Exercise 1:

**Encapsulation**

Nearly all classes have data associated with them. This means that you can set data about the object which it will remember and then you can read it back from the object later.

For example: Ticket class would have a ‘movieTitle’ data variable and an object of that class to represent a ticket to a movie Wicked, it would have ‘movieTitle’ variable set to “Wicked”.

The encapsulation, literally ‘to hold inside’ and the all idea about the encapsulation is to hide the details from users. Private data can only be accessed within the class. In order to access private data variables programmer have to write getters and setters.

Encapsulation improves maintainability, accessibility and flexibility.

**Inheritance**

The main purpose of Inheritance is code reusability. The code that is written in the superclass doesn’t have to be rewritten. Example:

Public class Animal{

Private int age;

Private String name;

Public Animal(int age, String name){

This.age = age;

This.name = name;

}

Public int getAge()

{

return age;

}

Public String getName(){

Return name;

}

}

Public class Dog extends Animal{

Public Dog(int age, String name){super(age,name);}

Public void details(){ System.out.print(“Dog name : ” super.getName());}

}

**Polymorphism**

Literally means a state of having many shapes or capacity to take on different shapes

2 users of polymorphism:

1. Polymorphic collections – allows to have an array (collection) of different types of objects. Example: Animal list = {

new Dog(), new Cat(), new Cow()

}

For(int i = 0; i < list.length; i++){

System.out.println(list[i].makeSound());

// Polymorphic calls are made to the makeSound()

// all animals has to implement make sound method

}

Every class inherits from java.land.Object, so we can have polymorphic collections of objects of any class.

1. Polymorphic parameters Example:

We want vet to be able to vaccinate various animals. It would be tedious to define overloaded methods for each class like Cat, Dog and Cow. Solution to that would be to take in polymorphic parameter that could refer to an instance of either the super class or the subclass.

Class Vet{

void vaccinate (Animal a){

if(a instanceOf Cat){

Cat c = (Cat) a;

System.out.print(“Cat is vaccinated!!” + c.makeSound());

}

else if(a instanceOf Dog){

Dog d = (Dog) a;

System.out.print(“Dog “ + d.name + ” is vaccinated!!”);

}

}

**Abstraction**

Classes are generally written so that objects can be instantiated from them to do something useful, but in some cases it is useful to define classes for which the programmer never intends to instantiate any objects. They are used as super classes in inheritance situation. No objects of abstract superclasses can be instantiated.

Example:

Class Shape could have 3 subclasses such as Square. Shape is too general and there wouldn’t be any point to call method drawShape() as it is too general. Those kind of classes useful just to take an advantage of polymorphism.

Public abstract class Shape{} // abstract and has abstract method drawShape()

Public abstract class TwoDimensional extends Shape{} // abstract and has abstract method drawShape()

Public class Square extends TwoDimensional{} // concrete and has to implement abstract drawShape() method.

In the example above of the Shape superclass and associated derived subclasses, reconsider the method drawShape() belonging to the superclass.

Rules:

* An abstract method is declared abstract and it has no body.
* An abstract method cannot appear in non-abstract classes.
* If an abstract class contains abstract methods, they must be over-ridden in the subclasses unless the subclasses are also abstract, then the concrete class has to implement an abstract method.