farm animals start to sing:");
                    myFarm.singChorus();
                    break;
                case 3:
                    System.out.println("Bye bye!");
                    break;
            }
        } while (userChoice != 3);

    }
}
```

**//Source Code for Q2**

***Member*** Class

```
public class Member {

    private static int _nextMemberID = 1000;
    protected static final double MIN_PAYMENT_RATE = 0.2;

    protected int _memberID;
    protected double _amtOutstanding;

    public Member() {
        _memberID = _nextMemberID;
        _amtOutstanding = 0;
        _nextMemberID += 10;
    }

    public boolean makePurchase(double amt) {
        if (amt <= 0) {
            return false;
        }
        _amtOutstanding += amt;
        return true;
    }

    public boolean makePayment(double amt) {
        if (amt < _amtOutstanding * MIN_PAYMENT_RATE) {
            return false;
        }
        _amtOutstanding -= amt;
        return true;
    }

    public void print() {
        System.out.println("Member's Identification Number: " + _memberID);
        System.out.printf("Amount Outstanding: $%.2f\n", _amtOutstanding);
    }

}

class AssociateMember extends Member {

    protected double _spendingLimit;

    public AssociateMember(double spendingLimit) {
        super();
        _spendingLimit = spendingLimit;
    }

    public boolean makePurchase(double amt) {
        if (amt <= 0 || _amtOutstanding + amt > _spendingLimit) {
            return false;
        }
        _amtOutstanding += amt;
        return true;
    }

    public void print() {
        super.print();
        System.out.printf("Spending Limit: $%.2f\n", _spendingLimit);
    }
}
```

```
class PlatinumMember extends Member {

    private static final double DISCOUNT_RATE = 0.1;

    public PlatinumMember() {
        super();
    }

    public boolean makePayment(double amt) {
        if (amt <= 0 || amt < _amtOutstanding * (1 - DISCOUNT_RATE) *
MIN_PAYMENT_RATE) {
            return false;
        }
        _amtOutstanding -= amt / (1 - DISCOUNT_RATE);
        return true;
    }

    public void print() {
        super.print();
        System.out.printf("Discount Rate: %.2f%c\n", DISCOUNT_RATE * 100 , '%');
    }
}
```

***MemberTest*** Class

```
public class MemberTest {

    public static void main(String[] args) {

        Member mem = new Member();
        AssociateMember aMem = new AssociateMember(500.00);

        //Simple code to show the differences

        System.out.println("Ordinary Member:");
        mem.print();
        System.out.println();
        System.out.println("Associate Member:");
        aMem.print();
        System.out.println();

        //Lets now try to make a purchase of $300.
        double purchaseAmt = 300.0;
        System.out.printf("Ordinary member attempt to make a purchase of $%.2f!\n",
purchaseAmt);
        if (mem.makePurchase(purchaseAmt)) {
            printMessage("Ordinary", purchaseAmt, 0);
        } else {
            printMessage("Ordinary", purchaseAmt, 1);
        }

        System.out.printf("\nAssociate member attempt to make a purchase of
$%.2f!\n", purchaseAmt);
        if (aMem.makePurchase(purchaseAmt)) {
            printMessage("Associate", purchaseAmt, 0);
        } else {
            printMessage("Associate", purchaseAmt, 1);
        }
```

```
        //What are the members' information after the purchase?
        System.out.println("\nAfter the purchase operation");
        System.out.println("Ordinary Member:");
        mem.print();
        System.out.println("\nAssociate Member:");
        aMem.print();
        System.out.println();

        //Lets now try to make another purchase of $300.
        System.out.printf("Ordinary member attempt to make a purchase of $%.2f!\n",
purchaseAmt);
        if (mem.makePurchase(purchaseAmt)) {
            printMessage("Ordinary", purchaseAmt, 0);
        } else {
            printMessage("Ordinary", purchaseAmt, 1);
        }

        System.out.printf("\nAssociate member attempt to make a purchase of
$%.2f!\n", purchaseAmt);
        if (aMem.makePurchase(purchaseAmt)) {
            printMessage("Associate", purchaseAmt, 0);
        } else {
            printMessage("Associate", purchaseAmt, 1);
        }

        //What are the members' information after the 2 purchases?
        System.out.println("\nAfter the purchase operation");
        System.out.println("Ordinary Member:");
        mem.print();
        System.out.println("\nAssociate Member:");
        aMem.print();
        System.out.println();

        //Now, lets now try to pay off the debt?
        double paymentAmt = 100.0;
        System.out.printf("Ordinary member attempt to pay $%.2f!\n", paymentAmt);
        if (mem.makePayment(paymentAmt)) {
            printMessage("Ordinary", paymentAmt, 2);
        } else {
            printMessage("Ordinary", paymentAmt, 3);
        }

        System.out.printf("\nAssociate member attempt to pay $%.2f!\n",
paymentAmt);
        if (aMem.makePayment(paymentAmt)) {
            printMessage("Associate", paymentAmt, 2);
        } else {
            printMessage("Associate", paymentAmt, 3);
        }

        //What are the members' information after all the transactions?
        System.out.println("\nAfter the payment operation");
        System.out.println("Ordinary Member:");
        mem.print();
        System.out.println("\nAssociate Member:");
        aMem.print();

    }
```

```
    // This method is responsible to print all the various messages
    // Input: A membership type (Ordinary/Associate/Platinum),
    // an Amount and a Message Code
    //      Code 0: Purchase Successful
    //      Code 1: Purchase Failed with Reason
    //      Code 2: Payment Successful
    //      Code 3: Payment Failed with Reason
    // Output: Respective message with regards to the message code.
    //
    // Note: printf (introduced in Java 1.5) is use instead of println because
    //       a floating number can be easily formatted. If you use println, we
    //       have to make use of the NumberFormat class.
    //       It is up to your preference.

    private static void printMessage(String memberType, double amt, int code) {
        System.out.print(memberType + " member ");
        switch (code) {
            case 0:
                System.out.printf("made a purchase of $%.2f!", amt);
                break;
            case 1:
                System.out.printf("FAILED to make a purchase of $%.2f,\n" +
                    "because the amount outstanding has exceeded the purchase
limit!", amt);
                break;
            case 2:
                System.out.printf("made a payment of $%.2f!", amt);
                break;
            case 3:
                System.out.printf("FAILED to make a payment of $%.2f,\n" +
                    "because the amount paid is less than the minimum amount
required!", amt);
                break;

        }
        System.out.println();
    }
}
```

**//Source Code for Q3**

***MembershipSystemApp*** Class

```
import java.util.Scanner;

class Member {

    //Constant Variable
    protected static final double MIN_PAYMENT_RATE = 0.2;

    //For a controlled memberID
    private static int _nextMemberID = 1000;

    //Instance Variables
    protected int _memberID;
    protected double _amtOutstanding;

    //Constructor
    public Member() {
        _memberID = _nextMemberID;
        _amtOutstanding = 0;
        _nextMemberID += 10;
    }

    //Public Methods
    public boolean makePurchase(double amt) {
        if (amt <= 0) {
            return false;
        }
        _amtOutstanding += amt;
        return true;
    }

    public boolean makePayment(double amt) {
        if (amt < _amtOutstanding * MIN_PAYMENT_RATE) {
            return false;
        }
        _amtOutstanding -= amt;
        return true;
    }

     //Tutorial 2, Q3, Part A

    public int getMemberID() {
        return _memberID;
    }

    public void print() {
        // You may choose to write a toString() method instead
        System.out.println("Member's Identification Number: " + _memberID);
        System.out.printf("Amount Outstanding: $%.2f\n", _amtOutstanding);
    }
}




public class MembershipSystemApp {

    //Purpose: Look for Member object with "targetMemberID".
    //Pre: arr_Members[0... currNumOfMembers-1] are valid Member references
    //Post: This function doesnt change the Member object(s)
```

```
    //Return:
    //    - index (0.. currNumOfMembers-1) of the Member object if found
    //    - -1, otherwise
     //Tutorial 2, Q3, Part B.i

    private static int findMemberID(Member[] arr_Members, int currNumOfMembers, int
targetMemberID) {
        for (int i = 0; i < currNumOfMembers; i++) {
            if (arr_Members[i].getMemberID() == targetMemberID) {
                return i;
            }
        }
        return -1;
    }

    //Purpose: Print out account information for Member objects
    //Pre: baArr[0... currNumOfMembers-1] are valid Member references
    //Post: This function doesnt change the Member object(s)
    //Tutorial 2, Q3, Part B.ii

    private static void printAllMembers(Member arr_Members[], int currNumOfMembers)
{
        for (int i = 0; i < currNumOfMembers; i++) {
            arr_Members[i].print();
            System.out.println(); //to print an empty line
        }
    }


     //Tutorial 2, Q3, Part B.iii

    public static void main(String[] args) {

        final int MAX_NUM_OF_MEMBER = 100;

        int currNumOfMembers = 0, targetMemberIdx = 0;
        Member[] arr_Members = new Member[MAX_NUM_OF_MEMBER];

        //variables for user input
        Scanner sc = new Scanner(System.in);
        int choice, userMemberID;
        double userAmt;

        //Good pratice: set reference to null if you dont know the
        //                starting value
        for (int i = 0; i < arr_Members.length; i++) {
            arr_Members[i] = null;
        }

        do {
            //Print "Menu" and get user choice
            System.out.println("**************************************");
            System.out.println("*     Membership System of CS1020    *");
            System.out.println("**************************************");
            System.out.println("Service available: ");
            System.out.println("1. New Member, 2. Make Purchase, 3. Make Payment,
4. Print All, 5. Exit\n");
            System.out.print("Your choice: ");
            choice = sc.nextInt();
            System.out.println();   //Print an empty line for output readability

            //Perform appropriate service;
            switch (choice) {
```

```
                case 1:         //New Member
                    if (cursNumOfMembers < MAX_NUM_OF_MEMBER) {
                        arr_Members[currNumOfMembers] = new Member();
                        currNumOfMembers++;

                        //Output result
                        System.out.println("Result: Member added successfully!");
                        arr_Members[currNumOfMembers - 1].print();
                        System.out.println();
                    }
                    else {
                        System.out.println("Result: Maximum Members reached!");
                    }
                    break;

                case 2:         //Make Purchase
                    System.out.print("Member's Identification Number: ");
                    userMemberID = sc.nextInt();
                    System.out.print("Purchase Amount: ");
                    userAmt = sc.nextDouble();
                    System.out.println();

                    //Look for Member object in the array first
                    targetMemberIdx = findMemberID(arr_Members, currNumOfMembers,
userMemberID);
                    if (targetMemberIdx != -1) {
                        if (arr_Members[targetMemberIdx].makePurchase(userAmt)) {
                            System.out.println("Result: Purchase successful!");
                        }
                        else {
                            System.out.println("Result: Purchase failed because
amount cannot be negative!");
                        }
                        arr_Members[targetMemberIdx].print();
                        System.out.println();
                    }
                    else {
                        System.out.println("Result: Invalid Member Identification
Number");
                    }
                    break;

                case 3:         //Make Payment
                    System.out.print("Member's Identification Number: ");
                    userMemberID = sc.nextInt();
                    System.out.print("Payment Amount: ");
                    userAmt = sc.nextDouble();
                    System.out.println();

                    //Look for Member object in the array first
                    targetMemberIdx = findMemberID(arr_Members, currNumOfMembers,
userMemberID);
                    if (targetMemberIdx != -1) {
                        if (arr_Members[targetMemberIdx].makePayment(userAmt)) {
                            System.out.println("Result: Payment successful!");
                        }
                        else {
                            System.out.println("Result: Payment failed because the
amount paid is less than the minimum amount required!");
                        }
                        arr_Members[targetMemberIdx].print();
                        System.out.println();
                    }
```

```
                    else {
                        System.out.println("Result: Invalid Member Identification 
Number");
                    }
                    break;

                case 4:         //Print All
                    System.out.println("Result: " + currNumOfMembers + " existing 
members.\n");
                    printAllMembers(arr_Members, currNumOfMembers);
                    break;

                case 5:         //Exit
                    System.out.println("Bye bye!");
                    break;

                default:
                    System.out.println("Result: Invalid choice!");
            }

        } while (choice != 5);
    }

}
```