**Exercise 1**

**Encapsulation:**

Encapsulation involves making properties private and giving a set of public mutator and accessor methods.

The following is a bank account example involving encapsulation:

public class Bank\_Account {

//Variables

private int acc\_num;

private int acc\_bal;

//Mutator method for depositing money

public void deposit(int dep ) {

acc\_bal = dep;

}

//Accessor method for account number

public int get\_accNum(){

return acc\_num;

}

//Accessor method for account balance

public void deposit(int amount ) {

if (amount < 0) {

System.out.println("Invalid amount");

}

else {

acc\_bal = acc\_bal + amount;

}

}

}

Now let us say there is a Hacker class that attempts to gain access and deposit an invalid amount into another user’s account. For example:

public class Hacker {

//Creating an instance of Bank\_Account class in an attempt to gain access to its properties

Bank\_Account b\_acc = new Bank\_Account();

b\_acc.acc\_bal = -200; //Invalid statement

b\_acc.deposit(-200); //Invalid statement

}

However the *b\_acc.acc\_bal = -200;* statement is illegal because the variables in the Bank\_Account are set to private and so they can only be accessed via corresponding methods within that class. Therefore such access through Hacker class is not permitted.

The Hacker then attempts use the deposit method to carry out the operation using the statement *b\_acc.deposit(-200);*. However the method implementation of the Bank\_Account class rejects this because the if statement within the deposit method checks for negative amount.

In conclusion, encapsulation makes your class more secure because you are hiding your properties of a class from rest of the program through private declaration. In addition to this, you are only making changes and access to these properties within the class through mutator and accessor methods, which have conditions only known within that class.

**Inheritance:**

Classes in java can be extended by creating new classes which inherit the characteristics of the base class. Base class contains the variables and methods that can be used in the subclasses and this is known as inheritance. Inheritance is effective because it allows you to gain access to common properties such as methods by the subclasses rather than creating it individually. So in other words, if classes that you intend to create have a common relationship, then it is important to establish inheritance.

The below example involves shapes, specifically triangle and rectangle. What they both share in common is that they are polygon and therefore inheritance can be established. They both also contain the property of width and height.

//Superclass  
public class Polygon {

// They are protected so subclass can gain access  
protected int height;

protected int width;

//Setting the width and height  
public void create\_val(int x, int y) {

height = x;  
 width = y;

}

}

//Subclass  
public class Rectangle extends Polygon {

public double get\_area() {

return (height \* width);

}

}

//Subclass  
public class Triangle extends Polygon {

public double get\_area() {

return (height \* width)/2;

}

}

Since height and width are not declared as variables in the subclasses, it automatically accesses the properties from the superclass because it is declared as protected.

Below is the main class program showing how inheritance is used.

public static void main (String[] args) {

//Inheritance also allows you to access the methods from the superclass while creating instances of //the subclasses as shown below:

Rectangle rect = new Rectangle();  
 Triangle tri = new Triangle();

rect.create\_val(20, 20);  
 tri.create\_val(20, 20);

System.out.println(rect.get\_area());  
 System.out.println(tri.get\_area());

}

**Polymorphism:**

Polymorphism is sharing a common interface for multiple types but having different implementations for different types. In the example below, polymorphism allows you to use the method from the base class Food called eat(). This is polymorphic because it can be either Chicken or Tuna. This is also an example of method overriding because eat() method is overridden in subclass Chicken and Tuna.

//Base class  
public class Food {

public void eat() {

System.out.println(“This food is great”);

}

}

//Sub class  
public class Chicken extends Food {

@Override   
public void eat() {

System.out.println("This chicken is great");

}

}

//Sub class  
public class Tuna extends Food {

@Override   
public void eat() {

System.out.println("This tuna is great");

}

}

The program below demonstrates the use of polymorphism. This is a polymorphic array example, which stores objects of different classes of superclass type.

public class PolymorphismExample {

public static void main (String args[]) {

//An array called meals of food (superclass)   
//Therefore it can hold objects of Chicken and Tuna  
Food meals [] = new Food [2];

meals [0] = new Chicken();

meals [1] = new Tuna();

//Looping through the array of objects which calls eat() method for each class  
//Doing this prevents you from creating the different objects separately   
for (int i = 0; i < 2; ++i) {

meals [i].eat();

}

}

}

In conclusion, for polymorphic array you can assign different objects to variables as long the reference variable is of the superclass type. That way, instead of having to build a new object for using a method, you can simply build one reference variable of the superclass and assign it to objects of subclasses.

**Abstraction:**

Let us first create some example classes:

public abstract class Game {

//Abstract method  
 public abstract void draw();

public static void main (String[] args) {

Game g\_obj = new Game(); //Invalid statement as it cannot be instantiated

Player p\_obj = new Player(); //Valid statement as Player object is instantiable

}

}

public class Player extends Game {

//Abstract method must be implemented, otherwise the Player class is invalid  
@Override  
public void draw() {

}

}

The purpose of an abstract class is to prevent you from making an instance of the object within the same class. Abstract classes exist to be extended by subclasses, they cannot be instantiated as shown in the Game class. However you can see that the Player object can be successfully created because the Player class is not abstract. An abstract class also exists to represent and prevent you from creating an instance of an entity that is abstract. Using the context above, a person would have a better idea in creating a new player than asking someone to create a new game object, because a new game object could be a player but it could also be many other instances such as enemy, UI, and so forth.

In abstract classes, you can also use abstract methods. The purpose of declaring abstract methods is that subclasses extending a class containing abstract method, such as Game, must implement its method. In this context, Player class must provide the implementation for draw() method using override.